Introduction to Software Engineering

Edited by R.P. Lano
(Version 0.1)

Table of Contents

NOTE: THIS IS STILL WORK IN PROGRESS!!!

Note: current version of this book can be found at
http://en.wikibooks.org/wiki/Introduction_to_Software_Engineering

Introduction

Preface

When preparing an undergraduate class on Software Engineering, I found two things: First, there are quite a few good articles in Wikipedia covering different aspects related to software engineering, and second, that for a beginner those articles contain too much information and too few examples. Hence the idea for this book came about, to take the relevant articles from Wikipedia, combine them, edit them, fill in the missing pieces, put them in context and create an introductory wikibook out of them.

The hope is that this can be used as a textbook for an introductory software engineering class. This is also, why in the final version I want to have a pdf version and a printed version, since this will make it more useful for a textbook.

As for the philosophy behind this book: brevity is preferred to completeness, and examples are preferred to theory. If this effort was successful, you be the judge of it, and if you have suggestions for improvement, just use the 'Edit' button!

My special thanks go to Adrignola and Kayau who did the tedious work of importing the original articles (with all their history) from Wikipedia to Wikibooks!

Introduction

Let us first look into the question of what is software engineering? It is based on software development, sometimes also called 'programming', but software development is just a part of software engineering, it is a pre-requisite to it.

Developers Work in Teams

In your beginning semesters you were coding as individuals. The problems you were solving were small enough so one person could master them. In the real world this is different: the problem sizes and time constraints are such that only teams can solve those problems.

For teams to work effectively they need a language to communicate (UML). Also teams do not consist only of developers, but also of testers, architects, system engineers and most importantly the customer. So we need to learn about what makes good teams, how to communicate with the customer, and how to document not only the source code, but everything related to the software project.

New Language

In previous courses we learned languages, such as Java or C++, and how to turn ideas into code. But these ideas are independent of the language. With UML we will see a way to describe code language independant, and more importantly, we learned to think in one higher level of abstraction. UML can be an invaluable communication and documentation tool.

We will learn to see the big picture: patterns. This gives us even one higher level of abstraction. Again this increases our vocabulary to give us the means to communicate with our peers more effectively. Also it is a fantastic way to learn from our seniors. This is essential for designing large software systems.

Measurement

Also just being able to write software, doesn't mean that the software is any good. Hence, we will discover what makes good software, and how to measure software quality. On one hand we should be able to analyse existing source code though static analysis and measuring metrics, but also how do we guarantee that our code meets certain quality standards? Testing is also important in this context, it guarantees high quality products.

New Tools

Up to now, you may known about an IDE, a compiler and a debugger. But there are many more tools at the disposal of a software engineer. There are tools that allow us to work in teams, to document our software, to assist and monitor the whole development effort. There are tools for software architects, tools for testing and profiling, automation and re-engineering.

History

When the first modern digital computers appeared in the early 1940s,[1] the instructions to make them operate were wired into the machine. Practitioners quickly realized that this design was not flexible and came up with the "stored program architecture" or von Neumann architecture. Thus the first division between "hardware" and "software" began with abstraction being used to deal with the complexity of computing.

Programming languages started to appear in the 1950s and this was also another major step in abstraction. Major languages such as Fortran, ALGOL, and COBOL were released in the late 1950s to deal with scientific, algorithmic, and business problems respectively. E.W. Dijkstra wrote his seminal paper, "Go To Statement Considered Harmful",[2] in 1968 and David Parnas introduced the key concept of modularity and information hiding in 1972[3] to help programmers deal with the ever increasing complexity of software systems. A software system for managing the hardware called an operating system was also introduced, most notably by Unix in 1969. In 1967, the Simula language introduced the object-oriented programming paradigm.

These advances in software were met with more advances in computer hardware. In the mid 1970s, the microcomputer was introduced, making it economical for hobbyists to obtain a computer and write software for it. This in turn led to the now famous Personal Computer (PC) and Microsoft

Windows. The Software Development Life Cycle or SDLC was also starting to appear as a consensus for centralized construction of software in the mid 1980s. The late 1970s and early 1980s saw the introduction of several new Simula-inspired object-oriented programming languages, including Smalltalk, Objective-C, and C++.

Open-source software started to appear in the early 90s in the form of Linux and other software introducing the "bazaar" or decentralized style of constructing software.[4] Then the World Wide Web and the popularization of the Internet hit in the mid 90s, changing the engineering of software once again. Distributed systems gained sway as a way to design systems, and the Java programming language was introduced with its own virtual machine as another step in abstraction. Programmers collaborated and wrote the Agile Manifesto, which favored more lightweight processes to create cheaper and more timely software.

The current definition of *software engineering* is still being debated by practitioners today as they struggle to come up with ways to produce software that is "cheaper, better, faster". Cost reduction has been a primary focus of the IT industry since the 1990s. Total cost of ownership represents the costs of more than just acquisition. It includes things like productivity impediments, upkeep efforts, and resources needed to support infrastructure. [5] [6]

References

1. ↑ Leondes (2002). *intelligent systems: technology and applications*. CRC Press. ISBN 9780849311215.
2. ↑ Dijkstra, E. W. (March 1968). "Go To Statement Considered Harmful". *Wikipedia:Communications of the ACM* **11** (3): 147–148. doi:10.1145/362929.362947. http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF. Retrieved 2009-08-10.
3. ↑ Parnas, David (December 1972). "On the Criteria To Be Used in Decomposing Systems into Modules". *Wikipedia:Communications of the ACM* **15** (12): 1053–1058. doi:10.1145/361598.361623. http://www.acm.org/classics/may96/. Retrieved 2008-12-26.
4. ↑ Raymond, Eric S. *The Cathedral and the Bazaar*. ed 3.0. 2000.
5. ↑ Software engineering
6. ↑ History of software engineering

Software Engineer

A **software engineer** is an engineer who applies the principles of software engineering to the design, development, testing, and evaluation of the software and systems that make computers or anything containing software work.

Overview

Prior to the mid-1990s, software practitioners called themselves *programmers* or *developers*, regardless of their actual jobs. Many people prefer to call themselves *software developer* and *programmer*, because most widely agree what these terms mean, while *software engineer* is still being debated. A prominent computing scientist, E. W. Dijkstra, wrote in a paper that the coining of the term *software engineer* was not a useful term since it was an inappropriate analogy, "The

existence of the mere term has been the base of a number of extremely shallow --and false-- analogies, which just confuse the issue...Computers are such exceptional gadgets that there is good reason to assume that most analogies with other disciplines are too shallow to be of any positive value, are even so shallow that they are only confusing."[1]

The term *programmer* has often been used as a pejorative term to refer to those without the tools, skills, education, or ethics to write good quality software. In response, many practitioners called themselves *software engineers* to escape the stigma attached to the word *programmer*. In many companies, the titles *programmer* and *software developer* were changed to *software engineer*, for many categories of programmers.

These terms cause confusion, because some denied any differences (arguing that everyone does essentially the same thing with software) while others use the terms to create a difference (because the terms mean completely different jobs).

A state of the art

In 2004, Keith Chapple of the U. S. Bureau of Labor Statistics counted 760,840 software engineers holding jobs in the U.S.; in the same period there were some 1.4 million practitioners employed in the U.S. in all other engineering disciplines combined.[2] The label software engineer is used very liberally in the corporate world. Very few of the practicing software engineers actually hold Engineering degrees from accredited universities. In fact, according to the Association for Computing Machinery, "most people who now function in the U.S. as serious software engineers have degrees in computer science, not in software engineering".

Education

About half of all practitioners today have computer science degrees. A small, but growing, number of practitioners have software engineering degrees. In 1987 Imperial College London introduced the first three-year software engineering Bachelor's degree in the UK and the world. Since then, software engineering undergraduate degrees have been established at many universities. A standard international curriculum for undergraduate software engineering degrees was recently defined by the CCSE. As of 2004, in the U.S., about 50 universities offer software engineering degrees, which teach both computer science and engineering principles and practices. Steve McConnell opines that because most universities teach computer science rather than software engineering, there is a shortage of true software engineers.[3] ETS University and UQAM were mandated by IEEE to develop the SoftWare Engineering BOdy of Knowledge SWEBOK, which has become an ISO standard describing the body of knowledge covered by a software engineer.

Other degrees

In business, some software engineering practitioners have Management Information Systems (MIS) degrees. In embedded systems, some have electrical engineering or computer engineering degrees, because embedded software often requires a detailed understanding of hardware. In medical software, practitioners may have medical informatics, general medical, or biology degrees. Some practitioners have mathematics, science, engineering, or technology degrees. Some have philosophy (logic in particular) or other non-technical degrees. And, others have no degrees.

Profession

Employment

Most software engineers work as employees or contractors. Software engineers work with businesses, government agencies (civilian or military), and non-profit organizations. Some software engineers work for themselves as freelancers. Some organizations have specialists to perform each of the tasks in the software development process. Other organizations required software engineers to do many or all of them. In large projects, people may specialize in only one role. In small projects, people may fill several or all roles at the same time. Specializations include: in industry (requirements analysts, software architects, software developers, software testers, technical support, project managers) and in academia (educators and researchers).

There is considerable debate over the future employment prospects for Software Engineers and other Information Technology (IT) Professionals. For example, an online futures market called the Future of IT Jobs in America attempts to answer whether there will be more IT jobs, including software engineers, in 2012 than there were in 2002.

Certification

Professional certification of software engineers is a contentious issue. Some see it as a tool to improve professional practice. Most successful certification programs in the software industry are oriented toward specific technologies, and are managed by the vendors of these technologies. These certification programs are tailored to the institutions that would employ people who use these technologies.

The Association for Computing Machinery (ACM) had a professional certification program in the early 1980s, which was discontinued due to lack of interest. Actually the ACM made an explicit decision *not* to continue with certification. As of 2006, the IEEE had certified over 575 software professionals.[4]

Impact of globalization

Many students in the developed world have avoided degrees related to software engineering because of the fear of offshore outsourci]] (importing software products or services from other countries) and of being displaced by foreign visa workers.[5] Although government statistics do not currently show a threat to software engineering itself; a related career, computer programming does appear to have been affected.[6][7] Often one is expected to start out as a computer programmer before being promoted to software engineer. Thus, the career path to software engineering may be rough, especially during recessions.

Some career counselors suggest a student also focus on "people skills" and business skills rather than purely technical skills because such "soft skills" are allegedly more difficult to offshore.[8] It is the quasi-management aspects of software engineering that appear to be what has kept it from being impacted by globalization.[9]

Debates within software engineering

Controversies over the term Engineer

Some people believe that *software engineering* implies a certain level of academic training, professional discipline, adherence to formal processes, and especially legal liability that often are not applied in cases of software development. A common analogy is that working in construction does not make one a civil engineer, and so writing code does not make one a software engineer. It is disputed by some - in particular by the Canadian Professional Engineers Ontario (PEO) body, that the field is mature enough to warrant the title "engineering". The PEO's position was that "software engineering" was not an appropriate name for the field since those who practiced in the field and called themselves "software engineers" were not properly licensed professional engineers, and that they should therefore not be allowed to use the name.[10]

References

1. ↑ http://www.cs.utexas.edu/users/EWD/transcriptions/EWD06xx/EWD690.html E.W.Dijkstra Archive: The pragmatic engineer versus the scientific designer
2. ↑ Bureau of Labor Statistics, U.S. Department of Labor, *USDL 05-2145: Occupational Employment and Wages, November 2004*
3. ↑ McConnell, Steve (July 10, 2003. *Professional Software Development: Shorter Schedules, Higher Quality Products, More Successful Projects, Enhanced Careers.* ISBN 978-0321193674.
4. ↑ IEEE Computer Society. "2006 IEEE COMPUTER SOCIETY REPORT TO THE IFIP GENERAL ASSEMBLY". http://www.ifip.org/minutes/GA2006/Tab18b-US-IEEE.pdf. Retrieved 2007-04-10.
5. ↑ As outsourcing gathers steam, computer science interest wanes
6. ↑ Computer Programmers
7. ↑ Software developer growth slows in North America | InfoWorld | News | 2007-03-13 | By Robert Mullins, IDG News Service
8. ↑ Hot Skills, Cold Skills
9. ↑ Dual Roles: The Changing Face of IT
10. ↑ Sayo, Mylene. "[http://www.peo.on.ca/enforcement/June112002newsrelease.html What's in a Name? Tech Sector battles Engineers on "software engineering""]. http://www.peo.on.ca/enforcement/June112002newsrelease.html. Retrieved 2008-07-24

UML

Introduction

Unified Modeling Language (UML) is a standardized general-purpose modeling language in the field of software engineering. The standard is managed, and was created by, the Object Management Group. UML includes a set of graphic notation techniques to create visual models of software-intensive systems.[1]

Overview

The Unified Modeling Language (UML) is used to specify, visualize, modify, construct and document the artifacts of an object-oriented software-intensive system under development.[2] UML offers a standard way to visualize a system's architectural blueprints, including elements such as:

- activities
- actors
- business processes
- database schemas
- (logical) components
- programming language statements
- reusable software components.[3]

UML combines techniques from data modeling (entity relationship diagrams), business modeling (work flows), object modeling, and component modeling. It can be used with all processes, throughout the software development life cycle, and across different implementation technologies. [4] UML has synthesized the notations of the Booch method, the Object-modeling technique (OMT) and Object-oriented software engineering (OOSE) by fusing them into a single, common and widely usable modeling language.[citation needed] UML aims to be a standard modeling language which can model concurrent and distributed systems. UML is a de facto industry standard, and is evolving under the auspices of the Object Management Group (OMG).

UML models may be automatically transformed to other representations (e.g. Java) by means of QVT-like transformation languages. UML is extensible, with two mechanisms for customization: profiles and stereotypes.

Modeling

It is important to distinguish between the UML model and the set of diagrams of a system. A diagram is a partial graphic representation of a system's model. The model also contains documentation that drive the model elements and diagrams (such as written use cases).

UML diagrams represent two different views of a system model [5]:

- Static (or structural) view: emphasizes the static structure of the system using objects, attributes, operations and relationships. The structural view includes class diagrams and composite structure diagrams.
- Dynamic (or behavioral) view: emphasizes the dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects. This view includes sequence diagrams, activity diagrams and state machine diagrams.

UML models can be exchanged among UML tools by using the XMI interchange format.

Diagrams overview

UML 2.2 has 14 types of diagrams divided into two categories.[6] Seven diagram types represent structural information, and the other seven represent general types of behavior, including four that represent different aspects of interactions.

UML does not restrict UML element types to a certain diagram type. In general, every UML element may appear on almost all types of diagrams; this flexibility has been partially restricted in UML 2.0. UML profiles may define additional diagram types or extend existing diagrams with additional notations.

Structure diagrams

Structure diagrams emphasize the things that must be present in the system being modeled. Since structure diagrams represent the structure, they are used extensively in documenting the software architecture of software systems.

- Class diagram: describes the structure of a system by showing the system's classes, their attributes, and the relationships among the classes.
- Component diagram: describes how a software system is split up into components and shows the dependencies among these components.
- Composite structure diagram: describes the internal structure of a class and the collaborations that this structure makes possible.
- Deployment diagram: describes the hardware used in system implementations and the execution environments and artifacts deployed on the hardware.
- Object diagram: shows a complete or partial view of the structure of a modeled system at a specific time.
- Package diagram: describes how a system is split up into logical groupings by showing the dependencies among these groupings.
- Profile diagram: operates at the metamodel level to show stereotypes as classes with the <<stereotype>> stereotype, and profiles as packages with the <<profile>> stereotype. The extension relation (solid line with closed, filled arrowhead) indicates what metamodel element a given stereotype is extending.

Behaviour diagrams

Behavior diagrams emphasize what must happen in the system being modeled. Since behavior diagrams illustrate the behavior of a system, they are used extensively to describe the functionality of software systems.

- Activity diagram: describes the business and operational step-by-step workflows of components in a system. An activity diagram shows the overall flow of control.
- state machine diagram: describes the states and state transitions of the system.
- Use case diagram: describes the functionality provided by a system in terms of actors, their goals represented as use cases, and any dependencies among those use cases.

Interaction diagrams

Interaction diagrams, a subset of behaviour diagrams, emphasize the flow of control and data among the things in the system being modeled:

- Communication diagram: shows the interactions between objects or parts in terms of sequenced messages. They represent a combination of information taken from Class, Sequence, and Use Case Diagrams describing both the static structure and dynamic behavior of a system.
- Interaction overview diagram: provides an overview in which the nodes represent communication diagrams.
- Sequence diagram: shows how objects communicate with each other in terms of a sequence of messages. Also indicates the lifespans of objects relative to those messages.
- Timing diagrams: a specific type of interaction diagram where the focus is on timing constraints.

UML Modelling Tools

The most well-known UML modelling tool is IBM Rational Rose.[citation needed] Other tools include Rational Rhapsody, MagicDraw UML, StarUML, ArgoUML, Umbrello, BOUML, PowerDesigner, and Dia. Some of popular development environments also offer UML modelling tools, i. e. Eclipse, NetBeans, and Visual Studio. [7]

References

1. ↑ http://en.wikipedia.org/w/index.php?title=Unified_Modeling_Language&oldid=413683022 Unified Modeling Language
2. ↑ Wikipedia:FOLDOC (2001). Unified Modeling Language last updated 2002-01-03. Accessed 6 feb 2009.
3. ↑ Grady Booch, Ivar Jacobson & Jim Rumbaugh (2000) OMG Unified Modeling Language Specification, Version 1.3 First Edition: March 2000. Retrieved 12 August 2008.
4. ↑ Satish Mishra (1997). "Visual Modeling & Unified Modeling Language (UML) : Introduction to UML". Rational Software Corporation. Accessed 9 Nov 2008.

5. ↑ Jon Holt Institution of Electrical Engineers (2004). *UML for Systems Engineering: Watching the Wheels* IET, 2004, ISBN 0863413544. p.58
6. ↑ *UML Superstructure Specification Version 2.2*. OMG, February 2009.
7. ↑ http://en.wikipedia.org/wiki/List_of_Unified_Modeling_Language_tools List of Unified Modeling Language Tools

Examples

From the previous example, you may have noticed that software engineers speak a funny language: UML. After this little introduction to UML, hopefully you will see that designing software is a little like writing a screenplay for a Holywood movie.

Example: Bill at the Restaurant

Bill is at a restaurant and wants to eat dinner. His waiter is Linus, who takes the orders and brings the food. In the kitchen is Larry the cook. Steve is the cashier.

Use Case Diagram

The first thing a software engineer does is to draw a Use Case diagram. All the actors are represented by little stick figures, all the 'actions' are represented by ovals and are called use cases. The actors and use cases are connected by lines. Very often there is also one or more system boundaries. Actors which are not part of the system are drawn outside the system area.

Game Time

Draw a Use Case diagram for our restaurant example.

Activity Diagram

The Activity Diagram captures the workflow of a process. In the Use Case diagram we have no timely order, the Activity Diagram gives more detail to the Use Case. An Activity Diagram has a beginning and an end, and it also depicts decisions and repetitions. It often depicts the flow of information, hence it is also called Flowchart. Activity Diagrams are not so important.

Game Time

Draw an Activity Diagram diagram for our restaurant example.

Sequence Diagram

The next step of refinement is the Sequence Diagram. It lists the actors/objects horizontally and then depicts the messages going back and forth between the objects. The timeline is progressing downwards.

This diagram is so important, because on the one hand it identifies our objects/classes and on the other hand it also gives us the methods of each of the classes, because each message turns into a method.

Game Time

Draw a Sequence Diagram for our restaurant example.

Collaboration Diagram

The Collaboration Diagram is similar to the Sequence Diagram, but it has a different layout. Instead of worrying about the timeline, we worry about the interactions between the objects. Each object is represented by a box, and interactions between the objects are shown by an arrow. This diagram shows the responsibility of objects. If an object has too much responsibility, probably something is wrong in your

Game Time

Draw a Collaboration Diagram for our restaurant example.

Class Diagram

For us as software engineers, the Class Diagram is the most important one. First, it shows classes (objects) as boxes, with attributes and methods as compartments. Second, it shows relationships between the objects as arrows, where a subtle distinction is being made between three different kind of relationships:

- Association (’has a’): a static relationship, usually one class is attribute of another class, or one class uses another class
- Aggregation (’consists of’’): for instance an order consists of order details
- Inheritance (’is a’): describes a hierarchy between classes

There are many more, but for now we do not need to worry about them.

Game Time

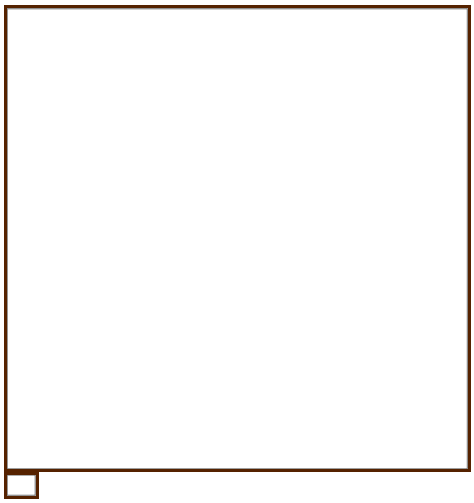
Draw a Class Diagram for our restaurant example.

Process & Methodology

Introduction

First we need to take a brief look at the big piture. The software development process is a structure imposed on the development of a software product. It is made up of a set of activities and steps with the goal to find repeatable, predictable processes that improve productivity and quality.

Software Development Activities



Model of the Systems Development Life Cycle with the Maintenance bubble highlighted.

The software development process consists of a set of activities and steps, which are

- Requirements
- Specification
- Architecture
- Design
- Implementation
- Testing
- Deployment
- Maintenance

Planning

The important task in creating a software product is extracting the requirements or requirements analysis. Customers typically have an abstract idea of what they want as an end result, but not what software should do. Incomplete, ambiguous, or even contradictory requirements are recognized by skilled and experienced software engineers at this point. Frequently demonstrating live code may help reduce the risk that the requirements are incorrect.

Once the general requirements are gathered from the client, an analysis of the scope of the development should be determined and clearly stated. This is often called a scope document.

Certain functionality may be out of scope of the project as a function of cost or as a result of unclear requirements at the start of development. If the development is done externally, this document can be considered a legal document so that if there are ever disputes, any ambiguity of what was promised to the client can be clarified.

Implementation, Testing and Documenting

Implementation is the part of the process where software engineers actually program the code for the project.

Software testing is an integral and important part of the software development process. This part of the process ensures that defects are recognized as early as possible.

Documenting the internal design of software for the purpose of future maintenance and enhancement is done throughout development. This may also include the writing of an API, be it external or internal. It is very important to document everything in the project.

Deployment and Maintenance

Deployment starts after the code is appropriately tested, is approved for release and sold or otherwise distributed into a production environment.

Software Training and Support is important and a lot of developers fail to realize that. It would not matter how much time and planning a development team puts into creating software if nobody in an organization ends up using it. People are often resistant to change and avoid venturing into an unfamiliar area, so as a part of the deployment phase, it is very important to have training classes for new clients of your software.

Maintaining and enhancing software to cope with newly discovered problems or new requirements can take far more time than the initial development of the software. It may be necessary to add code that does not fit the original design to correct an unforeseen problem or it may be that a customer is requesting more functionality and code can be added to accommodate their requests. If the labor cost of the maintenance phase exceeds 25% of the prior-phases' labor cost, then it is likely that the overall quality of at least one prior phase is poor. In that case, management should consider the option of rebuilding the system (or portions) before maintenance cost is out of control.

References

External links

- Don't Write Another Process
- No Silver Bullet: Essence and Accidents of Software Engineering", 1986
- Gerhard Fischer, "The Software Technology of the 21st Century: From Software Reuse to Collaborative Software Design", 2001
- Lydia Ash: *The Web Testing Companion: The Insider's Guide to Efficient and Effective Tests*, Wiley, May 2, 2003. ISBN 0-471-43021-8
- SaaSSDLC.com — Software as a Service Systems Development Life Cycle Project
- Software development life cycle (SDLC) [visual image], *software development life cycle*
- Selecting an SDLC", 2009
- Heraprocess.org — Hera is a light process solution for managing web projects

Methodology

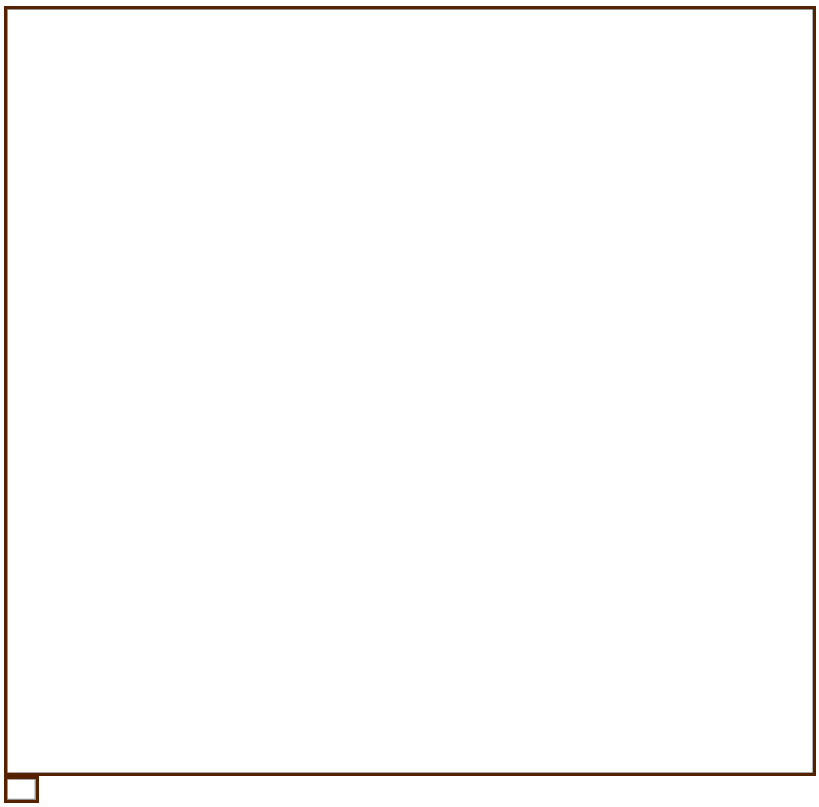
A **software development methodology** or **system development methodology** in software engineering is a framework that is used to structure, plan, and control the process of developing an information system.

History

The software development methodology framework didn't emerge until the 1960s. According to Elliott (2004) the systems development life cycle (SDLC) can be considered to be the oldest formalized methodology framework for building information systems. The main idea of the SDLC has been "to pursue the development of information systems in a very deliberate, structured and methodical way, requiring each stage of the life cycle from inception of the idea to delivery of the final system, to be carried out in rigidly and sequentially".[1] within the context of the framework being applied. The main target of this methodology framework in the 1960s was "to develop large scale functional business systems in an age of large scale business conglomerates. Information systems activities revolved around heavy data processing and number crunching routines".[1]

As a noun

As a noun, a software development methodology is a framework that is used to structure, plan, and control the process of developing an information system - this includes the pre-definition of specific deliverables and artifacts that are created and completed by a project team to develop or maintain an application.[2]



The three basic approaches applied to software development methodology frameworks.

A wide variety of such frameworks have evolved over the years, each with its own recognized strengths and weaknesses. One software development methodology framework is not necessarily suitable for use by all projects. Each of the available methodology frameworks are best suited to specific kinds of projects, based on various technical, organizational, project and team considerations.[2]

These software development frameworks are often bound to some kind of organization, which further develops, supports the use, and promotes the methodology framework. The methodology framework is often defined in some kind of formal documentation. Specific software development methodology frameworks (noun) include

- Rational Unified Process (RUP, IBM) since 1998.
- Agile Unified Process (AUP) since 2005 by Scott Ambler

As a verb

As a verb, the software development methodology is an approach used by organizations and project teams to apply the software development methodology framework (noun). Specific software development methodologies (verb) include:

1970s

- Structured programming since 1969
- Cap Gemini SDM, originally from PANDATA, the first English translation was published in 1974. SDM stands for System Development Methodology

1980s

- Structured Systems Analysis and Design Methodology (SSADM) from 1980 onwards
- Information Requirement Analysis/Soft systems methodology

1990s

- Object-oriented programming (OOP) has been developed since the early 1960s, and developed as a dominant programming approach during the mid-1990s
- Rapid application development (RAD) since 1991
- Scrum, since the late 1990s
- Team software process developed by Watts Humphrey at the SEI
- Extreme Programming since 1999

Verb approaches

Every software development methodology framework acts as a basis for applying specific approaches to develop and maintain software. Several software development approaches have been used since the origin of information technology. These are:[2]

- Waterfall: a linear framework
- Prototyping: an iterative framework
- Incremental: a combined linear-iterative framework
- Spiral: a combined linear-iterative framework
- Rapid application development (RAD): an iterative framework
- Extreme Programming

Waterfall development

The Waterfall model is a sequential development approach, in which development is seen as flowing steadily downwards (like a waterfall) through the phases of requirements analysis, design, implementation, testing (validation), integration, and maintenance. The first formal description of the method is often cited as an article published by Winston W. Royce[3] in 1970 although Royce did not use the term "waterfall" in this article.

The basic principles are:[2]

- Project is divided into sequential phases, with some overlap and splashback acceptable between phases.
- Emphasis is on planning, time schedules, target dates, budgets and implementation of an entire system at one time.
- Tight control is maintained over the life of the project via extensive written documentation, formal reviews, and approval/signoff by the user and information technology management occurring at the end of most phases before beginning the next phase.

Prototyping

Software prototyping, is the development approach of activities during software development, the creation of prototypes, i.e., incomplete versions of the software program being developed.

The basic principles are:[2]

- Not a standalone, complete development methodology, but rather an approach to handling selected parts of a larger, more traditional development methodology (i.e. incremental, spiral, or rapid application development (RAD)).
- Attempts to reduce inherent project risk by breaking a project into smaller segments and providing more ease-of-change during the development process.
- User is involved throughout the development process, which increases the likelihood of user acceptance of the final implementation.
- Small-scale mock-ups of the system are developed following an iterative modification process until the prototype evolves to meet the users' requirements.
- While most prototypes are developed with the expectation that they will be discarded, it is possible in some cases to evolve from prototype to working system.
- A basic understanding of the fundamental business problem is necessary to avoid solving the wrong problem.

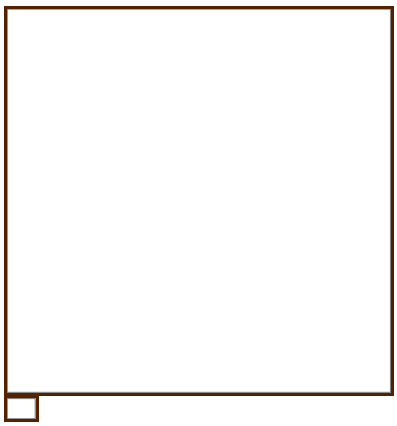
Incremental development

Various methods are acceptable for combining linear and iterative systems development methodologies, with the primary objective of each being to reduce inherent project risk by breaking a project into smaller segments and providing more ease-of-change during the development process.

The basic principles are:[2]

- A series of mini-Waterfalls are performed, where all phases of the Waterfall are completed for a small part of a system, before proceeding to the next increment, or
- Overall requirements are defined before proceeding to evolutionary, mini-Waterfall development of individual increments of a system, or
- The initial software concept, requirements analysis, and design of architecture and system core are defined via Waterfall, followed by iterative Prototyping, which culminates in installing the final prototype, a working system.

Spiral development



The spiral model.

The spiral model is a software development process combining elements of both design and prototyping-in-stages, in an effort to combine advantages of top-down and bottom-up concepts.

The basic principles are:[2]

- Focus is on risk assessment and on minimizing project risk by breaking a project into smaller segments and providing more ease-of-change during the development process, as well as providing the opportunity to evaluate risks and weigh consideration of project continuation throughout the life cycle.
- "Each cycle involves a progression through the same sequence of steps, for each part of the product and for each of its levels of elaboration, from an overall concept-of-operation document down to the coding of each individual program."[4]
- Each trip around the spiral traverses four basic quadrants: (1) determine objectives, alternatives, and constraints of the iteration; (2) evaluate alternatives; Identify and resolve risks; (3) develop and verify deliverables from the iteration; and (4) plan the next iteration.[4][5]
- Begin each cycle with an identification of stakeholders and their win conditions, and end each cycle with review and commitment.[6]

Rapid application development

Rapid application development (RAD) is a software development methodology, which involves iterative development and the construction of prototypes. Rapid application development is a term originally used to describe a software development process introduced by James Martin in 1991.

The basic principles are:[2]

- Key objective is for fast development and delivery of a high quality system at a relatively low investment cost.
- Attempts to reduce inherent project risk by breaking a project into smaller segments and providing more ease-of-change during the development process.

- Aims to produce high quality systems quickly, primarily via iterative Prototyping (at any stage of development), active user involvement, and computerized development tools. These tools may include Graphical User Interface (GUI) builders, Computer Aided Software Engineering (CASE) tools, Database Management Systems (DBMS), fourth-generation programming languages, code generators, and object-oriented techniques.
- Key emphasis is on fulfilling the business need, while technological or engineering excellence is of lesser importance.
- Project control involves prioritizing development and defining delivery deadlines or “timeboxes”. If the project starts to slip, emphasis is on reducing requirements to fit the timebox, not in increasing the deadline.
- Generally includes joint application design (JAD), where users are intensely involved in system design, via consensus building in either structured workshops, or electronically facilitated interaction.
- Active user involvement is imperative.
- Iteratively produces production software, as opposed to a throwaway prototype.
- Produces documentation necessary to facilitate future development and maintenance.
- Standard systems analysis and design methods can be fitted into this framework.

Other practices

Other methodology practices include:

- Object-oriented development methodologies, such as Grady Booch's object-oriented design (OOD), also known as object-oriented analysis and design (OOAD). The Booch model includes six diagrams: class, object, state transition, interaction, module, and process.[7]
- Top-down programming: evolved in the 1970s by IBM researcher Harlan Mills (and Niklaus Wirth) in developed structured programming.
- Unified Process (UP) is an iterative software development methodology framework, based on Unified Modeling Language (UML). UP organizes the development of software into four phases, each consisting of one or more executable iterations of the software at that stage of development: inception, elaboration, construction, and guidelines. Many tools and products exist to facilitate UP implementation. One of the more popular versions of UP is the Rational Unified Process (RUP).
- Agile software development refers to a group of software development methodologies based on iterative development, where requirements and solutions evolve via collaboration between self-organizing cross-functional teams. The term was coined in the year 2001 when the Agile Manifesto was formulated.
- Integrated software development refers to a deliverable based software development framework using the three primary IT (project management, software development, software testing) life cycles that can be leveraged using multiple (iterative, waterfall, spiral, agile) software development approaches, where requirements and solutions evolve via collaboration between self-organizing cross-functional teams.

Subtopics

View model

The TEAF Matrix of Views and Perspectives.

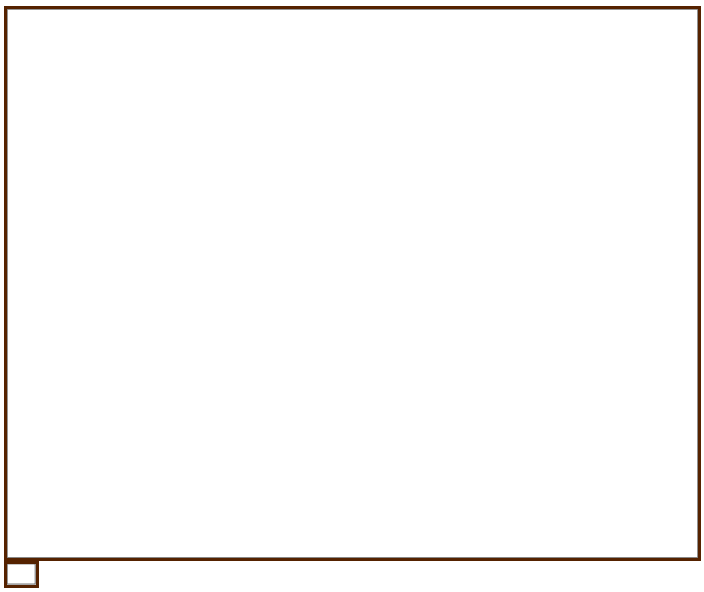
A view model is framework which provides the viewpoints on the system and its environment, to be used in the software development process. It is a graphical representation of the underlying semantics of a view.

The purpose of viewpoints and views is to enable human engineers to comprehend very complex systems, and to organize the elements of the problem and the solution around domains of expertise. In the engineering of physically intensive systems, viewpoints often correspond to capabilities and responsibilities within the engineering organization.[8]

Most complex system specifications are so extensive that no one individual can fully comprehend all aspects of the specifications. Furthermore, we all have different interests in a given system and different reasons for examining the system's specifications. A business executive will ask different questions of a system make-up than would a system implementer. The concept of viewpoints framework, therefore, is to provide separate viewpoints into the specification of a given complex system. These viewpoints each satisfy an audience with interest in some set of aspects of the system. Associated with each viewpoint is a viewpoint language that optimizes the vocabulary and presentation for the audience of that viewpoint.

Business process and data modelling

Graphical representation of the current state of information provides a very effective means for presenting information to both users and system developers.



example of the interaction between business process and data models.[9]

- A business model illustrates the functions associated with the business process being modeled and the organizations that perform these functions. By depicting activities and information flows, a foundation is created to visualize, define, understand, and validate the nature of a process.
- A data model provides the details of information to be stored, and is of primary use when the final product is the generation of computer software code for an application or the preparation of a functional specification to aid a computer software make-or-buy decision. See the figure on the right for an example of the interaction between business process and data models.[9]

Usually, a model is created after conducting an interview, referred to as business analysis. The interview consists of a facilitator asking a series of questions designed to extract required information that describes a process. The interviewer is called a facilitator to emphasize that it is the participants who provide the information. The facilitator should have some knowledge of the process of interest, but this is not as important as having a structured methodology by which the questions are asked of the process expert. The methodology is important because usually a team of facilitators is collecting information across the facility and the results of the information from all the interviewers must fit together once completed.[9]

The models are developed as defining either the current state of the process, in which case the final product is called the "as-is" snapshot model, or a collection of ideas of what the process should contain, resulting in a "what-can-be" model. Generation of process and data models can be used to determine if the existing processes and information systems are sound and only need minor modifications or enhancements, or if re-engineering is required as a corrective action. The creation of business models is more than a way to view or automate your information process. Analysis can be used to fundamentally reshape the way your business or organization conducts its operations.[9]

Computer-aided software engineering

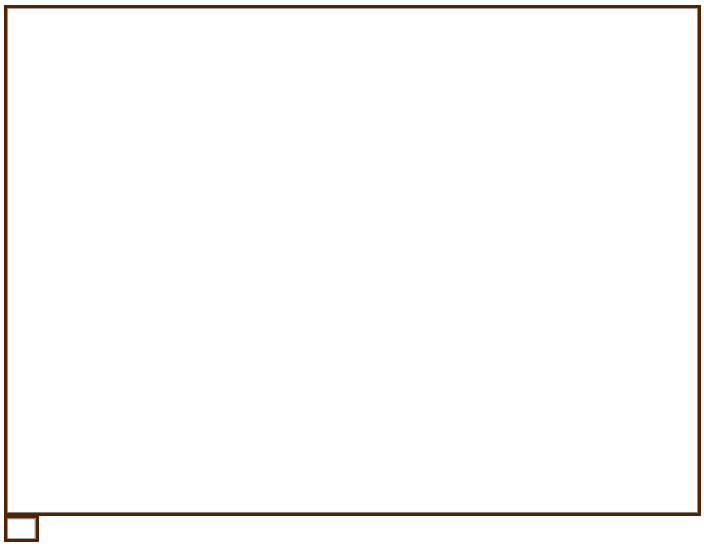
Computer-aided software engineering (CASE), in the field software engineering is the scientific application of a set of tools and methods to a software which results in high-quality, defect-free, and maintainable software products.[10] It also refers to methods for the development of information systems together with automated tools that can be used in the software development process.[11] The term "computer-aided software engineering" (CASE) can refer to the software used for the automated development of systems software, i.e., computer code. The CASE functions include analysis, design, and programming. CASE tools automate methods for designing, documenting, and producing structured computer code in the desired programming language.[12]

Two key ideas of Computer-aided Software System Engineering (CASE) are:[13]

- Foster computer assistance in software development and or software maintenance processes, and
- An engineering approach to software development and or maintenance.

Typical CASE tools exist for configuration management, data modeling, model transformation, refactoring, source code generation, and Unified Modeling Language.

Integrated development environment



Anjuta, a C and C++ IDE for the GNOME environment

An integrated development environment (IDE) also known as *integrated design environment* or *integrated debugging environment* is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of a:

- source code editor,
- compiler and/or interpreter,
- build automation tools, and
- debugger (usually).

IDEs are designed to maximize programmer productivity by providing tight-knit components with similar user interfaces. Typically an IDE is dedicated to a specific programming language, so as to provide a feature set which most closely matches the programming paradigms of the language.

Modeling language

A modeling language is any artificial language that can be used to express information or knowledge or systems in a structure that is defined by a consistent set of rules. The rules are used for interpretation of the meaning of components in the structure. A modeling language can be graphical or textual.[14] Graphical modeling languages use a diagram techniques with named symbols that represent concepts and lines that connect the symbols and that represent relationships and various other graphical annotation to represent constraints. Textual modeling languages typically use standardised keywords accompanied by parameters to make computer-interpretable expressions.

Example of graphical modelling languages in the field of software engineering are:

- Business Process Modeling Notation (BPMN, and the XML form BPML) is an example of a process modeling language.
- EXPRESS and EXPRESS-G (ISO 10303-11) is an international standard general-purpose data modeling language.
- Extended Enterprise Modeling Language (EEML) is commonly used for business process modeling across layers.
- Flowchart is a schematic representation of an algorithm or a stepwise process,
- Fundamental Modeling Concepts (FMC) modeling language for software-intensive systems.
- IDEF is a family of modeling languages, the most notable of which include IDEF0 for functional modeling, IDEF1X for information modeling, and IDEF5 for modeling ontologies.
- LePUS3 is an object-oriented visual Design Description Language and a formal specification language that is suitable primarily for modelling large object-oriented (Java, C++, C#) programs and design patterns.
- Specification and Description Language(SDL) is a specification language targeted at the unambiguous specification and description of the behaviour of reactive and distributed systems.
- Unified Modeling Language (UML) is a general-purpose modeling language that is an industry standard for specifying software-intensive systems. UML 2.0, the current version, supports thirteen different diagram techniques, and has widespread tool support.

Not all modeling languages are executable, and for those that are, using them doesn't necessarily mean that programmers are no longer needed. On the contrary, executable modeling languages are intended to amplify the productivity of skilled programmers, so that they can address more difficult problems, such as parallel computing and distributed systems.

Programming paradigm

A programming paradigm is a fundamental style of computer programming, in contrast to a software engineering methodology, which is a style of solving specific software engineering problems. Paradigms differ in the concepts and abstractions used to represent the elements of a program (such as objects, functions, variables, constraints...) and the steps that compose a computation (assignation, evaluation, continuations, data flows...).

A programming language can support multiple paradigms. For example programs written in C++ or Object Pascal can be purely procedural, or purely object-oriented, or contain elements of both paradigms. Software designers and programmers decide how to use those paradigm elements. In object-oriented programming, programmers can think of a program as a collection of interacting objects, while in functional programming a program can be thought of as a sequence of stateless function evaluations. When programming computers or systems with many processors, process-oriented programming allows programmers to think about applications as sets of concurrent processes acting upon logically shared data structures.

Just as different groups in software engineering advocate different *methodologies*, different programming languages advocate different *programming paradigms*. Some languages are designed to support one paradigm (Smalltalk supports object-oriented programming, Haskell supports functional programming), while other programming languages support multiple paradigms (such as Object Pascal, C++, C#, Visual Basic, Common Lisp, Scheme, Python, Ruby, and Oz).

Many programming paradigms are as well known for what methods they *forbid* as for what they enable. For instance, pure functional programming forbids using side-effects; structured programming forbids using goto statements. Partly for this reason, new paradigms are often regarded as doctrinaire or overly rigid by those accustomed to earlier styles.[*citation needed*] Avoiding certain methods can make it easier to prove theorems about a program's correctness, or simply to understand its behavior.

Software framework

A software framework is a re-usable design for a software system or subsystem. A software framework may include support programs, code libraries, a scripting language, or other software to help develop and *glue together* the different components of a software project. Various parts of the framework may be exposed via an API.

Software development process

A software development process is a framework imposed on the development of a software product. Synonyms include software life cycle and *software process*. There are several models for such processes, each describing approaches to a variety of tasks or activities that take place during the process.

A largely growing body of software development organizations implement process methodologies. Many of them are in the defense industry, which in the U.S. requires a rating based on 'process models' to obtain contracts. The international standard describing the method to select, implement and monitor the life cycle for software is ISO 12207.

A decades-long goal has been to find repeatable, predictable processes that improve productivity and quality. Some try to systematize or formalize the seemingly unruly task of writing software. Others apply project management methods to writing software. Without project management, software projects can easily be delivered late or over budget. With large numbers of software projects not meeting their expectations in terms of functionality, cost, or delivery schedule, effective project management appears to be lacking.

See also

Lists

- List of software engineering topics
- List of software development philosophies

Related topics

- Domain-specific modeling
- Lightweight methodology
- Object modeling language
- Structured programming
- Integrated IT Methodology

References

1. ↑ ***a*** ***b*** Geoffrey Elliott (2004) *Global Business Information Technology: an integrated systems approach*. Pearson Education. p.87.
2. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** Centers for Medicare & Medicaid Services (CMS) Office of Information Service (2008). *Selecting a development approach*. Webarticle. United States Department of Health and Human Services (HHS). Revalidated: March 27, 2008. Retrieved 27 Oct 2008.
3. ↑ Wasserfallmodell > Entstehungskontext, Markus Rerych, Institut für Gestaltungs- und Wirkungsforschung, TU-Wien. Accessed on line November 28, 2007.
4. ↑ ***a*** ***b*** Barry Boehm (1996., "A Spiral Model of Software Development and Enhancement". In: *ACM SIGSOFT Software Engineering Notes* (ACM) 11(4):14-24, August 1986
5. ↑ Richard H. Thayer, Barry W. Boehm (1986). *Tutorial: software engineering project management*. Computer Society Press of the IEEE. p.130
6. ↑ Barry W. Boehm (2000). *Software cost estimation with Cocomo II: Volume 1*.
7. ↑ Georges Gauthier Merx & Ronald J. Norman (2006). *Unified Software Engineering with Java*. p.201.
8. ↑ Edward J. Barkmeyer ea (2003). *Concepts for Automating Systems Integration* NIST 2003.
9. ↑ ***a*** ***b*** ***c*** ***d*** Paul R. Smith & Richard Sarfaty (1993). Creating a strategic plan for configuration management using Computer Aided Software Engineering (CASE) tools. Paper For 1993 National DOE/Contractors and Facilities CAD/CAE User's Group.
10. ↑ Kuhn, D.L (1989). "Selecting and effectively using a computer aided software engineering tool". Annual Westinghouse computer symposium; 6-7 Nov 1989; Pittsburgh, PA (USA); DOE Project.

11.↑ P. Loucopoulos and V. Karakostas (1995). *System Requirements Engineering*. McGraw-Hill.
12.↑ CASE definition In: Telecom Glossary 2000. *Retrieved 26 Oct 2008.*
13.↑ K. Robinson (1992). *Putting the Software Engineering into CASE*. New York : John Wiley and Sons Inc.
14.↑ Xiao He (2007). "A metamodel for the notation of graphical modeling languages". In: *Computer Software and Applications Conference, 2007. COMPSAC 2007 - Vol. 1. 31st Annual International*, Volume 1, Issue , 24–27 July 2007, pp 219-224.

External links

Wikimedia Commons has media related to: ***Software development methodology***

- Selecting a development approach at cms.hhs.gov.
- Software Methodologies Book Reviews An extensive set of book reviews related to software methodologies and processes

V-Model

The V-model of the Systems Engineering Process.[1]

The **V-model** represents a software development process (also applicable to hardware development) which may be considered an extension of the waterfall model. Instead of moving down in a linear way, the process steps are bent upwards after the coding phase, to form the typical V shape. The V-Model demonstrates the relationships between each phase of the development life cycle and its associated phase of testing. The horizontal and vertical axes represents time or project completeness (left-to-right) and level of abstraction (coarsest-grain abstraction uppermost), respectively.

Verification Phases

Requirements analysis

In the Requirements analysis phase, the requirements of the proposed system are collected by analyzing the needs of the user(s). This phase is concerned about establishing what the ideal system has to perform. However it does not determine how the software will be designed or built. Usually, the users are interviewed and a document called the user requirements document is generated.

The user requirements document will typically describe the system's functional, interface, performance, data, security, etc requirements as expected by the user. It is used by business analysts to communicate their understanding of the system to the users. The users carefully review this document as this document would serve as the guideline for the system designers in the system design phase. The user acceptance tests are designed in this phase. See also Functional requirements. this is parallel processing

There are different methods for gathering requirements of both soft and hard methodologies including; interviews, questionnaires, document analysis, observation, throw-away prototypes, use cases and status and dynamic views with users.

System Design

Systems design is the phase where system engineers analyze and understand the business of the proposed system by studying the user requirements document. They figure out possibilities and techniques by which the user requirements can be implemented. If any of the requirements are not feasible, the user is informed of the issue. A resolution is found and the user requirement document is edited accordingly.

The software specification document which serves as a blueprint for the development phase is generated. This document contains the general system organization, menu structures, data structures etc. It may also hold example business scenarios, sample windows, reports for the better understanding. Other technical documentation like entity diagrams, data dictionary will also be produced in this phase. The documents for system testing are prepared in this phase.

Architecture Design

The phase of the design of computer architecture and software architecture can also be referred to as high-level design. The baseline in selecting the architecture is that it should realize all which typically consists of the list of modules, brief functionality of each module, their interface relationships, dependencies, database tables, architecture diagrams, technology details etc. The integration testing design is carried out in the particular phase.

Module Design

The module design phase can also be referred to as low-level design. The designed system is broken up into smaller units or modules and each of them is explained so that the programmer can start coding directly. The low level design document or program specifications will contain a detailed functional logic of the module, in pseudocode:

- database tables, with all elements, including their type and size
- all interface details with complete API references
- all dependency issues
- error message listings
- complete input and outputs for a module.

The unit test design is developed in this stage.

Validation Phases

Unit Testing

In computer programming, unit testing is a method by which individual units of source code are tested to determine if they are fit for use. A unit is the smallest testable part of an application. In procedural programming a unit may be an individual function or procedure. Unit tests are created by programmers or occasionally by white box testers. The purpose is to verify the internal logic code by testing every possible branch within the function, also known as test coverage. Static analysis tools are used to facilitate in this process, where variations of input data are passed to the function to test every possible case of execution.

Integration Testing

In integration testing the separate modules will be tested together to expose faults in the interfaces and in the interaction between integrated components. Testing is usually black box as the code is not directly checked for errors.

System Testing

System testing will compare the system specifications against the actual system.After the integration test is completed, the next test level is the system test. System testing checks if the integrated product meets the specified requirements. Why is this still necessary after the component and integration tests? The reasons for this are as follows:

Reasons for system test

1. In the lower test levels, the testing was done against technical specifications, i.e., from the technical perspective of the software producer. The system test, though, looks at the system from the perspective of the customer and the future user. The testers validate whether the requirements are completely and appropriately met.
    - Example: The customer (who has ordered and paid for the system) and the user (who uses the system) can be different groups of people or organizations with their own specific interests and requirements of the system.
2. Many functions and system characteristics result from the interaction of all system components, consequently, they are only visible on the level of the entire system and can only be observed and tested there.

User Acceptance Testing

Acceptance testing is the phase of testing used to determine whether a system satisfies the requirements specified in the requirements analysis phase. The acceptance test design is derived from the requirements document. The acceptance test phase is the phase used by the customer to determine whether to accept the system or not.

Acceptance testing helps

- to determine whether a system satisfies its acceptance criteria or not.
- to enable the customer to determine whether to accept the system or not.
- to test the software in the "real world" by the intended audience.

Purpose of acceptance testing:

- to verify the system or changes according to the original needs.

Procedures

1. Define the acceptance criteria:
    - Functionality requirements.
    - Performance requirements.
    - Interface quality requirements.
    - Overall software quality requirements.
2. Develop an acceptance plan:
    - Project description.
    - User responsibilities.
    - Acceptance description.
    - Execute the acceptance test plan.

References

1. ↑ *Clarus Concept of Operations.* Publication No. FHWA-JPO-05-072, Federal Highway Administration (FHWA), 2005

Further reading

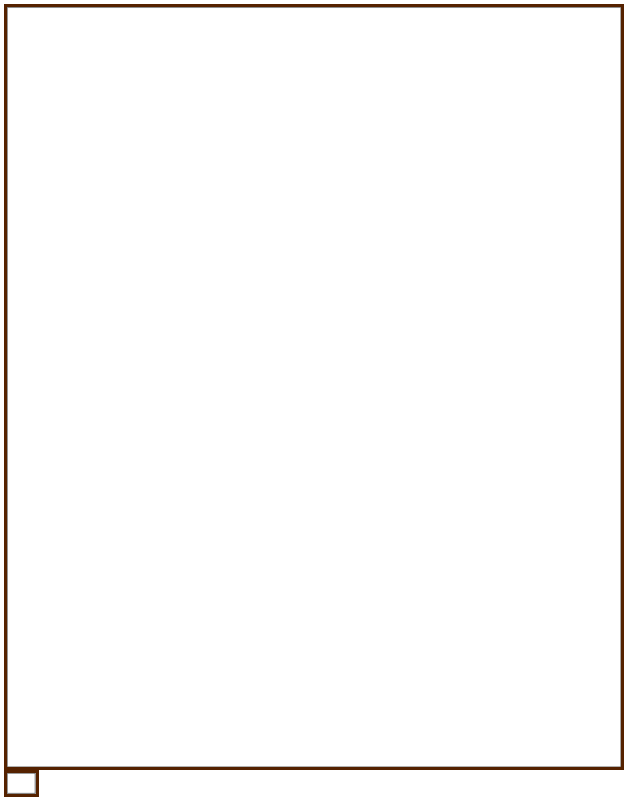
- Roger S. Pressman:*Software Engineering: A Practitioner's Approach*, The McGraw-Hill Companies, ISBN 007301933X
- Mark Hoffman & Ted Beaumont: *Application Development: Managing the Project Life Cycle*, Mc Press, ISBN 1883884454
- Boris Beizer: *Software Testing Techniques.* Second Edition, International Thomson Computer Press, 1990, ISBN 1-85032-880-3

External links

Wikimedia Commons has media related to: ***V-models***

- A paper by Christian Bucanac
- The SDLC and SixSigma
- SDLC for small and medium DB applications

Agile Model

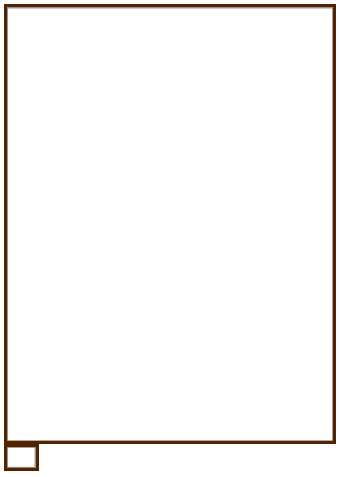


Agile software development poster

Agile software development is a group of software development methodologies based on iterative and incremental development, where requirements and solutions evolve through collaboration between self-organizing, cross-functional teams. The *Agile Manifesto*[1] introduced the term in 2001.

History

Predecessors



Jeff Sutherland, one of the developers of the Scrum agile software development process

Incremental software development methods have been traced back to 1957.[2] In 1974, a paper by E. A. Edmonds introduced an adaptive software development process.[3]

So-called "lightweight" software development methods evolved in the mid-1990s as a reaction against "heavyweight" methods, which were characterized by their critics as a heavily regulated, regimented, micromanaged, waterfall model of development. Proponents of lightweight methods (and now "agile" methods) contend that they are a return to development practices from early in the history of software development.[2]

Early implementations of lightweight methods include Scrum (1995), Crystal Clear, Extreme Programming (1996), Adaptive Software Development, Feature Driven Development, and Dynamic Systems Development Method (DSDM) (1995). These are now typically referred to as agile methodologies, after the Agile Manifesto published in 2001.[4]

Agile Manifesto

In February 2001, 17 software developers[5] met at a ski resort in Snowbird, Utah, to discuss lightweight development methods. They published the "Manifesto for Agile Software Development"[1] to define the approach now known as agile software development. Some of the manifesto's authors formed the Agile Alliance, a nonprofit organization that promotes software development according to the manifesto's principles.

Agile Manifesto reads, in its entirety, as follows:[1]

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> **Individuals and interactions** over processes and tools
>
> **Working software** over comprehensive documentation
>
> **Customer collaboration** over contract negotiation
>
> **Responding to change** over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.

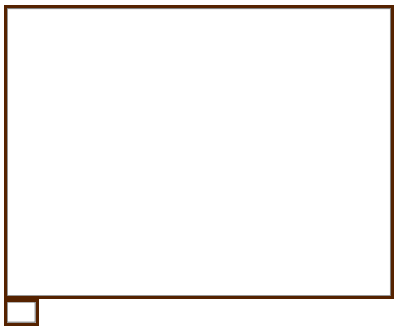
Twelve principles underlie the Agile Manifesto, including:[6]

- Customer satisfaction by rapid delivery of useful software
- Welcome changing requirements, even late in development
- Working software is delivered frequently (weeks rather than months)
- Working software is the principal measure of progress
- Sustainable development, able to maintain a constant pace
- Close, daily co-operation between business people and developers

- Face-to-face conversation is the best form of communication (co-location)
- Projects are built around motivated individuals, who should be trusted
- Continuous attention to technical excellence and good design
- Simplicity
- Self-organizing teams
- Regular adaptation to changing circumstances

In 2005, a group headed by Alistair Cockburn and Jim Highsmith wrote an addendum of project management principles, the Declaration of Interdependence,[7] to guide software project management according to agile development methods.

Characteristics



Pair programming, an XP development technique used by agile

There are many specific agile development methods. Most promote development, teamwork, collaboration, and process adaptability throughout the life-cycle of the project.

Agile methods break tasks into small increments with minimal planning, and do not directly involve long-term planning. Iterations are short time frames (timeboxes) that typically last from one to four weeks. Each iteration involves a team working through a full software development cycle including planning, requirements analysis, design, coding, unit testing, and acceptance testing when a working product is demonstrated to stakeholders. This minimizes overall risk and allows the project to adapt to changes quickly. Stakeholders produce documentation as required. An iteration may not add enough functionality to warrant a market release, but the goal is to have an available release (with minimal bugs) at the end of each iteration.[8] Multiple iterations may be required to release a product or new features.

Team composition in an agile project is usually cross-functional and self-organizing without consideration for any existing corporate hierarchy or the corporate roles of team members. Team members normally take responsibility for tasks that deliver the functionality an iteration requires. They decide individually how to meet an iteration's requirements.

Agile methods emphasize face-to-face communication over written documents when the team is all in the same location. Most agile teams work in a single open office (called a bullpen), which facilitates such communication. Team size is typically small (5-9 people) to simplify team communication and team collaboration. Larger development efforts may be delivered by multiple

teams working toward a common goal or on different parts of an effort. This may require a co-ordination of priorities across teams. When a team works in different locations, they maintain daily contact through videoconferencing, voice, e-mail, etc.

No matter what development disciplines are required, each agile team will contain a customer representative. This person is appointed by stakeholders to act on their behalf and makes a personal commitment to being available for developers to answer mid-iteration problem-domain questions. At the end of each iteration, stakeholders and the customer representative review progress and re-evaluate priorities with a view to optimizing the return on investment (ROI) and ensuring alignment with customer needs and company goals.

Most agile implementations use a routine and formal daily face-to-face communication among team members. This specifically includes the customer representative and any interested stakeholders as observers. In a brief session, team members report to each other what they did the previous day, what they intend to do today, and what their roadblocks are. This face-to-face communication exposes problems as they arise.

Agile development emphasizes working software as the primary measure of progress. This, combined with the preference for face-to-face communication, produces less written documentation than other methods. The agile method encourages stakeholders to prioritize wants with other iteration outcomes based exclusively on business value perceived at the beginning of the iteration.

Specific tools and techniques such as continuous integration, automated or xUnit test, pair programming, test driven development, design patterns, domain-driven design, code refactoring and other techniques are often used to improve quality and enhance project agility.

Comparison with other methods

Agile methods are sometimes characterized as being at the opposite end of the spectrum from "plan-driven" or "disciplined" methods; agile teams may, however, employ highly disciplined formal methods.[9] A more accurate distinction is that methods exist on a continuum from "adaptive" to "predictive".[10] Agile methods lie on the "adaptive" side of this continuum. Adaptive methods focus on adapting quickly to changing realities. When the needs of a project change, an adaptive team changes as well. An adaptive team will have difficulty describing exactly what will happen in the future. The further away a date is, the more vague an adaptive method will be about what will happen on that date. An adaptive team cannot report exactly what tasks are being done next week, but only which features are planned for next month. When asked about a release six months from now, an adaptive team may only be able to report the mission statement for the release, or a statement of expected value vs. cost.

Predictive methods, in contrast, focus on planning the future in detail. A predictive team can report exactly what features and tasks are planned for the entire length of the development process. Predictive teams have difficulty changing direction. The plan is typically optimized for the original destination and changing direction can require completed work to be started over. Predictive teams will often institute a change control board to ensure that only the most valuable changes are considered.

Formal methods, in contrast to adaptive and predictive methods, focus on computer science theory with a wide array of types of provers. A formal method attempts to prove the absence of errors with some level of determinism. Some formal methods are based on model checking and provide counter examples for code that cannot be proven. Generally, mathematical models (often supported through special languages see SPIN model checker) map to assertions about requirements. Formal methods are dependent on a tool driven approach, and may be combined with other development approaches. Some provers do not easily scale. Like agile methods, manifestos relevant to high integrity software have been proposed in Crosstalk.

Agile methods have much in common with the "Rapid Application Development" techniques from the 1980/90s as espoused by James Martin and others.

Agile methods

Well-known agile software development methods include:

- Agile Modeling
- Agile Unified Process (AUP)
- Dynamic Systems Development Method (DSDM)
- Essential Unified Process (EssUP)
- Extreme Programming (XP)
- Feature Driven Development (FDD)
- Open Unified Process (OpenUP)
- Scrum
- Velocity tracking

Method tailoring

In the literature, different terms refer to the notion of method adaptation, including ‘method tailoring’, ‘method fragment adaptation’ and ‘situational method engineering’. Method tailoring is defined as:

> A process or capability in which human agents through responsive changes in, and dynamic interplays between contexts, intentions, and method fragments determine a system development approach for a specific project situation.[11]

Potentially, almost all agile methods are suitable for method tailoring. Even the DSDM method is being used for this purpose and has been successfully tailored in a CMM context.[12] Situation-appropriateness can be considered as a distinguishing characteristic between agile methods and traditional software development methods, with the latter being relatively much more rigid and prescriptive. The practical implication is that agile methods allow project teams to adapt working practices according to the needs of individual projects. Practices are concrete activities and products that are part of a method framework. At a more extreme level, the philosophy behind the method, consisting of a number of principles, could be adapted (Aydin, 2004).[11]

Extreme Programming (XP) makes the need for method adaptation explicit. One of the fundamental ideas of XP is that no one process fits every project, but rather that practices should be tailored to the needs of individual projects. Partial adoption of XP practices, as suggested by Beck, has been reported on several occasions.[13] A tailoring practice is proposed by Mehdi Mirakhorli which provides sufficient roadmap and guideline for adapting all the practices. RDP Practice is designed for customizing XP. This practice, first proposed as a long research paper in the APSO workshop at the ICSE 2008 conference, is currently the only proposed and applicable method for customizing XP. Although it is specifically a solution for XP, this practice has the capability of extending to other methodologies. At first glance, this practice seems to be in the category of static method adaptation but experiences with RDP Practice says that it can be treated like dynamic method adaptation. The distinction between static method adaptation and dynamic method adaptation is subtle.[14] The key assumption behind static method adaptation is that the project context is given at the start of a project and remains fixed during project execution. The result is a static definition of the project context. Given such a definition, route maps can be used in order to determine which structured method fragments should be used for that particular project, based on predefined sets of criteria. Dynamic method adaptation, in contrast, assumes that projects are situated in an emergent context. An emergent context implies that a project has to deal with emergent factors that affect relevant conditions but are not predictable. This also means that a project context is not fixed, but changing during project execution. In such a case prescriptive route maps are not appropriate. The practical implication of dynamic method adaptation is that project managers often have to modify structured fragments or even innovate new fragments, during the execution of a project (Aydin et al., 2005).[14]

Measuring agility

While agility can be seen as a means to an end, a number of approaches have been proposed to quantify agility. Agility Index Measurements (AIM)[15] score projects against a number of agility factors to achieve a total. The similarly-named Agility Measurement Index,[16] scores developments against five dimensions of a software project (duration, risk, novelty, effort, and interaction). Other techniques are based on measurable goals.[17] Another study using fuzzy mathematics[18] has suggested that project velocity can be used as a metric of agility. There are agile self assessments to determine whether a team is using agile practices (Nokia test,[19] Karlskrona test,[20] 42 points test[21]).

While such approaches have been proposed to measure agility, the practical application of such metrics has yet to be seen.

Experience and reception

One of the early studies reporting gains in quality, productivity, and business satisfaction by using Agile methods was a survey conducted by Shine Technologies from November 2002 to January 2003.[22] A similar survey conducted in 2006 by Scott Ambler, the Practice Leader for Agile Development with IBM Rational's Methods Group reported similar benefits.[23] In a survey conducted by VersionOne in 2008, 55% of respondents answered that Agile methods had been successful in 90-100% of cases.[24] Others claim that agile development methods are still too young to require extensive academic proof of their success.[25]

Suitability

Large-scale agile software development remains an active research area.[26][27]

Agile development has been widely documented (see Experience Reports, below, as well as Beck[28] pg. 157, and Boehm and Turner[29]) as working well for small (<10 developers) co-located teams.

Some things that may negatively impact the success of an agile project are:

- Large-scale development efforts (>20 developers), though scaling strategies[27] and evidence of some large projects[30] have been described.
- Distributed development efforts (non-colocated teams). Strategies have been described in *Bridging the Distance*[31] and *Using an Agile Software Process with Offshore Development*[32]
- Forcing an agile process on a development team[33]
- Mission-critical systems where failure is not an option at any cost (e.g. software for surgical procedures).

Several successful large-scale agile projects have been documented.Template:Where BT has had several hundred developers situated in the UK, Ireland and India working collaboratively on projects and using Agile methods.[*citation needed*]

In terms of outsourcing agile development, Michael Hackett, Sr. Vice President of LogiGear Corporation has stated that "the offshore team. . . should have expertise, experience, good communication skills, inter-cultural understanding, trust and understanding between members and groups and with each other."[34]

Barry Boehm and Richard Turner suggest that risk analysis be used to choose between adaptive ("agile") and predictive ("plan-driven") methods.[29] The authors suggest that each side of the continuum has its own *home ground* as follows:

Agile home ground:[29]

- Low criticality
- Senior developers
- Requirements change often
- Small number of developers
- Culture that thrives on chaos

Plan-driven home ground:[29]

- High criticality
- Junior developers
- Requirements do not change often
- Large number of developers
- Culture that demands order

Formal methods:

- Extreme criticality
- Senior developers
- Limited requirements, limited features see Wirth's law
- Requirements that can be modeled
- Extreme quality

Experience reports

Agile development has been the subject of several conferences. Some of these conferences have had academic backing and included peer-reviewed papers, including a peer-reviewed experience report track. The experience reports share industry experiences with agile software development.

As of 2006, experience reports have been or will be presented at the following conferences:

- XP (2000,[35] 2001, 2002, 2003, 2004, 2005, 2006,[36] 2010 (proceedings published by IEEE)[37])
- XP Universe (2001[38])
- XP/Agile Universe (2002,[39] 2003,[40] 2004[41])
- Agile Development Conference[42] (2003,2004,2007,2008) (peer-reviewed; proceedings published by IEEE)

References

1. ↑ ***a*** ***b*** ***c*** Beck, Kent; et al. (2001). "Manifesto for Agile Software Development". Agile Alliance. http://agilemanifesto.org/. Retrieved 2010-06-14.
2. ↑ ***a*** ***b*** Gerald M. Weinberg, as quoted in Larman, Craig; Basili, Victor R. (June 2003). "Iterative and Incremental Development: A Brief History". *Computer* **36** (6): 47–56. doi:10.1109/MC.2003.1204375. ISSN 0018-9162. "We were doing incremental development as early as 1957, in Los Angeles, under the direction of Bernie Dimsdale [at IBM's ServiceBureau Corporation]. He was a colleague of John von Neumann, so perhaps he learned it there, or assumed it as totally natural. I do remember Herb Jacobs (primarily, though we all participated) developing a large simulation for Motorola, where the technique used was, as far as I can tell .... All of us, as far as I can remember, thought waterfalling of a huge project was rather stupid, or at least ignorant of the realities. I think what the waterfall description did for us was make us realize that we were doing something else, something unnamed except for 'software development.'".
3. ↑ Edmonds, E. A. (1974). "A Process for the Development of Software for Nontechnical Users as an Adaptive System". *General Systems* **19**: 215–18.
4. ↑ Larman, Craig (2004). *Agile and Iterative Development: A Manager's Guide*. Addison-Wesley. p. 27. ISBN 9780131111554
5. ↑ Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Stephen J. Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas

6. ↑ Beck, Kent; et al. (2001). "Principles behind the Agile Manifesto". Agile Alliance. http://www.agilemanifesto.org/principles.html. Retrieved 2010-06-06.
7. ↑ Anderson, David (2005). "Declaration of Interdependence". http://pmdoi.org.
8. ↑ Beck, Kent (1999). "Embracing Change with Extreme Programming". *Computer* **32** (10): 70–77. doi:10.1109/2.796139.
9. ↑ Black, S. E.; Boca., P. P.; Bowen, J. P.; Gorman, J.; Hinchey, M. G. (September 2009). "Formal versus agile: Survival of the fittest". *IEEE Computer* **49** (9): 39–45.
10. ↑ Boehm, B.; R. Turner (2004). *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston, MA: Addison-Wesley. ISBN 0-321-18612-5. Appendix A, pages 165-194
11. ↑ ***a*** ***b*** Aydin, M.N., Harmsen, F., Slooten, K. v., & Stagwee, R. A. (2004). An Agile Information Systems Development Method in use. *Turk J Elec Engin, 12(2)*, 127-138
12. ↑ Abrahamsson, P., Warsta, J., Siponen, M.T., & Ronkainen, J. (2003). New Directions on Agile Methods: A Comparative Analysis. *Proceedings of ICSE'03*, 244-254
13. ↑ Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J. (2002). Agile Software Development Methods: Review and Analysis. *VTT Publications 478*
14. ↑ ***a*** ***b*** Aydin, M.N., Harmsen, F., Slooten van K., & Stegwee, R.A. (2005). On the Adaptation of An Agile Information(Suren) Systems Development Method. *Journal of Database Management Special issue on Agile Analysis, Design, and Implementation, 16(4)*, 20-24
15. ↑ "David Bock's Weblog : Weblog". Jroller.com. http://jroller.com/page/bokmann?entry=improving_your_processes_aim_high. Retrieved 2010-04-02.
16. ↑ "Agility measurement index". Doi.acm.org. http://doi.acm.org/10.1145/1185448.1185509. Retrieved 2010-04-02.
17. ↑ Peter Lappo; Henry C.T. Andrew. "Assessing Agility". http://www.smr.co.uk/presentations/measure.pdf. Retrieved 2010-06-06.
18. ↑ Kurian, Tisni (2006). "Agility Metrics: A Quantitative Fuzzy Based Approach for Measuring Agility of a Software Process" *ISAM-Proceedings of International Conference on Agile Manufacturing'06(ICAM-2006)*, Norfolk, U.S.
19. ↑ Joe Little (2007-12-02). "Nokia test, A Scrum specific test". Agileconsortium.blogspot.com. http://agileconsortium.blogspot.com/2007/12/nokia-test.html. Retrieved 2010-06-06.
20. ↑ Mark Seuffert, Piratson Technologies, Sweden. "Karlskrona test, A generic agile adoption test". Piratson.se. http://www.piratson.se/archive/Agile_Karlskrona_Test.html. Retrieved 2010-06-06.
21. ↑ "How agile are you, A Scrum specific test". Agile-software-development.com. http://www.agile-software-development.com/2008/01/how-agile-are-you-take-this-42-point.html. Retrieved 2010-06-06.
22. ↑ "Agile Methodologies Survey Results" (PDF). Shine Technologies. 2003. http://www.shinetech.com/attachments/104_ShineTechAgileSurvey2003-01-17.pdf. Retrieved 2010-06-03. "95% [stated] that there was either no effect or a cost reduction . . . 93% stated that productivity was better or significantly better . . . 88% stated that quality was better or significantly better . . . 83% stated that business satisfaction was better or significantly better"
23. ↑ Ambler, Scott (August 3, 2006). "Survey Says: Agile Works in Practice". *Dr. Dobb's*. http://www.drdobbs.com/architecture-and-design/191800169;jsessionid=2QJ23QRYM3H4PQE1GHPCKH4ATMY32JVN?queryText=agile+survey. Retrieved 2010-06-03. "Only 6 percent indicated that their productivity was lowered . . . No change in productivity was reported by 34 percent of

respondents and 60 percent reported increased productivity. . . . 66 percent [responded] that the quality is higher. . . . 58 percent of organizations report improved satisfaction, whereas only 3 percent report reduced satisfaction."
24.↑ "The State of Agile Development" (PDF). VersionOne, Inc.. 2008. http://www.versionone.com/pdf/3rdAnnualStateOfAgile_FullDataReport.pdf. Retrieved 2010-07-03. "Agile delivers"
25.↑ "Answering the "Where is the Proof That Agile Methods Work" Question". Agilemodeling.com. 2007-01-19. http://www.agilemodeling.com/essays/proof.htm. Retrieved 2010-04-02.
26.↑ Agile Processes Workshop II Managing Multiple Concurrent Agile Projects. Washington: OOPSLA 2002
27.↑ ***a*** ***b*** W. Scott Ambler (2006) "Supersize Me" in Dr. Dobb's Journal, February 15, 2006.
28.↑ Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Boston, MA: Addison-Wesley. ISBN 0-321-27865-8.
29.↑ ***a*** ***b*** ***c*** ***d*** Boehm, B.; R. Turner (2004). *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston, MA: Addison-Wesley. pp. 55–57. ISBN 0-321-18612-5.
30.↑ Schaaf, R.J. (2007). "Agility XL", Systems and Software Technology Conference 2007, Tampa, FL
31.↑ "Bridging the Distance". Sdmagazine.com. http://www.drdobbs.com/architecture-and-design/184414899. Retrieved 2011-02-01.
32.↑ Martin Fowler. "Using an Agile Software Process with Offshore Development". Martinfowler.com. http://www.martinfowler.com/articles/agileOffshore.html. Retrieved 2010-06-06.
33.↑ [The Art of Agile Development James Shore & Shane Warden pg 47]
34.↑ [1] LogiGear, PC World Viet Nam, Jan 2011
35.↑ 2000
36.↑ "2006". Virtual.vtt.fi. http://virtual.vtt.fi/virtual/xp2006/. Retrieved 2010-06-06.
37.↑ "2010". Xp2010.org. http://www.xp2010.org/. Retrieved 2010-06-06.
38.↑ 2001
39.↑ 2002
40.↑ 2003
41.↑ 2004
42.↑ "Agile Development Conference". Agile200x.org. http://www.agile200x.org/. Retrieved 2010-06-06.

Further reading

- Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J. (2002). Agile Software Development Methods: Review and Analysis. *VTT Publications 478*.
- Cohen, D., Lindvall, M., & Costa, P. (2004). An introduction to agile methods. In *Advances in Computers* (pp. 1–66). New York: Elsevier Science.
- Dingsøyr, Torgeir, Dybå, Tore and Moe, Nils Brede (ed.): *Agile Software Develoment: Current Research and Future Directions*, Springer, Berlin Heidelberg, 2010.
- Fowler, Martin. *Is Design Dead?*. Appeared in *Extreme Programming Explained*, G. Succi and M. Marchesi, ed., Addison-Wesley, Boston. 2001.
- Larman, Craig and Basili, Victor R. *Iterative and Incremental Development: A Brief History* IEEE Computer, June 2003

- Riehle, Dirk. *A Comparison of the Value Systems of Adaptive Software Development and Extreme Programming: How Methodologies May Learn From Each Other*. Appeared in *Extreme Programming Explained*, G. Succi and M. Marchesi, ed., Addison-Wesley, Boston. 2001.
- Rother, Mike (2009). *Toyota Kata*. McGraw-Hill. ISBN 0071635238. http://books.google.com/?id=_1lhPgAACAAJ&dq=toyota+kata
- M. Stephens, D. Rosenberg. *Extreme Programming Refactored: The Case Against XP*. Apress L.P., Berkeley, California. 2003. (ISBN 1-59059-096-1)

External links

- Manifesto for Agile Software Development
- The Agile Alliance
- The Agile Executive
- Article Two Ways to Build a Pyramid by John Mayo-Smith
- Agile Software Development: A gentle introduction
- The New Methodology Martin Fowler's description of the background to agile methods
- Agile Journal - Largest online community focused specifically on agile development
- [9]]
- Agile Cookbook
- Ten Authors of The Agile Manifesto Celebrate its Tenth Anniversary

Standards

There are a few industry standards related to process improvement models we should mention briefly. For you as a beginner, it is enough to know they exist. However, if you start working for large corporations, you will find that many will follow one or the other of these standards.

Capability Maturity Model Integration

The Capability Maturity Model Integration (CMMI) is one of the leading models and based on best practice. Independent assessments grade organizations on how well they follow their defined processes, not on the quality of those processes or the software produced. CMMI has replaced CMM.

ISO 9000

ISO 9000 describes standards for a formally organized process to manufacture a product and the methods of managing and monitoring progress. Although the standard was originally created for the manufacturing sector, ISO 9000 standards have been applied to software development as well. Like CMMI, certification with ISO 9000 does not guarantee the quality of the end result, only that formalized business processes have been followed.

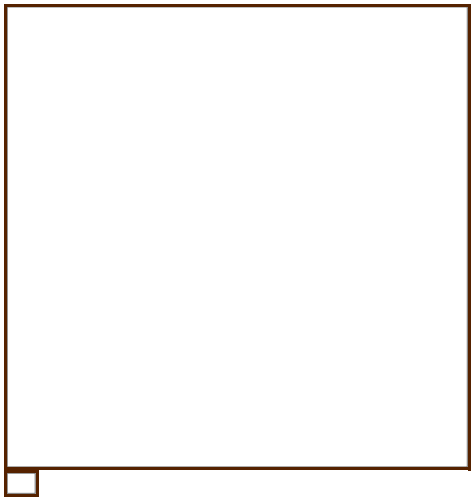
ISO 15504

ISO 15504, also known as Software Process Improvement Capability Determination (SPICE), is a "framework for the assessment of software processes". This standard is aimed at setting out a clear model for process comparison. SPICE is used much like CMMI. It models processes to manage, control, guide and monitor software development. This model is then used to measure what a development organization or project team actually does during software development. This information is analyzed to identify weaknesses and drive improvement. It also identifies strengths that can be continued or integrated into common practice for that organization or team.

External Links

- CMMI Official Website
- Introduction to Software Engineering/Print version at the Open Directory Project
- ISO 9000 at the Open Directory Project
- Introduction to ISO 9000 and ISO 14000
- ISO 15504 News (isospice)
- Automotive SPICE

Life Cycle



Model of the Systems Development Life Cycle with the Maintenance bubble highlighted.

The **Systems Development Life Cycle (SDLC)**, or *Software Development Life Cycle* in systems engineering, information systems and software engineering, is the process of creating or altering systems, and the models and methodologies that people use to develop these systems. The concept generally refers to computer or information systems.

In software engineering the SDLC concept underpins many kinds of software development methodologies. These methodologies form the framework for planning and controlling the creation of an information system[1]: the software development process.

Overview

Systems Development Life Cycle (SDLC) is a process used by a systems analyst to develop an information system, including requirements, validation, training, and user (stakeholder) ownership. Any SDLC should result in a high quality system that meets or exceeds customer expectations, reaches completion within time and cost estimates, works effectively and efficiently in the current and planned Information Technology infrastructure, and is inexpensive to maintain and cost-effective to enhance.[2]

Computer systems are complex and often (especially with the recent rise of Service-Oriented Architecture) link multiple traditional systems potentially supplied by different software vendors. To manage this level of complexity, a number of SDLC models have been created: "waterfall"; "fountain"; "spiral"; "build and fix"; "rapid prototyping"; "incremental"; and "synchronize and stabilize". [3]

SDLC models can be described along a spectrum of agile to iterative to sequential. Agile methodologies, such as XP and Scrum, focus on light-weight processes which allow for rapid changes along the development cycle. Iterative methodologies, such as Rational Unified Process and Dynamic Systems Development Method, focus on limited project scopes and expanding or improving products by multiple iterations. Sequential or big-design-up-front (BDUF) models, such as Waterfall, focus on complete and correct planning to guide large projects and risks to successful and predictable results [*citation needed*]. Other models, such as Anamorphic Development, tend to focus on a form of development that is guided by project scope and adaptive iterations of feature development.

In project management a project can be defined both with a project life cycle (PLC) and an SDLC, during which slightly different activities occur. According to Taylor (2004) "the project life cycle encompasses all the activities of the project, while the systems development life cycle focuses on realizing the product requirements".[4]

History

The **Systems Life Cycle (SLC)** is a type of methodology used to describe the process for building information systems, intended to develop information systems in a very deliberate, structured and methodical way, reiterating each stage of the life cycle. The systems development life cycle, according to Elliott & Strachan & Radford (2004), "originated in the 1960s,to develop large scale functional business systems in an age of large scale business conglomerates. Information systems activities revolved around heavy data processing and number crunching routines".[5]

Several systems development frameworks have been partly based on SDLC, such as the Structured Systems Analysis and Design Method (SSADM) produced for the UK government Office of Government Commerce in the 1980s. Ever since, according to Elliott (2004), "the traditional life cycle approaches to systems development have been increasingly replaced with alternative approaches and frameworks, which attempted to overcome some of the inherent deficiencies of the traditional SDLC".[5]

Systems development phases

The System Development Life Cycle framework provides a sequence of activities for system designers and developers to follow. It consists of a set of steps or phases in which each phase of the SDLC uses the results of the previous one.

A Systems Development Life Cycle (SDLC) adheres to important phases that are essential for developers, such as planning, analysis, design, and implementation, and are explained in the section below. A number of system development life cycle (SDLC) models have been created: waterfall, fountain, spiral, build and fix, rapid prototyping, incremental, and synchronize and stabilize. The oldest of these, and the best known, is the waterfall model: a sequence of stages in which the output of each stage becomes the input for the next. These stages can be characterized and divided up in different ways, including the following[6]:

- **Project planning, feasibility study**: Establishes a high-level view of the intended project and determines its goals.
- **Systems analysis, requirements definition**: Refines project goals into defined functions and operation of the intended application. Analyzes end-user information needs.
- **Systems design**: Describes desired features and operations in detail, including screen layouts, business rules, process diagrams, pseudocode and other documentation.
- **Implementation**: The real code is written here.
- **Integration and testing**: Brings all the pieces together into a special testing environment, then checks for errors, bugs and interoperability.
- **Acceptance, installation, deployment**: The final stage of initial development, where the software is put into production and runs actual business.
- **Maintenance**: What happens during the rest of the software's life: changes, correction, additions, moves to a different computing platform and more. This, the least glamorous and perhaps most important step of all, goes on seemingly forever.

In the following example (see picture) these stage of the Systems Development Life Cycle are divided in ten steps from definition to creation and modification of IT work products:

The tenth phase occurs when the system is disposed of and the task performed is either eliminated or transferred to other systems. The tasks and work products for each phase are described in subsequent chapters. [7]

Not every project will require that the phases be sequentially executed. However, the phases are interdependent. Depending upon the size and complexity of the project, phases may be combined or may overlap.[7]

System analysis

The goal of system analysis is to determine where the problem is in an attempt to fix the system. This step involves breaking down the system in different pieces to analyze the situation, analyzing project goals, breaking down what needs to be created and attempting to engage users so that definite requirements can be defined.

Requirements analysis sometimes requires individuals/teams from client as well as service provider sides to get detailed and accurate requirements; often there has to be a lot of communication to and from to understand these requirements. Requirement gathering is the most crucial aspect as many times communication gaps arise in this phase and this leads to validation errors and bugs in the software program.

Design

In systems design the design functions and operations are described in detail, including screen layouts, business rules, process diagrams and other documentation. The output of this stage will describe the new system as a collection of modules or subsystems.

The design stage takes as its initial input the requirements identified in the approved requirements document. For each requirement, a set of one or more design elements will be produced as a result of interviews, workshops, and/or prototype efforts.

Design elements describe the desired software features in detail, and generally include functional hierarchy diagrams, screen layout diagrams, tables of business rules, business process diagrams, pseudocode, and a complete entity-relationship diagram with a full data dictionary. These design elements are intended to describe the software in sufficient detail that skilled programmers may develop the software with minimal additional input design.

Implementation

Modular and subsystem programming code will be accomplished during this stage. Unit testing and module testing are done in this stage by the developers. This stage is intermingled with the next in that individual modules will need testing before integration to the main project.

Testing

The code is tested at various levels in software testing. Unit, system and user acceptance testings are often performed. This is a grey area as many different opinions exist as to what the stages of testing are and how much if any iteration occurs. Iteration is not generally part of the waterfall model, but usually some occur at this stage. In the testing the whole system is test one by one

Following are the types of testing:

- Defect testing
- Path testing
- Data set testing
- Unit testing
- System testing
- Integration testing
- Black box testing
- White box testing
- Regression testing
- Automation testing
- User acceptance testing
- Performance testing

Operations and maintenance

The deployment of the system includes changes and enhancements before the decommissioning or sunset of the system. Maintaining the system is an important aspect of SDLC. As key personnel change positions in the organization, new changes will be implemented, which will require system updates.

Systems Analysis and Design

The **Systems Analysis and Design (SAD)** is the process of developing Information Systems (IS) that effectively use of hardware, software, data, process, and people to support the company's business objectives.

Systems development life cycle topics

Management and control

SDLC Phases Related to Management Controls.[8]

The Systems Development Life Cycle (SDLC) phases serve as a programmatic guide to project activity and provide a flexible but consistent way to conduct projects to a depth matching the scope of the project. Each of the SDLC phase objectives are described in this section with key deliverables, a description of recommended tasks, and a summary of related control objectives for effective management. It is critical for the project manager to establish and monitor control objectives during each SDLC phase while executing projects. Control objectives help to provide a clear statement of the desired result or purpose and should be used throughout the entire SDLC process. Control objectives can be grouped into major categories (Domains), and relate to the SDLC phases as shown in the figure.[8]

To manage and control any SDLC initiative, each project will be required to establish some degree of a Work Breakdown Structure (WBS) to capture and schedule the work necessary to complete the project. The WBS and all programmatic material should be kept in the “Project Description” section of the project notebook. The WBS format is mostly left to the project manager to establish in a way that best describes the project work. There are some key areas that must be defined in the WBS as part of the SDLC policy. The following diagram describes three key areas that will be addressed in the WBS in a manner established by the project manager.[8]

Work breakdown structured organization

Work Breakdown Structure.[8]

The upper section of the Work Breakdown Structure (WBS) should identify the major phases and milestones of the project in a summary fashion. In addition, the upper section should provide an overview of the full scope and timeline of the project and will be part of the initial project description effort leading to project approval. The middle section of the WBS is based on the seven Systems Development Life Cycle (SDLC) phases as a guide for WBS task development. The WBS elements should consist of milestones and “tasks” as opposed to “activities” and have a definitive period (usually two weeks or more). Each task must have a measurable output (e.x. document, decision, or analysis). A WBS task may rely on one or more activities (e.g. software engineering, systems engineering) and may require close coordination with other tasks, either internal or external to the project. Any part of the project needing support from contractors should have a Statement of work (SOW) written to include the appropriate tasks from the SDLC phases. The development of a SOW does not occur during a specific phase of SDLC but is developed to include the work from the SDLC process that may be conducted by external resources such as contractors and struct.[8]

Baselines in the SDLC

Baselines are an important part of the Systems Development Life Cycle (SDLC). These baselines are established after four of the five phases of the SDLC and are critical to the iterative nature of the model .[9] Each baseline is considered as a milestone in the SDLC.

- Functional Baseline: established after the conceptual design phase.
- Allocated Baseline: established after the preliminary design phase.
- Product Baseline: established after the detail design and development phase.
- Updated Product Baseline: established after the production construction phase.

Complementary to SDLC

Complementary Software development methods to Systems Development Life Cycle (SDLC) are:

- Software Prototyping
- Joint Applications Design (JAD)
- Rapid Application Development (RAD)
- Extreme Programming (XP); extension of earlier work in Prototyping and RAD.
- Open Source Development
- End-user development
- Object Oriented Programming

Comparison of Methodology Approaches (Post, & Anderson 2006)[10]

	SDLC	RAD	Open Source	Objects	JAD	Prototyping	End User
Control	Formal	MIS	Weak	Standards	Joint	User	User
Time Frame	Long	Short	Medium	Any	Medium	Short	Short
Users	Many	Few	Few	Varies	Few	One or Two	One
MIS staff	Many	Few	Hundreds	Split	Few	One or Two	None
Transaction/DSS	Transaction	Both	Both	Both	DSS	DSS	DSS
Interface	Minimal	Minimal	Weak	Windows	Crucial	Crucial	Crucial
Documentation and training	Vital	Limited	Internal	In Objects	Limited	Weak	None
Integrity and security	Vital	Vital	Unknown	In Objects	Limited	Weak	Weak
Reusability	Limited	Some	Maybe	Vital	Limited	Weak	None

Strengths and weaknesses

Few people in the modern computing world would use a strict waterfall model for their Systems Development Life Cycle (SDLC) as many modern methodologies have superseded this thinking. Some will argue that the SDLC no longer applies to models like Agile computing, but it is still a term widely in use in Technology circles. The SDLC practice has advantages in traditional models of software development, that lends itself more to a structured environment. The disadvantages to using the SDLC methodology is when there is need for iterative development or (i.e. web development or e-commerce) where stakeholders need to review on a regular basis the software

being designed. Instead of viewing SDLC from a strength or weakness perspective, it is far more important to take the best practices from the SDLC model and apply it to whatever may be most appropriate for the software being designed.

A comparison of the strengths and weaknesses of SDLC:

Strength and Weaknesses of SDLC [10]

Strengths	Weaknesses
Control.	Increased development time.
Monitor Large projects.	Increased development cost.
Detailed steps.	Systems must be defined up front.
Evaluate costs and completion targets.	Rigidity.
Documentation.	Hard to estimate costs, project overruns.
Well defined user input.	User input is sometimes limited.
Ease of maintenance.	
Development and design standards.	
Tolerates changes in MIS staffing.	

An alternative to the SDLC is Rapid Application Development, which combines prototyping, Joint Application Development and implementation of CASE tools. The advantages of RAD are speed, reduced development cost, and active user involvement in the development process.

References

1. ↑ SELECTING A DEVELOPMENT APPROACH. Retrieved 27 October 2008.
2. ↑ "Systems Development Life Cycle". In: Foldoc(2000-12-24)
3. ↑ http://docs.google.com/viewer?a=v&q=cache:bfhOl8jp1S8J:condor.depaul.edu/~jpetlick/extra/394/Session2.ppt+&hl=en&pid=bl&srcid=ADGEEShCfW0_MLC4wRbczfUxrndHTkbwguF9fZuaUCe0RDyOCWyO2PTmaPhHnZ4jRhZZ75maVO_7gVAD2ex5-QIhrj1683hMefBNkak7FkQJCAwd-i0-_aQfEVEEKP177h4mmkvMMWJ7&sig=AHIEtbRhMlZ-TUyioKEhLQQxXk1WoSJXWA
4. ↑ James Taylor (2004). *Managing Information Technology Projects*. p.39..
5. ↑ ***a*** ***b*** Geoffrey Elliott & Josh Strachan (2004) *Global Business Information Technology*. p.87.
6. ↑ http://www.computerworld.com/s/article/71151/System_Development_Life_Cycle
7. ↑ ***a*** ***b*** US Department of Justice (2003). INFORMATION RESOURCES MANAGEMENT Chapter 1. Introduction.

8. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** U.S. House of Representatives (1999). *Systems Development Life-Cycle Policy*. p.13.
9. ↑ Blanchard, B. S., & Fabrycky, W. J.(2006) *Systems engineering and analysis* (4th ed.) New Jersey: Prentice Hall. p.31
10. ↑ ***a*** ***b*** Post, G., & Anderson, D., (2006). *Management information systems: Solving business problems with information technology*. (4th ed.). New York: McGraw-Hill Irwin.

Further reading

- Blanchard, B. S., & Fabrycky, W. J.(2006) *Systems engineering and analysis* (4th ed.) New Jersey: Prentice Hall.
- Cummings, Haag (2006). *Management Information Systems for the Information Age*. Toronto, McGraw-Hill Ryerson
- Beynon-Davies P. (2009). *Business Information Systems*. Palgrave, Basingstoke. ISBN 978-0-230-20368-6
- Computer World, 2002, Retrieved on June 22, 2006 from the World Wide Web:
- Management Information Systems, 2005, Retrieved on June 22, 2006 from the World Wide Web:

External links

Wikimedia Commons has media related to: ***Systems Development Life Cycle***

- The Agile System Development Lifecycle
- Pension Benefit Guaranty Corporation - Information Technology Solutions Lifecycle Methodology
- HHS Enterprise Performance Life Cycle Framework

Rapid Application Development

Rapid application development (**RAD**) refers to a type of software development methodology that uses minimal planning in favor of rapid prototyping. The "planning" of software developed using RAD is interleaved with writing the software itself. The lack of extensive pre-planning generally allows software to be written much faster, and makes it easier to change requirements.

Overview

Rapid application development is a software development methodology that involves methods like iterative development and software prototyping. According to Whitten (2004), it is a merger of various structured techniques, especially data-driven Information Engineering, with prototyping techniques to accelerate software systems development.[1]

In rapid application development, structured techniques and prototyping are especially used to define users' requirements and to design the final system. The development process starts with the development of preliminary data models and business process models using structured techniques. In the next stage, requirements are verified using prototyping, eventually to refine the data and process models. These stages are repeated iteratively; further development results in "a combined business requirements and technical design statement to be used for constructing new systems".[1]

RAD approaches may entail compromises in functionality and performance in exchange for enabling faster development and facilitating application maintenance.

History

Rapid application development is a term originally used to describe a software development process introduced by James Martin in 1991. Martin's methodology involves iterative development and the construction of prototypes. More recently, the term and its acronym have come to be used in a broader, general sense that encompasses a variety of methods aimed at speeding application development, such as the use of software frameworks of varied types, such as web application frameworks.

Rapid application development was a response to non-agile processes developed in the 1970s and 1980s, such as the Structured Systems Analysis and Design Method and other Waterfall models. One problem with previous methodologies was that applications took so long to build that requirements had changed before the system was complete, resulting in inadequate or even unusable systems. Another problem was the assumption that a methodical requirements analysis phase alone would identify all the critical requirements. Ample evidence[*citation needed*] attests to the fact that this is seldom the case, even for projects with highly experienced professionals at all levels.

Starting with the ideas of Brian Gallagher, Alex Balchin, Barry Boehm and Scott Shultz, James Martin developed the rapid application development approach during the 1980s at IBM and finally formalized it by publishing a book in 1991, *Rapid Application Development*.

Relative effectiveness

The shift from traditional session-based client/server development to open sessionless and collaborative development like Web 2.0 has increased the need for faster iterations through the phases of the SDLC.[2] This, coupled with the growing use of open source frameworks and products in core commercial development, has, for many developers, rekindled interest in finding a silver bullet RAD methodology.

Although most RAD methodologies foster software re-use, small team structure and distributed system development, most RAD practitioners recognize that, ultimately, no one “rapid” methodology can provide an order of magnitude improvement over any other development methodology.

All types of RAD have the potential for providing a good framework for faster product development with improved software quality, but successful implementation and benefits often hinge on project type, schedule, software release cycle and corporate culture. It may also be of

interest that some of the largest software vendors such as Microsoft[3] and IBM[4] do not extensively use RAD in the development of their flagship products and for the most part, they still primarily rely on traditional waterfall methodologies with some degree of spiraling.[5]

This table contains a high-level summary of some of the major types of RAD and their relative strengths and weaknesses.

Agile software development (Agile)	
Pros	Minimizes feature creep by developing in short intervals resulting in miniature software projects and releasing the product in mini-increments.
Cons	Short iteration may add too little functionality, leading to significant delays in final iterations. Since Agile emphasizes real-time communication (preferably face-to-face), using it is problematic for large multi-team distributed system development. Agile methods produce very little written documentation and require a significant amount of post-project documentation.
Extreme Programming (XP)	
Pros	Lowers the cost of changes through quick spirals of new requirements. Most design activity occurs incrementally and on the fly.
Cons	Programmers must work in pairs, which is difficult for some people. No up-front “detailed design” occurs, which can result in more redesign effort in the long term. The business champion attached to the project full time can potentially become a single point of failure for the project and a major source of stress for a team.
Joint application design (JAD)	
Pros	Captures the voice of the customer by involving them in the design and development of the application through a series of collaborative workshops called JAD sessions.
Cons	The client may create an unrealistic product vision and request extensive gold-plating, leading a team to over- or under-develop functionality.
Lean software development (LD)	
Pros	Creates minimalist solutions (i.e., needs determine technology) and delivers less functionality earlier; per the policy that 80% today is better than 100% tomorrow.
Cons	Product may lose its competitive edge because of insufficient core functionality and may exhibit poor overall quality.
Rapid application development (RAD)	
Pros	Promotes strong collaborative atmosphere and dynamic gathering of requirements. Business owner actively participates in prototyping, writing test cases and performing unit testing.
Cons	Dependence on strong cohesive teams and individual commitment to the project. Decision making relies on the feature functionality team and a communal decision-making process with lesser degree of centralized PM and

	engineering authority.
Scrum	
Pros	Improved productivity in teams previously paralyzed by heavy “process”, ability to prioritize work, use of backlog for completing items in a series of short iterations or sprints, daily measured progress and communications.
Cons	Reliance on facilitation by a master who may lack the political skills to remove impediments and deliver the sprint goal. Due to relying on self-organizing teams and rejecting traditional centralized "process control", internal power struggles can paralyze a team.

Table 1: Pros and Cons of various RAD types

Criticism

Since rapid application development is an iterative and incremental process, it can lead to a succession of prototypes that never culminate in a satisfactory production application. Such failures may be avoided if the application development tools are robust, flexible, and put to proper use. This is addressed in methods such as the 2080 Development method or other post-agile variants.

Practical implications

When organizations adopt rapid development methodologies, care must be taken to avoid role and responsibility confusion and communication breakdown within a development team, and between team and client. In addition, especially in cases where the client is absent or not able to participate with authority in the development process, the system analyst should be endowed with this authority on behalf of the client to ensure appropriate prioritisation of non-functional requirements. Furthermore, no increment of the system should be developed without a thorough and formally documented design phase.[6]

References

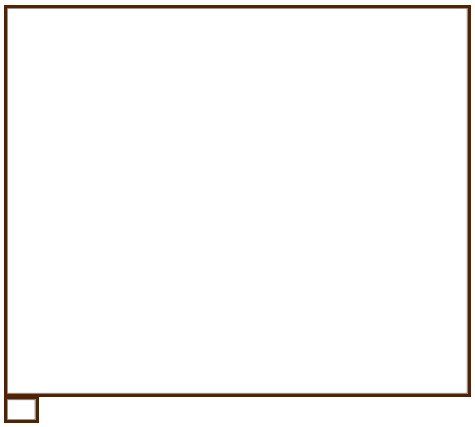
1. ↑ ***a*** ***b*** Whitten, Jeffrey L.; Lonnie D. Bentley, Kevin C. Dittman. (2004). *Systems Analysis and Design Methods*. 6th edition. ISBN 025619906X.
2. ↑ Maurer and S. Martel. (2002). "Extreme Programming: Rapid Development for Web-Based Applications". IEEE Internet Computing, 6(1) pp 86-91 January/February 2002.
3. ↑ Andrew Begel, Nachiappan Nagappan. "Usage and Perceptions of Agile Software Development in an Industrial Context: An Exploratory Study, *Microsoft Research*". http://research.microsoft.com/pubs/56015/AgileDevatMS-ESEM07.pdf. Retrieved 2008-11-15.
4. ↑ E. M. Maximilien and L. Williams. (2003). "Assessing Test-driven Development at IBM". Proceedings of International Conference of Software Engineering, Portland, OR, pp. 564-569, 2003.
5. ↑ M. Stephens, Rosenberg, D. (2003). "Extreme Programming Refactored: The Case Against XP". Apress, 2003.

6. ↑ Gerber, Aurona; Van der Merwe, Alta; Alberts, Ronell; (2007), Implications of Rapid Development Methodologies, CSITEd 2007, Mauritius, November 2007 [2]

Further reading

- Steve McConnell (1996). *Rapid Development: Taming Wild Software Schedules*, Microsoft Press Books, ISBN 978-1556159008
- Kerr, James M.; Hunter, Richard (1993). *Inside RAD: How to Build a Fully-Functional System in 90 Days or Less*. McGraw-Hill. ISBN 0070342237.
- Ellen Gottesdiener (1995). "RAD Realities: Beyond the Hype to How RAD Really Works" Application Development Trends
- Ken Schwaber (1996). *Agile Project Management with Scrum*, Microsoft Press Books, ISBN 978-0735619937
- Steve McConnell (2003). *Professional Software Development: Shorter Schedules, Higher Quality Products, More Successful Projects, Enhanced Careers*, Microsoft Prese s Books, ISBN 978-0321193674
- Dean Leffingwell (2007). *Scaling Software Agility: Best Practices for Large Enterprises*, Addison-Wesley Professional, ISBN 978-0321458193

Extreme Programming



Planning and feedback loops in Extreme Programming.

Extreme Programming (XP) is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements. As a type of agile software development,[1][2][3] it advocates frequent "releases" in short development cycles (timeboxing), which is intended to improve productivity and introduce checkpoints where new customer requirements can be adopted.

Other elements of extreme programming include: programming in pairs or doing extensive code review, unit testing of all code, avoiding programming of features until they are actually needed, a flat management structure, simplicity and clarity in code, expecting changes in the customer's requirements as time passes and the problem is better understood, and frequent communication with the customer and among programmers.[2][3][4] The methodology takes its name from the idea that the beneficial elements of traditional software engineering practices are taken to "extreme" levels,

on the theory that if some is good, more is better. It is unrelated to "cowboy coding", which is more free-form and unplanned. It does not advocate "death march" work schedules, but instead working at a sustainable pace.[5]

Critics have noted several potential drawbacks,[6] including problems with unstable requirements, no documented compromises of user conflicts, and a lack of an overall design specification or document.

History

Extreme Programming was created by Kent Beck during his work on the Chrysler Comprehensive Compensation System (C3) payroll project.[6] Beck became the C3 project leader in March 1996 and began to refine the development method used in the project and wrote a book on the method (in October 1999, *Extreme Programming Explained* was published).[6] Chrysler cancelled the C3 project in February 2000, after the company was acquired by Daimler-Benz.[7]

Although extreme programming itself is relatively new, many of its practices have been around for some time; the methodology, after all, takes "best practices" to extreme levels. For example, the "practice of test-first development, planning and writing tests before each micro-increment" was used as early as NASA's Project Mercury, in the early 1960s *(Larman 2003)*. To shorten the total development time, some formal test documents (such as for acceptance testing) have been developed in parallel (or shortly before) the software is ready for testing. A NASA independent test group can write the test procedures, based on formal requirements and logical limits, before the software has been written and integrated with the hardware. In XP, this concept is taken to the extreme level by writing automated tests (perhaps inside of software modules) which validate the operation of even small sections of software coding, rather than only testing the larger features. Some other XP practices, such as refactoring, modularity, bottom-up design, and incremental design were described by Leo Brodie in his book published in 1984.[8]

Origins

Software development in the 1990s was shaped by two major influences: internally, object-oriented programming replaced procedural programming as the programming paradigm favored by some in the industry; externally, the rise of the Internet and the dot-com boom emphasized speed-to-market and company-growth as competitive business factors. Rapidly-changing requirements demanded shorter product life-cycles, and were often incompatible with traditional methods of software development.

The Chrysler Comprehensive Compensation System was started in order to determine the best way to use object technologies, using the payroll systems at Chrysler as the object of research, with Smalltalk as the language and GemStone as the data access layer. They brought in Kent Beck,[6] a prominent Smalltalk practitioner, to do performance tuning on the system, but his role expanded as he noted several problems they were having with their development process. He took this opportunity to propose and implement some changes in their practices based on his work with his frequent collaborator, Ward Cunningham. Beck describes the early conception of the methods:[9]

> The first time I was asked to lead a team, I asked them to do a little bit of the things I thought were sensible, like testing and reviews. The second time there was a lot more on

> the line. I thought, "Damn the torpedoes, at least this will make a good article," [and] asked the team to crank up all the knobs to 10 on the things I thought were essential and leave out everything else.

Beck invited Ron Jeffries to the project to help develop and refine these methods. Jeffries thereafter acted as a coach to instill the practices as habits in the C3 team.

Information about the principles and practices behind XP was disseminated to the wider world through discussions on the original Wiki, Cunningham's WikiWikiWeb. Various contributors discussed and expanded upon the ideas, and some spin-off methodologies resulted (see agile software development). Also, XP concepts have been explained, for several years, using a hypertext system map on the XP website at "http://www.extremeprogramming.org" circa 1999.

Beck edited a series of books on XP, beginning with his own *Extreme Programming Explained* (1999, ISBN 0-201-61641-6), spreading his ideas to a much larger, yet very receptive, audience. Authors in the series went through various aspects attending XP and its practices. Even a book was written, critical of the practices.

Current state

XP created quite a buzz in the late 1990s and early 2000s, seeing adoption in a number of environments radically different from its origins.

The high discipline required by the original practices often went by the wayside, causing some of these practices, such as those thought too rigid, to be deprecated or reduced, or even left unfinished, on individual sites. For example, the practice of end-of-day integration tests, for a particular project, could be changed to an end-of-week schedule, or simply reduced to mutually agreed dates. Such a more relaxed schedule could avoid people feeling rushed to generate artificial stubs just to pass the end-of-day testing. A less rigid schedule allows, instead, for some complex features to be more fully developed over a several-day period. However, some level of periodic integration testing can detect groups of people working in non-compatible, tangent efforts before too much work is invested in divergent, wrong directions.

Meanwhile, other agile development practices have not stood still, and XP is still evolving, assimilating more lessons from experiences in the field, to use other practices. In the second edition of *Extreme Programming Explained*, Beck added more values and practices and differentiated between primary and corollary practices.

Concept

Goals

Extreme Programming Explained describes Extreme Programming as a software development discipline that organizes people to produce higher quality software more productively.

In traditional system development methods (such as SSADM or the waterfall model) the requirements for the system are determined at the beginning of the development project and often fixed from that point on. This means that the cost of changing the requirements at a later stage (a common feature of software engineering projects[*citation needed*]) will be high. Like other agile software development methods, XP attempts to reduce the cost of change by having multiple short development cycles, rather than one long one. In this doctrine changes are a natural, inescapable and desirable aspect of software development projects, and should be planned for instead of attempting to define a stable set of requirements.

Extreme programming also introduces a number of basic values, principles and practices on top of the agile programming framework.

Activities

XP describes four basic activities that are performed within the software development process: coding, testing, listening, and designing. Each of those activities is described below.

Coding

The advocates of XP argue that the only truly important product of the system development process is code - software instructions a computer can interpret. Without code, there is no working product.

Coding can also be used to figure out the most suitable solution. Coding can also help to communicate thoughts about programming problems. A programmer dealing with a complex programming problem and finding it hard to explain the solution to fellow programmers might code it and use the code to demonstrate what he or she means. Code, say the proponents of this position, is always clear and concise and cannot be interpreted in more than one way. Other programmers can give feedback on this code by also coding their thoughts.

Testing

One can not be certain that a function works unless one tests it. Bugs and design errors are pervasive problems in software development. Extreme programming's approach is that if a little testing can eliminate a few flaws, a lot of testing can eliminate many more flaws.

- Unit tests determine whether a given feature works as intended. A programmer writes as many automated tests as they can think of that might "break" the code; if all tests run successfully, then the coding is complete. Every piece of code that is written is tested before moving on to the next feature.
- Acceptance tests verify that the requirements as understood by the programmers satisfy the customer's actual requirements. These occur in the exploration phase of release planning.

A "testathon" is an event when programmers meet to do collaborative test writing, a kind of brainstorming relative to software testing.

Listening

Programmers must listen to what the customers need the system to do, what "business logic" is needed. They must understand these needs well enough to give the customer feedback about the technical aspects of how the problem might be solved, or cannot be solved. Communication between the customer and programmer is further addressed in the Planning Game.

Designing

From the point of view of simplicity, of course one could say that system development doesn't need more than coding, testing and listening. If those activities are performed well, the result should always be a system that works. In practice, this will not work. One can come a long way without designing but at a given time one will get stuck. The system becomes too complex and the dependencies within the system cease to be clear. One can avoid this by creating a design structure that organizes the logic in the system. Good design will avoid lots of dependencies within a system; this means that changing one part of the system will not affect other parts of the system.

Values

Extreme Programming initially recognized four values in 1999. A new value was added in the second edition of *Extreme Programming Explained*. The five values are:

Communication

Building software systems requires communicating system requirements to the developers of the system. In formal software development methodologies, this task is accomplished through documentation. Extreme programming techniques can be viewed as methods for rapidly building and disseminating institutional knowledge among members of a development team. The goal is to give all developers a shared view of the system which matches the view held by the users of the system. To this end, extreme programming favors simple designs, common metaphors, collaboration of users and programmers, frequent verbal communication, and feedback.

Simplicity

Extreme Programming encourages starting with the simplest solution. Extra functionality can then be added later. The difference between this approach and more conventional system development methods is the focus on designing and coding for the needs of today instead of those of tomorrow, next week, or next month. This is sometimes summed up as the "you ain't gonna need it" (YAGNI) approach.[5] Proponents of XP acknowledge the disadvantage that this can sometimes entail more effort tomorrow to change the system; their claim is that this is more than compensated for by the advantage of not investing in possible future requirements that might change before they become relevant. Coding and designing for uncertain future requirements implies the risk of spending resources on something that might not be needed. Related to the "communication" value, simplicity in design and coding should improve the quality of communication. A simple design with very simple code could be easily understood by most programmers in the team.

Feedback

Within extreme programming, feedback relates to different dimensions of the system development:

- Feedback from the system: by writing unit tests,[6] or running periodic integration tests, the programmers have direct feedback from the state of the system after implementing changes.
- Feedback from the customer: The functional tests (aka acceptance tests) are written by the customer and the testers. They will get concrete feedback about the current state of their system. This review is planned once in every two or three weeks so the customer can easily steer the development.
- Feedback from the team: When customers come up with new requirements in the planning game the team directly gives an estimation of the time that it will take to implement.

Feedback is closely related to communication and simplicity. Flaws in the system are easily communicated by writing a unit test that proves a certain piece of code will break. The direct feedback from the system tells programmers to recode this part. A customer is able to test the system periodically according to the functional requirements, known as *user stories*.[6] To quote Kent Beck, "Optimism is an occupational hazard of programming, feedback is the treatment."[*citation needed*]

Courage

Several practices embody courage. One is the commandment to always design and code for today and not for tomorrow. This is an effort to avoid getting bogged down in design and requiring a lot of effort to implement anything else. Courage enables developers to feel comfortable with refactoring their code when necessary.[6] This means reviewing the existing system and modifying it so that future changes can be implemented more easily. Another example of courage is knowing when to throw code away: courage to remove source code that is obsolete, no matter how much effort was used to create that source code. Also, courage means persistence: A programmer might be stuck on a complex problem for an entire day, then solve the problem quickly the next day, if only they are persistent.

Respect

The respect value includes respect for others as well as self-respect. Programmers should never commit changes that break compilation, that make existing unit-tests fail, or that otherwise delay the work of their peers. Members respect their own work by always striving for high quality and seeking for the best design for the solution at hand through refactoring.

Adopting the four earlier values leads to respect gained from others in the team. Nobody on the team should feel unappreciated or ignored. This ensures a high level of motivation and encourages loyalty toward the team and toward the goal of the project. This value is very dependent upon the other values, and is very much oriented toward people in a team.

Rules

The first version of rules for XP was published in 1999 by Don Wells[10] at the XP website. 29 rules are given in the categories of planning, managing, designing, coding, and testing. Planning, managing and designing are called out explicitly to counter claims that XP doesn't support those activities.

Another version of XP rules was proposed by Ken Auer[11] in XP/Agile Universe 2003. He felt XP was defined by its rules, not its practices (which are subject to more variation and ambiguity). He defined two categories: "Rules of Engagement" which dictate the environment in which software development can take place effectively, and "Rules of Play" which define the minute-by-minute activities and rules within the framework of the Rules of Engagement.

Principles

The principles that form the basis of XP are based on the values just described and are intended to foster decisions in a system development project. The principles are intended to be more concrete than the values and more easily translated to guidance in a practical situation.

Feedback

Extreme programming sees feedback as most useful if it is done rapidly and expresses that the time between an action and its feedback is critical to learning and making changes. Unlike traditional system development methods, contact with the customer occurs in more frequent iterations. The customer has clear insight into the system that is being developed. He or she can give feedback and steer the development as needed.

Unit tests also contribute to the rapid feedback principle. When writing code, the unit test provides direct feedback as to how the system reacts to the changes one has made. If, for instance, the changes affect a part of the system that is not in the scope of the programmer who made them, that programmer will not notice the flaw. There is a large chance that this bug will appear when the system is in production.

Assuming simplicity

This is about treating every problem as if its solution were "extremely simple". Traditional system development methods say to plan for the future and to code for reusability. Extreme programming rejects these ideas.

The advocates of extreme programming say that making big changes all at once does not work. Extreme programming applies incremental changes: for example, a system might have small releases every three weeks. When many little steps are made, the customer has more control over the development process and the system that is being developed.

Embracing change

The principle of embracing change is about not working against changes but embracing them. For instance, if at one of the iterative meetings it appears that the customer's requirements have changed dramatically, programmers are to embrace this and plan the new requirements for the next iteration.

Practices

Extreme programming has been described as having 12 practices, grouped into four areas:

Fine scale feedback

- Pair programming[6]
- Planning game
- Test-driven development
- Whole team

Continuous process

- Continuous integration
- Refactoring or design improvement[6]
- Small releases

Shared understanding

- Coding standards
- Collective code ownership[6]
- Simple design[6]
- System metaphor

Programmer welfare

- Sustainable pace

Coding

- The customer is always available
- Code the Unit test first
- Only one pair integrates code at a time
- Leave Optimization till last
- No Overtime

Testing

- All code must have Unit tests
- All code must pass all Unit tests before it can be released.
- When a Bug is found tests are created before the bug is addressed (a bug is not an error in logic, it is a test you forgot to write)
- Acceptance tests are run often and the results are published

Controversial aspects

The practices in XP have been heavily debated.[6] Proponents of extreme programming claim that by having the on-site customer[6] request changes informally, the process becomes flexible, and saves the cost of formal overhead. Critics of XP claim this can lead to costly rework and project scope creep beyond what was previously agreed or funded.

Change control boards are a sign that there are potential conflicts in project objectives and constraints between multiple users. XP's expedited methodology is somewhat dependent on programmers being able to assume a unified client viewpoint so the programmer can concentrate on coding rather than documentation of compromise objectives and constraints. This also applies when multiple programming organizations are involved, particularly organizations which compete for shares of projects.[*citation needed*]

Other potentially controversial aspects of extreme programming include:

- Requirements are expressed as automated acceptance tests rather than specification documents.
- Requirements are defined incrementally, rather than trying to get them all in advance.
- Software developers are usually required to work in pairs.
- There is no Big Design Up Front. Most of the design activity takes place on the fly and incrementally, starting with "the simplest thing that could possibly work" and adding complexity only when it's required by failing tests. Critics compare this to "debugging a system into appearance" and fear this will result in more re-design effort than only re-designing when requirements change.
- A customer representative is attached to the project. This role can become a single-point-of-failure for the project, and some people have found it to be a source of stress. Also, there is the danger of micro-management by a non-technical representative trying to dictate the use of technical software features and architecture.
- Dependence upon all other aspects of XP: "XP is like a ring of poisonous snakes, daisy-chained together. All it takes is for one of them to wriggle loose, and you've got a very angry, poisonous snake heading your way."[12]

Scalability

Historically, XP only works on teams of twelve or fewer people. One way to circumvent this limitation is to break up the project into smaller pieces and the team into smaller groups. It has been claimed that XP has been used successfully on teams of over a hundred developers[*citation needed*]. ThoughtWorks has claimed reasonable success on distributed XP projects with up to sixty people[*citation needed*].

In 2004 Industrial Extreme Programming (IXP)[13] was introduced as an evolution of XP. It is intended to bring the ability to work in large and distributed teams. It now has 23 practices and flexible values. As it is a new member of the Agile family, there is not enough data to prove its usability, however it claims to be an answer to what it sees as XP's imperfections.

Severability and responses

In 2003, Matt Stephens and Doug Rosenberg published *Extreme Programming Refactored: The Case Against XP* which questioned the value of the XP process and suggested ways in which it could be improved. This triggered a lengthy debate in articles, internet newsgroups, and web-site chat areas. The core argument of the book is that XP's practices are interdependent but that few practical organizations are willing/able to adopt all the practices; therefore the entire process fails. The book also makes other criticisms and it draws a likeness of XP's "collective ownership" model to socialism in a negative manner.

Certain aspects of XP have changed since the book *Extreme Programming Refactored* (2003) was published; in particular, XP now accommodates modifications to the practices as long as the required objectives are still met. XP also uses increasingly generic terms for processes. Some argue that these changes invalidate previous criticisms; others claim that this is simply watering the process down.

RDP Practice is a technique for tailoring extreme programming. This practice was initially proposed as a long research paper in a workshop organized by Philippe Kruchten and Steve Adolph(See APSO workshop at ICSE 2008) and yet it is the only proposed and applicable method for customizing XP. The valuable concepts behind RDP practice, in a short time provided the rationale for applicability of it in industries. RDP Practice tries to customize XP by relying on technique XP Rules.

Other authors have tried to reconcile XP with the older methods in order to form a unified methodology. Some of these XP sought to replace, such as the waterfall method; example: Project Lifecycles: Waterfall, Rapid Application Development, and All That. JPMorgan Chase & Co. tried combining XP with the computer programming methodologies of Capability Maturity Model Integration (CMMI), and Six Sigma. They found that the three systems reinforced each other well, leading to better development, and did not mutually contradict.[14]

Criticism

Extreme programming's initial buzz and controversial tenets, such as pair programming and continuous design, have attracted particular criticisms, such as the ones coming from McBreen[15] and Boehm and Turner.[16] Many of the criticisms, however, are believed by Agile practitioners to be misunderstandings of agile development.[17]

In particular, extreme programming is reviewed and critiqued by Matt Stephens's and Doug Rosenberg's *Extreme Programming Refactored*.[18]

Criticisms include:

- A methodology is only as effective as the people involved, Agile does not solve this
- Often used as a means to bleed money from customers through lack of defining a deliverable
- Lack of structure and necessary documentation
- Only works with senior-level developers
- Incorporates insufficient software design
- Requires meetings at frequent intervals at enormous expense to customers
- Requires too much cultural change to adopt
- Can lead to more difficult contractual negotiations
- Can be very inefficient—if the requirements for one area of code change through various iterations, the same programming may need to be done several times over. Whereas if a plan were there to be followed, a single area of code is expected to be written once.
- Impossible to develop realistic estimates of work effort needed to provide a quote, because at the beginning of the project no one knows the entire scope/requirements
- Can increase the risk of scope creep due to the lack of detailed requirements documentation
- Agile is feature driven; non-functional quality attributes are hard to be placed as user stories

References

1. ↑ "Human Centred Technology Workshop 2005", 2005, PDF webpage: Informatics-UK-report-cdrp585.
2. ↑ **a** **b** "Design Patterns and Refactoring", University of Pennsylvania, 2003, webpage: UPenn-Lectures-design-patterns.
3. ↑ **a** **b** "Extreme Programming" (lecture paper), USFCA.edu, webpage: USFCA-edu-601-lecture.
4. ↑ "Manifesto for Agile Software Development", Agile Alliance, 2001, webpage: Manifesto-for-Agile-Software-Dev
5. ↑ **a** **b** "Everyone's a Programmer" by Clair Tristram. *Technology Review*, Nov 2003. p. 39
6. ↑ **a** **b** **c** **d** **e** **f** **g** **h** **i** **j** **k** **l** **m** "Extreme Programming", *Computerworld* (online), December 2001, webpage: Computerworld-appdev-92.
7. ↑ *Extreme Programming Refactored: The Case Against XP*. ISBN 1590590961.
8. ↑ Brodie, Leo (1984) (paperback). *Thinking Forth*. Prentice-Hall. ISBN 0-13-917568-7. http://thinking-forth.sourceforge.net. Retrieved 2006-06-19.
9. ↑ Interview with Kent Beck and Martin Fowler
10. ↑ Don Wells
11. ↑ Ken Auer

12.↑ The Case Against Extreme Programming: A Self-Referential Safety Net
13.↑ Cutter Consortium :: Industrial XP: Making XP Work in Large Organizations
14.↑ Extreme Programming (XP) Six Sigma CMMI.
15.↑ McBreen, P. (2003). *Questioning Extreme Programming*. Boston, MA: Addison-Wesley. ISBN 0-201-84457-5.
16.↑ Boehm, B.; R. Turner (2004). *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston, MA: Addison-Wesley. ISBN 0-321-18612-5.
17.↑ sdmagazine
18.↑ Extreme Programming Refactored, Matt Stephens and Doug Rosenberg, Publisher: Apress L.P.

Further reading

- Ken Auer and Roy Miller. *Extreme Programming Applied: Playing To Win*, Addison-Wesley.
- Kent Beck: *Extreme Programming Explained: Embrace Change*, Addison-Wesley.
- Kent Beck and Martin Fowler: *Planning Extreme Programming*, Addison-Wesley.
- Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change, Second Edition*, Addison-Wesley.
- Alistair Cockburn: *Agile Software Development*, Addison-Wesley.
- Martin Fowler: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley.
- Harvey Herela (2005). Case Study: The Chrysler Comprehensive Compensation System. Galen Lab, U.C. Irvine.
- Jim Highsmith. *Agile Software Development Ecosystems*, Addison-Wesley.
- Ron Jeffries, Ann Anderson and Chet Hendrickson (2000), *Extreme Programming Installed*, Addison-Wesley.
- Mehdi Mirakhorli (2008). *RDP technique: a practice to customize xp*, International Conference on Software Engineering, Proceedings of the 2008 international workshop on Scrutinizing agile practices or shoot-out at the agile corral, Leipzig, Germany 2008, Pages 23–32.
- Craig Larman & V. Basili (2003). "Iterative and Incremental Development: A Brief History", Computer (IEEE Computer Society) 36 (6): 47-56.
- Matt Stephens and Doug Rosenberg (2003). *Extreme Programming Refactored: The Case Against XP*, Apress.
- Waldner, JB. (2008). "Nanocomputers and Swarm Intelligence". In: ISTE, 225-256.

External links

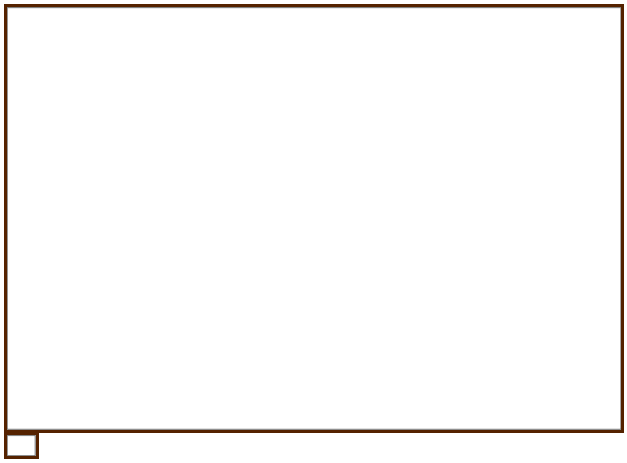
Wikimedia Commons has media related to: ***Extreme Programming***

- Extreme Programming
- A gentle introduction
- Industrial eXtreme Programming
- XP magazine
- Problems and Solutions to XP implementation

- Using an Agile Software Process with Offshore Development - ThoughtWorks' experiences with implementing XP in large distributed projects

Planning

Requirements



Requirements analysis is the first stage in the systems engineering process and software development process.[1]

Requirements analysis in systems engineering and software engineering, encompasses those tasks that go into determining the needs or conditions to meet for a new or altered product, taking account of the possibly conflicting requirements of the various stakeholders, such as beneficiaries or users.

Requirements analysis is critical to the success of a development project.[2] Requirements must be documented, actionable, measurable, testable, related to identified business needs or opportunities, and defined to a level of detail sufficient for system design. Requirements can be architectural, structural, behavioral, functional, and non-functional.

Overview

Conceptually, requirements analysis includes three types of activity:

- Eliciting requirements: the task of communicating with customers and users to determine what their requirements are. This is sometimes also called requirements gathering.
- Analyzing requirements: determining whether the stated requirements are unclear, incomplete, ambiguous, or contradictory, and then resolving these issues.
- Recording requirements: Requirements might be documented in various forms, such as natural-language documents, use cases, user stories, or process specifications.

Requirements analysis can be a long and arduous process during which many delicate psychological skills are involved. New systems change the environment and relationships between people, so it is important to identify all the stakeholders, take into account all their needs and ensure they understand the implications of the new systems. Analysts can employ several techniques to elicit the requirements from the customer. Historically, this has included such things as holding interviews, or holding focus groups (more aptly named in this context as requirements workshops)

and creating requirements lists. More modern techniques include prototyping, and use cases. Where necessary, the analyst will employ a combination of these methods to establish the exact requirements of the stakeholders, so that a system that meets the business needs is produced.

Requirements engineering

Systematic requirements analysis is also known as *requirements engineering*.[3] It is sometimes referred to loosely by names such as *requirements gathering*, *requirements capture*, or requirements specification. The term *requirements analysis* can also be applied specifically to the analysis proper, as opposed to elicitation or documentation of the requirements, for instance. Requirements Engineering can be divided into discrete chronological steps:

- Requirements elicitation,
- Requirements analysis and negotiation,
- Requirements specification,
- System modeling,
- Requirements validation,
- Requirements management.

Requirement engineering according to Laplante (2007) is "a subdiscipline of systems engineering and software engineering that is concerned with determining the goals, functions, and constraints of hardware and software systems."[4] In some life cycle models, the requirement engineering process begins with a feasibility study activity, which leads to a feasibility report. If the feasibility study suggests that the product should be developed, then requirement analysis can begin. If requirement analysis precedes feasibility studies, which may foster outside the box thinking, then feasibility should be determined before requirements are finalized.

Requirements analysis topics

Stakeholder identification

See Stakeholder analysis for a discussion of business uses. Stakeholders (SH) are people or organizations (legal entities such as companies, standards bodies) which have a valid interest in the system. They may be affected by it either directly or indirectly. A major new emphasis in the 1990s was a focus on the identification of *stakeholders*. It is increasingly recognized that stakeholders are not limited to the organization employing the analyst. Other stakeholders will include:

- anyone who operates the system (normal and maintenance operators)
- anyone who benefits from the system (functional, political, financial and social beneficiaries)
- anyone involved in purchasing or procuring the system. In a mass-market product organization, product management, marketing and sometimes sales act as surrogate consumers (mass-market customers) to guide development of the product
- organizations which regulate aspects of the system (financial, safety, and other regulators)
- people or organizations opposed to the system (negative stakeholders; see also Misuse case)

- organizations responsible for systems which interface with the system under design
- those organizations who integrate horizontally with the organization for whom the analyst is designing the system

Stakeholder interviews

Stakeholder interviews are a common technique used in requirement analysis. Though they are generally idiosyncratic in nature and focused upon the perspectives and perceived needs of the stakeholder, very often without larger enterprise or system context, this perspective deficiency has the general advantage of obtaining a much richer understanding of the stakeholder's unique business processes, decision-relevant business rules, and perceived needs. Consequently this technique can serve as a means of obtaining the highly focused knowledge that is often not elicited in Joint Requirements Development sessions, where the stakeholder's attention is compelled to assume a more cross-functional context. Moreover, the in-person nature of the interviews provides a more relaxed environment where lines of thought may be explored at length.

Joint Requirements Development (JRD) Sessions

Requirements often have cross-functional implications that are unknown to individual stakeholders and often missed or incompletely defined during stakeholder interviews. These cross-functional implications can be elicited by conducting JRD sessions in a controlled environment, facilitated by a trained facilitator, wherein stakeholders participate in discussions to elicit requirements, analyze their details and uncover cross-functional implications. A dedicated scribe and Business Analyst should be present to document the discussion. Utilizing the skills of a trained facilitator to guide the discussion frees the Business Analyst to focus on the requirements definition process.

JRD Sessions are analogous to Joint Application Design Sessions. In the former, the sessions elicit requirements that guide design, whereas the latter elicit the specific design features to be implemented in satisfaction of elicited requirements.

Contract-style requirement lists

One traditional way of documenting requirements has been contract style requirement lists. In a complex system such requirements lists can run to hundreds of pages.

An appropriate metaphor would be an extremely long shopping list. Such lists are very much out of favour in modern analysis; as they have proved spectacularly unsuccessful at achieving their aims; but they are still seen to this day.

Strengths

- Provides a checklist of requirements.
- Provide a contract between the project sponsor(s) and developers.
- For a large system can provide a high level description.

Weaknesses

- Such lists can run to hundreds of pages. It is virtually impossible to read such documents as a whole and have a coherent understanding of the system.
- Such requirements lists abstract all the requirements and so there is little context
    - This abstraction makes it impossible to see how the requirements fit or work together.
    - This abstraction makes it difficult to prioritize requirements properly; while a list does make it easy to prioritize each individual item, removing one item out of context can render an entire use case or business requirement useless.
    - This abstraction increases the likelihood of misinterpreting the requirements; as more people read them, the number of (different) interpretations of the envisioned system increase.
    - This abstraction means that it's extremely difficult to be sure that you have the majority of the requirements. Necessarily, these documents speak in generality; but the devil, as they say, is in the details.
- These lists create a false sense of mutual understanding between the stakeholders and developers.
- These contract style lists give the stakeholders a false sense of security that the developers must achieve certain things. However, due to the nature of these lists, they inevitably miss out crucial requirements which are identified later in the process. Developers can use these discovered requirements to renegotiate the terms and conditions in their favour.
- These requirements lists are no help in system design, since they do not lend themselves to application.

Alternative to requirement lists

As an alternative to the large, pre-defined requirement lists Agile Software Development uses User stories to define a requirement in every day language.

Measurable goals

Best practices take the composed list of requirements merely as clues and repeatedly ask "why?" until the actual business purposes are discovered. Stakeholders and developers can then devise tests to measure what level of each goal has been achieved thus far. Such goals change more slowly than the long list of specific but unmeasured requirements. Once a small set of critical, measured goals has been established, rapid prototyping and short iterative development phases may proceed to deliver actual stakeholder value long before the project is half over.

Prototypes

In the mid-1980s, prototyping was seen as the best solution to the requirements analysis problem. Prototypes are Mockups of an application. Mockups allow users to visualize an application that hasn't yet been constructed. Prototypes help users get an idea of what the system will look like, and make it easier for users to make design decisions without waiting for the system to be built. Major

improvements in communication between users and developers were often seen with the introduction of prototypes. Early views of applications led to fewer changes later and hence reduced overall costs considerably.

However, over the next decade, while proving a useful technique, prototyping did not solve the requirements problem:

- Managers, once they see a prototype, may have a hard time understanding that the finished design will not be produced for some time.
- Designers often feel compelled to use patched together prototype code in the real system, because they are afraid to 'waste time' starting again.
- Prototypes principally help with design decisions and user interface design. However, they can not tell you what the requirements originally were.
- Designers and end-users can focus too much on user interface design and too little on producing a system that serves the business process.
- Prototypes work well for user interfaces, screen layout and screen flow but are not so useful for batch or asynchronous processes which may involve complex database updates and/or calculations.

Prototypes can be flat diagrams (often referred to as wireframes) or working applications using synthesized functionality. Wireframes are made in a variety of graphic design documents, and often remove all color from the design (i.e. use a greyscale color palette) in instances where the final software is expected to have graphic design applied to it. This helps to prevent confusion over the final visual look and feel of the application.

Use cases

A use case is a technique for documenting the potential requirements of a new system or software change. Each use case provides one or more *scenarios* that convey how the system should interact with the end-user or another system to achieve a specific business goal. Use cases typically avoid technical jargon, preferring instead the language of the end-user or *domain expert*. Use cases are often co-authored by requirements engineers and stakeholders.

Use cases are deceptively simple tools for describing the behavior of software or systems. A use case contains a textual description of all of the ways which the intended users could work with the software or system. Use cases do not describe any internal workings of the system, nor do they explain how that system will be implemented. They simply show the steps that a user follows to perform a task. All the ways that users interact with a system can be described in this manner.

Software requirements specification

A software requirements specification (SRS) is a complete description of the behavior of the system to be developed. It includes a set of use cases that describe all of the interactions that the users will have with the software. Use cases are also known as functional requirements. In addition to use cases, the SRS also contains nonfunctional (or supplementary) requirements. Non-functional requirements are requirements which impose constraints on the design or implementation (such as performance requirements, quality standards, or design constraints).

Recommended approaches for the specification of software requirements are described by IEEE 830-1998. This standard describes possible structures, desirable contents, and qualities of a software requirements specification.

Types of Requirements

Requirements are categorized in several ways. The following are common categorizations of requirements that relate to technical management:[1]

Customer Requirements
: Statements of fact and assumptions that define the expectations of the system in terms of mission objectives, environment, constraints, and measures of effectiveness and suitability (MOE/MOS). The customers are those that perform the eight primary functions of systems engineering, with special emphasis on the operator as the key customer. Operational requirements will define the basic need and, at a minimum, answer the questions posed in the following listing:[1]

- *Operational distribution or deployment*: Where will the system be used?
- *Mission profile or scenario*: How will the system accomplish its mission objective?
- *Performance and related parameters*: What are the critical system parameters to accomplish the mission?
- *Utilization environments*: How are the various system components to be used?
- *Effectiveness requirements*: How effective or efficient must the system be in performing its mission?
- *Operational life cycle*: How long will the system be in use by the user?
- *Environment*: What environments will the system be expected to operate in an effective manner?

Architectural Requirements
: Architectural requirements explain what has to be done by identifying the necessary system architecture of a system.

Structural Requirements
: Structural requirements explain what has to be done by identifying the necessary structure of a system.

Behavioral Requirements
: Behavioral requirements explain what has to be done by identifying the necessary behavior of a system.

Functional Requirements
: Functional requirements explain what has to be done by identifying the necessary task, action or activity that must be accomplished. Functional requirements analysis will be used as the toplevel functions for functional analysis.[1]

Non-functional Requirements
: Non-functional requirements are requirements that specify criteria that can be used to judge the operation of a system, rather than specific behaviors.

Performance Requirements
: The extent to which a mission or function must be executed; generally measured in terms of quantity, quality, coverage, timeliness or readiness. During requirements analysis, performance (how well does it have to be done) requirements will be interactively developed

across all identified functions based on system life cycle factors; and characterized in terms of the degree of certainty in their estimate, the degree of criticality to system success, and their relationship to other requirements.[1]

Design Requirements

The “build to,” “code to,” and “buy to” requirements for products and “how to execute” requirements for processes expressed in technical data packages and technical manuals.[1]

Derived Requirements

Requirements that are implied or transformed from higher-level requirement. For example, a requirement for long range or high speed may result in a design requirement for low weight.[1]

Allocated Requirements

A requirement that is established by dividing or otherwise allocating a high-level requirement into multiple lower-level requirements. Example: A 100-pound item that consists of two subsystems might result in weight requirements of 70 pounds and 30 pounds for the two lower-level items.[1]

Well-known requirements categorization models include FURPS and FURPS+, developed at Hewlett-Packard.

Requirements analysis issues

Stakeholder issues

Steve McConnell, in his book *Rapid Development*, details a number of ways users can inhibit requirements gathering:

- Users do not understand what they want or users don't have a clear idea of their requirements
- Users will not commit to a set of written requirements
- Users insist on new requirements after the cost and schedule have been fixed
- Communication with users is slow
- Users often do not participate in reviews or are incapable of doing so
- Users are technically unsophisticated
- Users do not understand the development process
- Users do not know about present technology

This may lead to the situation where user requirements keep changing even when system or product development has been started.

Engineer/developer issues

Possible problems caused by engineers and developers during requirements analysis are:

- Technical personnel and end-users may have different vocabularies. Consequently, they may wrongly believe they are in perfect agreement until the finished product is supplied.
- Engineers and developers may try to make the requirements fit an existing system or model, rather than develop a system specific to the needs of the client.

- Analysis may often be carried out by engineers or programmers, rather than personnel with the people skills and the domain knowledge to understand a client's needs properly.

Attempted solutions

One attempted solution to communications problems has been to employ specialists in business or system analysis.

Techniques introduced in the 1990s like prototyping, Unified Modeling Language (UML), use cases, and Agile software development are also intended as solutions to problems encountered with previous methods.

Also, a new class of application simulation or application definition tools have entered the market. These tools are designed to bridge the communication gap between business users and the IT organization — and also to allow applications to be 'test marketed' before any code is produced. The best of these tools offer:

- electronic whiteboards to sketch application flows and test alternatives
- ability to capture business logic and data needs
- ability to generate high fidelity prototypes that closely imitate the final application
- interactivity
- capability to add contextual requirements and other comments
- ability for remote and distributed users to run and interact with the simulation

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

Further reading

- Laplante, Phil (2009). *Requirements Engineering for Software and Systems* (1st ed.). Redmond, WA: CRC Press. ISBN 1-42006-467-3. http://beta.crcpress.com/product/isbn/9781420064674.

- McConnell, Steve (1996). *Rapid Development: Taming Wild Software Schedules* (1st ed.). Redmond, WA: Microsoft Press. ISBN 1-55615-900-5. http://www.stevemcconnell.com/.
- Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
- Andrew Stellman and Jennifer Greene (2005). *Applied Software Project Management*. Cambridge, MA: O'Reilly Media. ISBN 0-596-00948-8. http://www.stellman-greene.com.
- Brian Berenbach, Daniel Paulish, Juergen Katzmeier, Arnold Rudorfer (2009). *Software & Systems Requirements Engineering: In Practice*. New York: McGraw-Hill Professional. ISBN 0-07-1605479. http://www.mhprofessional.com.
- Walter Sobkiw (2008). *Sustainable Development Possible with Creative System Engineering*. New Jersey: CassBeth. ISBN 0615216307. http://books.google.com/books?id=7WJppXs-LzEC.

External links

Wikimedia Commons has media related to: ***Requirements analysis***

- Software Requirement Analysis using UML article by Dhiraj Shetty.
- Requirements Engineering Process "Goodies"
- Requirements Engineering: A Roadmap (PDF) article by Bashar Nuseibeh and Steve Easterbrook, 2000.

Requirements Management

Requirements management is the process of documenting, analyzing, tracing, prioritizing and agreeing on requirements and then controlling change and communicating to relevant stakeholders. It is a continuous process throughout a project. A requirement is a capability to which a project outcome (product or service) should conform.

Overview

The purpose of requirements management is to assure the organization documents, verifies and meets the needs and expectations of its customers and internal or external stakeholders[1]. Requirements management begins with the analysis and elicitation of the objectives and constraints of the organization. Requirements management further includes supporting planning for requirements, integrating requirements and the organization for working with them (attributes for requirements), as well as relationships with other information delivering against requirements, and changes for these.

The traceability thus established is used in managing requirements to report back fulfillment of company and stakeholder interests, in terms of compliance, completeness, coverage and consistency. Traceabilities also support change management as part of requirements management in

understanding the impacts of changes through requirements or other related elements (e.g., functional impacts through relations to functional architecture), and facilitating introducing these changes.[2]

Requirements management involves communication between the project team members and stakeholders, and adjustment to requirements changes throughout the course of the project[3]. To prevent one class of requirements from overriding another, constant communication among members of the development team is critical. For example, in software development for internal applications, the business has such strong needs that it may ignore user requirements, or believe that in creating use cases, the user requirements are being taken care of.

Traceability

Requirements traceability is concerned with documenting the life of a requirement. It should be possible to trace back to the origin of each requirement and every change made to the requirement should therefore be documented in order to achieve traceability. Even the use of the requirement after the implemented features have been deployed and used should be traceable[4].

Requirements come from different sources, like the business person ordering the product, the marketing manager and the actual user. These people all have different requirements for the product. Using requirements traceability, an implemented feature can be traced back to the person or group that wanted it during the requirements elicitation. This can, for example, be used during the development process to prioritize the requirement, determining how valuable the requirement is to a specific user. It can also be used after the deployment when user studies show that a feature is not used, to see why it was required in the first place.

Requirements activities

At each stage in a development process, there are key requirements management activities and methods. To illustrate, consider a standard five-phase development process with Investigation, Feasibility, Design, Construction and Test, and Release stages.

Investigation

In Investigation, the first three classes of requirements are gathered from the users, from the business and from the development team. In each area, similar questions are asked; what are the goals, what are the constraints, what are the current tools or processes in place, and so on. Only when these requirements are well understood can functional requirements be developed.

A caveat is required here: no matter how hard a team tries, requirements cannot be fully defined at the beginning of the project. Some requirements will change, either because they simply weren’t extracted, or because internal or external forces at work affect the project in mid-cycle. Thus, the team members must agree at the outset that a prime condition for success is flexibility in thinking and operation.

The deliverable from the Investigation stage is a requirements document that has been approved by all members of the team. Later, in the thick of development, this document will be critical in preventing scope creep or unnecessary changes. As the system develops, each new feature opens a world of new possibilities, so the requirements specification anchors the team to the original vision and permits a controlled discussion of scope change.

While many organizations still use only documents to manage requirements, others manage their requirements baselines using software tools. These tools allow requirements to be managed in a database, and usually have functions to automate traceability (e.g., by allowing electronic links to be created between parent and child requirements, or between test cases and requirements), electronic baseline creation, version control, and change management. Usually such tools contain an export function that allows a specification document to be created by exporting the requirements data into a standard document application.

Feasibility

In the Feasibility stage, costs of the requirements are determined. For user requirements, the current cost of work is compared to the future projected costs once the new system is in place. Questions such as these are asked: “What are data entry errors costing us now?” Or “What is the cost of scrap due to operator error with the current interface?” Actually, the need for the new tool is often recognized as these questions come to the attention of financial people in the organization.

Business costs would include, “What department has the budget for this?” “What is the expected rate of return on the new product in the marketplace?” “What’s the internal rate of return in reducing costs of training and support if we make a new, easier-to-use system?”

Technical costs are related to software development costs and hardware costs. “Do we have the right people to create the tool?” “Do we need new equipment to support expanded software roles?” This last question is an important type. The team must inquire into whether the newest automated tools will add sufficient processing power to shift some of the burden from the user to the system in order to save people time.

The question also points out a fundamental point about requirements management. A human and a tool form a system, and this realization is especially important if the tool is a computer or a new application on a computer. The human mind excels in parallel processing and interpretation of trends with insufficient data. The CPU excels in serial processing and accurate mathematical computation. The overarching goal of the requirements management effort for a software project would thus be to make sure the work being automated gets assigned to the proper processor. For instance, “Don’t make the human remember where she is in the interface. Make the interface report the human’s location in the system at all times.” Or “Don’t make the human enter the same data in two screens. Make the system store the data and fill in the second screen as needed.”

The deliverable from the Feasibility stage is the budget and schedule for the project.

Design

Assuming that costs are accurately determined and benefits to be gained are sufficiently large, the project can proceed to the Design stage. In Design, the main requirements management activity is comparing the results of the design against the requirements document to make sure that work is staying in scope.

Again, flexibility is paramount to success. Here's a classic story of scope change in mid-stream that actually worked well. Ford auto designers in the early '80s were expecting gasoline prices to hit $3.18 per gallon by the end of the decade. Midway through the design of the Ford Taurus, prices had centered to around $1.50 a gallon. The design team decided they could build a larger, more comfortable, and more powerful car if the gas prices stayed low, so they redesigned the car. The Taurus launch set nationwide sales records when the new car came out, primarily because it was so roomy and comfortable to drive.

In most cases, however, departing from the original requirements to that degree does not work. So the requirements document becomes a critical tool that helps the team make decisions about design changes.

Construction and test

In the construction and testing stage, the main activity of requirements management is to make sure that work and cost stay within schedule and budget, and that the emerging tool does in fact meet requirements. A main tool used in this stage is prototype construction and iterative testing. For a software application, the user interface can be created on paper and tested with potential users while the framework of the software is being built. Results of these tests are recorded in a user interface design guide and handed off to the design team when they are ready to develop the interface. This saves their time and makes their jobs much easier.

Release

Requirements management does not end with product release. From that point on, the data coming in about the application's acceptability is gathered and fed into the Investigation phase of the next generation or release. Thus the process begins again.

Tools

There exist both desktop and Web-based tools for requirements management. A Web-based requirements tool can be installed at the customer's datacenter or can be offered as an on-demand requirements management platform which in some cases is completely free.[5]

Modeling Languages

The system engineering modeling language SysML incorporates a requirements diagram allowing the developer to graphically organize, manage, and trace requirements.

On-demand requirements management platforms

An on-demand requirements management platform is a fully hosted requirements management solution, where the only system requirements would normally be Internet access and a standard Web browser.

The service would normally include all special hardware and software. Other services may include technology and processes designed to secure your data against physical loss and unauthorized use, 24×7 data availability, and assurance that the service will scale as you add users, applications, and additional activities.

Some on-demand requirements management platforms charge a fee while others are free to use.

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

Further reading

- CMMI Product Team (August 2006) (PDF). *CMMI for Development, Version 1.2*. Technical Report CMU/SEI-2006-TR-008. Software Engineering Institute. http://www.sei.cmu.edu/library/abstracts/reports/06tr008.cfm. Retrieved 2008-01-22.
- Colin Hood, Simon Wiedemann, Stefan Fichtinger, Urte Pautz *Requirements Management: Interface Between Requirements Development and All Other Engineering Processes* Springer, Berlin 2007, ISBN 354047689X

External links

- Critical Issues in Requirements Management - Panel Discussion with Executives from IBM/ Rational, Cognition Corporation, PTC, Chrysler, and Siemens.
- Web 2.0 Requirements Management - What does it look like and why is it relevant?
- Forbes Requirements Management Software Directory
- INCOSE Requirements Tools Survey
- Jiludwig Requirements Management Tools Directory

- Requirements Management Tool Resources
- Washington State Information Services Board (ISB)policy: CMM Key Practices for Level 2 - Requirements Management
- U.K. Office of Government Commerce (OGC) - Requirements management
- Requirement Writing 101 for Product Management

Specification

Systems engineering model of Specification and Levels of Development. During system development a series of specifications are generated to describe the system at different levels of detail. These program unique specifications form the core of the configuration baselines. As shown here, in addition to referring to different levels within the system hierarchy, these baselines are defined at different phases of the design process.[6]

A **functional specification** (also, *functional spec*, *specs*, *functional specifications document (FSD)*, or *Program specification*) in systems engineering and software development is the documentation that describes the requested behavior of an engineering system. The documentation typically describes what is needed by the system user as well as requested properties of inputs and outputs (e.g. of the software system).

Overview

In systems engineering a specification is a document that clearly and accurately describes the essential technical requirements for items, materials, or services including the procedures by which it can be determined that the requirements have been met. Specifications help avoid duplication and inconsistencies, allow for accurate estimates of necessary work and resources, act as a negotiation and reference document for engineering changes, provide documentation of configuration, and allow for consistent communication among those responsible for the eight primary functions of Systems Engineering. They provide a precise idea of the problem to be solved so that they can efficiently design the system and estimate the cost of design alternatives. They provide guidance to testers for verification (qualification) of each technical requirement.[6]

A functional specification does not define the inner workings of the proposed system; it does not include the specification how the system function will be implemented. Instead, it focuses on what various outside agents (people using the program, computer peripherals, or other computers, for example) might "observe" when interacting with the system. A typical functional specification might state the following:

> *When the user clicks the OK button, the dialog is closed and the focus is returned to the main window in the state it was in before this dialog was displayed.*

Such a requirement describes an interaction between an external agent (the user) and the software system. When the user provides input to the system by clicking the OK button, the program responds (or should respond) by closing the dialog window containing the OK button.

It can be *informal*, in which case it can be considered as a blueprint or user manual from a developer point of view, or *formal*, in which case it has a definite meaning defined in mathematical or programmatic terms. In practice, most successful specifications are written to understand and fine-tune applications that were already well-developed, although safety-critical software systems are often carefully specified prior to application development. Specifications are most important for external interfaces that must remain stable.

Functional specification topics

Purpose

There are many purposes for functional specifications. One of the primary purposes on team projects is to achieve some form of team consensus on what the program is to achieve before making the more time-consuming effort of writing source code and test cases, followed by a period of debugging. Typically, such consensus is reached after one or more reviews by the stakeholders on the project at hand after having negotiated a cost-effective way to achieve the requirements the software needs to fulfill.

Process

In the ordered industrial software engineering life-cycle (waterfall model), functional specification describes what has to be implemented. The next system specification document describes how the functions will be realized using a chosen software environment. In not industrial, prototypical systems development, functional specifications are typically written after or as part of requirements analysis.

When the team agrees that functional specification consensus is reached, the functional spec is typically declared "complete" or "signed off". After this, typically the software development and testing team write source code and test cases using the functional specification as the reference. While testing is performed the behavior of the program is compared against the expected behavior as defined in the functional specification.

Types of software development specifications

- Advanced Microcontroller Bus Architecture
- Bit specification
- Design specification
- Diagnostic design specification
- Multiboot Specification
- Product design specification
- Real-time specification for Java
- Software Requirements Specification

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

External links

- Writing functional specifications Tutorial
- Painless Functional Specifications, 4-part series by Joel Spolsky

Architecture & Design

Introduction

The **software architecture** of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both. [7] The term also refers to documentation of a system's software architecture. Documenting software architecture facilitates communication between stakeholders, documents early decisions about high-level design, and allows reuse of design components and patterns between projects.[8]

Overview

The field of computer science has come across problems associated with complexity since its formation.[9] Earlier problems of complexity were solved by developers by choosing the right data structures, developing algorithms, and by applying the concept of separation of concerns. Although the term “software architecture” is relatively new to the industry, the fundamental principles of the field have been applied sporadically by software engineering pioneers since the mid 1980s. Early attempts to capture and explain software architecture of a system were imprecise and disorganized, often characterized by a set of box-and-line diagrams.[10] During the 1990s there was a concentrated effort to define and codify fundamental aspects of the discipline. Initial sets of design patterns, styles, best practices, description languages, and formal logic were developed during that time.

The software architecture discipline is centered on the idea of reducing complexity through abstraction and separation of concerns. To date there is still no agreement on the precise definition of the term “software architecture”.[11]

As a maturing discipline with no clear rules on the right way to build a system, designing software architecture is still a mix of art and science. The “art” aspect of software architecture is because a commercial software system supports some aspect of a business or a mission. How a system supports key business drivers is described via scenarios as non-functional requirements of a system, also known as quality attributes, determine how a system will behave.[12] Every system is unique due to the nature of the business drivers it supports, as such the degree of quality attributes exhibited by a system such as fault-tolerance, backward compatibility, extensibility, reliability, maintainability, availability, security, usability, and such other –ilities will vary with each implementation.[12] To bring a software architecture user's perspective into the software architecture, it can be said that software architecture gives the direction to take steps and do the tasks involved in each such user's speciality area and interest e.g. the stakeholders of software systems, the software developer, the software system operational support group, the software maintenance specialists, the deployer, the tester and also the business end user[*citation needed*]. In this sense software architecture is really the amalgamation of the multiple perspectives a system always embodies. The fact that those several different perspectives can be put together into a software architecture stands as the vindication of the need and justification of creation of software architecture before the software development in a project attains maturity.

History

The origin of software architecture as a concept was first identified in the research work of Edsger Dijkstra in 1968 and David Parnas in the early 1970s. These scientists emphasized that the structure of a software system matters and getting the structure right is critical. The study of the field increased in popularity since the early 1990s with research work concentrating on architectural styles (patterns), architecture description languages, architecture documentation, and formal methods[13].

Research institutions have played a prominent role in furthering software architecture as a discipline. Mary Shaw and David Garlan of Carnegie Mellon wrote a book titled *Software Architecture: Perspectives on an Emerging Discipline* in 1996, which brought forward the concepts in Software Architecture, such as components, connectors, styles and so on. The University of California, Irvine's Institute for Software Research's efforts in software architecture research is directed primarily in architectural styles, architecture description languages, and dynamic architectures.

The IEEE 1471: ANSI/IEEE 1471-2000: Recommended Practice for Architecture Description of Software-Intensive Systems is the first formal standard in the area of software architecture, and was adopted in 2007 by ISO as *ISO/IEC 42010:2007 (IEEE 1471).*

Software architecture topics

Architecture description languages

Architecture description languages (**ADLs**) are used to describe a Software Architecture. Several different ADLs have been developed by different organizations, including AADL (SAE standard), Wright (developed by Carnegie Mellon), Acme (developed by Carnegie Mellon), xADL (developed by UCI), Darwin (developed by Imperial College London), DAOP-ADL (developed by University of Málaga), and ByADL (University of L'Aquila, Italy). Common elements of an ADL are component, connector and configuration.

Views

Software architecture is commonly organized in views, which are analogous to the different types of blueprints made in building architecture. A view is a representation of a set of system components and relationships among them. [7] Within the ontology established by ANSI/IEEE 1471-2000, *views* are responses to *viewpoints*, where a viewpoint is a specification that describes the architecture in question from the perspective of a given set of stakeholders and their concerns. The viewpoint specifies not only the concerns addressed but the presentation, model kinds used, conventions used and any consistency (correspondence) rules to keep a view consistent with other views.

Some possible views (actually, *viewpoints* in the 1471 ontology) are:

- Functional/logic view

- Code/module view
- Development/structural view
- Concurrency/process/runtime/thread view
- Physical/deployment/install view
- User action/feedback view
- Data view/data model

Several languages for describing software architectures ('architecture description language' in ISO/IEC 42010 / IEEE-1471 terminology) have been devised, but no consensus exists on which symbol-set or language should be used to describe each architecture view. The UML is a standard that can be used for *"for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes."* Thus, the UML is a visual language that can be used to create software architecture views.

Architecture frameworks

Frameworks related to the domain of software architecture are:

- 4+1
- RM-ODP (Reference Model of Open Distributed Processing)
- Service-Oriented Modeling Framework (SOMF)

Other architectures such as the Zachman Framework, DODAF, and TOGAF relate to the field of Enterprise architecture.

The distinction from functional design

The IEEE Std 610.12-1990 Standard Glossary of Software Engineering Terminology defines the following distinctions:

- Architectural Design: the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.
- Detailed Design: the process of refining and expanding the preliminary design of a system or component to the extent that the design is sufficiently complete to begin implementation.
- Functional Design: the process of defining the working relationships among the components of a system.
- Preliminary Design: the process of analyzing design alternatives and defining the architecture, components, interfaces, and timing/sizing estimates for a system or components.

Software architecture, also described as strategic design, is an activity concerned with global requirements governing *how* a solution is implemented such as programming paradigms, architectural styles, component-based software engineering standards, architectural patterns, security, scale, integration, and law-governed regularities. Functional design, also described as

tactical design, is an activity concerned with local requirements governing *what* a solution does such as algorithms, design patterns, programming idioms, refactorings, and low-level implemenation.

According to the Intension/Locality Hypothesis[14], the distinction between architectural and detailed design is defined by the Locality Criterion[14], according to which a statement about software design is non-local (architectural) if and only if a program that satisfies it can be expanded into a program which does not. For example, the client–server style is architectural (strategic) because a program that is built on this principle can be expanded into a program which is not client–server; for example, by adding peer-to-peer nodes.

Architecture is design but not all design is architectural.[7] In practice, the architect is the one who draws the line between software architecture (architectural design) and detailed design (non-architectural design). There aren't rules or guidelines that fit all cases. Examples of rules or heuristics that architects (or organizations) can establish when they want to distinguish between architecture and detailed design include:

- Architecture is driven by non-functional requirements, while functional design is driven by functional requirements.
- Pseudo-code belongs in the detailed design document.
- UML component, deployment, and package diagrams generally appear in software architecture documents; UML class, object, and behavior diagrams appear in detailed functional design documents.

Examples of architectural styles and patterns

There are many common ways of designing computer software modules and their communications, among them:

- Blackboard
- Client–server model (2-tier, n-tier, peer-to-peer, cloud computing all use this model)
- Database-centric architecture (broad division can be made for programs which have database at its center and applications which don't have to rely on databases, E.g. desktop application programs, utility programs etc.)
- Distributed computing
- Event-driven architecture
- Front end and back end
- Implicit invocation
- Monolithic application
- Peer-to-peer
- Pipes and filters
- Plugin
- Representational State Transfer
- Rule evaluation
- Search-oriented architecture (A pure SOA implements a service for every data access point.)
- Service-oriented architecture
- Shared nothing architecture

- Software componentry
- Space based architecture
- Structured (module-based but usually monolithic within modules)
- Three-tier model (An architecture with Presentation, Business Logic and Database tiers)

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

Further reading

- Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, Judith Stafford: *Documenting Software Architectures: Views and Beyond, Second Edition*. Addison-Wesley, 2010, ISBN 0321552687. This book describes what is software architecture and shows how to document it in multiple views, using UML and other notations. It also explains how to complement the architecture views with behavior, software interface, and rationale documentation. Accompanying the book is a wiki that contains an example of software architecture documentation.
- Len Bass, Paul Clements, Rick Kazman: *Software Architecture in Practice, Second Edition*. Addison Wesley, Reading 5/9/2003 ISBN 0-321-15495-9 (This book, now in second edition, eloquently covers the fundamental concepts of the discipline. The theme is centered around achieving quality attributes of a system.)
- Amnon H. Eden, Rick Kazman. *Architecture, Design, Implementation.* On the distinction between architectural design and detailed design.
- Garzás, Javier, and Piattini, Mario. An ontology for micro-architectural design knowledge, IEEE Software Magazine, Volume: 22, Issue: 2, March-April 2005. pp. 28 – 33.
- Philippe Kruchten: *Architectural Blueprints - the 4+1 View Model of Software Architecture*. In: IEEE Software. 12 (6) November 1995, pp. 42–50 (also available online at the Rational website(PDF))
- Tony Shan and Winnie Hua (2006). *Solution Architecting Mechanism*. Proceedings of the 10th IEEE International EDOC Enterprise Computing Conference (EDOC 2006), October 2006, p23-32
- SOMF: Bell, Michael (2008). "Service-Oriented Modeling: Service Analysis, Design, and Architecture". Wiley. http://www.amazon.com/Service-Oriented-Modeling-Service-Analysis-Architecture/dp/0470141115/ref=pd_bbs_2.

External links

- Excellent explanation on IBM Developerworks
- Collection of software architecture definitions at Software Engineering Institute (SEI), Carnegie Mellon University (CMU)
- Software architecture vs. software design: The Intension/Locality Hypothesis
- Worldwide Institute of Software Architects (WWISA)
- International Association of Software Architects (IASA)
- SoftwareArchitecturePortal.org — website of IFIP Working Group 2.10 on Software Architecture
- Software Architecture — practical resources for Software Architects
- SoftwareArchitectures.com — independent resource of information on the discipline
- Microsoft Architecture Journal
- Architectural Patterns
- Software Architecture, chapter 1 of Roy Fielding's REST dissertation
- DiaSpec, an approach and tool to generate a distributed framework from a software architecture
- When Good Architecture Goes Bad
- Software Architecture and Related Concerns, What is Software Architecture? And What Software Architecture ***Is Not***
- Handbook of Software Architecture
- The Spiral Architecture Driven Development - the SDLC based on Spiral model is to reduce the risks of ineffective architecture
- Rationale focused software architecture documentation method

Design

Software design is a process of problem-solving and planning for a software solution. After the purpose and specifications of software are determined, software developers will design or employ designers to develop a plan for a solution. It includes low-level component and algorithm implementation issues as well as the architectural view.

Overview

The software requirements analysis (SRA) step of a software development process yields specifications that are used in software engineering. If the software is "semiautomated" or user centered, software design may involve user experience design yielding a story board to help determine those specifications. If the software is completely automated (meaning no user or user interface), a software design may be as simple as a flow chart or text describing a planned sequence of events. There are also semi-standard methods like Unified Modeling Language and Fundamental modeling concepts. In either case some documentation of the plan is usually the product of the design.

A software design may be platform-independent or platform-specific, depending on the availability of the technology called for by the design.

Software Design Topics

Design concepts

The design concepts provide the software designer with a foundation from which more sophisticated methods can be applied. A set of fundamental design concepts has evolved. They are:

1. Abstraction - Abstraction is the process or result of generalization by reducing the information content of a concept or an observable phenomenon, typically in order to retain only information which is relevant for a particular purpose.
2. Refinement - It is the process of elaboration. A hierarchy is developed by decomposing a macroscopic statement of function in a stepwise fashion until programming language statements are reached. In each step, one or several instructions of a given program are decomposed into more detailed instructions. Abstraction and Refinement are complementary concepts.
3. Modularity - Software architecture is divided into components called modules.
4. Software Architecture - It refers to the overall structure of the software and the ways in which that structure provides conceptual integrity for a system. A good software architecture will yield a good return on investment with respect to the desired outcome of the project, e.g. in terms of performance, quality, schedule and cost.
5. Control Hierarchy - A program structure that represent the organization of a program components and implies a hierarchy of control.
6. Structural Partitioning - The program structure can be divided both horizontally and vertically. Horizontal partitions define separate branches of modular hierarchy for each major program function. Vertical partitioning suggests that control and work should be distributed top down in the program structure.
7. Data Structure - It is a representation of the logical relationship among individual elements of data.
8. Software Procedure - It focuses on the processing of each modules individually
9. Information Hiding - Modules should be specified and designed so that information contained within a module is inaccessible to other modules that have no need for such information.

Design considerations

There are many aspects to consider in the design of a piece of software. The importance of each should reflect the goals the software is trying to achieve. Some of these aspects are:

- **Compatibility** - The software is able to operate with other products that are designed for interoperability with another product. For example, a piece of software may be backward-compatible with an older version of itself.
- **Extensibility** - New capabilities can be added to the software without major changes to the underlying architecture.
- **Fault-tolerance** - The software is resistant to and able to recover from component failure.
- **Maintainability** - The software can be restored to a specified condition within a specified period of time. For example, antivirus software may include the ability to periodically receive virus definition updates in order to maintain the software's effectiveness.

- **Modularity** - the resulting software comprises well defined, independent components. That leads to better maintainability. The components could be then implemented and tested in isolation before being integrated to form a desired software system. This allows division of work in a software development project.
- **Packaging** - Printed material such as the box and manuals should match the style designated for the target market and should enhance usability. All compatibility information should be visible on the outside of the package. All components required for use should be included in the package or specified as a requirement on the outside of the package.
- **Reliability** - The software is able to perform a required function under stated conditions for a specified period of time.
- **Reusability** - the software is able to add further features and modification with slight or no modification.
- **Robustness** - The software is able to operate under stress or tolerate unpredictable or invalid input. For example, it can be designed with a resilience to low memory conditions.
- **Security** - The software is able to withstand hostile acts and influences.
- **Usability** - The software user interface must be usable for its target user/audience. Default values for the parameters must be chosen so that they are a good choice for the majority of the users.

Modeling language

A modeling language is any artificial language that can be used to express information or knowledge or systems in a structure that is defined by a consistent set of rules. The rules are used for interpretation of the meaning of components in the structure. A modeling language can be graphical or textual. Examples of graphical modelling languages for software design are:

- Business Process Modeling Notation (BPMN) is an example of a Process Modeling language.
- EXPRESS and EXPRESS-G (ISO 10303-11) is an international standard general-purpose data modeling language.
- Extended Enterprise Modeling Language (EEML) is commonly used for business process modeling across a number of layers.
- Flowchart is a schematic representation of an algorithm or a stepwise process,
- Fundamental Modeling Concepts (FMC) modeling language for software-intensive systems.
- IDEF is a family of modeling languages, the most notable of which include IDEF0 for functional modeling, IDEF1X for information modeling, and IDEF5 for modeling ontologies.
- Jackson Structured Programming (JSP) is a method for structured programming based on correspondences between data stream structure and program structure
- LePUS3 is an object-oriented visual Design Description Language and a formal specification language that is suitable primarily for modelling large object-oriented (Java, C++, C#) programs and design patterns.
- Unified Modeling Language (UML) is a general modeling language to describe software both structurally and behaviorally. It has a graphical notation and allows for extension with a Profile (UML).
- Alloy (specification language) is a general purpose specification language for expressing complex structural constraints and behavior in a software system. It provides a concise language based on first-order relational logic.

- Systems Modeling Language (SysML) is a new general-purpose modeling language for systems engineering.

flexibility

Design patterns

A software designer or architect may identify a design problem which has been solved by others before. A template or pattern describing a solution to a common problem is known as a design pattern. The reuse of such patterns can speed up the software development process, having been tested and proved in the past.

Usage

Software design documentation may be reviewed or presented to allow constraints, specifications and even requirements to be adjusted prior to programming. Redesign may occur after review of a programmed simulation or prototype. It is possible to design software in the process of programming, without a plan or requirement analysis, but for more complex projects this would not be considered a professional approach. A separate design prior to programming allows for multidisciplinary designers and Subject Matter Experts (SMEs) to collaborate with highly-skilled programmers for software that is both useful and technically sound.

Design Patterns

In software engineering, a **design pattern** is a general reusable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Many patterns imply object-orientation or more generally mutable state, and so may not be as applicable in functional programming languages, in which data is immutable or treated as such.

Design patterns reside in the domain of modules and interconnections. At a higher level there are architectural patterns that are larger in scope, usually describing an overall pattern followed by an entire system.[15]

There are many types of design patterns including: structural design patterns, computational design patterns, algorithm strategy patterns, implementation strategy patterns and execution patterns. Structural patterns address concerns related to the high level structure of an application being developed . Computational patterns address concerns related to the identification of key computations. Algorithm strategy patterns address concerns related to high level strategies that describe how to exploit application characteristic on a computation platform. Implementation strategy patterns address concerns related to the realization of the source code to support (i) how the program itself is organized and (ii) the common data structures specific to parallel programming.

Execution patterns address concerns related to the support of the execution of an application, including the strategies in executing streams of tasks and building blocks to support the synchronization between tasks.

History

Patterns originated as an architectural concept by Christopher Alexander (1977/79). In 1987, Kent Beck and Ward Cunningham began experimenting with the idea of applying patterns to programming and presented their results at the OOPSLA conference that year.[16][17] In the following years, Beck, Cunningham and others followed up on this work.

Design patterns gained popularity in computer science after the book *Design Patterns: Elements of Reusable Object-Oriented Software* was published in 1994 by the so-called "Gang of Four" (Gamma et al.). That same year, the first Pattern Languages of Programming Conference was held and the following year, the Portland Pattern Repository was set up for documentation of design patterns. The scope of the term remains a matter of dispute. Notable books in the design pattern genre include:

- Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.
- Buschmann, Frank; Regine Meunier, Hans Rohnert, Peter Sommerlad (1996). *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons. ISBN 0-471-95869-7.
- Schmidt, Douglas C.; Michael Stal, Hans Rohnert, Frank Buschmann (2000). *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons. ISBN 0-471-60695-2.
- Fowler, Martin (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley. ISBN 978-0321127426.
- Hohpe, Gregor; Bobby Woolf (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley. ISBN 0-321-20068-3.
- Freeman, Eric T; Elisabeth Robson, Bert Bates, Kathy Sierra (2004). *Head First Design Patterns*. O'Reilly Media. ISBN 0-596-00712-4.

Although design patterns have been applied practically for a long time, formalization of the concept of design patterns languished for several years.[18]

In 2009 over 30 contributors collaborated with Thomas Erl on his book, *SOA Design Patterns* [19]. The goal of this book was to establish a de facto catalog of design patterns for SOA and service-orientation[20]. (Over 200+ IT professionals participated world-wide in reviewing Erl's book and patterns.) These patterns are also published and discussed on the community research site soapatterns.org

Practice

Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems, and it also improves code readability for coders and architects who are familiar with the patterns.

In order to achieve flexibility, design patterns usually introduce additional levels of indirection, which in some cases may complicate the resulting designs and hurt application performance.

By definition, a pattern must be programmed anew into each application that uses it. Since some authors see this as a step backward from software reuse as provided by components, researchers have worked to turn patterns into components. Meyer and Arnout were able to provide full or partial componentization of two-thirds of the patterns they attempted.[21]

Often, people only understand how to apply certain software design techniques to certain problems [*citation needed*]. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that does not require specifics tied to a particular problem.

Structure

Design patterns are composed of several sections (see Documentation below). Of particular interest are the Structure, Participants, and Collaboration sections. These sections describe a *design motif*: a prototypical *micro-architecture* that developers copy and adapt to their particular designs to solve the recurrent problem described by the design pattern. A micro-architecture is a set of program constituents (e.g., classes, methods...) and their relationships. Developers use the design pattern by introducing in their designs this prototypical micro-architecture, which means that micro-architectures in their designs will have structure and organization similar to the chosen design motif.

In addition to this, patterns allow developers to communicate using well-known, well understood names for software interactions. Common design patterns can be improved over time, making them more robust than *ad-hoc* designs.

Domain-specific patterns

Efforts have also been made to codify design patterns in particular domains, including use of existing design patterns as well as domain specific design patterns. Examples include user interface design patterns,[22] information visualization [23], secure design[24], "secure usability"[25], web design [26] and business model design.[27]

The annual Pattern Languages of Programming Conference proceedings [28] include many examples of domain specific patterns.

Classification and list

Design patterns were originally grouped into the categories: creational patterns, structural patterns, and behavioral patterns, and described using the concepts of delegation, aggregation, and consultation. For further background on object-oriented design, see coupling and cohesion, inheritance, interface, and polymorphism. Another classification has also introduced the notion of architectural design pattern that may be applied at the architecture level of the software such as the Model-View-Controller pattern.

Name	Description	In *Design Patterns*	In *Code Complete*[29]	Other
Creational patterns				
Abstract factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.	Yes	Yes	N/A
Builder	Separate the construction of a complex object from its representation allowing the same construction process to create various representations.	Yes	No	N/A
Factory method	Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.	Yes	Yes	N/A
Lazy initialization	Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed.	No	No	*PoEAA*[30]
Multiton	Ensure a class has only named instances, and provide global point of access to them.	No	No	N/A
Object pool	Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalisation of connection pool and thread pool patterns.	No	No	N/A
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.	Yes	No	N/A
Resource	Ensure that resources are properly released by	No	No	N/A

acquisition is initialization	tying them to the lifespan of suitable objects.			
Singleton	Ensure a class has only one instance, and provide a global point of access to it.	Yes	Yes	N/A
Structural patterns				
Adapter or Wrapper	Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.	Yes	Yes	N/A
Bridge	Decouple an abstraction from its implementation allowing the two to vary independently.	Yes	Yes	N/A
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.	Yes	Yes	N/A
Decorator	Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.	Yes	Yes	N/A
Facade	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.	Yes	Yes	N/A
Front Controller	Provide a unified interface to a set of interfaces in a subsystem. Front Controller defines a higher-level interface that makes the subsystem easier to use.	No	Yes	N/A
Flyweight	Use sharing to support large numbers of fine-grained objects efficiently.	Yes	No	N/A
Proxy	Provide a surrogate or placeholder for another object to control access to it.	Yes	No	N/A
Behavioral patterns				
Blackboard	Generalized observer, which allows multiple readers and writers. Communicates	No	No	N/A

	information system-wide.			
Chain of responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.	Yes	No	N/A
Command	Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.	Yes	No	N/A
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.	Yes	No	N/A
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.	Yes	Yes	N/A
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.	Yes	No	N/A
Memento	Without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later.	Yes	No	N/A
Null object	Avoid null references by providing a default object.	No	No	N/A
Observer or Publish/subscribe	Define a one-to-many dependency between objects where a state change in one object results with all its dependents being notified and updated automatically.	Yes	Yes	N/A
Servant	Define common functionality for a group of classes	No	No	N/A
Specification	Recombinable business logic in a boolean fashion	No	No	N/A
State	Allow an object to alter its behavior when its internal state changes. The object will appear	Yes	No	N/A

	to change its class.			
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.	Yes	Yes	N/A
Template method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.	Yes	Yes	N/A
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.	Yes	No	N/A

Name	Description	In *POSA2*[31]	Other
Concurrency patterns			
Active Object	Decouples method execution from method invocation that reside in their own thread of control. The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests.	Yes	N/A
Balking	Only execute an action on an object when the object is in a particular state.	No	N/A
Binding Properties	Combining multiple observers to force properties in different objects to be synchronized or coordinated in some way.[32]	No	N/A
Messaging pattern	The messaging design pattern (MDP) allows the interchange of information (i.e. messages) between components and applications.	No	N/A
Double-checked locking	Reduce the overhead of acquiring a lock by first testing the locking criterion (the 'lock hint') in an unsafe manner; only if that succeeds does the actual lock proceed. Can be unsafe when implemented in some language/hardware combinations. It can therefore sometimes be considered an anti-pattern.	Yes	N/A

Event-based asynchronous	Addresses problems with the Asynchronous pattern that occur in multithreaded programs.[33]	No	N/A
Guarded suspension	Manages operations that require both a lock to be acquired and a precondition to be satisfied before the operation can be executed.	No	N/A
Lock	One thread puts a "lock" on a resource, preventing other threads from accessing or modifying it.[34]	No	*PoEAA*[30]
Monitor object	An object whose methods are subject to mutual exclusion, thus preventing multiple objects from erroneously trying to use it at the same time.	Yes	N/A
Reactor	A reactor object provides an asynchronous interface to resources that must be handled synchronously.	Yes	N/A
Read-write lock	Allows concurrent read access to an object but requires exclusive access for write operations.	No	N/A
Scheduler	Explicitly control when threads may execute single-threaded code.	No	N/A
Thread pool	A number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads. Can be considered a special case of the object pool pattern.	No	N/A
Thread-specific storage	Static or "global" memory local to a thread.	Yes	N/A

Documentation

The documentation for a design pattern describes the context in which the pattern is used, the forces within the context that the pattern seeks to resolve, and the suggested solution.[35] There is no single, standard format for documenting design patterns. Rather, a variety of different formats have been used by different pattern authors. However, according to Martin Fowler certain pattern forms have become more well-known than others, and consequently become common starting points for new pattern writing efforts.[36] One example of a commonly used documentation format is the one used by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides (collectively known as the "Gang of Four", or GoF for short) in their book *Design Patterns*. It contains the following sections:

- **Pattern Name and Classification:** A descriptive and unique name that helps in identifying and referring to the pattern.
- **Intent:** A description of the goal behind the pattern and the reason for using it.
- **Also Known As:** Other names for the pattern.

- **Motivation (Forces):** A scenario consisting of a problem and a context in which this pattern can be used.
- **Applicability:** Situations in which this pattern is usable; the context for the pattern.
- **Structure:** A graphical representation of the pattern. Class diagrams and Interaction diagrams may be used for this purpose.
- **Participants:** A listing of the classes and objects used in the pattern and their roles in the design.
- **Collaboration:** A description of how classes and objects used in the pattern interact with each other.
- **Consequences:** A description of the results, side effects, and trade offs caused by using the pattern.
- **Implementation:** A description of an implementation of the pattern; the solution part of the pattern.
- **Sample Code:** An illustration of how the pattern can be used in a programming language.
- **Known Uses:** Examples of real usages of the pattern.
- **Related Patterns:** Other patterns that have some relationship with the pattern; discussion of the differences between the pattern and similar patterns.

References

1. ↑ Stellman, Andrew; Greene, Jennifer (2005). *Applied Software Project Management*. O'Reilly Media. ISBN 978-0-596-00948-9. http://www.stellman-greene.com/aspm/.
2. ↑ "Requirements management". UK Office of Government Commerce. http://www.ogc.gov.uk/delivery_lifecycle_requirements_management.asp. Retrieved 2009-11-10.
3. ↑ *A Guide to the Project Management Body of Knowledge* (4th ed.). Project Management Institute. 2008. ISBN 978-1-933-89051-7. http://www.pmi.org/.
4. ↑ Gotel, O., Finkelstein, A. An Analysis of the Requirements Traceability Problem Proc. of First International Conference on Requirements Engineering, 1994, pages 94-101
5. ↑ "Requirements Management Tools Survey". International Council on Systems Engineering. http://www.incose.org/ProductsPubs/products/rmsurvey.aspx. Retrieved 2009-11-10.
6. ↑ ***a*** ***b*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
7. ↑ ***a*** ***b*** ***c*** Clements, Paul; Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord, Judith Stafford (2010). *Documenting Software Architectures: Views and Beyond, Second Edition*. Boston: Addison-Wesley. ISBN 0321552687.
8. ↑ Bass, Len; Paul Clements, Rick Kazman (2003). *Software Architecture In Practice, Second Edition*. Boston: Addison-Wesley. pp. 21–24. ISBN 0-321-15495-9.
9. ↑ University of Waterloo (2006). "A Very Brief History of Computer Science". http://www.cs.uwaterloo.ca/~shallit/Courses/134/history.html. Retrieved 2006-09-23.
10. ↑ IEEE Transactions on Software Engineering (2006). "Introduction to the Special Issue on Software Architecture". http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/trans/ts/&toc=comp/trans/ts/1995/04/e4toc.xml&DOI=10.1109/TSE.1995.10003. Retrieved 2006-09-23.
11. ↑ SEI (2006). "How do you define Software Architecture?". http://www.sei.cmu.edu/architecture/start/definitions.cfm. Retrieved 2006-09-23.

12.↑ ***a*** ***b*** SoftwareArchitectures.com (2006). "Intro to Software Quality Attributes". http://www.softwarearchitectures.com/one/Designing+Architecture/78.aspx. Retrieved 2006-09-23.
13.↑ Garlan & Shaw (1994). "An Introduction to Software Architecture". http://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf. Retrieved 2006-09-25.
14.↑ ***a*** ***b*** Amnon H. Eden, Rick Kazman (2003). "Architecture Design Implementation". http://www.eden-study.org/articles/2003/icse03.pdf.
15.↑ Martin, Robert C.. "Design Principles and Design Patterns". http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf. Retrieved 2000.
16.↑ Smith, Reid (October 1987). "Panel on design methodology". *OOPSLA '87 Addendum to the Proceedings*. OOPSLA '87. doi:10.1145/62138.62151. , *"Ward cautioned against requiring too much programming at, what he termed, 'the high level of wizards.' He pointed out that a written 'pattern language' can significantly improve the selection and application of abstractions. He proposed a 'radical shift in the burden of design and implementation' basing the new methodology on an adaptation of Christopher Alexander's work in pattern languages and that programming-oriented pattern languages developed at Tektronix has significantly aided their software development efforts."*
17.↑ Beck, Kent; Ward Cunningham (September 1987). "Using Pattern Languages for Object-Oriented Program". *OOPSLA '87 workshop on* Specification and Design for Object-Oriented Programming. OOPSLA '87. http://c2.com/doc/oopsla87.html. Retrieved 2006-05-26.
18.↑ Baroni, Aline Lúcia; Yann-Gaël Guéhéneuc and Hervé Albin-Amiot (June 2003). "Design Patterns Formalization" (PDF). Nantes: École Nationale Supérieure des Techniques Industrielles et des Mines de Nantes. http://www.iro.umontreal.ca/~ptidej/Publications/Documents/Research+report+Metamodeling+June03.doc.pdf. Retrieved 2007-12-29.
19.↑ Erl, Thomas (2009). *SOA Design Patterns*. New York: Prentice Hall/PearsonPTR. pp. 864. ISBN 0-13-613516-1.
20.↑ http://soa.sys-con.com/node/809800
21.↑ Meyer, Bertrand; Karine Arnout (July 2006). "Componentization: The Visitor Example". *IEEE Computer* (IEEE) **39** (7): 23–30. http://se.ethz.ch/~meyer/publications/computer/visitor.pdf.
22.↑ Laakso, Sari A. (2003-09-16). "Collection of User Interface Design Patterns". University of Helsinki, Dept. of Computer Science. http://www.cs.helsinki.fi/u/salaakso/patterns/index.html. Retrieved 2008-01-31.
23.↑ Heer, J.; M. Agrawala (2006). "Software Design Patterns for Information Visualization". *IEEE Transactions on Visualization and Computer Graphics* **12** (5): 853. doi:10.1109/TVCG.2006.178. PMID 17080809. http://vis.berkeley.edu/papers/infovis_design_patterns/.
24.↑ Chad Dougherty et al (2009). *Secure Design Patterns*. http://www.cert.org/archive/pdf/09tr010.pdf.
25.↑ Simson L. Garfinkel (2005). *Design Principles and Patterns for Computer Systems That Are Simultaneously Secure and Usable*. http://www.simson.net/thesis/.
26.↑ "Yahoo! Design Pattern Library". http://developer.yahoo.com/ypatterns/. Retrieved 2008-01-31.
27.↑ "How to design your Business Model as a Lean Startup?". http://torgronsund.wordpress.com/2010/01/06/lean-startup-business-model-pattern/. Retrieved 2010-01-06.
28.↑ Pattern Languages of Programming, Conference proceedings (annual, 1994—) [3]

29.↑ McConnell, Steve (June 2004). "Design in Construction". *Code Complete* (2nd ed.). Microsoft Press. pp. 104. ISBN 978-0735619678. "Table 5.1 Popular Design Patterns"
30.↑ **a** **b** Fowler, Martin (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley. ISBN 978-0321127426. http://martinfowler.com/.
31.↑ Schmidt, Douglas C.; Michael Stal, Hans Rohnert, Frank Buschmann (2000). *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons. ISBN 0-471-60695-2.
32.↑ http://c2.com/cgi/wiki?BindingProperties
33.↑ Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson, and Morgan Skinner (2008). "Event-based Asynchronous Pattern". *Professional C# 2008*. Wiley. pp. 570–571. ISBN 0470191376.
34.↑ http://c2.com/cgi/wiki?LockPattern
35.↑ Gabriel, Dick. "A Pattern Definition". Archived from the original on 2007-02-09. http://web.archive.org/web/20070209224120/http://hillside.net/patterns/definition.html. Retrieved 2007-03-06.
36.↑ Fowler, Martin (2006-08-01). "Writing Software Patterns". http://www.martinfowler.com/articles/writingPatterns.html. Retrieved 2007-03-06.

Further reading

Books

- Alexander, Christopher; Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel (1977). *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press. ISBN 978-0195019193.
- Alur, Deepak; John Crupi, Dan Malks (May 2003). *Core J2EE Patterns: Best Practices and Design Strategies (2nd Edition)*. Prentice Hall. ISBN 0131422464.
- Beck, Kent (October 2007). *Implementation Patterns*. Addison-Wesley. ISBN 978-0321413093.
- Beck, Kent; R. Crocker, G. Meszaros, J.O. Coplien, L. Dominick, F. Paulisch, and J. Vlissides (March 1996). *Proceedings of the 18th International Conference on Software Engineering*. pp. 25–30.
- Borchers, Jan (2001). *A Pattern Approach to Interaction Design*. John Wiley & Sons. ISBN 0-471-49828-9.
- Coplien, James O.; Douglas C. Schmidt (1995). *Pattern Languages of Program Design*. Addison-Wesley. ISBN 0-201-60734-4.
- Coplien, James O.; John M. Vlissides, and Norman L. Kerth (1996). *Pattern Languages of Program Design 2*. Addison-Wesley. ISBN 0-201-89527-7.
- Fowler, Martin (1997). *Analysis Patterns: Reusable Object Models*. Addison-Wesley. ISBN 0-201-89542-0.
- Fowler, Martin (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley. ISBN 978-0321127426.
- Freeman, Eric; Elisabeth Freeman, Kathy Sierra, and Bert Bates (2004). *Head First Design Patterns*. O'Reilly Media. ISBN 0-596-00712-4.
- Hohmann, Luke; Martin Fowler and Guy Kawasaki (2003). *Beyond Software Architecture*. Addison-Wesley. ISBN 0-201-77594-8.
- Alur, Deepak; Elisabeth Freeman, Kathy Sierra, and Bert Bates (2004). *Head First Design Patterns*. O'Reilly Media. ISBN 0-596-00712-4.

- Gabriel, Richard (1996) (PDF). *Patterns of Software: Tales From The Software Community*. Oxford University Press. pp. 235. ISBN 0-19-512123-6. http://www.dreamsongs.com/NewFiles/PatternsOfSoftware.pdf.
- Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.
- Hohpe, Gregor; Bobby Woolf (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley. ISBN 0-321-20068-3.
- Holub, Allen (2004). *Holub on Patterns*. Apress. ISBN 1-59059-388-X.
- Kircher, Michael; Markus Völter and Uwe Zdun (2005). *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*. John Wiley & Sons. ISBN 0-470-85662-9.
- Larman, Craig (2005). *Applying UML and Patterns*. Prentice Hall. ISBN 0-13-148906-2.
- Liskov, Barbara; John Guttag (2000). *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.*. Addison-Wesley. ISBN 0-201-65768-6.
- Manolescu, Dragos; Markus Voelter and James Noble (2006). *Pattern Languages of Program Design 5*. Addison-Wesley. ISBN 0-321-32194-4.
- Marinescu, Floyd (2002). *EJB Design Patterns: Advanced Patterns, Processes and Idioms*. John Wiley & Sons. ISBN 0-471-20831-0.
- Martin, Robert Cecil; Dirk Riehle and Frank Buschmann (1997). *Pattern Languages of Program Design 3*. Addison-Wesley. ISBN 0-201-31011-2.
- Mattson, Timothy G; Beverly A. Sanders and Berna L. Massingill (2005). *Patterns for Parallel Programming*. Addison-Wesley. ISBN 0-321-22811-1.
- Shalloway, Alan; James R. Trott (2001). *Design Patterns Explained, Second Edition: A New Perspective on Object-Oriented Design*. Addison-Wesley. ISBN 0-321-24714-0.
- Vlissides, John M. (1998). *Pattern Hatching: Design Patterns Applied*. Addison-Wesley. ISBN 0-201-43293-5.
- Weir, Charles; James Noble (2000). *Small Memory Software: Patterns for systems with limited memory*. Addison-Wesley. ISBN 0201596075. http://www.cix.co.uk/~smallmemory/.

Web sites

- "History of Patterns". *Portland Pattern Repository*. http://www.c2.com/cgi-bin/wiki?HistoryOfPatterns. Retrieved 2005-07-28.
- "Are Design Patterns Missing Language Features?". Cunningham & Cunningham, Inc.. http://www.c2.com/cgi/wiki?AreDesignPatternsMissingLanguageFeatures. Retrieved 2006-01-20.
- "Show Trial of the Gang of Four". Cunningham & Cunningham, Inc.. http://www.c2.com/cgi/wiki?ShowTrialOfTheGangOfFour. Retrieved 2006-01-20.
- "Design Patterns in Modern Day Software Factories (WCSF)". XO Software, Ltd. http://www.xosoftware.co.uk/Articles/WCSFDesignPatterns/. Retrieved 2010-02-22.

External links

- Directory of websites that provide pattern catalogs at hillside.net.
- Ward Cunningham's *Portland Pattern Repository*.

- Messaging Design Pattern Published in the 17th conference on Pattern Languages of Programs (PLoP 2010).
- Patterns and Anti-Patterns at the Open Directory Project
- PerfectJPattern Open Source Project Design Patterns library that aims to provide full or partial componentized version of all known Patterns in Java.
- Lean Startup Business Model Pattern Example of design pattern thinking applied to business models
- Jt J2EE Pattern Oriented Framework
- Printable Design Patterns Quick Reference Cards
- 101 Design Patterns & Tips for Developers
- On Patterns and Pattern Languages by Buschmann, Henney, and Schmidt
- Patterns for Scripted Applications
- Design Patterns Reference at oodesign.com
- Design Patterns for 70% programmers in the world by Saurabh Verma at slideshare.com

Anti-Patterns

In software engineering, an **anti-pattern** (or **antipattern**) is a pattern that may be commonly used but is ineffective and/or counterproductive in practice.[1][2]

The term was coined in 1995 by Andrew Koenig,[3] inspired by Gang of Four's book *Design Patterns*, which developed the concept of design patterns in the software field. The term was widely popularized three years later by the book *AntiPatterns*, which extended the use of the term beyond the field of software design and into general social interaction. According to the authors of the latter, there must be at least two key elements present to formally distinguish an actual anti-pattern from a simple bad habit, bad practice, or bad idea:

- Some repeated pattern of action, process or structure that initially appears to be beneficial, but ultimately produces more bad consequences than beneficial results, and
- A refactored solution exists that is clearly documented, proven in actual practice and repeatable.

Often pejoratively named with clever oxymoronic neologisms, many anti-pattern ideas amount to little more than mistakes, rants, unsolvable problems, or bad practices to be avoided if possible. Sometimes called *pitfalls* or *dark patterns*, this informal use of the term has come to refer to classes of commonly reinvented bad solutions to problems. Thus, many candidate anti-patterns under debate would not be formally considered anti-patterns.

By formally describing repeated mistakes, one can recognize the forces that lead to their repetition and learn how others have refactored themselves out of these broken patterns.

Known anti-patterns

Organizational anti-patterns

- Analysis paralysis: Devoting disproportionate effort to the analysis phase of a project
- Cash cow: A profitable legacy product that often leads to complacency about new products
- Design by committee: The result of having many contributors to a design, but no unifying vision
- Escalation of commitment: Failing to revoke a decision when it proves wrong
- Management by perkele: Authoritarian style of management with no tolerance of dissent
- Matrix Management: Unfocused organizational structure that results in divided loyalties and lack of direction
- Moral hazard: Insulating a decision-maker from the consequences of his or her decision
- Mushroom management: Keeping employees uninformed and misinformed (kept in the dark and fed manure)
- Stovepipe or Silos: A structure that supports mostly up-down flow of data but inhibits cross organizational communication
- Vendor lock-in: Making a system excessively dependent on an externally supplied component[4]

Project management anti-patterns

- Death march: Everyone knows that the project is going to be a disaster – except the CEO. However, the truth remains hidden and the project is artificially kept alive until the Day Zero finally comes ("Big Bang"). Alternative definition: Employees are pressured to work late nights and weekends on a project with an unreasonable deadline.
- Groupthink: During groupthink, members of the group avoid promoting viewpoints outside the comfort zone of consensus thinking
- Smoke and mirrors: Demonstrating how unimplemented functions will appear
- Software bloat: Allowing successive versions of a system to demand ever more resources
- Waterfall model: An older method of software development that inadequately deals with unanticipated change

Analysis anti-patterns

- Bystander apathy: When a requirement or design decision is wrong, but the people who notice this do nothing because it affects a larger number of people

Software design anti-patterns

- Abstraction inversion: Not exposing implemented functionality required by users, so that they re-implement it using higher level functions
- Ambiguous viewpoint: Presenting a model (usually Object-oriented analysis and design (OOAD)) without specifying its viewpoint

- Big ball of mud: A system with no recognizable structure
- Database-as-IPC: Using a database as the message queue for routine interprocess communication where a much more lightweight mechanism would be suitable
- Gold plating: Continuing to work on a task or project well past the point at which extra effort is adding value
- Inner-platform effect: A system so customizable as to become a poor replica of the software development platform
- Input kludge: Failing to specify and implement the handling of possibly invalid input
- Interface bloat: Making an interface so powerful that it is extremely difficult to implement
- Magic pushbutton: Coding implementation logic directly within interface code, without using abstraction
- Race hazard: Failing to see the consequence of different orders of events
- Stovepipe system: A barely maintainable assemblage of ill-related components

Object-oriented design anti-patterns

- Anemic Domain Model: The use of domain model without any business logic. The domain model's objects cannot guarantee their correctness at any moment, because their validation and mutation logic is placed somewhere outside (most likely in multiple places).
- BaseBean: Inheriting functionality from a utility class rather than delegating to it
- Call super: Requiring subclasses to call a superclass's overridden method
- Circle-ellipse problem: Subtyping variable-types on the basis of value-subtypes
- Circular dependency: Introducing unnecessary direct or indirect mutual dependencies between objects or software modules
- Constant interface: Using interfaces to define constants
- God object: Concentrating too many functions in a single part of the design (class)
- Object cesspool: Reusing objects whose state does not conform to the (possibly implicit) contract for re-use
- Object orgy: Failing to properly encapsulate objects permitting unrestricted access to their internals
- Poltergeists: Objects whose sole purpose is to pass information to another object
- Sequential coupling: A class that requires its methods to be called in a particular order
- Yo-yo problem: A structure (e.g., of inheritance) that is hard to understand due to excessive fragmentation

Programming anti-patterns

- Accidental complexity: Introducing unnecessary complexity into a solution
- Action at a distance: Unexpected interaction between widely separated parts of a system
- Blind faith: Lack of checking of (a) the correctness of a bug fix or (b) the result of a subroutine
- Boat anchor: Retaining a part of a system that no longer has any use
- Busy spin: Consuming CPU while waiting for something to happen, usually by repeated checking instead of messaging
- Caching failure: Forgetting to reset an error flag when an error has been corrected
- Cargo cult programming: Using patterns and methods without understanding why

- Coding by exception: Adding new code to handle each special case as it is recognized
- Error hiding: Catching an error message before it can be shown to the user and either showing nothing or showing a meaningless message
- Hard code: Embedding assumptions about the environment of a system in its implementation
- Lava flow: Retaining undesirable (redundant or low-quality) code because removing it is too expensive or has unpredictable consequences[5][6]
- Loop-switch sequence: Encoding a set of sequential steps using a switch within a loop statement
- Magic numbers: Including unexplained numbers in algorithms
- Magic strings: Including literal strings in code, for comparisons, as event types etc.
- Soft code: Storing business logic in configuration files rather than source code[7]
- Spaghetti code: Programs whose structure is barely comprehensible, especially because of misuse of code structures

Methodological anti-patterns

- Copy and paste programming: Copying (and modifying) existing code rather than creating generic solutions
- Golden hammer: Assuming that a favorite solution is universally applicable (See: Silver Bullet)
- Improbability factor: Assuming that it is improbable that a known error will occur
- Not Invented Here (NIH) syndrome: The tendency towards *reinventing the wheel* (Failing to adopt an existing, adequate solution)
- Premature optimization: Coding early-on for perceived efficiency, sacrificing good design, maintainability, and sometimes even real-world efficiency
- Programming by permutation (or "programming by accident"): Trying to approach a solution by successively modifying the code to see if it works
- Reinventing the wheel: Failing to adopt an existing, adequate solution
- Reinventing the square wheel: Failing to adopt an existing solution and instead adopting a custom solution which performs much worse than the existing one
- Silver bullet: Assuming that a favorite technical solution can solve a larger process or problem
- Tester Driven Development: Software projects in which new requirements are specified in bug reports

Configuration management anti-patterns

- Dependency hell: Problems with versions of required products
- DLL hell: Inadequate management of dynamic-link libraries (DLLs), specifically on Microsoft Windows
- Extension conflict: Problems with different extensions to pre-Mac OS X versions of the Mac OS attempting to patch the same parts of the operating system
- JAR hell: Overutilization of the multiple JAR files, usually causing versioning and location problems because of misunderstanding of the Java class loading model

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

Further reading

1. Laplante, Phillip A.; Colin J. Neill (2005). *Antipatterns: Identification, Refactoring and Management*. Auerbach Publications. ISBN 0-8493-2994-9.
2. Brown, William J.; Raphael C. Malveau, Hays W. "Skip" McCormick, Scott W. Thomas, Theresa Hudson (ed). (2000). *Anti-Patterns in Project Management*. John Wiley & Sons, ltd. ISBN 0-471-36366-9.

External links

- Anti-pattern at WikiWikiWeb
- Anti-patterns catalog
- AntiPatterns.com Web site for the AntiPatterns book
- Patterns of Toxic Behavior

Implementation

Introduction

Computer programming (often shortened to **programming** or **coding**) is the process of designing, writing, testing, debugging / troubleshooting, and maintaining the source code of computer programs. This source code is written in a programming language. The purpose of programming is to create a program that exhibits a certain desired behaviour. The process of writing source code often requires expertise in many different subjects, including knowledge of the application domain, specialized algorithms and formal logic.

Definition

Hoc and Nguyen-Xuan define computer programming as "the process of transforming a mental plan in familiar terms into one compatible with the computer." [8] Said another way, programming is the craft of transforming requirements into something that a computer can execute.

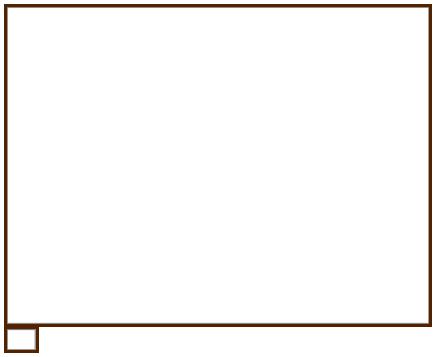
Overview

Within software engineering, programming (the *implementation*) is regarded as one phase in a software development process.

There is an ongoing debate on the extent to which the writing of programs is an art, a craft or an engineering discipline.[9] In general, good programming is considered to be the measured application of all three, with the goal of producing an efficient and evolvable software solution (the criteria for "efficient" and "evolvable" vary considerably). The discipline differs from many other technical professions in that programmers, in general, do not need to be licensed or pass any standardized (or governmentally regulated) certification tests in order to call themselves "programmers" or even "software engineers." However, representing oneself as a "Professional Software Engineer" without a license from an accredited institution is illegal in many parts of the world. However, because the discipline covers many areas, which may or may not include critical applications, it is debatable whether licensing is required for the profession as a whole. In most cases, the discipline is self-governed by the entities which require the programming, and sometimes very strict environments are defined (e.g. United States Air Force use of AdaCore and security clearance).

Another ongoing debate is the extent to which the programming language used in writing computer programs affects the form that the final program takes. This debate is analogous to that surrounding the Sapir–Whorf hypothesis [10] in linguistics, which postulates that a particular spoken language's nature influences the habitual thought of its speakers. Different language patterns yield different patterns of thought. This idea challenges the possibility of representing the world perfectly with language, because it acknowledges that the mechanisms of any language condition the thoughts of its speaker community.

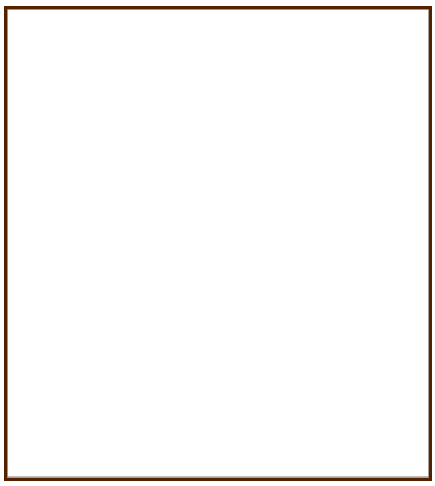
History



Wired plug board for an IBM 402 Accounting Machine.

The Antikythera mechanism from ancient Greece was a calculator utilizing gears of various sizes and configuration to determine its operation,[11] which tracked the metonic cycle still used in lunar-to-solar calendars, and which is consistent for calculating the dates of the Olympiads.[12] Al-Jazari built programmable Automata in 1206. One system employed in these devices was the use of pegs and cams placed into a wooden drum at specific locations. which would sequentially trigger levers that in turn operated percussion instruments. The output of this device was a small drummer playing various rhythms and drum patterns.[13][14] The Jacquard Loom, which Joseph Marie Jacquard developed in 1801, uses a series of pasteboard cards with holes punched in them. The hole pattern represented the pattern that the loom had to follow in weaving cloth. The loom could produce entirely different weaves using different sets of cards. Charles Babbage adopted the use of punched cards around 1830 to control his Analytical Engine. The synthesis of numerical calculation, predetermined operation and output, along with a way to organize and input instructions in a manner relatively easy for humans to conceive and produce, led to the modern development of computer programming. Development of computer programming accelerated through the Industrial Revolution.

In the late 1880s, Herman Hollerith invented the recording of data on a medium that could then be read by a machine. Prior uses of machine readable media, above, had been for control, not data. "After some initial trials with paper tape, he settled on punched cards..."[15] To process these punched cards, first known as "Hollerith cards" he invented the tabulator, and the keypunch machines. These three inventions were the foundation of the modern information processing industry. In 1896 he founded the *Tabulating Machine Company* (which later became the core of IBM). The addition of a control panel (plugboard) to his 1906 Type I Tabulator allowed it to do different jobs without having to be physically rebuilt. By the late 1940s, there were a variety of plug-board programmable machines, called unit record equipment, to perform data-processing tasks (card reading). Early computer programmers used plug-boards for the variety of complex calculations requested of the newly invented machines.



Data and instructions could be stored on external punched cards, which were kept in order and arranged in program decks.

The invention of the von Neumann architecture allowed computer programs to be stored in computer memory. Early programs had to be painstakingly crafted using the instructions (elementary operations) of the particular machine, often in binary notation. Every model of computer would likely use different instructions (machine language) to do the same task. Later, assembly languages were developed that let the programmer specify each instruction in a text format, entering abbreviations for each operation code instead of a number and specifying addresses in symbolic form (e.g., ADD X, TOTAL). Entering a program in assembly language is usually more convenient, faster, and less prone to human error than using machine language, but because an assembly language is little more than a different notation for a machine language, any two machines with different instruction sets also have different assembly languages.

In 1954, FORTRAN was invented; it was the first high level programming language to have a functional implementation, as opposed to just a design on paper.[16][17] (A high-level language is, in very general terms, any programming language that allows the programmer to write programs in terms that are more abstract than assembly language instructions, i.e. at a level of abstraction "higher" than that of an assembly language.) It allowed programmers to specify calculations by entering a formula directly (e.g. Y = X*2 + 5*X + 9). The program text, or *source*, is converted into machine instructions using a special program called a compiler, which translates the FORTRAN program into machine language. In fact, the name FORTRAN stands for "Formula Translation". Many other languages were developed, including some for commercial programming, such as COBOL. Programs were mostly still entered using punched cards or paper tape. (See computer programming in the punch card era). By the late 1960s, data storage devices and computer terminals became inexpensive enough that programs could be created by typing directly into the computers. Text editors were developed that allowed changes and corrections to be made much more easily than with punched cards. (Usually, an error in punching a card meant that the card had to be discarded and a new one punched to replace it.)

As time has progressed, computers have made giant leaps in the area of processing power. This has brought about newer programming languages that are more abstracted from the underlying hardware. Although these high-level languages usually incur greater overhead, the increase in speed of modern computers has made the use of these languages much more practical than in the past. These increasingly abstracted languages typically are easier to learn and allow the programmer to

develop applications much more efficiently and with less source code. However, high-level languages are still impractical for a few programs, such as those where low-level hardware control is necessary or where maximum processing speed is vital.

Throughout the second half of the twentieth century, programming was an attractive career in most developed countries. Some forms of programming have been increasingly subject to offshore outsourcing (importing software and services from other countries, usually at a lower wage), making programming career decisions in developed countries more complicated, while increasing economic opportunities in less developed areas. It is unclear how far this trend will continue and how deeply it will impact programmer wages and opportunities. [*citation needed*]

Modern programming

Quality requirements

Whatever the approach to software development may be, the final program must satisfy some fundamental properties. The following properties are among the most relevant:

- **Efficiency**/**performance**: the amount of system resources a program consumes (processor time, memory space, slow devices such as disks, network bandwidth and to some extent even user interaction): the less, the better. This also includes correct disposal of some resources, such as cleaning up temporary files and lack of memory leaks.
- **Reliability**: how often the results of a program are correct. This depends on conceptual correctness of algorithms, and minimization of programming mistakes, such as mistakes in resource management (e.g., buffer overflows and race conditions) and logic errors (such as division by zero or off-by-one errors).
- **Robustness**: how well a program anticipates problems not due to programmer error. This includes situations such as incorrect, inappropriate or corrupt data, unavailability of needed resources such as memory, operating system services and network connections, and user error.
- **Usability**: the ergonomics of a program: the ease with which a person can use the program for its intended purpose, or in some cases even unanticipated purposes. Such issues can make or break its success even regardless of other issues. This involves a wide range of textual, graphical and sometimes hardware elements that improve the clarity, intuitiveness, cohesiveness and completeness of a program's user interface.
- **Portability**: the range of computer hardware and operating system platforms on which the source code of a program can be compiled/interpreted and run. This depends on differences in the programming facilities provided by the different platforms, including hardware and operating system resources, expected behaviour of the hardware and operating system, and availability of platform specific compilers (and sometimes libraries) for the language of the source code
- **Maintainability**: the ease with which a program can be modified by its present or future developers in order to make improvements or customizations, fix bugs and security holes, or adapt it to new environments. Good practices during initial development make the difference in this regard. This quality may not be directly apparent to the end user but it can significantly affect the fate of a program over the long term.

Algorithmic complexity

The academic field and the engineering practice of computer programming are both largely concerned with discovering and implementing the most efficient algorithms for a given class of problem. For this purpose, algorithms are classified into *orders* using so-called Big O notation, *O(n)*, which expresses resource use, such as execution time or memory consumption, in terms of the size of an input. Expert programmers are familiar with a variety of well-established algorithms and their respective complexities and use this knowledge to choose algorithms that are best suited to the circumstances.

Methodologies

The first step in most formal software development projects is requirements analysis, followed by testing to determine value modeling, implementation, and failure elimination (debugging). There exist a lot of differing approaches for each of those tasks. One approach popular for requirements analysis is Use Case analysis. Nowadays many programmers use forms of Agile software development where the various stages of formal software development are more integrated together into short cycles that take a few weeks rather than years. There are many approaches to the Software development process

Popular modeling techniques include Object-Oriented Analysis and Design (OOAD) and Model-Driven Architecture (MDA). The Unified Modeling Language (UML) is a notation used for both the OOAD and MDA.

A similar technique used for database design is Entity-Relationship Modeling (ER Modeling).

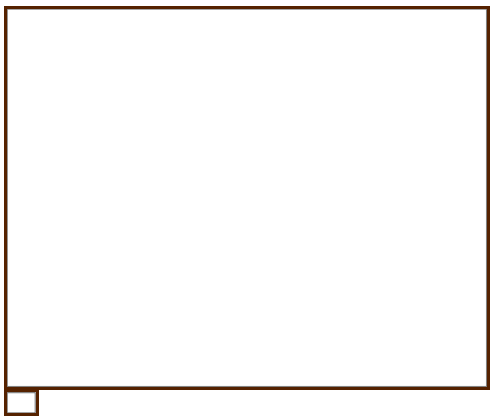
Implementation techniques include imperative languages (object-oriented or procedural), functional languages, and logic languages.

Measuring language usage

It is very difficult to determine what are the most popular of modern programming languages. Some languages are very popular for particular kinds of applications (e.g., COBOL is still strong in the corporate data center, often on large mainframes, FORTRAN in engineering applications, scripting languages in web development, and C in embedded applications), while some languages are regularly used to write many different kinds of applications.

Methods of measuring programming language popularity include: counting the number of job advertisements that mention the language,[18] the number of books teaching the language that are sold (this overestimates the importance of newer languages), and estimates of the number of existing lines of code written in the language (this underestimates the number of users of business languages such as COBOL).

Debugging



A bug, which was debugged in 1947.

Debugging is a very important task in the software development process, because an incorrect program can have significant consequences for its users. Some languages are more prone to some kinds of faults because their specification does not require compilers to perform as much checking as other languages. Use of a static analysis tool can help detect some possible problems.

Debugging is often done with IDEs like Eclipse, Kdevelop, NetBeans, and Visual Studio. Standalone debuggers like gdb are also used, and these often provide less of a visual environment, usually using a command line.

Programming languages

Different programming languages support different styles of programming (called *programming paradigms*). The choice of language used is subject to many considerations, such as company policy, suitability to task, availability of third-party packages, or individual preference. Ideally, the programming language best suited for the task at hand will be selected. Trade-offs from this ideal involve finding enough programmers who know the language to build a team, the availability of compilers for that language, and the efficiency with which programs written in a given language execute. Languages form an approximate spectrum from "low-level" to "high-level"; "low-level" languages are typically more machine-oriented and faster to execute, whereas "high-level" languages are more abstract and easier to use but execute less quickly.

Allen Downey, in his book *How To Think Like A Computer Scientist*, writes:

> The details look different in different languages, but a few basic instructions appear in just about every language:
>
> - **input**: Get data from the keyboard, a file, or some other device.
> - **output**: Display data on the screen or send data to a file or other device.
> - **arithmetic**: Perform basic arithmetical operations like addition and multiplication.
> - **conditional execution**: Check for certain conditions and execute the appropriate sequence of statements.
> - **repetition**: Perform some action repeatedly, usually with some variation.

Many computer languages provide a mechanism to call functions provided by libraries. Provided the functions in a library follow the appropriate run time conventions (e.g., method of passing arguments), then these functions may be written in any other language.

Programmers

Computer programmers are those who write computer software. Their jobs usually involve:

- Coding
- Compilation
- Debugging
- Documentation
- Integration
- Maintenance
- Requirements analysis
- Software architecture
- Software testing
- Specification

References

1. ↑ Budgen, D. (2003). *Software design*. Harlow, Eng.: Addison-Wesley. p. 225. ISBN 0-201-72219-4. http://books.google.com/?id=bnY3vb606bAC&pg=PA225&dq=%22anti-pattern%22+date:1990-2003. "As described in Long (2001), design anti-patterns are 'obvious, but wrong, solutions to recurring problems'."
2. ↑ Scott W. Ambler (1998). *Process patterns: building large-scale systems using object technology*. Cambridge, UK: Cambridge University Press. p. 4. ISBN 0-521-64568-9. http://books.google.com/?id=qJJk2yEeoZoC&pg=PA4&dq=%22anti-pattern%22+date:1990-2001. "...common approaches to solving recurring problems that prove to be ineffective. These approaches are called antipatterns."
3. ↑ Koenig, Andrew (March/April 1995). "Patterns and Antipatterns". *Journal of Object-Oriented Programming* **8,** (1): 46–48. ; was later re-printed in the: Rising, Linda (1998). *The patterns handbook: techniques, strategies, and applications*. Cambridge, U.K.: Cambridge University Press. p. 387. ISBN 0-521-64818-1. http://books.google.com/?id=HBAuixGMYWEC&pg=PT1&dq=0-521-64818-1. "Anti-pattern is just like pattern, except that instead of solution it gives something thats looks superficially like a solution, but isn't one."
4. ↑ Vendor Lock-In at antipatterns.com
5. ↑ Lava Flow at antipatterns.com
6. ↑ "Undocumented 'lava flow' antipatterns complicate process". Icmgworld.com. 2002-01-14. http://www.icmgworld.com/corp/news/Articles/RS/jan_0202.asp. Retrieved 2010-05-03.
7. ↑ Papadimoulis, Alex (2007-04-10). "Soft Coding". Worsethanfailure.com. http://worsethanfailure.com/Articles/Soft_Coding.aspx. Retrieved 2010-05-03.
8. ↑ Hoc, J.-M. and Nguyen-Xuan, A. Language semantics, mental models and analogy. J.-M. Hoc et al., Eds. Psychology of Programming. Academic Press. London, 1990, 139–156, cited through Brad A. Myers , John F. Pane , Andy Ko, Natural programming languages and environments, Communications of the ACM, v.47 n.9, September 2004

9. ↑ Paul Graham (2003). *Hackers and Painters*. http://www.paulgraham.com/hp.html. Retrieved 2006-08-22.
10. ↑ Kenneth E. Iverson, the originator of the APL programming language, believed that the Sapir–Whorf hypothesis applied to computer languages (without actually mentioning the hypothesis by name). His Turing award lecture, "Notation as a tool of thought", was devoted to this theme, arguing that more powerful notations aided thinking about computer algorithms. Iverson K.E.,"Notation as a tool of thought", *Communications of the ACM*, 23: 444-465 (August 1980).
11. ↑ "Ancient Greek Computer's Inner Workings Deciphered". National Geographic News. November 29, 2006.
12. ↑ Freeth, Tony; Jones, Alexander; Steele, John M.; Bitsakis, Yanis (July 31, 2008). "Calendars with Olympiad display and eclipse prediction on the Antikythera Mechanism". *Nature* **454** (7204): 614–617. doi:10.1038/nature07130. PMID 18668103. http://www.nature.com/nature/journal/v454/n7204/full/nature07130.html.
13. ↑ A 13th Century Programmable Robot, University of Sheffield
14. ↑ Fowler, Charles B. (October 1967). "The Museum of Music: A History of Mechanical Instruments". *Music Educators Journal* (Music Educators Journal, Vol. 54, No. 2) **54** (2): 45–49. doi:10.2307/3391092. http://jstor.org/stable/3391092
15. ↑ "Columbia University Computing History - Herman Hollerith". Columbia.edu. http://www.columbia.edu/acis/history/hollerith.html. Retrieved 2010-04-25.
16. ↑ 12:10 p.m. ET (2007-03-20). "Fortran creator John Backus dies - Tech and gadgets- msnbc.com". MSNBC. http://www.msnbc.msn.com/id/17704662/. Retrieved 2010-04-25.
17. ↑ "CSC-302 99S : Class 02: A Brief History of Programming Languages". Math.grin.edu. http://www.math.grin.edu/~rebelsky/Courses/CS302/99S/Outlines/outline.02.html. Retrieved 2010-04-25.
18. ↑ Survey of Job advertisements mentioning a given language>

Further reading

- Weinberg, Gerald M., *The Psychology of Computer Programming*, New York: Van Nostrand Reinhold, 1971

External links

- How to Think Like a Computer Scientist - by Jeffrey Elkner, Allen B. Downey and Chris Meyers

Code Convention

Coding conventions are a set of guidelines for a specific programming language that recommend programming style, practices and methods for each aspect of a piece program written in this language. These conventions usually cover file organization, indentation, comments, declarations, statements, white space, naming conventions, programming practices, programming principles, programming rules of thumb, etc. Software programmers are highly recommended to follow these guidelines to help improve the readability of their source code and make software maintenance easier. Coding conventions are only applicable to the human maintainers and peer reviewers of a

software project. Conventions may be formalized in a documented set of rules that an entire team or company follows, or may be as informal as the habitual coding practices of an individual. Coding conventions are not enforced by compilers. As a result, not following some or all of the rules has no impact on the executable programs created from the source code.

Software maintenance

Reducing the cost of software maintenance is the most often cited reason for following coding conventions. In their introduction to code conventions for the Java Programming Language, Sun Microsystems provides the following rationale:[1]

> Code conventions are important to programmers for a number of reasons:
>
> - 80% of the lifetime cost of a piece of software goes to maintenance.
> - Hardly any software is maintained for its whole life by the original author.
> - Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
> - If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

Quality

Software peer review frequently involves reading source code. This type of peer review is primarily a defect detection activity. By definition, only the original author of a piece of code has read the source file before the code is submitted for review. Code that is written using consistent guidelines is easier for other reviewers to understand and assimilate, improving the efficacy of the defect detection process.

Even for the original author, consistently coded software eases maintainability. There is no guarantee that an individual will remember the precise rationale for *why* a particular piece of code was written in a certain way long after the code was originally written. Coding conventions can help. Consistent use of whitespace improves readability and reduces the time it takes to understand the software.

Refactoring

Refactoring refers to a software maintenance activity where source code is modified to improve readability or improve its structure. Software is often refactored to bring it into conformance with a team's stated coding standards after its initial release. Any change that does not alter the behavior of the software can be considered refactoring. Common refactoring activities are changing variable names, renaming methods, moving methods or whole classes and breaking large methods (or functions) into smaller ones.

Agile software development methodologies plan for regular (or even continuous) refactoring making it an integral part of the team software development process.[2]

Task automation

Coding conventions allow to have simple scripts or programs whose job is to process source code for some purpose other than compiling it into an executable. It is common practice to count the software size (Source lines of code) to track current project progress or establish a baseline for future project estimates.

Consistent coding standards can, in turn, make the measurements more consistent. Special tags within source code comments are often used to process documentation, two notable examples are javadoc and doxygen. The tools specifies the use of a set of tags, but their use within a project is determined by convention.

Coding conventions simplify writing new software whose job is to process existing software. Use of static code analysis has grown consistently since the 1950s. Some of the growth of this class of development tools stems from increased maturity and sophistication of the practitioners themselves (and the modern focus on safety and security), but also from the nature of the languages themselves.

Language factors

All software practitioners must grapple with the problems of organizing and managing very many detailed instructions, each of which will eventually be processed in order to perform the task for which it was written. For all but the smallest software projects, source code (instructions) are partitioned into separate files and frequently among many directories. It was natural for programmers to collect closely related functions (behaviors) in the same file and to collect related files into directories. As software development evolved from purely procedural programming (such as found in FORTRAN) towards more object-oriented constructs (such as found in C++), it became the practice to write the code for a single (public) class in a single file (the 'one class per file' convention).[3][4] Java has gone one step further - the Java compiler returns an error if it finds more than one public class per file.

A convention in one language may be a requirement in another. Language conventions also affect individual source files. Each compiler (or interpreter) used to process source code is unique. The rules a compiler applies to the source creates implicit standards. For example, Python code is much more consistently indented than, say Perl, because whitespace (indentation) is actually significant to the interpreter. Python does not use the brace syntax Perl uses to delimit functions. Changes in indentation serve as the delimiters.[5][6] Tcl, which uses a brace syntax similar to Perl or C/C++ to delimit functions, does not allow the following, which seems fairly reasonable to a C programmer:

```
set i 0
while {$i < 10}
{
   puts "$i squared = [expr $i*$i]"
   incr i
}
```

The reason is that in Tcl, curly braces are not used only to delimit functions as in C or Java. More generally, curly braces are used to group words together into a single argument.[7][8] In Tcl, the *word* **while** takes two arguments, a *condition* and an *action*. In the example above, **while** is missing its second argument, its *action* (because the Tcl also uses the newline character to delimit the end of a command).

Common conventions

As mentioned above, common coding conventions may cover the following areas:

- Comment conventions
- Indent style conventions
- Naming conventions
- Programming practices
- Programming principles
- Programming rules of thumb
- Programming style conventions

Examples

Only one statement should occur per line

For example, in Java this would involve having statements written like this:

```
a++;
b = a;
```

But not like this:

```
a++; b = a;
```

Boolean values in decision structures

Some programmers suggest that coding where the result of a decision is merely the computation of a Boolean value, are overly verbose and error prone. They prefer to have the decision in the computation itself, like this:

```
return (hours < 24) && (minutes < 60) && (seconds < 60);
```

The difference is entirely stylistic, because optimizing compilers may produce identical object code for both forms. However, stylistically, programmers disagree which form is easier to read and maintain.

Arguments in favor of the longer form include: it is then possible to set a per-line breakpoint on one branch of the decision; further lines of code could be added to one branch without refactoring the return line, which would increase the chances of bugs being introduced; the longer form would always permit a debugger to step to a line where the variables involved are still in scope.

Left-hand comparisons

In languages which use one symbol (typically a single equals sign, (=)) for assignment and another (typically two equals signs, (==) for comparison (e.g. C/C++, Java, ActionScript 3, PHP, Perl numeric context, and most languages in the last 15 years), and where assignments may be made within control structures, there is an advantage to adopting the left-hand comparison style: to place constants or expressions to the left in any comparison. [9] [10]

Here are both left and right-hand comparison styles, applied to a line of Perl code. In both cases, this compares the value in the variable `$a` against 42, and if it matches, executes the code in the subsequent block.

```
if ( $a == 42 ) { ... }  # A right-hand comparison checking if $a equals 42.
if ( 42 == $a ) { ... }  # Recast, using the left-hand comparison style.
```

The difference occurs when a developer accidentally types = instead of ==:

```
if ( $a = 42 ) { ... }  # Inadvertent assignment which is often hard to debug
if ( 42 = $a ) { ... }  # Compile time error indicates source of problem
```

The first (right-hand) line now contains a potentially subtle flaw: rather than the previous behaviour, it now sets the value of `$a` to be 42, and then always runs the code in the following block. As this is syntactically legitimate, the error may go unnoticed by the programmer, and the software may ship with a bug.

The second (left-hand) line contains a semantic error, as numeric values cannot be assigned to. This will result in a diagnostic message being generated when the code is compiled, so the error cannot go unnoticed by the programmer.

Some languages have built-in protections against inadvertent assignment. Java and C#, for example, do not support automatic conversion to boolean for just this reason.

The risk can also be mitigated by use of static code analysis tools that can detect this issue.

Looping and control structures

The use of logical control structures for looping adds to good programming style as well. It helps someone reading code to better understand the program's sequence of execution (in imperative programming languages). For example, in pseudocode:

```
i = 0

while i < 5
  print i * 2
  i = i + 1
```

```
end while

print "Ended loop"
```

The above snippet obeys the naming and indentation style guidelines, but the following use of the "for" construct may be considered easier to read:

```
for i = 0, i < 5, i=i+1
  print i * 2

print "Ended loop"
```

In many languages, the often used "for each element in a range" pattern can be shortened to:

```
for i = 0 to 5
  print i * 2

print "Ended loop"
```

In programming languages that allow curly brackets, it has become common for style documents to require that even where optional, curly brackets be used with all control flow constructs.

```
for (i = 0 to 5) {
  print i * 2;
}

print "Ended loop";
```

This prevents program-flow bugs which can be time-consuming to track down, such as where a terminating semicolon is introduced at the end of the construct (a common typo):

```
 for (i = 0; i < 5; ++i);
    printf("%d\n", i*2);    /* The incorrect indentation hides the fact
                               that this line is not part of the loop body. */

 printf("Ended loop");
```

...or where another line is added before the first:

```
 for (i = 0; i < 5; ++i)
    fprintf(logfile, "loop reached %d\n", i);
    printf("%d\n", i*2);    /* The incorrect indentation hides the fact
                               that this line is not part of the loop body. */

 printf("Ended loop");
```

Lists

Where items in a list are placed on separate lines, it is sometimes considered good practice to add the item-separator after the final item, as well as between each item, at least in those languages where doing so is supported by the syntax (e.g., C, Java)

```
const char *array[] = {
    "item1",
    "item2",
    "item3",  /* still has the comma after it */
};
```

This prevents syntax errors or subtle string-concatenation bugs when the list items are re-ordered or more items are added to the end, without the programmer's noticing the "missing" separator on the line which was previously last in the list. However, this technique can result in a syntax error (or misleading semantics) in some languages. Even for languages that do support trailing commas, not all list-like syntactical constructs in those languages may support it.

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

External links

Coding conventions for languages

Also see the *Ada Style Guide* book.

- ActionScript: Flex SDK coding conventions and best practices
- Ada: Ada 95 Quality and Style Guide: Guidelines for Professional Programmers
- Ada: Guide for the use of the Ada programming language in high integrity systems (ISO/IEC TR 15942:2000)
- Ada: NASA Flight Software Branch — Ada Coding Standard
- Ada: European Space Agency's Ada Coding Standard (BSSC(98)3)
- C: Ganssle Group's Firmware Development Standard
- C: Netrino Embedded C Coding Standard
- C: Micrium C Coding Standard
- C++: Quantum Leaps C/C++ Coding Standard
- C++: GeoSoft's C++ Programming Style Guidelines
- C++: Google's C++ Style Guide
- C#: Design Guidelines for Developing Class Libraries
- C#: Microsoft, Philips Healthcare
- D: The D Style
- Erlang: Erlang Programming Rules and Conventions
- Flex: Code conventions for the Flex SDK

- GML: Game Maker Language
- Java: Ambysoft's Coding Standards for Java
- Java: Code Conventions for the Java Programming Language
- Java: GeoSoft's Java Programming Style Guidelines
- Java: Java Coding Standards at the Open Directory Project
- Lisp: Riastradh's Lisp Style Rules
- Mono: Programming style for Mono
- Object Pascal: Object Pascal Style Guide
- Perl: Perl Style Guide
- PHP::PEAR: PHP::PEAR Coding Standards
- Python: Style Guide for Python Code
- Ruby: The Unofficial Ruby Usage Guide
- Ruby: Good API Design

Coding conventions for projects

- Apache Developers' C Language Style Guide
- Drupal PHP Coding Standards
- Linux Kernel Coding Style (or Documentation/CodingStyle in the Linux Kernel source tree)
- ModuLiq Zero Indent Coding Style
- Mozilla Coding Style Guide
- Road Intranet's C++ Guidelines
- The NetBSD source code style guide (formerly known as the BSD Kernel Normal Form)
- "GNAT Coding Style: A Guide for GNAT Developers". *GCC online documentation*. Free Software Foundation. http://gcc.gnu.org/onlinedocs/gnat-style/. Retrieved 2009-01-19. (PDF)

Good Coding

Introduction to Software Engineering/Implementation/Good Coding

Documentation

Software documentation or **source code documentation** is written text that accompanies computer software. It either explains how it operates or how to use it, and may mean different things to people in different roles.

Involvement of people in software life

Documentation is an important part of software engineering. Types of documentation include:

1. Requirements - Statements that identify attributes, capabilities, characteristics, or qualities of a system. This is the foundation for what shall be or has been implemented.

2. Architecture/Design - Overview of softwares. Includes relations to an environment and construction principles to be used in design of software components.
3. Technical - Documentation of code, algorithms, interfaces, and APIs.
4. End User - Manuals for the end-user, system administrators and support staff.
5. Marketing - How to market the product and analysis of the market demand.

Requirements documentation

Requirements documentation is the description of what a particular software does or shall do. It is used throughout development to communicate what the software does or shall do. It is also used as an agreement or as the foundation for agreement on what the software shall do. Requirements are produced and consumed by everyone involved in the production of software: end users, customers, product managers, project managers, sales, marketing, software architects, usability engineers, interaction designers, developers, and testers, to name a few. Thus, requirements documentation has many different purposes.

Requirements come in a variety of styles, notations and formality. Requirements can be goal-like (e.g., *distributed work environment*), close to design (e.g., *builds can be started by right-clicking a configuration file and select the 'build' function*), and anything in between. They can be specified as statements in natural language, as drawn figures, as detailed mathematical formulas, and as a combination of them all.

The variation and complexity of requirements documentation makes it a proven challenge. Requirements may be implicit and hard to uncover. It is difficult to know exactly how much and what kind of documentation is needed and how much can be left to the architecture and design documentation, and it is difficult to know how to document requirements considering the variety of people that shall read and use the documentation. Thus, requirements documentation is often incomplete (or non-existent). Without proper requirements documentation, software changes become more difficult—and therefore more error prone (decreased software quality) and time-consuming (expensive).

The need for requirements documentation is typically related to the complexity of the product, the impact of the product, and the life expectancy of the software. If the software is very complex or developed by many people (e.g., mobile phone software), requirements can help to better communicate what to achieve. If the software is safety-critical and can have negative impact on human life (e.g., nuclear power systems, medical equipment), more formal requirements documentation is often required. If the software is expected to live for only a month or two (e.g., very small mobile phone applications developed specifically for a certain campaign) very little requirements documentation may be needed. If the software is a first release that is later built upon, requirements documentation is very helpful when managing the change of the software and verifying that nothing has been broken in the software when it is modified.

Traditionally, requirements are specified in requirements documents (e.g. using word processing applications and spreadsheet applications). To manage the increased complexity and changing nature of requirements documentation (and software documentation in general), database-centric systems and special-purpose requirements management tools are advocated.

Architecture/Design documentation

Architecture documentation is a special breed of design document. In a way, architecture documents are third derivative from the code (design document being second derivative, and code documents being first). Very little in the architecture documents is specific to the code itself. These documents do not describe how to program a particular routine, or even why that particular routine exists in the form that it does, but instead merely lays out the general requirements that would motivate the existence of such a routine. A good architecture document is short on details but thick on explanation. It may suggest approaches for lower level design, but leave the actual exploration trade studies to other documents.

Another breed of design docs is the comparison document, or trade study. This would often take the form of a *whitepaper*. It focuses on one specific aspect of the system and suggests alternate approaches. It could be at the user interface, code, design, or even architectural level. It will outline what the situation is, describe one or more alternatives, and enumerate the pros and cons of each. A good trade study document is heavy on research, expresses its idea clearly (without relying heavily on obtuse jargon to dazzle the reader), and most importantly is impartial. It should honestly and clearly explain the costs of whatever solution it offers as best. The objective of a trade study is to devise the best solution, rather than to push a particular point of view. It is perfectly acceptable to state no conclusion, or to conclude that none of the alternatives are sufficiently better than the baseline to warrant a change. It should be approached as a scientific endeavor, not as a marketing technique.

A very important part of the design document in enterprise software development is the Database Design Document (DDD). It contains Conceptual, Logical, and Physical Design Elements. The DDD includes the formal information that the people who interact with the database need. The purpose of preparing it is to create a common source to be used by all players within the scene. The potential users are:

- Database Designer
- Database Developer
- Database Administrator
- Application Designer
- Application Developer

When talking about Relational Database Systems, the document should include following parts:

- Entity - Relationship Schema, including following information and their clear definitions:
    - Entity Sets and their attributes
    - Relationships and their attributes
    - Candidate keys for each entity set
    - Attribute and Tuple based constraints
- Relational Schema, including following information:
    - Tables, Attributes, and their properties
    - Views
    - Constraints such as primary keys, foreign keys,
    - Cardinality of referential constraints
    - Cascading Policy for referential constraints

- Primary keys

It is very important to include all information that is to be used by all actors in the scene. It is also very important to update the documents as any change occurs in the database as well.

Technical documentation

This is what most programmers mean when using the term *software documentation*. When creating software, code alone is insufficient. There must be some text along with it to describe various aspects of its intended operation. It is important for the code documents to be thorough, but not so verbose that it becomes difficult to maintain them. Several How-to and overview documentation are found specific to the software application or software product being documented by API Writers. This documentation may be used by developers, testers and also the end customers or clients using this software application. Today, we see lot of high end applications in the field of power, energy, transportation, networks, aerospace, safety, security, industry automation and a variety of other domains. Technical documentation has become important within such organizations as the basic and advanced level of information may change over a period of time with architecture changes. Hence, technical documentation has gained lot of importance in recent times, especially in the software field.

Often, tools such as Doxygen, NDoc, javadoc, EiffelStudio, Sandcastle, ROBODoc, POD, TwinText, or Universal Report can be used to auto-generate the code documents—that is, they extract the comments and software contracts, where available, from the source code and create reference manuals in such forms as text or HTML files. Code documents are often organized into a *reference guide* style, allowing a programmer to quickly look up an arbitrary function or class.

The idea of auto-generating documentation is attractive to programmers for various reasons. For example, because it is extracted from the source code itself (for example, through comments), the programmer can write it while referring to the code, and use the same tools used to create the source code to make the documentation. This makes it much easier to keep the documentation up-to-date.

Of course, a downside is that only programmers can edit this kind of documentation, and it depends on them to refresh the output (for example, by running a cron job to update the documents nightly). Some would characterize this as a pro rather than a con.

Donald Knuth has insisted on the fact that documentation can be a very difficult afterthought process and has advocated literate programming, writing at the same time and location as the source code and extracted by automatic means.

Elucidative Programming is the result of practical applications of Literate Programming in real programming contexts. The Elucidative paradigm proposes that source code and documentation be stored separately. This paradigm was inspired by the same experimental findings that produced Kelp. Often, software developers need to be able to create and access information that is not going to be part of the source file itself. Such annotations are usually part of several software development activities, such as code walks and porting, where third party source code is analysed in a functional way. Annotations can therefore help the developer during any stage of software development where a formal documentation system would hinder progress. Kelp stores annotations in separate files, linking the information to the source code dynamically.

User documentation

Unlike code documents, user documents are usually far more diverse with respect to the source code of the program, and instead simply describe how it is used.

In the case of a software library, the code documents and user documents could be effectively equivalent and are worth conjoining, but for a general application this is not often true.

Typically, the user documentation describes each feature of the program, and assists the user in realizing these features. A good user document can also go so far as to provide thorough troubleshooting assistance. It is very important for user documents to not be confusing, and for them to be up to date. User documents need not be organized in any particular way, but it is very important for them to have a thorough index. Consistency and simplicity are also very valuable. User documentation is considered to constitute a contract specifying what the software will do. API Writers are very well accomplished towards writing good user documents as they would be well aware of the software architecture and programming techniques used. See also Technical Writing.

There are three broad ways in which user documentation can be organized.

1. **Tutorial:** A tutorial approach is considered the most useful for a new user, in which they are guided through each step of accomplishing particular tasks [11].
2. **Thematic:** A thematic approach, where chapters or sections concentrate on one particular area of interest, is of more general use to an intermediate user. Some authors prefer to convey their ideas through a knowledge based article to facilitating the user needs. This approach is usually practiced by a dynamic industry, such as Information technology, where the user population is largely correlated with the troubleshooting demands [12], [13].
3. **List or Reference:** The final type of organizing principle is one in which commands or tasks are simply listed alphabetically or logically grouped, often via cross-referenced indexes. This latter approach is of greater use to advanced users who know exactly what sort of information they are looking for.

A common complaint among users regarding software documentation is that only one of these three approaches was taken to the near-exclusion of the other two. It is common to limit provided software documentation for personal computers to online help that give only reference information on commands or menu items. The job of tutoring new users or helping more experienced users get the most out of a program is left to private publishers, who are often given significant assistance by the software developer.

Marketing documentation

For many applications it is necessary to have some promotional materials to encourage casual observers to spend more time learning about the product. This form of documentation has three purposes:-

1. To excite the potential user about the product and instill in them a desire for becoming more involved with it.
2. To inform them about what exactly the product does, so that their expectations are in line with what they will be receiving.
3. To explain the position of this product with respect to other alternatives.

One good marketing technique is to provide clear and memorable *catch phrases* that exemplify the point we wish to convey, and also emphasize the interoperability of the program with anything else provided by the manufacturer.

Notes

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

External links

- kelp - a source code annotation framework for architectural, design and technical documentation.
- ISO documentation standards committee - International Organization for Standardization committee which develops user documentation standards.

Testing

Introduction

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test.[14] Software testing also provides an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Test techniques include, but are not limited to, the process of executing a program or application with the intent of finding software bugs.

Software testing can also be stated as the process of validating and verifying that a software program/application/product:

1. meets the business and technical requirements that guided its design and development;
2. works as expected; and
3. can be implemented with the same characteristics.

Software testing, depending on the testing method employed, can be implemented at any time in the development process. However, most of the test effort occurs after the requirements have been defined and the coding process has been completed. As such, the methodology of the test is governed by the software development methodology adopted.

Different software development models will focus the test effort at different points in the development process. Newer development models, such as Agile, often employ test driven development and place an increased portion of the testing in the hands of the developer, before it reaches a formal team of testers. In a more traditional model, most of the test execution occurs after the requirements have been defined and the coding process has been completed.

Overview

Testing can never completely identify all the defects within software. Instead, it furnishes a *criticism* or *comparison* that compares the state and behavior of the product against oracles—principles or mechanisms by which someone might recognize a problem. These oracles may include (but are not limited to) specifications, contracts,[15] comparable products, past versions of the same product, inferences about intended or expected purpose, user or customer expectations, relevant standards, applicable laws, or other criteria.

Every software product has a target audience. For example, the audience for video game software is completely different from banking software. Therefore, when an organization develops or otherwise invests in a software product, it can assess whether the software product will be acceptable to its end users, its target audience, its purchasers, and other stakeholders. **Software testing** is the process of attempting to make this assessment.

A study conducted by NIST in 2002 reports that software bugs cost the U.S. economy $59.5 billion annually. More than a third of this cost could be avoided if better software testing was performed.[16]

History

The separation of debugging from testing was initially introduced by Glenford J. Myers in 1979.[17] Although his attention was on breakage testing ("a successful test is one that finds a bug"[17][18]) it illustrated the desire of the software engineering community to separate fundamental development activities, such as debugging, from that of verification. Dave Gelperin and William C. Hetzel classified in 1988 the phases and goals in software testing in the following stages:[19]

- Until 1956 - Debugging oriented[20]
- 1957–1978 - Demonstration oriented[21]
- 1979–1982 - Destruction oriented[22]
- 1983–1987 - Evaluation oriented[23]
- 1988–2000 - Prevention oriented[24]

Software testing topics

Scope

A primary purpose of testing is to detect software failures so that defects may be discovered and corrected. This is a non-trivial pursuit. Testing cannot establish that a product functions properly under all conditions but can only establish that it does not function properly under specific conditions.[25] The scope of software testing often includes examination of code as well as execution of that code in various environments and conditions as well as examining the aspects of code: does it do what it is supposed to do and do what it needs to do. In the current culture of software development, a testing organization may be separate from the development team. There are various roles for testing team members. Information derived from software testing may be used to correct the process by which software is developed.[26]

Functional vs non-functional testing

Functional testing refers to activities that verify a specific action or function of the code. These are usually found in the code requirements documentation, although some development methodologies work from use cases or user stories. Functional tests tend to answer the question of "can the user do this" or "does this particular feature work".

Non-functional testing refers to aspects of the software that may not be related to a specific function or user action, such as scalability or security. Non-functional testing tends to answer such questions as "how many people can log in at once".

Defects and failures

Not all software defects are caused by coding errors. One common source of expensive defects is caused by requirement gaps, e.g., unrecognized requirements, that result in errors of omission by the program designer.[27] A common source of requirements gaps is non-functional requirements such as testability, scalability, maintainability, usability, performance, and security.

Software faults occur through the following processes. A programmer makes an error (mistake), which results in a defect (fault, bug) in the software source code. If this defect is executed, in certain situations the system will produce wrong results, causing a failure.[28] Not all defects will necessarily result in failures. For example, defects in dead code will never result in failures. A defect can turn into a failure when the environment is changed. Examples of these changes in environment include the software being run on a new hardware platform, alterations in source data or interacting with different software.[28] A single defect may result in a wide range of failure symptoms.

Finding faults early

It is commonly believed that the earlier a defect is found the cheaper it is to fix it.[29] The following table shows the cost of fixing the defect depending on the stage it was found.[30] For example, if a problem in the requirements is found only post-release, then it would cost 10–100 times more to fix than if it had already been found by the requirements review.

Cost to fix a defect		Time detected				
		Requirements	**Architecture**	**Construction**	**System test**	**Post-release**
Time introduced	**Requirements**	1×	3×	5–10×	10×	10–100×
	Architecture	-	1×	10×	15×	25–100×
	Construction	-	-	1×	10×	10–25×

Compatibility

A common cause of software failure (real or perceived) is a lack of compatibility with other application software, operating systems (or operating system versions, old or new), or target environments that differ greatly from the original (such as a terminal or GUI application intended to be run on the desktop now being required to become a web application, which must render in a web browser). For example, in the case of a lack of backward compatibility, this can occur because the programmers develop and test software only on the latest version of the target environment, which not all users may be running. This results in the unintended consequence that the latest work may not function on earlier versions of the target environment, or on older hardware that earlier versions of the target environment was capable of using. Sometimes such issues can be fixed by proactively abstracting operating system functionality into a separate program module or library.

Input combinations and preconditions

A very fundamental problem with software testing is that testing under *all* combinations of inputs and preconditions (initial state) is not feasible, even with a simple product.[25][31] This means that the number of defects in a software product can be very large and defects that occur infrequently are difficult to find in testing. More significantly, non-functional dimensions of quality (how it is supposed to *be* versus what it is supposed to *do*)—usability, scalability, performance, compatibility, reliability—can be highly subjective; something that constitutes sufficient value to one person may be intolerable to another.

Static vs. dynamic testing

There are many approaches to software testing. Reviews, walkthroughs, or inspections are considered as static testing, whereas actually executing programmed code with a given set of test cases is referred to as dynamic testing. Static testing can be (and unfortunately in practice often is) omitted. Dynamic testing takes place when the program itself is used for the first time (which is generally considered the beginning of the testing stage). Dynamic testing may begin before the program is 100% complete in order to test particular sections of code (modules or discrete functions). Typical techniques for this are either using stubs/drivers or execution from a debugger environment. For example, spreadsheet programs are, by their very nature, tested to a large extent interactively ("on the fly"), with results displayed immediately after each calculation or text manipulation.

Software verification and validation

Software testing is used in association with verification and validation:[32]

- Verification: Have we built the software right? (i.e., does it match the specification).
- Validation: Have we built the right software? (i.e., is this what the customer wants).

The terms verification and validation are commonly used interchangeably in the industry; it is also common to see these two terms incorrectly defined. According to the IEEE Standard Glossary of Software Engineering Terminology:

> Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
> Validation is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.

The software testing team

Software testing can be done by software testers. Until the 1980s the term "software tester" was used generally, but later it was also seen as a separate profession. Regarding the periods and the different goals in software testing,[33] different roles have been established: *manager*, *test lead*, *test designer*, *tester*, *automation developer*, and *test administrator*.

Software quality assurance (SQA)

Though controversial, software testing is a part of the software quality assurance (SQA) process. [25] In SQA, software process specialists and auditors are concerned for the software development process rather than just the artefacts such as documentation, code and systems. They examine and change the software engineering process itself to reduce the amount of faults that end up in the delivered software: the so-called *defect rate.*

What constitutes an "acceptable defect rate" depends on the nature of the software; A flight simulator video game would have much higher defect tolerance than software for an actual airplane.

Although there are close links with SQA, testing departments often exist independently, and there may be no SQA function in some companies.

Software testing is a task intended to detect defects in software by contrasting a computer program's expected results with its actual results for a given set of inputs. By contrast, QA (quality assurance) is the implementation of policies and procedures intended to prevent defects from occurring in the first place.

Testing methods

The box approach

Software testing methods are traditionally divided into white- and black-box testing. These two approaches are used to describe the point of view that a test engineer takes when designing test cases.

White box testing

White box testing is when the tester has access to the internal data structures and algorithms including the code that implement these.

Types of white box testing

The following types of white box testing exist:

- API testing (application programming interface) - testing of the application using public and private APIs
- Code coverage - creating tests to satisfy some criteria of code coverage (e.g., the test designer can create tests to cause all statements in the program to be executed at least once)
- Fault injection methods - improving the coverage of a test by introducing faults to test code paths
- Mutation testing methods
- Static testing - White box testing includes all static testing

Test coverage

White box testing methods can also be used to evaluate the completeness of a test suite that

was created with black box testing methods. This allows the software team to examine parts of a system that are rarely tested and ensures that the most important function points have been tested.[34]

Two common forms of code coverage are:

- *Function coverage*, which reports on functions executed
- *Statement coverage*, which reports on the number of lines executed to complete the test

They both return a code coverage metric, measured as a percentage.

Black box testing

Black box testing treats the software as a "black box"—without any knowledge of internal implementation. Black box testing methods include: equivalence partitioning, boundary value analysis, all-pairs testing, fuzz testing, model-based testing, exploratory testing and specification-based testing.

Specification-based testing: Specification-based testing aims to test the functionality of software according to the applicable requirements.[35] Thus, the tester inputs data into, and only sees the output from, the test object. This level of testing usually requires thorough test cases to be provided to the tester, who then can simply verify that for a given input, the output value (or behavior), either "is" or "is not" the same as the expected value specified in the test case.

Specification-based testing is necessary, but it is insufficient to guard against certain risks.[36]

Advantages and disadvantages: The black box tester has no "bonds" with the code, and a tester's perception is very simple: a code *must* have bugs. Using the principle, "Ask and you shall receive," black box testers find bugs where programmers do not. On the other hand, black box testing has been said to be "like a walk in a dark labyrinth without a flashlight," because the tester doesn't know how the software being tested was actually constructed. As a result, there are situations when (1) a tester writes many test cases to check something that could have been tested by only one test case, and/or (2) some parts of the back-end are not tested at all.

Therefore, black box testing has the advantage of "an unaffiliated opinion", on the one hand, and the disadvantage of "blind exploring", on the other. [37]

Grey box testing

Grey box testing (American spelling: **gray box testing**) involves having knowledge of internal data structures and algorithms for purposes of designing the test cases, but testing at the user, or black-box level. Manipulating input data and formatting output do not qualify as grey box, because the input and output are clearly outside of the "black-box" that we are calling the system under test. This distinction is particularly important when conducting integration testing between two modules of code written by two different developers, where only the interfaces are exposed for test.

However, modifying a data repository does qualify as grey box, as the user would not normally be able to change the data outside of the system under test. Grey box testing may also include reverse engineering to determine, for instance, boundary values or error messages.

Testing levels

Tests are frequently grouped by where they are added in the software development process, or by the level of specificity of the test.

Unit testing

Unit testing refers to tests that verify the functionality of a specific section of code, usually at the function level. In an object-oriented environment, this is usually at the class level, and the minimal unit tests include the constructors and destructors.[38]

These type of tests are usually written by developers as they work on code (white-box style), to ensure that the specific function is working as expected. One function might have multiple tests, to catch corner cases or other branches in the code. Unit testing alone cannot verify the functionality of a piece of software, but rather is used to assure that the building blocks the software uses work independently of each other.

Unit testing is also called *component testing*.

Integration testing

Integration testing is any type of software testing that seeks to verify the interfaces between components against a software design. Software components may be integrated in an iterative way or all together ("big bang"). Normally the former is considered a better practice since it allows interface issues to be localised more quickly and fixed.

Integration testing works to expose defects in the interfaces and interaction between integrated components (modules). Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system. [39]

System testing

System testing tests a completely integrated system to verify that it meets its requirements.[40]

System integration testing

System integration testing verifies that a system is integrated to any external or third-party systems defined in the system requirements. [*citation needed*]

Regression testing

Regression testing focuses on finding defects after a major code change has occurred. Specifically, it seeks to uncover software regressions, or old bugs that have come back. Such regressions occur whenever software functionality that was previously working correctly stops working as intended. Typically, regressions occur as an unintended consequence of program changes, when the newly developed part of the software collides with the previously existing code. Common methods of regression testing include re-running previously run tests and checking whether previously fixed faults have re-emerged. The depth of testing depends on the phase in the release process and the risk of the added features. They can either be complete, for changes added late in the release or deemed to be risky, to very shallow, consisting of positive tests on each feature, if the changes are early in the release or deemed to be of low risk.

Acceptance testing

Acceptance testing can mean one of two things:

1. A smoke test is used as an acceptance test prior to introducing a new build to the main testing process, i.e. before integration or regression.
2. Acceptance testing performed by the customer, often in their lab environment on their own hardware, is known as user acceptance testing (UAT). Acceptance testing may be performed as part of the hand-off process between any two phases of development. [*citation needed*]

Alpha testing

Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing.[41]

Beta testing

Beta testing comes after alpha testing and can be considered a form of external user acceptance testing. Versions of the software, known as beta versions, are released to a limited audience outside of the programming team. The software is released to groups of people so that further testing can ensure the product has few faults or bugs. Sometimes, beta versions are made available to the open public to increase the feedback field to a maximal number of future users.[*citation needed*]

Non-functional testing

Special methods exist to test non-functional aspects of software. In contrast to functional testing, which establishes the correct operation of the software (correct in that it matches the expected behavior defined in the design requirements), non-functional testing verifies that the software functions properly even when it receives invalid or unexpected inputs. Software fault injection, in the form of fuzzing, is an example of non-functional testing. Non-functional testing, especially for software, is designed to establish whether the device under test can tolerate invalid or unexpected

inputs, thereby establishing the robustness of input validation routines as well as error-handling routines. Various commercial non-functional testing tools are linked from the software fault injection page; there are also numerous open-source and free software tools available that perform non-functional testing.

Software performance testing and load testing

Performance testing is executed to determine how fast a system or sub-system performs under a particular workload. It can also serve to validate and verify other quality attributes of the system, such as scalability, reliability and resource usage. Load testing is primarily concerned with testing that can continue to operate under a specific load, whether that be large quantities of data or a large number of users. This is generally referred to as software scalability. The related load testing activity of when performed as a non-functional activity is often referred to as *endurance testing*.

Volume testing is a way to test functionality. *Stress testing* is a way to test reliability. *Load testing* is a way to test performance. There is little agreement on what the specific goals of load testing are. The terms load testing, performance testing, reliability testing, and volume testing, are often used interchangeably.

Stability testing

Stability testing checks to see if the software can continuously function well in or above an acceptable period. This activity of non-functional software testing is often referred to as load (or endurance) testing.

Usability testing

Usability testing is needed to check if the user interface is easy to use and understand.It approach towards the use of the application.

Security testing

Security testing is essential for software that processes confidential data to prevent system intrusion by hackers.

Internationalization and localization

The general ability of software to be internationalized and localized can be automatically tested without actual translation, by using pseudolocalization. It will verify that the application still works, even after it has been translated into a new language or adapted for a new culture (such as different currencies or time zones).[42]

Actual translation to human languages must be tested, too. Possible localization failures include:

- Software is often localized by translating a list of strings out of context, and the translator may choose the wrong translation for an ambiguous source string.
- If several people translate strings, technical terminology may become inconsistent.
- Literal word-for-word translations may sound inappropriate, artificial or too technical in the target language.
- Untranslated messages in the original language may be left hard coded in the source code.
- Some messages may be created automatically in run time and the resulting string may be ungrammatical, functionally incorrect, misleading or confusing.
- Software may use a keyboard shortcut which has no function on the source language's keyboard layout, but is used for typing characters in the layout of the target language.
- Software may lack support for the character encoding of the target language.
- Fonts and font sizes which are appropriate in the source language, may be inappropriate in the target language; for example, CJK characters may become unreadable if the font is too small.
- A string in the target language may be longer than the software can handle. This may make the string partly invisible to the user or cause the software to fail.
- Software may lack proper support for reading or writing bi-directional text.
- Software may display images with text that wasn't localized.
- Localized operating systems may have differently-named system configuration files and environment variables and different formats for date and currency.

To avoid these and other localization problems, a tester who knows the target language must run the program with all the possible use cases for translation to see if the messages are readable, translated correctly in context and don't cause failures.

Destructive testing

Destructive testing attempts to cause the software or a sub-system to fail, in order to test its robustness.

The testing process

Traditional CMMI or waterfall development model

A common practice of software testing is that testing is performed by an independent group of testers after the functionality is developed, before it is shipped to the customer.[43] This practice often results in the testing phase being used as a project buffer to compensate for project delays, thereby compromising the time devoted to testing.[44]

Another practice is to start software testing at the same moment the project starts and it is a continuous process until the project finishes.[45]

Agile or Extreme development model

In counterpoint, some emerging software disciplines such as extreme programming and the agile software development movement, adhere to a "test-driven software development" model. In this process, unit tests are written first, by the software engineers (often with pair programming in the extreme programming methodology). Of course these tests fail initially; as they are expected to. Then as code is written it passes incrementally larger portions of the test suites. The test suites are continuously updated as new failure conditions and corner cases are discovered, and they are integrated with any regression tests that are developed. Unit tests are maintained along with the rest of the software source code and generally integrated into the build process (with inherently interactive tests being relegated to a partially manual build acceptance process). The ultimate goal of this test process is to achieve continuous deployment where software updates can be published to the public frequently. [46] [47]

A sample testing cycle

Although variations exist between organizations, there is a typical cycle for testing.[48] The sample below is common among organizations employing the Waterfall development model.

- **Requirements analysis**: Testing should begin in the requirements phase of the software development life cycle. During the design phase, testers work with developers in determining what aspects of a design are testable and with what parameters those tests work.
- **Test planning**: Test strategy, test plan, testbed creation. Since many activities will be carried out during testing, a plan is needed.
- **Test development**: Test procedures, test scenarios, test cases, test datasets, test scripts to use in testing software.
- **Test execution**: Testers execute the software based on the plans and test documents then report any errors found to the development team.
- **Test reporting**: Once testing is completed, testers generate metrics and make final reports on their test effort and whether or not the software tested is ready for release.
- **Test result analysis**: Or Defect Analysis, is done by the development team usually along with the client, in order to decide what defects should be treated, fixed, rejected (i.e. found software working properly) or deferred to be dealt with later.
- **Defect Retesting**: Once a defect has been dealt with by the development team, it is retested by the testing team. AKA Resolution testing.
- **Regression testing**: It is common to have a small test program built of a subset of tests, for each integration of new, modified, or fixed software, in order to ensure that the latest delivery has not ruined anything, and that the software product as a whole is still working correctly.
- **Test Closure**: Once the test meets the exit criteria, the activities such as capturing the key outputs, lessons learned, results, logs, documents related to the project are archived and used as a reference for future projects.

Automated testing

Many programming groups are relying more and more on automated testing, especially groups that use test-driven development. There are many frameworks to write tests in, and continuous integration software will run tests automatically every time code is checked into a version control system.

While automation cannot reproduce everything that a human can do (and all the ways they think of doing it), it can be very useful for regression testing. However, it does require a well-developed test suite of testing scripts in order to be truly useful.

Testing tools

Program testing and fault detection can be aided significantly by testing tools and debuggers. Testing/debug tools include features such as:

- Program monitors, permitting full or partial monitoring of program code including:
  - Instruction set simulator, permitting complete instruction level monitoring and trace facilities
  - Program animation, permitting step-by-step execution and conditional breakpoint at source level or in machine code
  - Code coverage reports
- Formatted dump or symbolic debugging, tools allowing inspection of program variables on error or at chosen points
- Automated functional GUI testing tools are used to repeat system-level tests through the GUI
- Benchmarks, allowing run-time performance comparisons to be made
- Performance analysis (or profiling tools) that can help to highlight hot spots and resource usage

Some of these features may be incorporated into an Integrated Development Environment (IDE).

- A regression testing technique is to have a standard set of tests, which cover existing functionality that result in persistent tabular data, and to compare pre-change data to post-change data, where there should not be differences, using a tool like diffkit. Differences detected indicate unexpected functionality changes or "regression".

Measurement in software testing

Usually, quality is constrained to such topics as correctness, completeness, security,[*citation needed*] but can also include more technical requirements as described under the ISO standard ISO/IEC 9126, such as capability, reliability, efficiency, portability, maintainability, compatibility, and usability.

There are a number of frequently-used software measures, often called *metrics*, which are used to assist in determining the state of the software or the adequacy of the testing.

Testing artifacts

Software testing process can produce several artifacts.

Test plan
: A test specification is called a test plan. The developers are well aware what test plans will be executed and this information is made available to management and the developers. The idea is to make them more cautious when developing their code or making additional changes. Some companies have a higher-level document called a test strategy.

Traceability matrix
: A traceability matrix is a table that correlates requirements or design documents to test documents. It is used to change tests when the source documents are changed, or to verify that the test results are correct.

Test case
: A test case normally consists of a unique identifier, requirement references from a design specification, preconditions, events, a series of steps (also known as actions) to follow, input, output, expected result, and actual result. Clinically defined a test case is an input and an expected result.[49] This can be as pragmatic as 'for condition x your derived result is y', whereas other test cases described in more detail the input scenario and what results might be expected. It can occasionally be a series of steps (but often steps are contained in a separate test procedure that can be exercised against multiple test cases, as a matter of economy) but with one expected result or expected outcome. The optional fields are a test case ID, test step, or order of execution number, related requirement(s), depth, test category, author, and check boxes for whether the test is automatable and has been automated. Larger test cases may also contain prerequisite states or steps, and descriptions. A test case should also contain a place for the actual result. These steps can be stored in a word processor document, spreadsheet, database, or other common repository. In a database system, you may also be able to see past test results, who generated the results, and what system configuration was used to generate those results. These past results would usually be stored in a separate table.

Test script
: The test script is the combination of a test case, test procedure, and test data. Initially the term was derived from the product of work created by automated regression test tools. Today, test scripts can be manual, automated, or a combination of both.

Test suite
: The most common term for a collection of test cases is a test suite. The test suite often also contains more detailed instructions or goals for each collection of test cases. It definitely contains a section where the tester identifies the system configuration used during testing. A group of test cases may also contain prerequisite states or steps, and descriptions of the following tests.

Test data
: In most cases, multiple sets of values or data are used to test the same functionality of a particular feature. All the test values and changeable environmental components are collected in separate files and stored as test data. It is also useful to provide this data to the client and with the product or a project.

Test harness
: The software, tools, samples of data input and output, and configurations are all referred to collectively as a test harness.

Certifications

Several certification programs exist to support the professional aspirations of software testers and quality assurance specialists. No certification currently offered actually requires the applicant to demonstrate the ability to test software. No certification is based on a widely accepted body of knowledge. This has led some to declare that the testing field is not ready for certification.[50] Certification itself cannot measure an individual's productivity, their skill, or practical knowledge, and cannot guarantee their competence, or professionalism as a tester.[51]

Software testing certification types

- *Exam-based*: Formalized exams, which need to be passed; can also be learned by self-study [e.g., for ISTQB or QAI][52]
- *Education-based*: Instructor-led sessions, where each course has to be passed [e.g., International Institute for Software Testing (IIST)].

Testing certifications

- Certified Associate in Software Testing (CAST) offered by the Quality Assurance Institute (QAI)[53]
- CATe offered by the *International Institute for Software Testing*[54]
- Certified Manager in Software Testing (CMST) offered by the Quality Assurance Institute (QAI)[53]
- Certified Software Tester (CSTE) offered by the Quality Assurance Institute (QAI) [53]
- Certified Software Test Professional (CSTP) offered by the *International Institute for Software Testing*[54]
- CSTP (TM) (Australian Version) offered by *K. J. Ross & Associates*[55]
- ISEB offered by the Information Systems Examinations Board
- ISTQB Certified Tester, Foundation Level (CTFL) offered by the International Software Testing Qualification Board [56][57]
- ISTQB Certified Tester, Advanced Level (CTAL) offered by the International Software Testing Qualification Board [56][57]
- TMPF TMap Next Foundation offered by the *Examination Institute for Information Science*[58]
- TMPA TMap Next Advanced offered by the *Examination Institute for Information Science*[58]

Quality assurance certifications

- CMSQ offered by the *Quality Assurance Institute* (QAI).[53]
- CSQA offered by the *Quality Assurance Institute* (QAI)[53]
- CSQE offered by the American Society for Quality (ASQ)[59]
- CQIA offered by the American Society for Quality (ASQ)[59]

Controversy

Some of the major software testing controversies include:

What constitutes responsible software testing?

Members of the "context-driven" school of testing[60] believe that there are no "best practices" of testing, but rather that testing is a set of skills that allow the tester to select or invent testing practices to suit each unique situation.[61]

Agile vs. traditional

Should testers learn to work under conditions of uncertainty and constant change or should they aim at process "maturity"? The agile testing movement has received growing popularity since 2006 mainly in commercial circles,[62][63] whereas government and military[64] software providers use this methodology but also the traditional test-last models (e.g. in the Waterfall model).[*citation needed*]

Exploratory test vs. scripted[65]

Should tests be designed at the same time as they are executed or should they be designed beforehand?

Manual testing vs. automated

Some writers believe that test automation is so expensive relative to its value that it should be used sparingly.[66] More in particular, test-driven development states that developers should write unit-tests of the XUnit type before coding the functionality. The tests then can be considered as a way to capture and implement the requirements.

Software design vs. software implementation

Should testing be carried out only at the end or throughout the whole process?

Who watches the watchmen?

The idea is that any form of observation is also an interaction—the act of testing can also affect that which is being tested.[67]

References

1. ↑ "Code Conventions for the Java Programming Language : Why Have Code Conventions". Sun Microsystems, Inc.. 1999-04-20. http://java.sun.com/docs/codeconv/html/CodeConventions.doc.html#16712.
2. ↑ Jeffries, Ron (2001-11-08). "What is Extreme Programming? : Design Improvement". XP Magazine. http://www.xprogramming.com/xpmag/whatisxp.htm#design.
3. ↑ Hoff, Todd (2007-01-09). "C++ Coding Standard : Naming Class Files". http://www.possibility.com/Cpp/CppCodingStandard.html#cflayout.
4. ↑ FIFE coding standards
5. ↑ van Rossum, Guido; Fred L. Drake, Jr., editor (2006-09-19). "Python Tutorial : First Steps Towards Programming". Python Software Foundation. http://docs.python.org/tut/node5.html#SECTION005200000000000000000.
6. ↑ Raymond, Eric (2000-05-01). "Why Python?". Linux Journal. http://www.linuxjournal.com/article/3882.
7. ↑ Tcl Developer Xchange. "Summary of Tcl language syntax". ActiveState. http://www.tcl.tk/man/tcl8.4/TclCmd/Tcl.htm.
8. ↑ Staplin, George Peter (2006-07-16). "Why can I not start a new line before a brace group". 'the Tcler's Wiki'. http://wiki.tcl.tk/8344.
9. ↑ Sklar, David; Adam Trachtenberg (2003). *PHP Cookbook*. O'Reilly. , recipe 5.1 "Avoiding == Versus = Confusion", p118

10.↑ "C Programming FAQs: Frequently Asked Questions". Addison-Wesley, 1995. Nov. 2010. http://c-faq.com/style/revtest.html.
11.↑ Woelz, Carlos. "The KDE Documentation Primer". http://i18n.kde.org/docs/doc-primer/index.html. Retrieved 15 June 2009.
12.↑ Microsoft. "Knowledge Base Articles for Driver Development". http://www.microsoft.com/whdc/driver/kernel/kb-drv.mspx. Retrieved 15 June 2009.
13.↑ Prekaski, Todd. "Building web and Adobe AIR applications from a shared Flex code base". http://www.adobe.com/devnet/air/flex/articles/flex_air_codebase.html. Retrieved 15 June 2009.
14.↑ Exploratory Testing, Cem Kaner, Florida Institute of Technology, *Quality Assurance Institute Worldwide Annual Software Testing Conference*, Orlando, FL, November 2006
15.↑ Leitner, A., Ciupa, I., Oriol, M., Meyer, B., Fiva, A., "Contract Driven Development = Test Driven Development - Writing Test Cases", Proceedings of ESEC/FSE'07: European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering 2007, (Dubrovnik, Croatia), September 2007
16.↑ Software errors cost U.S. economy $59.5 billion annually, NIST report
17.↑ ***a*** ***b*** Myers, Glenford J. (1979). *The Art of Software Testing*. John Wiley and Sons. ISBN 0-471-04328-1.
18.↑ Company, People's Computer (1987). "Dr. Dobb's journal of software tools for the professional programmer". *Dr. Dobb's journal of software tools for the professional programmer* (M&T Pub) **12** (1-6): 116. http://books.google.com/?id=7RoIAAAAIAAJ.
19.↑ Gelperin, D.; B. Hetzel (1988). "The Growth of Software Testing". *CACM* **31** (6). ISSN 0001-0782.
20.↑ *until 1956 it was the debugging oriented period, when testing was often associated to debugging: there was no clear difference between testing and debugging.* Gelperin, D.; B. Hetzel (1988). "The Growth of Software Testing". *CACM* **31** (6). ISSN 0001-0782.
21.↑ *From 1957–1978 there was the demonstration oriented period where debugging and testing was distinguished now - in this period it was shown, that software satisfies the requirements.* Gelperin, D.; B. Hetzel (1988). "The Growth of Software Testing". *CACM* **31** (6). ISSN 0001-0782.
22.↑ *The time between 1979–1982 is announced as the destruction oriented period, where the goal was to find errors.* Gelperin, D.; B. Hetzel (1988). "The Growth of Software Testing". *CACM* **31** (6). ISSN 0001-0782.
23.↑ *1983–1987 is classified as the evaluation oriented period: intention here is that during the software lifecycle a product evaluation is provided and measuring quality.* Gelperin, D.; B. Hetzel (1988). "The Growth of Software Testing". *CACM* **31** (6). ISSN 0001-0782.
24.↑ *From 1988 on it was seen as prevention oriented period where tests were to demonstrate that software satisfies its specification, to detect faults and to prevent faults.* Gelperin, D.; B. Hetzel (1988). "The Growth of Software Testing". *CACM* **31** (6). ISSN 0001-0782.
25.↑ ***a*** ***b*** ***c*** Kaner, Cem; Falk, Jack and Nguyen, Hung Quoc (1999). *Testing Computer Software, 2nd Ed.*. New York, et al: John Wiley and Sons, Inc.. pp. 480 pages. ISBN 0-471-35846-0.
26.↑ Kolawa, Adam; Huizinga, Dorota (2007). *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press. pp. 41–43. ISBN 0470042125. http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470042125.html.
27.↑ Kolawa, Adam; Huizinga, Dorota (2007). *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press. p. 86. ISBN 0470042125. http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470042125.html.
28.↑ ***a*** ***b*** Section 1.1.2, Certified Tester Foundation Level Syllabus, International Software Testing Qualifications Board

29.↑ Kaner, Cem; James Bach, Bret Pettichord (2001). *Lessons Learned in Software Testing: A Context-Driven Approach*. John Wiley & Sons. p. 4. ISBN 0-471-08112-4.
30.↑ McConnell, Steve (2004). *Code Complete* (2nd ed.). Microsoft Press. pp. 960. ISBN 0-7356-1967-0.
31.↑ Principle 2, Section 1.3, Certified Tester Foundation Level Syllabus, International Software Testing Qualifications Board
32.↑ Tran, Eushiuan (1999). "Verification/Validation/Certification". in Koopman, P.. *Topics in Dependable Embedded Systems*. USA: Carnegie Mellon University. http://www.ece.cmu.edu/~koopman/des_s99/verification/index.html. Retrieved 2008-01-13.
33.↑ see D. Gelperin and W.C. Hetzel
34.↑ Introduction, Code Coverage Analysis, Steve Cornett
35.↑ Laycock, G. T. (1993) (PostScript). *The Theory and Practice of Specification Based Software Testing*. Dept of Computer Science, Sheffield University, UK. http://www.mcs.le.ac.uk/people/gtl1/thesis.ps.gz. Retrieved 2008-02-13.
36.↑ Bach, James (June 1999). "Risk and Requirements-Based Testing" (PDF). *Computer* **32** (6): 113–114. http://www.satisfice.com/articles/requirements_based_testing.pdf. Retrieved 2008-08-19.
37.↑ Savenkov, Roman (2008). *How to Become a Software Tester*. Roman Savenkov Consulting. p. 159. ISBN 978-0-615-23372-7.
38.↑ Binder, Robert V. (1999). *Testing Object-Oriented Systems: Objects, Patterns, and Tools*. Addison-Wesley Professional. p. 45. ISBN 0-201-80938-9.
39.↑ Beizer, Boris (1990). *Software Testing Techniques* (Second ed.). New York: Van Nostrand Reinhold. pp. 21,430. ISBN 0-442-20672-0.
40.↑ IEEE (1990). *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. New York: IEEE. ISBN 1559370793.
41.↑ van Veenendaal, Erik. "Standard glossary of terms used in Software Testing". http://www.astqb.org/educational-resources/glossary.php#A. Retrieved 17 June 2010.
42.↑ Globalization Step-by-Step: The World-Ready Approach to Testing. Microsoft Developer Network
43.↑ e)Testing Phase in Software Testing:-
44.↑ Myers, Glenford J. (1979). *The Art of Software Testing*. John Wiley and Sons. pp. 145–146. ISBN 0-471-04328-1.
45.↑ Dustin, Elfriede (2002). *Effective Software Testing*. Addison Wesley. p. 3. ISBN 0-20179-429-2.
46.↑ Marchenko, Artem (November 16, 2007). "XP Practice: Continuous Integration". http://agilesoftwaredevelopment.com/xp/practices/continuous-integration. Retrieved 2009-11-16.
47.↑ Gurses, Levent (February 19, 2007). "Agile 101: What is Continuous Integration?". http://www.jacoozi.com/blog/?p=18. Retrieved 2009-11-16.
48.↑ Pan, Jiantao (Spring 1999). "Software Testing (18-849b Dependable Embedded Systems)". *Topics in Dependable Embedded Systems*. Electrical and Computer Engineering Department, Carnegie Mellon University. http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/.
49.↑ IEEE (1998). *IEEE standard for software test documentation*. New York: IEEE. ISBN 0-7381-1443-X.
50.↑ Kaner, Cem (2001). "NSF grant proposal to "lay a foundation for significant improvements in the quality of academic and commercial courses in software testing"" (pdf). http://www.testingeducation.org/general/nsf_grant.pdf.
51.↑ Kaner, Cem (2003). "Measuring the Effectiveness of Software Testers" (pdf). http://www.testingeducation.org/a/mest.pdf.

52.↑ Black, Rex (December 2008). *Advanced Software Testing- Vol. 2: Guide to the ISTQB Advanced Certification as an Advanced Test Manager*. Santa Barbara: Rocky Nook Publisher. ISBN 1933952369.
53.↑ **a** **b** **c** **d** **e** Quality Assurance Institute
54.↑ **a** **b** International Institute for Software Testing
55.↑ K. J. Ross & Associates
56.↑ **a** **b** "ISTQB". http://www.istqb.org/.
57.↑ **a** **b** "ISTQB in the U.S.". http://www.astqb.org/.
58.↑ **a** **b** EXIN: Examination Institute for Information Science
59.↑ **a** **b** American Society for Quality
60.↑ context-driven-testing.com
61.↑ Article on taking agile traits without the agile method.
62.↑ "We're all part of the story" by David Strom, July 1, 2009
63.↑ IEEE article about differences in adoption of agile trends between experienced managers vs. young students of the Project Management Institute. See also Agile adoption study from 2007
64.↑ Willison, John S. (April 2004). "Agile Software Development for an Agile Force". *CrossTalk* (STSC) (April 2004). Archived from the original on unknown. http://web.archive.org/web/20051029135922/http://www.stsc.hill.af.mil/crosstalk/2004/04/0404willison.html.
65.↑ IEEE article on Exploratory vs. Non Exploratory testing
66.↑ An example is Mark Fewster, Dorothy Graham: *Software Test Automation*. Addison Wesley, 1999, ISBN 0-201-33140-3.
67.↑ Microsoft Development Network Discussion on exactly this topic

External links

- Software testing tools and products at the Open Directory Project
- "Software that makes Software better" Economist.com
- Automated software testing metrics including manual testing metrics

Unit Tests

In computer programming, **unit testing** is a method by which individual units of source code are tested to determine if they are fit for use. A unit is the smallest testable part of an application. In procedural programming a unit may be an individual function or procedure. Unit tests are created by programmers or occasionally by white box testers.

Ideally, each test case is independent from the others: substitutes like method stubs, mock objects, [1] fakes and test harnesses can be used to assist testing a module in isolation. Unit tests are typically written and run by software developers to ensure that code meets its design and behaves as intended. Its implementation can vary from being very manual (pencil and paper) to being formalized as part of build automation.

Benefits

The goal of unit testing is to isolate each part of the program and show that the individual parts are correct.[2] A unit test provides a strict, written contract that the piece of code must satisfy. As a result, it affords several benefits. Unit tests find problems early in the development cycle.

Facilitates change

Unit testing allows the programmer to refactor code at a later date, and make sure the module still works correctly (e.g., in regression testing). The procedure is to write test cases for all functions and methods so that whenever a change causes a fault, it can be quickly identified and fixed.

Readily-available unit tests make it easy for the programmer to check whether a piece of code is still working properly.

In continuous unit testing environments, through the inherent practice of sustained maintenance, unit tests will continue to accurately reflect the intended use of the executable and code in the face of any change. Depending upon established development practices and unit test coverage, up-to-the-second accuracy can be maintained.

Simplifies integration

Unit testing may reduce uncertainty in the units themselves and can be used in a bottom-up testing style approach. By testing the parts of a program first and then testing the sum of its parts, integration testing becomes much easier.

An elaborate hierarchy of unit tests does not equal integration testing. Integration with peripheral units should be included in integration tests, but not in unit tests.[*citation needed*] Integration testing typically still relies heavily on humans testing manually; high-level or global-scope testing can be difficult to automate, such that manual testing often appears faster and cheaper.[*citation needed*]

Documentation

Unit testing provides a sort of living documentation of the system. Developers looking to learn what functionality is provided by a unit and how to use it can look at the unit tests to gain a basic understanding of the unit's API.

Unit test cases embody characteristics that are critical to the success of the unit. These characteristics can indicate appropriate/inappropriate use of a unit as well as negative behaviors that are to be trapped by the unit. A unit test case, in and of itself, documents these critical characteristics, although many software development environments do not rely solely upon code to document the product in development.

By contrast, ordinary narrative documentation is more susceptible to drifting from the implementation of the program and will thus become outdated (e.g., design changes, feature creep, relaxed practices in keeping documents up-to-date).

Design

When software is developed using a test-driven approach, the unit test may take the place of formal design. Each unit test can be seen as a design element specifying classes, methods, and observable behaviour. The following Java example will help illustrate this point.

Here is a test class that specifies a number of elements of the implementation. First, that there must be an interface called Adder, and an implementing class with a zero-argument constructor called AdderImpl. It goes on to assert that the Adder interface should have a method called add, with two integer parameters, which returns another integer. It also specifies the behaviour of this method for a small range of values.

```
public class TestAdder {
    public void testSum() {
        Adder adder = new AdderImpl();
        assert(adder.add(1, 1) == 2);
        assert(adder.add(1, 2) == 3);
        assert(adder.add(2, 2) == 4);
        assert(adder.add(0, 0) == 0);
        assert(adder.add(-1, -2) == -3);
        assert(adder.add(-1, 1) == 0);
        assert(adder.add(1234, 988) == 2222);
    }
}
```

In this case the unit test, having been written first, acts as a design document specifying the form and behaviour of a desired solution, but not the implementation details, which are left for the programmer. Following the "do the simplest thing that could possibly work" practice, the easiest solution that will make the test pass is shown below.

```
interface Adder {
    int add(int a, int b);
}
class AdderImpl implements Adder {
    int add(int a, int b) {
        return a + b;
    }
}
```

Unlike other diagram-based design methods, using a unit-test as a design has one significant advantage. The design document (the unit-test itself) can be used to verify that the implementation adheres to the design. With the unit-test design method, the tests will never pass if the developer does not implement the solution according to the design.

It is true that unit testing lacks some of the accessibility of a diagram, but UML diagrams are now easily generated for most modern languages by free tools (usually available as extensions to IDEs). Free tools, like those based on the xUnit framework, outsource to another system the graphical rendering of a view for human consumption.

Separation of interface from implementation

Because some classes may have references to other classes, testing a class can frequently spill over into testing another class. A common example of this is classes that depend on a database: in order to test the class, the tester often writes code that interacts with the database. This is a mistake, because a unit test should usually not go outside of its own class boundary, and especially should

not cross such process/network boundaries because this can introduce unacceptable performance problems to the unit test-suite. Crossing such unit boundaries turns unit tests into integration tests, and when test cases fail, makes it less clear which component is causing the failure. See also Fakes, mocks and integration tests

Instead, the software developer should create an abstract interface around the database queries, and then implement that interface with their own mock object. By abstracting this necessary attachment from the code (temporarily reducing the net effective coupling), the independent unit can be more thoroughly tested than may have been previously achieved. This results in a higher quality unit that is also more maintainable.

Unit testing limitations

Testing cannot be expected to catch every error in the program: it is impossible to evaluate every execution path in all but the most trivial programs. The same is true for unit testing. Additionally, unit testing by definition only tests the functionality of the units themselves. Therefore, it will not catch integration errors or broader system-level errors (such as functions performed across multiple units, or non-functional test areas such as performance). Unit testing should be done in conjunction with other software testing activities. Like all forms of software testing, unit tests can only show the presence of errors; they cannot show the absence of errors.

Software testing is a combinatorial problem. For example, every boolean decision statement requires at least two tests: one with an outcome of "true" and one with an outcome of "false". As a result, for every line of code written, programmers often need 3 to 5 lines of test code.[3] This obviously takes time and its investment may not be worth the effort. There are also many problems that cannot easily be tested at all – for example those that are nondeterministic or involve multiple threads. In addition, writing code for a unit test is as likely to be at least as buggy as the code it is testing. Fred Brooks in The Mythical Man-Month quotes: *never take two chronometers to sea. Always take one or three.* Meaning, if two chronometers contradict, how do you know which one is correct?

To obtain the intended benefits from unit testing, rigorous discipline is needed throughout the software development process. It is essential to keep careful records not only of the tests that have been performed, but also of all changes that have been made to the source code of this or any other unit in the software. Use of a version control system is essential. If a later version of the unit fails a particular test that it had previously passed, the version-control software can provide a list of the source code changes (if any) that have been applied to the unit since that time.

It is also essential to implement a sustainable process for ensuring that test case failures are reviewed daily and addressed immediately.[4] If such a process is not implemented and ingrained into the team's workflow, the application will evolve out of sync with the unit test suite, increasing false positives and reducing the effectiveness of the test suite.

Applications

Extreme Programming

Unit testing is the cornerstone of Extreme Programming, which relies on an automated unit testing framework. This automated unit testing framework can be either third party, e.g., xUnit, or created within the development group.

Extreme Programming uses the creation of unit tests for test-driven development. The developer writes a unit test that exposes either a software requirement or a defect. This test will fail because either the requirement isn't implemented yet, or because it intentionally exposes a defect in the existing code. Then, the developer writes the simplest code to make the test, along with other tests, pass.

Most code in a system is unit tested, but not necessarily all paths through the code. Extreme Programming mandates a "test everything that can possibly break" strategy, over the traditional "test every execution path" method. This leads developers to develop fewer tests than classical methods, but this isn't really a problem, more a restatement of fact, as classical methods have rarely ever been followed methodically enough for all execution paths to have been thoroughly tested. [*citation needed*] Extreme Programming simply recognizes that testing is rarely exhaustive (because it is often too expensive and time-consuming to be economically viable) and provides guidance on how to effectively focus limited resources.

Crucially, the test code is considered a first class project artifact in that it is maintained at the same quality as the implementation code, with all duplication removed. Developers release unit testing code to the code repository in conjunction with the code it tests. Extreme Programming's thorough unit testing allows the benefits mentioned above, such as simpler and more confident code development and refactoring, simplified code integration, accurate documentation, and more modular designs. These unit tests are also constantly run as a form of regression test.

Techniques

Unit testing is commonly automated, but may still be performed manually. The IEEE does not favor one over the other.[5] A manual approach to unit testing may employ a step-by-step instructional document. Nevertheless, the objective in unit testing is to isolate a unit and validate its correctness. Automation is efficient for achieving this, and enables the many benefits listed in this article. Conversely, if not planned carefully, a careless manual unit test case may execute as an integration test case that involves many software components, and thus preclude the achievement of most if not all of the goals established for unit testing.

To fully realize the effect of isolation while using an automated approach, the unit or code body under test is executed within a framework outside of its natural environment. In other words, it is executed outside of the product or calling context for which it was originally created. Testing in such an isolated manner reveals unnecessary dependencies between the code being tested and other units or data spaces in the product. These dependencies can then be eliminated.

Using an automation framework, the developer codes criteria into the test to verify the unit's correctness. During test case execution, the framework logs tests that fail any criterion. Many frameworks will also automatically flag these failed test cases and report them in a summary. Depending upon the severity of a failure, the framework may halt subsequent testing.

As a consequence, unit testing is traditionally a motivator for programmers to create decoupled and cohesive code bodies. This practice promotes healthy habits in software development. Design patterns, unit testing, and refactoring often work together so that the best solution may emerge.

Unit testing frameworks

Unit testing frameworks are most often third-party products that are not distributed as part of the compiler suite. They help simplify the process of unit testing, having been developed for a wide variety of languages. Examples of testing frameworks include open source solutions such as the various code-driven testing frameworks known collectively as xUnit, and proprietary/commercial solutions such as TBrun, Testwell CTA++ and VectorCAST/C++.

It is generally possible to perform unit testing without the support of a specific framework by writing client code that exercises the units under test and uses assertions, exception handling, or other control flow mechanisms to signal failure. Unit testing without a framework is valuable in that there is a barrier to entry for the adoption of unit testing; having scant unit tests is hardly better than having none at all, whereas once a framework is in place, adding unit tests becomes relatively easy. [6] In some frameworks many advanced unit test features are missing or must be hand-coded.

Language-level unit testing support

Some programming languages support unit testing directly (Eg. Java). Their grammar allows the direct declaration of unit tests without importing a library (whether third party or standard). Additionally, the boolean conditions of the unit tests can be expressed in the same syntax as boolean expressions used in non-unit test code, such as what is used for `if` and `while` statements.

Languages that directly support unit testing include:

- Cobra
- D

Notes

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."

3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

External links

- The evolution of Unit Testing Syntax and Semantics
- Unit Testing Guidelines from GeoSoft
- Test Driven Development (Ward Cunningham's Wiki)
- Unit Testing 101 for the Non-Programmer
- Step-by-Step Guide to JPA-Enabled Unit Testing (Java EE)

Profiling

In software engineering, **program profiling**, **software profiling** or simply **profiling**, a form of dynamic program analysis (as opposed to static code analysis), is the investigation of a program's behavior using information gathered as the program executes. The usual purpose of this analysis is to determine which sections of a program to optimize - to increase its overall speed, decrease its memory requirement or sometimes both.

- A **(code) profiler** is a **performance analysis** tool that, most commonly, measures only the frequency and duration of function calls, but there are other specific types of profilers (e.g. memory profilers) in addition to more comprehensive profilers, capable of gathering extensive performance data.
- An instruction set simulator which is also — by necessity — a profiler, can measure the totality of a program's behaviour from invocation to termination.

Gathering program events

Profilers use a wide variety of techniques to collect data, including hardware interrupts, code instrumentation, instruction set simulation, operating system hooks, and performance counters. The usage of profilers is 'called out' in the performance engineering process.

Use of profilers

Program analysis tools are extremely important for understanding program behavior. Computer architects need such tools to evaluate how well programs will perform on new architectures. Software writers need tools to analyze their programs and identify critical sections of code. Compiler writers often use such tools to find out how well their instruction scheduling or branch prediction algorithm is performing... (ATOM, PLDI, '94)

The output of a profiler may be:-

- A statistical *summary* of the events observed (a **profile**)

  Summary profile information is often shown annotated against the source code statements where the events occur, so the size of measurement data is linear to the code size of the program.

```
/* ------------ source------------------------- count */
0001             IF X = "A"                     0055
0002                THEN DO
0003                  ADD 1 to XCOUNT           0032
0004                ELSE
0005             IF X = "B"                     0055
```

- A stream of recorded events (a **trace**)

  For sequential programs, a summary profile is usually sufficient, but performance problems in parallel programs (waiting for messages or synchronization issues) often depend on the time relationship of events, thus requiring a full trace to get an understanding of what is happening. The size of a (full) trace is linear to the program's instruction path length, making it somewhat impractical. A trace may therefore be initiated at one point in a program and terminated at another point to limit the output.

- An ongoing interaction with the hypervisor (continuous or periodic monitoring via on-screen display for instance)

  This provides the opportunity to switch a trace on or off at any desired point during execution in addition to viewing on-going metrics about the (still executing) program. It also provides the opportunity to suspend asynchronous processes at critical points to examine interactions with other parallel processes in more detail.

History

Performance analysis tools existed on IBM/360 and IBM/370 platforms from the early 1970s, usually based on timer interrupts which recorded the Program status word (PSW) at set timer intervals to detect "hot spots" in executing code. This was an early example of sampling (see below). In early 1974, Instruction Set Simulators permitted full trace and other performance monitoring features.

Profiler-driven program analysis on Unix dates back to at least 1979, when Unix systems included a basic tool "prof" that listed each function and how much of program execution time it used. In 1982, gprof extended the concept to a complete call graph analysis [7]

In 1994, Amitabh Srivastava and Alan Eustace of Digital Equipment Corporation published a paper describing ATOM.[8] ATOM is a platform for converting a program into its own profiler. That is, at compile time, it inserts code into the program to be analyzed. That inserted code outputs analysis data. This technique - modifying a program to analyze itself - is known as "instrumentation".

In 2004, both the gprof and ATOM papers appeared on the list of the 50 most influential PLDI papers of all time.[9]

Profiler types based on output

Flat profiler

Flat profilers compute the average call times, from the calls, and do not break down the call times based on the callee or the context.

Call-graph profiler

Call graph profilers show the call times, and frequencies of the functions, and also the call-chains involved based on the callee. However context is not preserved.

Methods of data gathering

Event-based profilers

The programming languages listed here have event-based profilers:

- Java: the JVMTI (JVM Tools Interface) API, formerly JVMPI (JVM Profiling Interface), provides hooks to profilers, for trapping events like calls, class-load, unload, thread enter leave.
- .NET: Can attach a profiling agent as a COM server to the CLR. Like Java, the runtime then provides various callbacks into the agent, for trapping events like method JIT / enter / leave, object creation, etc. Particularly powerful in that the profiling agent can rewrite the target application's bytecode in arbitrary ways.
- Python: Python profiling includes the profile module, hotshot (which is call-graph based), and using the 'sys.setprofile' function to trap events like c_{call,return,exception}, python_{call,return,exception}.
- Ruby: Ruby also uses a similar interface like Python for profiling. Flat-profiler in profile.rb, module, and ruby-prof a C-extension are present.

Statistical profilers

Some profilers operate by sampling. A sampling profiler probes the target program's program counter at regular intervals using operating system interrupts. Sampling profiles are typically less numerically accurate and specific, but allow the target program to run at near full speed.

The resulting data are not exact, but a statistical approximation. *The actual amount of error is usually more than one sampling period. In fact, if a value is n times the sampling period, the expected error in it is the square-root of n sampling periods.* [10]

In practice, sampling profilers can often provide a more accurate picture of the target program's execution than other approaches, as they are not as intrusive to the target program, and thus don't have as many side effects (such as on memory caches or instruction decoding pipelines). Also since

they don't affect the execution speed as much, they can detect issues that would otherwise be hidden. They are also relatively immune to over-evaluating the cost of small, frequently called routines or 'tight' loops. They can show the relative amount of time spent in user mode versus interruptible kernel mode such as system call processing.

Still, kernel code to handle the interrupts entails a minor loss of CPU cycles, diverted cache usage, and is unable to distinguish the various tasks occurring in uninterruptible kernel code (microsecond-range activity).

Dedicated hardware can go beyond this: some recent MIPS processors JTAG interface have a PCSAMPLE register, which samples the program counter in a truly undetectable manner.

Some of the most commonly used statistical profilers are AMD CodeAnalyst, Apple Inc. Shark, gprof, Intel VTune and Parallel Amplifier (part of Intel Parallel Studio).

Instrumenting profilers

Some profilers **instrument** the target program with additional instructions to collect the required information.

Instrumenting the program can cause changes in the performance of the program, potentially causing inaccurate results and heisenbugs. Instrumenting will always have some impact on the program execution, typically always slowing it. However, instrumentation can be very specific and be carefully controlled to have a minimal impact. The impact on a particular program depends on the placement of instrumentation points and the mechanism used to capture the trace. Hardware support for trace capture means that on some targets, instrumentation can be on just one machine instruction. The impact of instrumentation can often be deducted (i.e. eliminated by subtraction) from the results.

gprof is an example of a profiler that uses both instrumentation and sampling. Instrumentation is used to gather caller information and the actual timing values are obtained by statistical sampling.

Instrumentation

- **Manual**: Performed by the programmer, e.g. by adding instructions to explicitly calculate runtimes, simply count events or calls to measurement APIs such as the Application Response Measurement standard.
- **Automatic source level**: instrumentation added to the source code by an automatic tool according to an instrumentation policy.
- **Compiler assisted**: Example: "gcc -pg ..." for gprof, "quantify g++ ..." for Quantify
- **Binary translation**: The tool adds instrumentation to a compiled binary. Example: ATOM
- **Runtime instrumentation**: Directly before execution the code is instrumented. The program run is fully supervised and controlled by the tool. Examples: Pin, Valgrind
- **Runtime injection**: More lightweight than runtime instrumentation. Code is modified at runtime to have jumps to helper functions. Example: DynInst

Interpreter instrumentation

- **Interpreter debug** options can enable the collection of performance metrics as the interpreter encounters each target statement. A bytecode, control table or JIT interpreters are three examples that usually have complete control over execution of the target code, thus enabling extremely comprehensive data collection opportunities.

Hypervisor/Simulator

- **Hypervisor**: Data are collected by running the (usually) unmodified program under a hypervisor. Example: SIMMON
- **Simulator** and **Hypervisor**: Data collected interactively and selectively by running the unmodified program under an Instruction Set Simulator. Examples: SIMON (Batch Interactive test/debug) and IBM OLIVER (CICS interactive test/debug).

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

- Dunlavey, “Performance tuning with instruction-level cost derived from call-stack sampling”, ACM SIGPLAN Notices 42, 8 (August, 2007), pp. 4–8.
- Dunlavey, “Performance Tuning: Slugging It Out!”, Dr. Dobb's Journal, Vol 18, #12, November 1993, pp 18–26.

External links

- Article "Need for speed — Eliminating performance bottlenecks" on doing execution time analysis of Java applications using IBM Rational Application Developer.
- Profiling Runtime Generated and Interpreted Code using the VTune™ Performance Analyzer

Test-driven Development

Test-driven development (**TDD**) is a software development process that relies on the repetition of a very short development cycle: first the developer writes a failing automated test case that defines a desired improvement or new function, then produces code to pass that test and finally refactors the new code to acceptable standards. Kent Beck, who is credited with having developed or 'rediscovered' the technique, stated in 2003 that TDD encourages simple designs and inspires confidence.[11]

Test-driven development is related to the test-first programming concepts of extreme programming, begun in 1999,[12] but more recently has created more general interest in its own right.[13]

Programmers also apply the concept to improving and debugging legacy code developed with older techniques.[14]

Requirements

Test-driven development requires developers to create automated unit tests that define code requirements (immediately) before writing the code itself. The tests contain assertions that are either true or false. Passing the tests confirms correct behavior as developers evolve and refactor the code. Developers often use testing frameworks, such as xUnit, to create and automatically run sets of test cases.

Test-driven development cycle

The following sequence is based on the book *Test-Driven Development by Example*[11].

Add a test

In **test-driven development**, each new feature begins with writing a test. This test must inevitably fail because it is written before the feature has been implemented. (If it does not fail, then either the proposed “new” feature already exists or the test is defective.) To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through use cases and user stories that cover the requirements and exception conditions. This could also imply a variant, or modification of an existing test. This is a differentiating feature of test-driven development versus writing unit tests *after* the code is written: it makes the developer focus on the requirements *before* writing the code, a subtle but important difference.

Run all tests and see if the new one fails

This validates that the test harness is working correctly and that the new test does not mistakenly pass without requiring any new code. This step also tests the test itself, in the negative: it rules out the possibility that the new test will always pass, and therefore be worthless. The new test should also fail for the expected reason. This increases confidence (although it does not entirely guarantee) that it is testing the right thing, and will pass only in intended cases.

Write some code

The next step is to write some code that will cause the test to pass. The new code written at this stage will not be perfect and may, for example, pass the test in an inelegant way. That is acceptable because later steps will improve and hone it.

It is important that the code written is *only* designed to pass the test; no further (and therefore untested) functionality should be predicted and 'allowed for' at any stage.

Run the automated tests and see them succeed

If all test cases now pass, the programmer can be confident that the code meets all the tested requirements. This is a good point from which to begin the final step of the cycle.

Refactor code

Now the code can be cleaned up as necessary. By re-running the test cases, the developer can be confident that code refactoring is not damaging any existing functionality. The concept of removing duplication is an important aspect of any software design. In this case, however, it also applies to removing any duplication between the test code and the production code — for example magic numbers or strings that were repeated in both, in order to make the test pass in step 3.

Repeat

Starting with another new test, the cycle is then repeated to push forward the functionality. The size of the steps should always be small, with as few as 1 to 10 edits between each test run. If new code does not rapidly satisfy a new test, or other tests fail unexpectedly, the programmer should undo or revert in preference to excessive debugging. Continuous Integration helps by providing revertible checkpoints. When using external libraries it is important not to make increments that are so small as to be effectively merely testing the library itself,[13] unless there is some reason to believe that the library is buggy or is not sufficiently feature-complete to serve all the needs of the main program being written.

Development style

There are various aspects to using test-driven development, for example the principles of "keep it simple, stupid" (KISS) and "You ain't gonna need it" (YAGNI). By focusing on writing only the code necessary to pass tests, designs can be cleaner and clearer than is often achieved by other methods.[11] In *Test-Driven Development by Example* Kent Beck also suggests the principle "Fake it till you make it".

To achieve some advanced design concept (such as a design pattern), tests are written that will generate that design. The code may remain simpler than the target pattern, but still pass all required tests. This can be unsettling at first but it allows the developer to focus only on what is important.

Write the tests first. The tests should be written before the functionality that is being tested. This has been claimed to have two benefits. It helps ensure that the application is written for testability, as the developers must consider how to test the application from the outset, rather than worrying about it later. It also ensures that tests for every feature will be written. When writing feature-first code, there is a tendency by developers and the development organisations to push the developer onto the next feature, neglecting testing entirely.

First fail the test cases. The idea is to ensure that the test really works and can catch an error. Once this is shown, the underlying functionality can be implemented. This has been coined the "test-driven development mantra", known as red/green/refactor where red means *fail* and green is *pass*.

Test-driven development constantly repeats the steps of adding test cases that fail, passing them, and refactoring. Receiving the expected test results at each stage reinforces the programmer's mental model of the code, boosts confidence and increases productivity.

Advanced practices of test-driven development can lead to Acceptance Test-driven development (ATDD) where the criteria specified by the customer are automated into acceptance tests, which then drive the traditional unit test-driven development (UTDD) process.[15] This process ensures the customer has an automated mechanism to decide whether the software meets their requirements. With ATDD, the development team now has a specific target to satisfy, the acceptance tests, which keeps them continuously focused on what the customer really wants from that user story.

Benefits

A 2005 study found that using TDD meant writing more tests and, in turn, programmers that wrote more tests tended to be more productive.[16] Hypotheses relating to code quality and a more direct correlation between TDD and productivity were inconclusive.[17]

Programmers using pure TDD on new ("greenfield") projects report they only rarely feel the need to invoke a debugger. Used in conjunction with a version control system, when tests fail unexpectedly, reverting the code to the last version that passed all tests may often be more productive than debugging.[18]

Test-driven development offers more than just simple validation of correctness, but can also drive the design of a program. By focusing on the test cases first, one must imagine how the functionality will be used by clients (in the first case, the test cases). So, the programmer is concerned with the interface before the implementation. This benefit is complementary to Design by Contract as it approaches code through test cases rather than through mathematical assertions or preconceptions.

Test-driven development offers the ability to take small steps when required. It allows a programmer to focus on the task at hand as the first goal is to make the test pass. Exceptional cases and error handling are not considered initially, and tests to create these extraneous circumstances are implemented separately. Test-driven development ensures in this way that all written code is covered by at least one test. This gives the programming team, and subsequent users, a greater level of confidence in the code.

While it is true that more code is required with TDD than without TDD because of the unit test code, total code implementation time is typically shorter.[19] Large numbers of tests help to limit the number of defects in the code. The early and frequent nature of the testing helps to catch defects

early in the development cycle, preventing them from becoming endemic and expensive problems. Eliminating defects early in the process usually avoids lengthy and tedious debugging later in the project.

TDD can lead to more modularized, flexible, and extensible code. This effect often comes about because the methodology requires that the developers think of the software in terms of small units that can be written and tested independently and integrated together later. This leads to smaller, more focused classes, looser coupling, and cleaner interfaces. The use of the mock object design pattern also contributes to the overall modularization of the code because this pattern requires that the code be written so that modules can be switched easily between mock versions for unit testing and "real" versions for deployment.

Because no more code is written than necessary to pass a failing test case, automated tests tend to cover every code path. For example, in order for a TDD developer to add an `else` branch to an existing `if` statement, the developer would first have to write a failing test case that motivates the branch. As a result, the automated tests resulting from TDD tend to be very thorough: they will detect any unexpected changes in the code's behaviour. This detects problems that can arise where a change later in the development cycle unexpectedly alters other functionality.

Vulnerabilities

- Test-driven development is difficult to use in situations where full functional tests are required to determine success or failure. Examples of these are user interfaces, programs that work with databases, and some that depend on specific network configurations. TDD encourages developers to put the minimum amount of code into such modules and to maximise the logic that is in testable library code, using fakes and mocks to represent the outside world.
- Management support is essential. Without the entire organization believing that test-driven development is going to improve the product, management may feel that time spent writing tests is wasted.[20]
- Unit tests created in a test-driven development environment are typically created by the developer who will also write the code that is being tested. The tests may therefore share the same blind spots with the code: If, for example, a developer does not realize that certain input parameters must be checked, most likely neither the test nor the code will verify these input parameters. If the developer misinterprets the requirements specification for the module being developed, both the tests and the code will be wrong.
- The high number of passing unit tests may bring a false sense of security, resulting in fewer additional software testing activities, such as integration testing and compliance testing.
- The tests themselves become part of the maintenance overhead of a project. Badly written tests, for example ones that include hard-coded error strings or which are themselves prone to failure, are expensive to maintain. There is a risk that tests that regularly generate false failures will be ignored, so that when a real failure occurs it may not be detected. It is possible to write tests for low and easy maintenance, for example by the reuse of error strings, and this should be a goal during the code refactoring phase described above.
- The level of coverage and testing detail achieved during repeated TDD cycles cannot easily be re-created at a later date. Therefore these original tests become increasingly precious as time goes by. If a poor architecture, a poor design or a poor testing strategy leads to a late

change that makes dozens of existing tests fail, it is important that they are individually fixed. Merely deleting, disabling or rashly altering them can lead to undetectable holes in the test coverage.

Code Visibility

Test suite code clearly has to be able to access the code it is testing. On the other hand normal design criteria such as information hiding, encapsulation and the separation of concerns should not be compromised. Therefore unit test code for TDD is usually written within the same project or module as the code being tested.

In object oriented design this still does not provide access to `private` data and methods. Therefore, extra work may be necessary for unit tests. In Java and other languages, a developer can use reflection to access fields that are marked `private`.[21] Alternatively, an inner class can be used to hold the unit tests so they will have visibility of the enclosing class's members and attributes. In the .NET Framework and some other programming languages, partial classes may be used to expose private methods and data for the tests to access.

It is important that such testing hacks do not remain in the production code. In C and other languages, compiler directives such as `#if DEBUG ... #endif` can be placed around such additional classes and indeed all other test-related code to prevent them being compiled into the released code. This then means that the released code is not exactly the same as that which is unit tested. The regular running of fewer but more comprehensive, end-to-end, integration tests on the final release build can then ensure (among other things) that no production code exists that subtly relies on aspects of the test harness.

There is some debate among practitioners of TDD, documented in their blogs and other writings, as to whether it is wise to test private and protected methods and data anyway. Some argue that it should be sufficient to test any class through its public interface as the private members are a mere implementation detail that may change, and should be allowed to do so without breaking numbers of tests. Others say that crucial aspects of functionality may be implemented in private methods, and that developing this while testing it indirectly via the public interface only obscures the issue: unit testing is about testing the smallest unit of functionality possible.[22][23]

Fakes, mocks and integration tests

Unit tests are so named because they each test *one unit* of code. A complex module may have a thousand unit tests and a simple one only ten. The tests used for TDD should never cross process boundaries in a program, let alone network connections. Doing so introduces delays that make tests run slowly and discourage developers from running the whole suite. Introducing dependencies on external modules or data also turns *unit tests* into *integration tests*. If one module misbehaves in a chain of interrelated modules, it is not so immediately clear where to look for the cause of the failure.

When code under development relies on a database, a web service, or any other external process or service, enforcing a unit-testable separation is also an opportunity and a driving force to design more modular, more testable and more reusable code.[24] Two steps are necessary:

1. Whenever external access is going to be needed in the final design, an interface should be defined that describes the access that will be available. See the dependency inversion principle for a discussion of the benefits of doing this regardless of TDD.
2. The interface should be implemented in two ways, one of which really accesses the external process, and the other of which is a fake or mock. Fake objects need do little more than add a message such as "Person object saved" to a trace log, against which a test assertion can be run to verify correct behaviour. Mock objects differ in that they themselves contain test assertions that can make the test fail, for example, if the person's name and other data are not as expected. Fake and mock object methods that return data, ostensibly from a data store or user, can help the test process by always returning the same, realistic data that tests can rely upon. They can also be set into predefined fault modes so that error-handling routines can be developed and reliably tested. Fake services other than data stores may also be useful in TDD: Fake encryption services may not, in fact, encrypt the data passed; fake random number services may always return 1. Fake or mock implementations are examples of dependency injection.

A corollary of such dependency injection is that the actual database or other external-access code is never tested by the TDD process itself. To avoid errors that may arise from this, other tests are needed that instantiate the test-driven code with the "real" implementations of the interfaces discussed above. These tests are quite separate from the TDD unit tests, and are really integration tests. There will be fewer of them, and they need to be run less often than the unit tests. They can nonetheless be implemented using the same testing framework, such as xUnit.

Integration tests that alter any persistent store or database should always be designed carefully with consideration of the initial and final state of the files or database, even if any test fails. This is often achieved using some combination of the following techniques:

- The `TearDown` method, which is integral to many test frameworks.
- `try...catch...finally` exception handling structures where available.
- Database transactions where a transaction atomically includes perhaps a write, a read and a matching delete operation.
- Taking a "snapshot" of the database before running any tests and rolling back to the snapshot after each test run. This may be automated using a framework such as Ant or NAnt or a continuous integration system such as CruiseControl.
- Initialising the database to a clean state *before* tests, rather than cleaning up *after* them. This may be relevant where cleaning up may make it difficult to diagnose test failures by deleting the final state of the database before detailed diagnosis can be performed.

Frameworks such as Moq, jMock, NMock, EasyMock, Typemock, jMockit, Unitils, Mockito, Mockachino, PowerMock or Rhino Mocks exist to make the process of creating and using complex mock objects easier.

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001

2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

External links

- [10] on WikiWikiWeb
- Test or spec? Test and spec? Test from spec!, by Bertrand Meyer (September 2004)
- Microsoft Visual Studio Team Test from a TDD approach
- Write Maintainable Unit Tests That Will Save You Time And Tears
- Improving Application Quality Using Test-Driven Development (TDD)

Refactoring

Code refactoring is "a disciplined way to restructure code",[25] undertaken in order to improve some of the *nonfunctional* attributes of the software. Typically, this is done by applying series of "refactorings", each of which is a (usually) tiny change in a computer program's source code that does not modify its *functional requirements*. Advantages include improved code readability and reduced complexity to improve the maintainability of the source code, as well as a more expressive internal architecture or object model to improve extensibility.

> *By continuously improving the design of code, we make it easier and easier to work with. This is in sharp contrast to what typically happens: little refactoring and a great deal of attention paid to expediently adding new features. If you get into the hygienic habit of refactoring continuously, you'll find that it is easier to extend and maintain code.*

—- Joshua Kerievsky, *Refactoring to Patterns* [26]

Refactoring does not take place in a vacuum, but typically the refactoring process takes place in a context of adding features to software:

- "... refactoring and adding new functionality are two different but complementary tasks" -- Scott Ambler

Overview

Refactoring is usually motivated by noticing a code smell.[27] For example the method at hand may be very long, or it may be a near duplicate of another nearby method. Once recognized, such problems can be addressed by *refactoring* the source code, or transforming it into a new form that behaves the same as before but that no longer "smells". For a long routine, extract one or more smaller subroutines. Or for duplicate routines, remove the duplication and utilize one shared function in their place. Failure to perform refactoring can result in accumulating technical debt.

There are two general categories of benefits to the activity of refactoring.

1. Maintainability. It is easier to fix bugs because the source code is easy to read and the intent of its author is easy to grasp.[28] This might be achieved by reducing large monolithic routines into a set of individually concise, well-named, single-purpose methods. It might be achieved by moving a method to a more appropriate class, or by removing misleading comments.
2. Extensibility. It is easier to extend the capabilities of the application if it uses recognizable design patterns, and it provides some flexibility where none before may have existed.[26]

Before refactoring a section of code, a solid set of automatic unit tests is needed. The tests should demonstrate in a few seconds [*citation needed*] that the behavior of the module is correct. The process is then an iterative cycle of making a small program transformation, testing it to ensure correctness, and making another small transformation. If at any point a test fails, you undo your last small change and try again in a different way. Through many small steps the program moves from where it was to where you want it to be. Proponents of extreme programming and other agile methodologies describe this activity as an integral part of the software development cycle.

List of refactoring techniques

Here are some examples of code refactorings; some of these may only apply to certain languages or language types. A longer list can be found in Fowler's Refactoring book[27] and on Fowler's Refactoring Website.[29]

- Techniques that allow for more abstraction
    - Encapsulate Field – force code to access the field with getter and setter methods
    - Generalize Type – create more general types to allow for more code sharing
    - Replace type-checking code with State/Strategy[30]
    - Replace conditional with polymorphism[31]
- Techniques for breaking code apart into more logical pieces
    - Extract Method, to turn part of a larger method into a new method. By breaking down code in smaller pieces, it is more easily understandable. This is also applicable to functions.
    - Extract Class moves part of the code from an existing class into a new class.
- Techniques for improving names and location of code
    - Move Method or Move Field – move to a more appropriate Class or source file
    - Rename Method or Rename Field – changing the name into a new one that better reveals its purpose

- Pull Up – in OOP, move to a superclass
- Push Down – in OOP, move to a subclass

Hardware refactoring

While the term *refactoring* originally referred exclusively to refactoring of software code, in recent years code written in hardware description languages (HDLs) has also been refactored. The term *hardware refactoring* is used as a shorthand term for refactoring of code in hardware description languages. Since HDLs are not considered to be programming languages by most hardware engineers,[32] hardware refactoring is to be considered a separate field from traditional code refactoring.

Automated refactoring of analog hardware descriptions (in VHDL-AMS) has been proposed by Zeng and Huss.[33] In their approach, refactoring preserves the simulated behavior of a hardware design. The non-functional measurement that improves is that refactored code can be processed by standard synthesis tools, while the original code cannot. Refactoring of digital HDLs, albeit manual refactoring, has also been investigated by Synopsys fellow Mike Keating.[34][35] His target is to make complex systems easier to understand, which increases the designers' productivity.

In the summer of 2008, there was an intense discussion about refactoring of VHDL code on the news://comp.lang.vhdl newsgroup.[36] The discussion revolved around a specific manual refactoring performed by one engineer, and the question to whether or not automated tools for such refactoring exist.

As of late 2009, Sigasi is offering automated tool support for VHDL refactoring.[37]

History

In the past refactoring was avoided in development processes. One example of this is that CVS (created in 1984) does not version the moving or renaming of files and directories.

Although refactoring code has been done informally for years, William Opdyke's 1992 Ph.D. dissertation[38] is the first known paper to specifically examine refactoring,[39] although all the theory and machinery have long been available as program transformation systems. All of these resources provide a catalog of common methods for refactoring; a refactoring method has a description of how to apply the method and indicators for when you should (or should not) apply the method.

Martin Fowler's book *Refactoring: Improving the Design of Existing Code*[27] is the canonical reference.

The first known use of the term "refactoring" in the published literature was in a September, 1990 article by William F. Opdyke and Ralph E. Johnson.[40] Opdyke's Ph.D. thesis,[38] published in 1992, also used this term.[39]

The term "factoring" has been used in the Forth community since at least the early 1980s[*citation needed*]. Chapter Six of Leo Brodie's book *Thinking Forth* (1984) is dedicated to the subject.

In extreme programming, the Extract Method refactoring technique has essentially the same meaning as factoring in Forth; to break down a "word" (or function) into smaller, more easily maintained functions.

Automated code refactoring

Many software editors and IDEs have automated refactoring support. Here is a list of a few of these editors, or so-called refactoring browsers.

- IntelliJ IDEA (for Java)
- Eclipse's Java Development Toolkit (JDT)
- NetBeans (for Java)
  - and RefactoringNG, a Netbeans module for refactoring where you can write transformations rules of the program's abstract syntax tree.
- Embarcadero Delphi
- Visual Studio (for .NET)
- JustCode (addon for Visual Studio)
- ReSharper (addon for Visual Studio)
- Coderush (addon for Visual Studio)
- Visual Assist (addon for Visual Studio with refactoring support for VB, VB.NET. C# and C++)
- DMS Software Reengineering Toolkit (Implements large-scale refactoring for C, C++, C#, COBOL, Java, PHP and other languages)
- Photran a Fortran plugin for the Eclipse IDE
- SharpSort addin for Visual Studio 2008
- Sigasi HDT (for VHDL)
- XCode
- Smalltalk Refactoring Browser (for Smalltalk)
- Simplifide (for Verilog, VHDL and SystemVerilog)
- Tidier (for Erlang)

References

1. ↑ Fowler, Martin (2007-01-02). "Mocks aren't Stubs". http://martinfowler.com/articles/mocksArentStubs.html. Retrieved 2008-04-01.
2. ↑ Kolawa, Adam; Huizinga, Dorota (2007). *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press. p. 75. ISBN 0470042125. http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470042125.html.
3. ↑ Cramblitt, Bob (2007-09-20). "Alberto Savoia sings the praises of software testing". http://searchsoftwarequality.techtarget.com/originalContent/0,289142,sid92_gci1273161,00.html. Retrieved 2007-11-29.

4. ↑ daVeiga, Nada (2008-02-06). "Change Code Without Fear: Utilize a regression safety net". http://www.ddj.com/development-tools/206105233. Retrieved 2008-02-08.
5. ↑ IEEE Standards Board, "IEEE Standard for Software Unit Testing: An American National Standard, ANSI/IEEE Std 1008-1987" in *IEEE Standards: Software Engineering, Volume Two: Process Standards; 1999 Edition; published by The Institute of Electrical and Electronics Engineers, Inc.* Software Engineering Technical Committee of the IEEE Computer Society.
6. ↑ Bullseye Testing Technology (2006–2008). "Intermediate Coverage Goals". http://www.bullseye.com/coverage.html#intermediate. Retrieved 24 March 2009.
7. ↑ *gprof: a Call Graph Execution Profiler*
8. ↑ Atom: A system for building customized program analysis tools, Amitabh Srivastava and Alan Eustace, 1994 (download)
9. ↑ 20 Years of PLDI (1979 - 1999): A Selection, Kathryn S. McKinley, Editor
10. ↑ Statistical Inaccuracy of `gprof` Output
11. ↑ ***a*** ***b*** ***c*** Beck, K. Test-Driven Development by Example, Addison Wesley, 2003
12. ↑ Lee Copeland (December 2001). "Extreme Programming". Computerworld. http://www.computerworld.com/softwaretopics/software/appdev/story/0,10801,66192,00.html. Retrieved January 11, 2011.
13. ↑ ***a*** ***b*** Newkirk, JW and Vorontsov, AA. *Test-Driven Development in Microsoft .NET*, Microsoft Press, 2004.
14. ↑ Feathers, M. Working Effectively with Legacy Code, Prentice Hall, 2004
15. ↑ Koskela, L. "Test Driven: TDD and Acceptance TDD for Java Developers", Manning Publications, 2007
16. ↑ Erdogmus, Hakan; Morisio, Torchiano. "On the Effectiveness of Test-first Approach to Programming". Proceedings of the IEEE Transactions on Software Engineering, 31(1). January 2005. (NRC 47445). http://iit-iti.nrc-cnrc.gc.ca/publications/nrc-47445_e.html. Retrieved 2008-01-14. "We found that test-first students on average wrote more tests and, in turn, students who wrote more tests tended to be more productive."
17. ↑ Proffitt, Jacob. "TDD Proven Effective! Or is it?". http://theruntime.com/blogs/jacob/archive/2008/01/22/tdd-proven-effective-or-is-it.aspx. Retrieved 2008-02-21. "So TDD's relationship to quality is problematic at best. Its relationship to productivity is more interesting. I hope there's a follow-up study because the productivity numbers simply don't add up very well to me. There is an undeniable correlation between productivity and the number of tests, but that correlation is actually stronger in the non-TDD group (which had a single outlier compared to roughly half of the TDD group being outside the 95% band)."
18. ↑ Llopis, Noel (20 February 2005). "Stepping Through the Looking Glass: Test-Driven Game Development (Part 1)". Games from Within. http://www.gamesfromwithin.com/articles/0502/000073.html. Retrieved 2007-11-01. "Comparing [TDD] to the non-test-driven development approach, you're replacing all the mental checking and debugger stepping with code that verifies that your program does exactly what you intended it to do."
19. ↑ Müller, Matthias M.; Padberg, Frank. "About the Return on Investment of Test-Driven Development" (PDF). Universität Karlsruhe, Germany. pp. 6. http://www.ipd.uka.de/mitarbeiter/muellerm/publications/edser03.pdf. Retrieved 2007-11-01.
20. ↑ Loughran, Steve (November 6th, 2006). "Testing" (PDF). HP Laboratories. http://people.apache.org/~stevel/slides/testing.pdf. Retrieved 2009-08-12.

21.↑ Burton, Ross (11/12/2003). "Subverting Java Access Protection for Unit Testing". O'Reilly Media, Inc.. http://www.onjava.com/pub/a/onjava/2003/11/12/reflection.html. Retrieved 2009-08-12.
22.↑ Newkirk, James (7 June 2004). "Testing Private Methods/Member Variables - Should you or shouldn't you". Microsoft Corporation. http://blogs.msdn.com/jamesnewkirk/archive/2004/06/07/150361.aspx. Retrieved 2009-08-12.
23.↑ Stall, Tim (1 Mar 2005). "How to Test Private and Protected methods in .NET". CodeProject. http://www.codeproject.com/KB/cs/testnonpublicmembers.aspx. Retrieved 2009-08-12.
24.↑ Fowler, Martin (1999). *Refactoring - Improving the design of existing code*. Boston: Addison Wesley Longman, Inc.. ISBN 0-201-48567-2.
25.↑ Scott Ambler
26.↑ ***a*** ***b*** Kerievsky, Joshua (2004). *Refactoring to Patterns*. Addison Wesley.
27.↑ ***a*** ***b*** ***c*** Fowler, Martin (1999). *Refactoring: Improving the design of existing code*. Addison Wesley.
28.↑ Martin, Robert (2009). *Clean Code*. Prentice Hall.
29.↑ Refactoring techniques in Fowler's refactoring Website
30.↑ Replace type-checking code with State/Strategy
31.↑ Replace conditional with polymorphism
32.↑ Hardware description languages and programming languages
33.↑ Kaiping Zeng, Sorin A. Huss, "Architecture refinements by code refactoring of behavioral VHDL-AMS models". ISCAS 2006
34.↑ M. Keating :"Complexity, Abstraction, and the Challenges of Designing Complex Systems", in DAC'08 tutorial [4]"Bridging a Verification Gap: C++ to RTL for Practical Design"
35.↑ M. Keating, P. Bricaud: *Reuse Methodology Manual for System-on-a-Chip Designs*, Kluwer Academic Publishers, 1999.
36.↑ http://newsgroups.derkeiler.com/Archive/Comp/comp.lang.vhdl/2008-06/msg00173.html
37.↑ www.eetimes.com/news/latest/showArticle.jhtml?articleID=222001855
38.↑ ***a*** ***b*** Opdyke, William F (June 1992) (compressed Postscript). *Refactoring Object-Oriented Frameworks*. Ph.D. thesis. University of Illinois at Urbana-Champaign. ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z. Retrieved 2008-02-12.
39.↑ ***a*** ***b*** Martin Fowler, "MF Bliki: EtymologyOfRefactoring"
40.↑ Opdyke, William F.; Johnson, Ralph E. (September 1990). "Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems". *Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPPA)*. ACM.

Further reading

- Fowler, Martin (1999). *Refactoring. Improving the Design of Existing Code*. Addison-Wesley. ISBN 0-201-48567-2.
- Wake, William C. (2003). *Refactoring Workbook*. Addison-Wesley. ISBN 0-321-10929-5.
- Mens, Tom and Tourwé, Tom (2004) *A Survey of Software Refactoring*, IEEE Transactions on Software Engineering, February 2004 (vol. 30 no. 2), pp. 126-139
- Feathers, Michael C (2004). *Working Effectively with Legacy Code*. Prentice Hall. ISBN 0-13-117705-2.

- Kerievsky, Joshua (2004). *Refactoring To Patterns*. Addison-Wesley. ISBN 0-321-21335-1.
- Arsenovski, Danijel (2008). *Professional Refactoring in Visual Basic*. Wrox. ISBN 0-47-017979-1.
- Arsenovski, Danijel (2009). *Professional Refactoring in C# and ASP.NET*. Wrox. ISBN 978-0470434529.
- Ritchie, Peter (2010). *Refactoring with Visual Studio 2010*. Packt. ISBN 978-1849680103.

External links

- What Is Refactoring? (c2.com article)
- Martin Fowler's homepage about refactoring
- Aspect-Oriented Refactoring by Ramnivas Laddad
- A Survey of Software Refactoring by Tom Mens and Tom Tourwé
- Refactoring at the Open Directory Project
- Refactoring Java Code
- Refactoring To Patterns Catalog
- Extract Boolean Variable from Conditional (a refactoring pattern not listed in the above catalog)
- Test-Driven Development With Refactoring
- Revisiting Fowler's Video Store: Refactoring Code, Refining Abstractions

Software Quality

Introduction

In the context of software engineering, **software quality** measures how well software is designed (*quality of design*), and how well the software conforms to that design (*quality of conformance*),[1] although there are several different definitions. It is often described as the 'fitness for purpose' of a piece of software.

Whereas *quality of conformance* is concerned with implementation (see Software Quality Assurance), *quality of design* measures how valid the design and requirements are in creating a worthwhile product.[2]

Definition

One of the challenges of software quality is that "everyone feels they understand it".[3]

Software quality may be defined as conformance to explicitly stated functional and performance requirements, explicitly documented development standards and implicit characteristics that are expected of all professionally developed software.

The three key points in this definition:

1. Software requirements are the foundations from which quality is measured.

   Lack of conformance to requirement is lack of quality.

2. Specified standards define a set of development criteria that guide the manager is software engineering.

   If criteria are not followed lack of quality will usually result.

3. A set of implicit requirements often goes unmentioned, for example ease of use, maintainability etc.

   If software confirms to its explicit requirement but fails to meet implicit requirements, software quality is suspected.

A definition in Steve McConnell's *Code Complete* divides software into two pieces: **internal** and **external quality characteristics**. External quality characteristics are those parts of a product that face its users, where internal quality characteristics are those that do not.[4]

Another definition by Dr. Tom DeMarco says "a product's quality is a function of how much it changes the world for the better."[5] This can be interpreted as meaning that user satisfaction is more important than anything in determining software quality.[1]

Another definition, coined by Gerald Weinberg in *Quality Software Management: Systems Thinking*, is "Quality is value to some person." This definition stresses that quality is inherently subjective - different people will experience the quality of the same software very differently. One strength of this definition is the questions it invites software teams to consider, such as "Who are the people we want to value our software?" and "What will be valuable to *them*?"

History

Software product quality

- Product quality
    - conformance to requirements or program specification; related to Reliability
- Scalability
- Correctness
- Completeness
- Absence of bugs
- Fault-tolerance
    - Extensibility
    - Maintainability
- Documentation

The Consortium for IT Software Quality (CISQ) was launched in 2009 to standardize the measurement of software product quality. The Consortium's goal is to bring together industry executives from Global 2000 IT organizations, system integrators, outsourcers, and package vendors to jointly address the challenge of standardizing the measurement of IT software quality and to promote a market-based ecosystem to support its deployment.

Source code quality

A computer has no concept of "well-written" source code. However, from a human point of view source code can be written in a way that has an effect on the effort needed to comprehend its behavior. Many source code programming style guides, which often stress readability and usually language-specific conventions are aimed at reducing the cost of source code maintenance. Some of the issues that affect code quality include:

- Readability
- Ease of maintenance, testing, debugging, fixing, modification and portability
- Low complexity
- Low resource consumption: memory, CPU
- Number of compilation or lint warnings
- Robust input validation and error handling, established by software fault injection

Methods to improve the quality:

- Refactoring

- Code Inspection or software review
- Documenting code

Software reliability

Software **reliability** is an important facet of software quality. It is defined as "the probability of failure-free operation of a computer program in a specified environment for a specified time".[6]

One of reliability's distinguishing characteristics is that it is objective, measurable, and can be estimated, whereas much of software quality is subjective criteria.[7] This distinction is especially important in the discipline of Software Quality Assurance. These measured criteria are typically called software metrics.

History

With software embedded into many devices today, software failure has caused more than inconvenience. Software errors have even caused human fatalities. The causes have ranged from poorly designed user interfaces to direct programming errors. An example of a programming error that lead to multiple deaths is discussed in Dr. Leveson's paper [11] (PDF). This has resulted in requirements for development of some types software. In the United States, both the Food and Drug Administration (FDA) and Federal Aviation Administration (FAA) have requirements for software development.

Goal of reliability

The need for a means to objectively determine software reliability comes from the desire to apply the techniques of contemporary engineering fields to the development of software. That desire is a result of the common observation, by both lay-persons and specialists, that computer software does not work the way it ought to. In other words, software is seen to exhibit undesirable behaviour, up to and including outright failure, with consequences for the data which is processed, the machinery on which the software runs, and by extension the people and materials which those machines might negatively affect. The more critical the application of the software to economic and production processes, or to life-sustaining systems, the more important is the need to assess the software's reliability.

Regardless of the criticality of any single software application, it is also more and more frequently observed that software has penetrated deeply into most every aspect of modern life through the technology we use. It is only expected that this infiltration will continue, along with an accompanying dependency on the software by the systems which maintain our society. As software becomes more and more crucial to the operation of the systems on which we depend, the argument goes, it only follows that the software should offer a concomitant level of dependability. In other words, the software should behave in the way it is intended, or even better, in the way it should.

Challenge of reliability

The circular logic of the preceding sentence is not accidental—it is meant to illustrate a fundamental problem in the issue of measuring software reliability, which is the difficulty of determining, in advance, exactly how the software is intended to operate. The problem seems to stem from a common conceptual error in the consideration of software, which is that software in some sense takes on a role which would otherwise be filled by a human being. This is a problem on two levels. Firstly, most modern software performs work which a human could never perform, especially at the high level of reliability that is often expected from software in comparison to humans. Secondly, software is fundamentally incapable of most of the mental capabilities of humans which separate them from mere mechanisms: qualities such as adaptability, general-purpose knowledge, a sense of conceptual and functional context, and common sense.

Nevertheless, most software programs could safely be considered to have a particular, even singular purpose. If the possibility can be allowed that said purpose can be well or even completely defined, it should present a means for at least considering objectively whether the software is, in fact, reliable, by comparing the expected outcome to the actual outcome of running the software in a given environment, with given data. Unfortunately, it is still not known whether it is possible to exhaustively determine either the expected outcome or the actual outcome of the entire set of possible environment and input data to a given program, without which it is probably impossible to determine the program's reliability with any certainty.

However, various attempts are in the works to attempt to rein in the vastness of the space of software's environmental and input variables, both for actual programs and theoretical descriptions of programs. Such attempts to improve software reliability can be applied at different stages of a program's development, in the case of real software. These stages principally include: requirements, design, programming, testing, and runtime evaluation. The study of theoretical software reliability is predominantly concerned with the concept of correctness, a mathematical field of computer science which is an outgrowth of language and automata theory.

Reliability in program development

Requirements

A program cannot be expected to work as desired if the developers of the program do not, in fact, know the program's desired behaviour in advance, or if they cannot at least determine its desired behaviour in parallel with development, in sufficient detail. What level of detail is considered sufficient is hotly debated. The idea of perfect detail is attractive, but may be impractical, if not actually impossible. This is because the desired behaviour tends to change as the possible range of the behaviour is determined through actual attempts, or more accurately, failed attempts, to achieve it.

Whether a program's desired behaviour can be successfully specified in advance is a moot point if the behaviour cannot be specified at all, and this is the focus of attempts to formalize the process of creating requirements for new software projects. In situ with the formalization effort is an attempt to help inform non-specialists, particularly non-programmers, who commission software projects

without sufficient knowledge of what computer software is in fact capable. Communicating this knowledge is made more difficult by the fact that, as hinted above, even programmers cannot always know in advance what is actually possible for software in advance of trying.

Design

While requirements are meant to specify what a program should do, design is meant, at least at a high level, to specify how the program should do it. The usefulness of design is also questioned by some, but those who look to formalize the process of ensuring reliability often offer good software design processes as the most significant means to accomplish it. Software design usually involves the use of more abstract and general means of specifying the parts of the software and what they do. As such, it can be seen as a way to break a large program down into many smaller programs, such that those smaller pieces together do the work of the whole program.

The purposes of high-level design are as follows. It separates what are considered to be problems of architecture, or overall program concept and structure, from problems of actual coding, which solve problems of actual data processing. It applies additional constraints to the development process by narrowing the scope of the smaller software components, and thereby—it is hoped—removing variables which could increase the likelihood of programming errors. It provides a program template, including the specification of interfaces, which can be shared by different teams of developers working on disparate parts, such that they can know in advance how each of their contributions will interface with those of the other teams. Finally, and perhaps most controversially, it specifies the program independently of the implementation language or languages, thereby removing language-specific biases and limitations which would otherwise creep into the design, perhaps unwittingly on the part of programmer-designers.

Programming

The history of computer programming language development can often be best understood in the light of attempts to master the complexity of computer programs, which otherwise becomes more difficult to understand in proportion (perhaps exponentially) to the size of the programs. (Another way of looking at the evolution of programming languages is simply as a way of getting the computer to do more and more of the work, but this may be a different way of saying the same thing). Lack of understanding of a program's overall structure and functionality is a sure way to fail to detect errors in the program, and thus the use of better languages should, conversely, reduce the number of errors by enabling a better understanding.

Improvements in languages tend to provide incrementally what software design has attempted to do in one fell swoop: consider the software at ever greater levels of abstraction. Such inventions as statement, sub-routine, file, class, template, library, component and more have allowed the arrangement of a program's parts to be specified using abstractions such as layers, hierarchies and modules, which provide structure at different granularities, so that from any point of view the program's code can be imagined to be orderly and comprehensible.

In addition, improvements in languages have enabled more exact control over the shape and use of data elements, culminating in the abstract data type. These data types can be specified to a very fine degree, including how and when they are accessed, and even the state of the data before and after it is accessed..

Software Build and Deployment

Many programming languages such as C and Java require the program "source code" to be translated in to a form that can be executed by a computer. This translation is done by a program called a compiler. Additional operations may be involved to associate, bind, link or package files together in order to create a usable runtime configuration of the software application. The totality of the compiling and assembly process is generically called "building" the software.

The software build is critical to software quality because if any of the generated files are incorrect the software build is likely to fail. And, if the incorrect version of a program is inadvertently used, then testing can lead to false results.

Software builds are typically done in work area unrelated to the runtime area, such as the application server. For this reason, a deployment step is needed to physically transfer the software build products to the runtime area. The deployment procedure may also involve technical parameters, which, if set incorrectly, can also prevent software testing from beginning. For example, a Java application server may have options for parent-first or parent-last class loading. Using the incorrect parameter can cause the application to fail to execute on the application server.

The technical activities supporting software quality including build, deployment, change control and reporting are collectively known as Software configuration management. A number of software tools have arisen to help meet the challenges of configuration management including file control tools and build control tools.

Testing

Software testing, when done correctly, can increase overall software *quality of conformance* by testing that the product conforms to its requirements. Testing includes, but is not limited to:

1. Unit Testing
2. Functional Testing
3. Regression Testing
4. Performance Testing
5. Failover Testing
6. Usability Testing

A number of agile methodologies use testing early in the development cycle to ensure quality in their products. For example, the test-driven development practice, where tests are written before the code they will test, is used in Extreme Programming to ensure quality.

Runtime

runtime reliability determinations are similar to tests, but go beyond simple confirmation of behaviour to the evaluation of qualities such as performance and interoperability with other code or particular hardware configurations.

Software quality factors

A software quality factor is a non-functional requirement for a software program which is not called up by the customer's contract, but nevertheless is a desirable requirement which enhances the quality of the software program. Note that none of these factors are binary; that is, they are not "either you have it or you don't" traits. Rather, they are characteristics that one seeks to maximize in one's software to optimize its quality. So rather than asking whether a software product "has" factor *x*, ask instead the *degree* to which it does (or does not).

Some software quality factors are listed here:

Understandability
: Clarity of purpose. This goes further than just a statement of purpose; all of the design and user documentation must be clearly written so that it is easily understandable. This is obviously subjective in that the user context must be taken into account: for instance, if the software product is to be used by software engineers it is not required to be understandable to the layman.

Completeness
: Presence of all constituent parts, with each part fully developed. This means that if the code calls a subroutine from an external library, the software package must provide reference to that library and all required parameters must be passed. All required input data must also be available.

Conciseness
: Minimization of excessive or redundant information or processing. This is important where memory capacity is limited, and it is generally considered good practice to keep lines of code to a minimum. It can be improved by replacing repeated functionality by one subroutine or function which achieves that functionality. It also applies to documents.

Portability
: Ability to be run well and easily on multiple computer configurations. Portability can mean both between different hardware—such as running on a PC as well as a smartphone—and between different operating systems—such as running on both Mac OS X and GNU/Linux.

Consistency
: Uniformity in notation, symbology, appearance, and terminology within itself.

Maintainability
: Propensity to facilitate updates to satisfy new requirements. Thus the software product that is maintainable should be well-documented, should not be complex, and should have spare capacity for memory, storage and processor utilization and other resources.

Testability
: Disposition to support acceptance criteria and evaluation of performance. Such a characteristic must be built-in during the design phase if the product is to be easily testable; a complex design leads to poor testability.

Usability

Convenience and practicality of use. This is affected by such things as the human-computer interface. The component of the software that has most impact on this is the user interface (UI), which for best usability is usually graphical (i.e. a GUI).

Reliability
: Ability to be expected to perform its intended functions satisfactorily. This implies a time factor in that a reliable product is expected to perform correctly over a period of time. It also encompasses environmental considerations in that the product is required to perform correctly in whatever conditions it finds itself (sometimes termed robustness).

Efficiency
: Fulfillment of purpose without waste of resources, such as memory, space and processor utilization, network bandwidth, time, etc.

Security
: Ability to protect data against unauthorized access and to withstand malicious or inadvertent interference with its operations. Besides the presence of appropriate security mechanisms such as authentication, access control and encryption, security also implies resilience in the face of malicious, intelligent and adaptive attackers.

Measurement of software quality factors

There are varied perspectives within the field on measurement. There are a great many measures that are valued by some professionals—or in some contexts, that are decried as harmful by others. Some believe that quantitative measures of software quality are essential. Others believe that contexts where quantitative measures are useful are quite rare, and so prefer qualitative measures. Several leaders in the field of software testing have written about the difficulty of measuring what we truly want to measure well.[8][9]

One example of a popular metric is the number of faults encountered in the software. Software that contains few faults is considered by some to have higher quality than software that contains many faults. Questions that can help determine the usefulness of this metric in a particular context include:

1. What constitutes “many faults?” Does this differ depending upon the purpose of the software (e.g., blogging software vs. navigational software)? Does this take into account the size and complexity of the software?
2. Does this account for the importance of the bugs (and the importance to the stakeholders of the people those bugs bug)? Does one try to weight this metric by the severity of the fault, or the incidence of users it affects? If so, how? And if not, how does one know that 100 faults discovered is better than 1000?
3. If the count of faults being discovered is shrinking, how do I know what that means? For example, does that mean that the product is now higher quality than it was before? Or that this is a smaller/less ambitious change than before? Or that fewer tester-hours have gone into the project than before? Or that this project was tested by less skilled testers than before? Or that the team has discovered that fewer faults reported is in their interest?

This last question points to an especially difficult one to manage. All software quality metrics are in some sense measures of human behavior, since humans create software.[8] If a team discovers that they will benefit from a drop in the number of reported bugs, there is a strong tendency for the team to start reporting fewer defects. That may mean that email begins to circumvent the bug tracking system, or that four or five bugs get lumped into one bug report, or that testers learn not to report minor annoyances. The difficulty is measuring what we mean to measure, without creating incentives for software programmers and testers to consciously or unconsciously "game" the measurements.

Software quality factors cannot be measured because of their vague definitions. It is necessary to find measurements, or metrics, which can be used to quantify them as non-functional requirements. For example, reliability is a software quality factor, but cannot be evaluated in its own right. However, there are related attributes to reliability, which can indeed be measured. Some such attributes are mean time to failure, rate of failure occurrence, and availability of the system. Similarly, an attribute of portability is the number of target-dependent statements in a program.

A scheme that could be used for evaluating software quality factors is given below. For every characteristic, there are a set of questions which are relevant to that characteristic. Some type of scoring formula could be developed based on the answers to these questions, from which a measurement of the characteristic can be obtained.

Understandability

Are variable names descriptive of the physical or functional property represented? Do uniquely recognisable functions contain adequate comments so that their purpose is clear? Are deviations from forward logical flow adequately commented? Are all elements of an array functionally related?...

Completeness

Are all necessary components available? Does any process fail for lack of resources or programming? Are all potential pathways through the code accounted for, including proper error handling?

Conciseness

Is all code reachable? Is any code redundant? How many statements within loops could be placed outside the loop, thus reducing computation time? Are branch decisions too complex?

Portability

Does the program depend upon system or library routines unique to a particular installation? Have machine-dependent statements been flagged and commented? Has dependency on internal bit representation of alphanumeric or special characters been avoided? How much effort would be required to transfer the program from one hardware/software system or environment to another?

Consistency

Is one variable name used to represent different logical or physical entities in the program? Does the program contain only one representation for any given physical or mathematical constant? Are functionally similar arithmetic expressions similarly constructed? Is a consistent scheme used for indentation, nomenclature, the color palette, fonts and other visual elements?

Maintainability

Has some memory capacity been reserved for future expansion? Is the design cohesive—i.e., does each module have distinct, recognizable functionality? Does the software allow for a change in data structures (object-oriented designs are more likely to allow for this)? If the code is procedure-based (rather than object-oriented), is a change likely to require restructuring the main program, or just a module?

Testability

Are complex structures employed in the code? Does the detailed design contain clear pseudo-code? Is the pseudo-code at a higher level of abstraction than the code? If tasking is used in concurrent designs, are schemes available for providing adequate test cases?

Usability

Is a GUI used? Is there adequate on-line help? Is a user manual provided? Are meaningful error messages provided?

Reliability

Are loop indexes range-tested? Is input data checked for range errors? Is divide-by-zero avoided? Is exception handling provided? It is the probability that the software performs its intended functions correctly in a specified period of time under stated operation conditions, but there could also be a problem with the requirement document...

Efficiency

Have functions been optimized for speed? Have repeatedly used blocks of code been formed into subroutines? Has the program been checked for memory leaks or overflow errors?

Security

Does the software protect itself and its data against unauthorized access and use? Does it allow its operator to enforce security policies? Are security mechanisms appropriate, adequate and correctly implemented? Can the software withstand attacks that can be anticipated in its intended environment?

User's perspective

In addition to the technical qualities of software, the end user's experience also determines the quality of software. This aspect of software quality is called usability. It is hard to quantify the usability of a given software product. Some important questions to be asked are:

- Is the user interface intuitive (self-explanatory/self-documenting)?
- Is it easy to perform simple operations?
- Is it feasible to perform complex operations?
- Does the software give sensible error messages?
- Do widgets behave as expected?
- Is the software well documented?
- Is the user interface responsive or too slow?

Also, the availability of (free or paid) support may factor into the usability of the software.

References

Notes

1. ↑ ***a*** ***b*** Pressman 2005, p. 746
2. ↑ Pressman 2005, p. 388
3. ↑ Crosby, P., *Quality is Free*, McGraw-Hill, 1979
4. ↑ McConnell 1993, p. 558
5. ↑ DeMarco, T., *Management Can Make Quality (Im)possible*, Cutter IT Summit, Boston, April 1999
6. ↑ J.D. Musa, A. Iannino, and K. Okumoto, *Engineering and Managing Software with Reliability Measures*, McGraw-Hill, 1987
7. ↑ Pressman 2005, p. 762
8. ↑ ***a*** ***b*** Cem Kaner http://www.kaner.com/pdfs/metrics2004.pdf
9. ↑ Douglass Hoffman http://www.softwarequalitymethods.com/Papers/DarkMets%20Paper.pdf

Bibliography

- McConnell, Steve (1993), *Code Complete* (First ed.), Microsoft Press
- Pressman, Scott (2005), *Software Engineering: A Practitioner's Approach* (Sixth, International ed.), McGraw-Hill Education

Further reading

- International Organization for Standardization. *Software Engineering—Product Quality—Part 1: Quality Model*. ISO, Geneva, Switzerland, 2001. ISO/IEC 9126-1:2001(E).
- Diomidis Spinellis. *Code Quality: The Open Source Perspective*. Addison Wesley, Boston, MA, 2006.

- Ho-Won Jung, Seung-Gweon Kim, and Chang-Sin Chung. Measuring software product quality: A survey of ISO/IEC 9126. *IEEE Software*, 21(5):10–13, September/October 2004.
- Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley, Boston, MA, second edition, 2002.
- Robert L. Glass. *Building Quality Software*. Prentice Hall, Upper Saddle River, NJ, 1992.
- Roland Petrasch, "The Definition of, Software Quality': A Practical Approach", ISSRE, 1999

External links

- Linux: Fewer Bugs Than Rivals Wired Magazine, 2004

Static Analysis

Static program analysis is the analysis of computer software that is performed without actually executing programs built from that software (analysis performed on executing programs is known as dynamic analysis). In most cases the analysis is performed on some version of the source code and in the other cases some form of the object code. The term is usually applied to the analysis performed by an automated tool, with human analysis being called program understanding, program comprehension or code review.

The sophistication of the analysis performed by tools varies from those that only consider the behavior of individual statements and declarations, to those that include the complete source code of a program in their analysis. Uses of the information obtained from the analysis vary from highlighting possible coding errors (e.g., the lint tool) to formal methods that mathematically prove properties about a given program (e.g., its behavior matches that of its specification).

It can be argued that software metrics and reverse engineering are forms of static analysis.

A growing commercial use of static analysis is in the verification of properties of software used in safety-critical computer systems and locating potentially vulnerable code. For example, medical software is increasing in sophistication and complexity, and the U.S. Food and Drug Administration (FDA) has identified the use of static code analysis as a means of improving the quality of software[1].

Formal methods

Formal methods is the term applied to the analysis of software (and hardware) whose results are obtained purely through the use of rigorous mathematical methods. The mathematical techniques used include denotational semantics, axiomatic semantics, operational semantics, and abstract interpretation.

It has been proven that, barring some hypothesis that the state space of programs is finite, finding all possible run-time errors, or more generally any kind of violation of a specification on the final result of a program, is undecidable: there is no mechanical method that can always answer truthfully

whether a given program may or may not exhibit runtime errors. This result dates from the works of Church, Kurt Gödel and Turing in the 1930s (see the halting problem and Rice's theorem). As with most[*citation needed*] undecidable questions, one can still attempt to give useful approximate solutions.

Some of the implementation techniques of formal static analysis include:

- Model checking considers systems that have finite state or may be reduced to finite state by abstraction;
- Data-flow analysis is a lattice-based technique for gathering information about the possible set of values;
- Abstract interpretation models the effect that every statement has on the state of an abstract machine (i.e., it 'executes' the software based on the mathematical properties of each statement and declaration). This abstract machine over-approximates the behaviours of the system: the abstract system is thus made simpler to analyze, at the expense of *incompleteness* (not every property true of the original system is true of the abstract system). If properly done, though, abstract interpretation is *sound* (every property true of the abstract system can be mapped to a true property of the original system)[2]. The Frama-c framework and Polyspace heavily rely on abstract interpretation.
- Use of assertions in program code as first suggested by Hoare logic. There is tool support for some programming languages (e.g., the SPARK programming language (a subset of Ada) and the Java Modeling Language — JML — using ESC/Java and ESC/Java2, ANSI/ISO C Specification Language for the C language).

References

1. ↑ FDA (2010-09-08). "Infusion Pump Software Safety Research at FDA". Food and Drug Administration. http://www.fda.gov/MedicalDevices/ProductsandMedicalProcedures/GeneralHospitalDevicesandSupplies/InfusionPumps/ucm202511.htm. Retrieved 2010-09-09.
2. ↑ Jones, Paul (2010-02-09). "A Formal Methods-based verification approach to medical device software analysis". Embedded Systems Design. http://embeddeddsp.embedded.com/design/opensource/222700533. Retrieved 2010-09-09.

Bibliography

- Syllabus and readings for Alex Aiken's Stanford CS295 course.
- Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, William Pugh, "Using Static Analysis to Find Bugs," IEEE Software, vol. 25, no. 5, pp. 22-29, Sep./Oct. 2008, doi:10.1109/MS.2008.130
- Brian Chess, Jacob West (Fortify Software) (2007). *Secure Programming with Static Analysis*. Addison-Wesley. ISBN 978-0321424778.
- Adam Kolawa (Parasoft), Static Analysis Best Practices white paper
- Improving Software Security with Precise Static and Runtime Analysis, Benjamin Livshits, section 7.3 "Static Techniques for Security," Stanford doctoral thesis, 2006.
- Flemming Nielson, Hanne R. Nielson, Chris Hankin (1999, corrected 2004). *Principles of Program Analysis*. Springer. ISBN 978-3540654100.

- "Abstract interpretation and static analysis," International Winter School on Semantics and Applications 2003, by David A. Schmidt

External links

- The SAMATE Project, a resource for Automated Static Analysis tools
- Integrate static analysis into a software development process
- Code Quality Improvement - Coding standards conformance checking (DDJ)
- Episode 59: Static Code Analysis Interview (Podcast) at *Software Engineering Radio*
- Implementing Automated Governance for Coding Standards Explains why and how to integrate static code analysis into the build process

Metrics

A **software metric** is a measure of some property of a piece of software or its specifications. Since quantitative measurements are essential in all sciences, there is a continuous effort by computer science practitioners and theoreticians to bring similar approaches to software development. The goal is obtaining objective, reproducible and quantifiable measurements, which may have numerous valuable applications in schedule and budget planning, cost estimation, quality assurance testing, software debugging, software performance optimization, and optimal personnel task assignments.

Common software measurements

Common software measurements include:

- Balanced scorecard
- Bugs per line of code
- COCOMO
- Code coverage
- Cohesion
- Comment density[1]
- Connascent software components
- Coupling
- Cyclomatic complexity
- Function point analysis
- Halstead Complexity
- Instruction path length
- Number of classes and interfaces
- Number of lines of code
- Number of lines of customer requirements
- Program execution time
- Program load time
- Binary file|Program size (binary)
- Robert Cecil Martin's software package metrics

- Weighted Micro Function Points

Limitations

As software development is a complex process, with high variance on both methodologies and objectives, it is difficult to define or measure software qualities and quantities and to determine a valid and concurrent measurement metric, especially when making such a prediction prior to the detail design. Another source of difficulty and debate is in determining which metrics matter, and what they mean.[2][3] The practical utility of *software* measurements has thus been limited to narrow domains where they include:

- Schedule
- Size/Complexity
- Cost
- Quality

Common goal of measurement may target one or more of the above aspects, or the balance between them as indicator of team's motivation or project performance.

Acceptance and Public Opinion

Some software development practitioners point out that simplistic measurements can cause more harm than good.[4] Others have noted that metrics have become an integral part of the software development process.[2] Impact of measurement on programmers psychology have raised concerns for harmful effects to performance due to stress, performance anxiety, and attempts to cheat the metrics, while others find it to have positive impact on developers value towards their own work, and prevent them being undervalued.[5] Some argue that the definition of many measurement methodologies are imprecise, and consequently it is often unclear how tools for computing them arrive at a particular result,[6] while others argue that imperfect quantification is better than none ("You can't control what you can't measure.")[7]. Evidence shows that software metrics are being widely used by government agencies, the US military, NASA[8], IT consultants, academic institutions[9], and commercial and academic development estimation software.

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.

4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

External links

- "Minimal Essential Software Quality Metrics" by Vijayan Reddy. Covers a minimal set of essential metrics for a successful product delivery.
- Definitions of software metrics in .NET
- International Function Point Users Group
- What is FPA at Nesma website
- Estimating With Use Case Points by Mike Cohn. Describes the process to measure the size of an application modeled with UML, using use cases.
- OO & Agile Metrics Resources - includes workshop material on gaming metrics to improve their design
- Further defines the term Software Metrics with examples.
- Software Engineering Metrics: What do they measure and how do we know - An intellectually rigorous treatment of software engineering metrics

Software Package Metrics

This article describes various **software package metrics**. They have been mentioned by Robert Cecil Martin in his *Agile Software Development: Principles, Patterns, and Practices* book (2002).

The term *software package*, as it is used here, refers to a group of related classes (in the field of object-oriented programming).

- **Number of Classes and Interfaces:** The number of concrete and abstract classes (and interfaces) in the package is an indicator of the extensibility of the package.
- **Afferent Couplings (Ca):** The number of other packages that depend upon classes within the package is an indicator of the package's responsibility.
- **Efferent Couplings (Ce):** The number of other packages that the classes in the package depend upon is an indicator of the package's independence.
- **Abstractness (A):** The ratio of the number of abstract classes (and interfaces) in the analyzed package to the total number of classes in the analyzed package. The range for this metric is 0 to 1, with A=0 indicating a completely concrete package and A=1 indicating a completely abstract package.
- **Instability (I):** The ratio of efferent coupling (Ce) to total coupling (Ce + Ca) such that I = Ce / (Ce + Ca). This metric is an indicator of the package's resilience to change. The range for this metric is 0 to 1, with I=0 indicating a completely stable package and I=1 indicating a completely instable package.
- **Distance from the Main Sequence (D):** The perpendicular distance of a package from the idealized line A + I = 1. This metric is an indicator of the package's balance between abstractness and stability. A package squarely on the main sequence is optimally balanced with respect to its abstractness and stability. Ideal packages are either completely abstract

and stable (x=0, y=1) or completely concrete and instable (x=1, y=0). The range for this metric is 0 to 1, with D=0 indicating a package that is coincident with the main sequence and D=1 indicating a package that is as far from the main sequence as possible.

- **Package Dependency Cycles:** Package dependency cycles are reported along with the hierarchical paths of packages participating in package dependency cycles.

References

- Robert Cecil Martin (2002). *Agile Software Development: Principles, Patterns and Practices*. Pearson Education. ISBN 0-13-597444-5.

External links

- OO Metrics tutorial explains package metrics with examples
- JHawk - Java Metrics tool, All the most important code metrics. Eclipse, stand alone and command line versions
- Lattix - Architecture tool that supports a variety of architecture metrics including package dependency metrics.
- NDepend - .NET application that supports the package dependency metrics.
- CppDepend - C++ Metrics tool that supports all the most important code metrics.
- JDepend - Java application that supports the package dependency metrics.
- STAN - Structure Analysis for Java. Eclipse integrated and standalone visual dependency analysis, quality metrics and reporting.
- SourceMonitor - Something for C++, C, C#, VB.NET, Java, Delphi, Visual Basic (VB6)
- PHP Depend - PHP version of JDepend that supports the package dependency metrics.

Visualization

Software visualization[10] is the static or animated 2-D or 3-D[11] visual representation of information about software systems based on their structure,[12] size,[13] history,[14] or behavior.[15]

Typically, the information used for visualization is software metric data from measurement activities or from reverse engineering. Visualization is inherently not a method for software quality assurance but can be used to manually discover anomalies similar to the process of visual data mining.[16]

The objectives of software visualizations are to support the understanding of software systems (i.e., its structure) and algorithms (e.g., by animating the behavior of sorting algorithms) as well as the analysis of software systems and their anomalies (e.g., by showing classes with high coupling).

Types

NDepend Graph and Dependency Matrix interaction

Single component

Tool for software visualization might be used to visualize source code and quality defects during software development and maintenance activities. Their target is the automatic discovery and visualization of quality defects in object-oriented software systems and services. Designed as a plugin for an IDE (e.g., Visual Studio, Eclipse) they visualized the direct relationship of a class and its methods with other classes in the software system and mark potential quality defects to warn the developer. A further benefit is the support for visual navigation through the software system.

Whole (sub-)systems

Other more powerful tools are used to visualize a whole system or subsystem to explore the architecture or to apply visual data mining or visual analytics techniques for defect discovery.

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

Further reading

- Diehl, S. (2002). *Software Visualization*. International Seminar. Revised Papers (LNCS Vol. 2269), Dagstuhl Castle, Germany, 20-25 May 2001 (Dagstuhl Seminar Proceedings).
- Diehl, S. (2007). *Software Visualization — Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007, ISBN 978-3-540-46504-1
- Gîrba, T., Kuhn, A., Seeberger, M., and Ducasse, S., "How Developers Drive Software Evolution," Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2005), IEEE Computer Society Press, 2005, pp. 113–122. PDF
- Keim, D. A. (2002). *Information visualization and visual data mining*. IEEE Transactions on Visualization and Computer Graphics, USA * vol 8 (Jan. March 2002), no 1, p 1 8, 67 refs.
- Knight, C. (2002). *System and Software Visualization*. In *Handbook of software engineering & knowledge engineering*. Vol. 2, Emerging technologies (Vol. 2): World Scientific Publishing Company.
- Kuhn, A., and Greevy, O., "Exploiting the Analogy Between Traces and Signal Processing," Proceedings IEEE International Conference on Software Maintenance (ICSM 2006), IEEE Computer Society Press, Los Alamitos CA, September 2006. PDF
- Lanza, M. (2004). *CodeCrawler — polymetric views in action*. Proceedings. 19th International Conference on Automated Software Engineering, Linz, Austria, 20 24 Sept. 2004 * Los Alamitos, CA, USA: IEEE Comput. Soc, 2004, p 394 5.
- Lopez, F. L., Robles, G., & Gonzalez, B. J. M. (2004). *Applying social network analysis to the information in CVS repositories*. "International Workshop on Mining Software Repositories (MSR 2004)" W17S Workshop 26th International Conference on Software Engineering, Edinburgh, Scotland, UK, 25 May 2004 * Stevenage, UK: IEE, 2004, p 101 5.
- Marcus, A., Feng, L., & Maletic, J. I. (2003). *3D representations for software visualization*. Paper presented at the Proceedings of the 2003 ACM symposium on Software visualization, San Diego, California.
- Soukup, T. (2002). *Visual data mining : techniques and tools for data visualization and mining*. New York: Chichester.
- Staples, M. L., & Bieman, J. M. (1999). *3-D Visualization of Software Structure*. In *Advances in Computers* (Vol. 49, pp. 96–143): Academic Press, London.
- Stasko, J. T., Brown, M. H., & Price, B. A. (1997). *Software Visualization*: MIT Press.
- Van Rysselberghe, F. (2004). *Studying Software Evolution Information By Visualizing the Change History*. Proceedings. 20th International Conference On Software Maintenance. pp 328–337, IEEE Computer Society Press, 2004
- Wettel, R., and Lanza, M., *Visualizing Software Systems as Cities*. In Proceedings of VISSOFT 2007 (4th IEEE International Workshop on Visualizing Software For Understanding and Analysis), pp. 92 – 99, IEEE Computer Society Press, 2007.

External links

- EPDV Eclipse Project Dependencies Viewer
- SoftVis is the second meeting in a planned series of biennial conferences.
- The Program Visualization Workshops aim to bring together researchers who design and construct program, algorithm, or data structure visualizations or animations as well as educators who use or evaluate visualization or animations in their teaching.
- CppDepend - useful C++ tool to visualize dependencies.

Code Review

Code review is systematic examination (often as peer review) of computer source code. It is intended to find and fix mistakes overlooked in the initial development phase, improving both the overall quality of software and the developers' skills. Reviews are done in various forms such as pair programming, informal walkthroughs, and formal inspections.[17]

Introduction

Code reviews can often find and remove common vulnerabilities such as format string exploits, race conditions, memory leaks and buffer overflows, thereby improving software security. Online software repositories based on Subversion (with Redmine or Trac), Mercurial, Git or others allow groups of individuals to collaboratively review code. Additionally, specific tools for collaborative code review can facilitate the code review process.

Automated code reviewing software lessens the task of reviewing large chunks of code on the developer by systematically checking source code for known vulnerabilities.

Capers Jones' ongoing analysis of over 12,000 software development projects showed that the latent defect discovery rate of formal inspection is in the 60-65% range. For informal inspection, the figure is less than 50%. [*citation needed*] The latent defect discovery rate for most forms of testing is about 30%. [18]

Typical code review rates are about 150 lines of code per hour. Inspecting and reviewing more than a few hundred lines of code per hour for critical software (such as safety critical embedded software) may be too fast to find errors. [19] Industry data indicate that code review can accomplish at most an 85% defect removal rate with an average rate of about 65%. [20]

Types

Code review practices fall into three main categories: pair programming, formal code review and lightweight code review.[17]

Formal code review, such as a Fagan inspection, involves a careful and detailed process with multiple participants and multiple phases. Formal code reviews are the traditional method of review, in which software developers attend a series of meetings and review code line by line, usually using printed copies of the material. Formal inspections are extremely thorough and have been proven effective at finding defects in the code under review.

Lightweight code review typically requires less overhead than formal code inspections, though it can be equally effective when done properly.[*citation needed*] Lightweight reviews are often conducted as part of the normal development process:

- Over-the-shoulder – One developer looks over the author's shoulder as the latter walks through the code.

- Email pass-around – Source code management system emails code to reviewers automatically after checkin is made.
- Pair Programming – Two authors develop code together at the same workstation, such is common in Extreme Programming.
- Tool-assisted code review – Authors and reviewers use specialized tools designed for peer code review.

Some of these may also be labeled a "Walkthrough" (informal) or "Critique" (fast and informal).

Many teams that eschew traditional, formal code review use one of the above forms of lightweight review as part of their normal development process. A code review case study published in the book *Best Kept Secrets of Peer Code Review* found that lightweight reviews uncovered as many bugs as formal reviews, but were faster and more cost-effective.

Criticism

Historically, formal code reviews have required a considerable investment in preparation for the review event and execution time.

Some believe that skillful, disciplined use of a number of other development practices can result in similarly high latent defect discovery/avoidance rates. Further, XP proponents might argue, layering additional XP practices, such as refactoring and test-driven development will result in latent defect levels rivaling those achievable with more traditional approaches, without the investment. [*citation needed*]

Use of code analysis tools can support this activity. Especially tools that work in the IDE as they provide direct feedback to developers of coding standard compliance.

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

- Jason Cohen (2006). *Best Kept Secrets of Peer Code Review (Modern Approach. Practical Advice.)*. Smartbearsoftware.com. ISBN 1599160676.

External links

- *Security Code Review FAQs*
- Security code review guidelines
- Lightweight Tool Support for Effective Code Reviews white paper
- Code Review Best Practices white paper by Adam Kolawa
- Best Practices for Peer Code Review white paper
- Code review case study
- "A Guide to Code Inspections" (Jack G. Ganssle)
- Article Four Ways to a Practical Code Review

Code Inspection

Inspection in software engineering, refers to peer review of any work product by trained individuals who look for defects using a well defined process. An inspection might also be referred to as a Fagan inspection after Michael Fagan, the creator of a very popular software inspection process.

Introduction

An inspection is one of the most common sorts of review practices found in software projects. The goal of the inspection is for all of the inspectors to reach consensus on a work product and approve it for use in the project. Commonly inspected work products include software requirements specifications and test plans. In an inspection, a work product is selected for review and a team is gathered for an inspection meeting to review the work product. A moderator is chosen to moderate the meeting. Each inspector prepares for the meeting by reading the work product and noting each defect. **The goal of the inspection is to identify defects**. In an inspection, a defect is any part of the work product that will keep an inspector from approving it. For example, if the team is inspecting a software requirements specification, each defect will be text in the document which an inspector disagrees with.

The process

The inspection process was developed by Michael Fagan in the mid-1970s and it has later been extended and modified.

The process should have entry criteria that determine if the inspection process is ready to begin. This prevents unfinished work products from entering the inspection process. The entry criteria might be a checklist including items such as "The document has been spell-checked".

The stages in the inspections process are: Planning, Overview meeting, Preparation, Inspection meeting, Rework and Follow-up. The Preparation, Inspection meeting and Rework stages might be iterated.

- **Planning:** The inspection is planned by the moderator.

- **Overview meeting:** The author describes the background of the work product.
- **Preparation:** Each inspector examines the work product to identify possible defects.
- **Inspection meeting:** During this meeting the reader reads through the work product, part by part and the inspectors point out the defects for every part.
- **Rework:** The author makes changes to the work product according to the action plans from the inspection meeting.
- **Follow-up:** The changes by the author are checked to make sure everything is correct.

The process is ended by the moderator when it satisfies some predefined exit criteria.

Inspection roles

During an inspection the following roles are used.

- **Author:** The person who created the work product being inspected.
- **Moderator:** This is the leader of the inspection. The moderator plans the inspection and coordinates it.
- **Reader:** The person reading through the documents, one item at a time. The other inspectors then point out defects.
- **Recorder/Scribe:** The person that documents the defects that are found during the inspection.
- **Inspector:** The person that examines the work product to identify possible defects.

Related inspection types

Code review

A code review can be done as a special kind of inspection in which the team examines a sample of code and fixes any defects in it. In a code review, a defect is a block of code which does not properly implement its requirements, which does not function as the programmer intended, or which is not incorrect but could be improved (for example, it could be made more readable or its performance could be improved). In addition to helping teams find and fix bugs, code reviews are useful for both cross-training programmers on the code being reviewed and for helping junior developers learn new programming techniques.

Peer Reviews

Peer Reviews are considered an industry best-practice for detecting software defects early and learning about software artifacts. Peer Reviews are composed of software walkthroughs and software inspections and are integral to software product engineering activities. A collection of coordinated knowledge, skills, and behaviors facilitates the best possible practice of Peer Reviews. The elements of Peer Reviews include the structured review process, standard of excellence product checklists, defined roles of participants, and the forms and reports.

Software inspections are the most rigorous form of Peer Reviews and fully utilize these elements in detecting defects. Software walkthroughs draw selectively upon the elements in assisting the producer to obtain the deepest understanding of an artifact and reaching a consensus among participants. Measured results reveal that Peer Reviews produce an attractive return on investment obtained through accelerated learning and early defect detection. For best results, Peer Reviews are rolled out within an organization through a defined program of preparing a policy and procedure, training practitioners and managers, defining measurements and populating a database structure, and sustaining the roll out infrastructure.

External links

- Review and inspection practices
- Article Software Inspections by Ron Radice
- Comparison of different inspection and review techniques

Deployment & Maintenance

Introduction

Software deployment is all of the activities that make a software system available for use.

The general deployment process consists of several interrelated activities with possible transitions between them. These activities can occur at the producer site or at the consumer site or both. Because every software system is unique, the precise processes or procedures within each activity can hardly be defined. Therefore, "deployment" should be interpreted as a *general process* that has to be customized according to specific requirements or characteristics. A brief description of each activity will be presented later.

Deployment activities

Release
: The release activity follows from the completed development process. It includes all the operations to prepare a system for assembly and transfer to the customer site. Therefore, it must determine the resources required to operate at the customer site and collect information for carrying out subsequent activities of deployment process.

Install and activate
: Activation is the activity of starting up the executable component of software. For simple system, it involves establishing some form of command for execution. For complex systems, it should make all the supporting systems ready to use.

  In larger software deployments, the working copy of the software might be installed on a production server in a production environment. Other versions of the deployed software may be installed in a test environment, development environment and disaster recovery environment.

Deactivate
: Deactivation is the inverse of activation, and refers to shutting down any executing components of a system. Deactivation is often required to perform other deployment activities, e.g., a software system may need to be deactivated before an update can be performed. The practice of removing infrequently used or obsolete systems from service is often referred to as application retirement or application decommissioning.

Adapt
: The adaptation activity is also a process to modify a software system that has been previously installed. It differs from updating in that adaptations are initiated by local events such as changing the environment of customer site, while updating is mostly started from remote software producer.

Update
: The update process replaces an earlier version of all or part of a software system with a newer release.

Built-In
: Mechanisms for installing updates are built into some software systems. Automation of these update processes ranges from fully automatic to user initiated and controlled. Norton Internet

Security is an example of a system with a semi-automatic method for retrieving and installing updates to both the antivirus definitions and other components of the system. Other software products provide query mechanisms for determining when updates are available.

Version tracking

Version tracking systems help the user find and install updates to software systems installed on PCs and local networks.

- Web based version tracking systems notify the user when updates are available for software systems installed on a local system. For example: VersionTracker Pro checks software versions on a user's computer and then queries its database to see if any updates are available.
- Local version tracking system notifies the user when updates are available for software systems installed on a local system. For example: Software Catalog stores version and other information for each software package installed on a local system. One click of a button launches a browser window to the upgrade web page for the application, including auto-filling of the user name and password for sites that require a login.
- Browser based version tracking systems notify the user when updates are available for software packages installed on a local system. For example: wfx-Versions is a Firefox extension which helps the user find the current version number of any program listed on the web.

Uninstall

Uninstallation is the inverse of installation. It is the removal of a system that is no longer required. It also involves some reconfiguration of other software systems in order to remove the uninstalled system’s files and dependencies.

Retire

Ultimately, a software system is marked as obsolete and support by the producers is withdrawn. It is the end of the life cycle of a software product.

Deployment roles

The complexity and variability of software products has necessitated the creation of specialized roles for coordinating and engineering the deployment process. For desktop systems, an end user is frequently also the "software deployer" when they install the software package on their machine. For enterprise software, there are many more roles involved. Additionally, the roles involved typically change as the application progresses from test (pre-production) to production environments. The typical roles involved in software deployments for enterprise applications are:

- Pre-production environments
  - Application developers: see Software development process
  - Build and release engineers: see Release engineering
  - Release managers: see Release management
  - Deployment coordinators: see DevOps
- Production environments
  - System administrator
  - Database administrator
  - Release coordinators: see DevOps
  - Operations project managers: see Information Technology Infrastructure Library

Examples

- FAI OpenSource Software Linux
- M23 OpenSource Software Linux
- Open PC Server Integration (opsi) OpenSource Software Windows
- RPM with YUM OpenSource Software Linux
- MS SCCM Microsoft Windows
- HP OpenView (Hewlett-Packard)
- Tivoli Provisioning Manager and IBM Tivoli Intelligent Orchestrator
- DX-Union (Materna)
- Novell ZENworks (Novell) Zero Effort Networks
- Garibaldi (Software) (INOSOFT AG)
- Client Management Suite (Baramundi Software AG, Augsburg)
- Blackberry MDS Suite Research In Motion (RIM)
- Intellisync Mobile Suite Nokia
- Mobile Device Manager 2008 Microsoft
- ubi-Suite ubitexx
- Java Web Start

References

External links

- Standardization efforts
  - Solution Installation Schema Submission request to W3C
  - OASIS Solution Deployment Descriptor TC
  - OMG Specification for Deployment and Configuration of Component-based Distributed Applications (OMG D&C)
  - JSR 88: Java EE Application Deployment
- Articles
  - The Future of Software Delivery - free developerWorks whitepaper
  - Carzaniga A., Fuggetta A., Hall R. S., Van Der Hoek A., Heimbigner D., Wolf A. L. — A Characterization Framework for Software Deployment Technologies — Technical Report CU-CS-857-98, Dept. of Computer Science, University of Colorado, April 1998. http://serl.cs.colorado.edu/~carzanig/papers/CU-CS-857-98.pdf
- Resources
  - Microsoft's resource page on Client Deployment

Maintenance

Software maintenance in software engineering is the modification of a software product after delivery to correct faults, to improve performance or other attributes.[21]

A common perception of maintenance is that it is merely fixing bugs. However, studies and surveys over the years have indicated that the majority, over 80%, of the maintenance effort is used for non-corrective actions (Pigosky 1997). This perception is perpetuated by users submitting problem reports that in reality are functionality enhancements to the system.

Software maintenance and evolution of systems was first addressed by Meir M. Lehman in 1969. Over a period of twenty years, his research led to the formulation of eight Laws of Evolution (Lehman 1997). Key findings of his research include that maintenance is really evolutionary developments and that maintenance decisions are aided by understanding what happens to systems (and software) over time. Lehman demonstrated that systems continue to evolve over time. As they evolve, they grow more complex unless some action such as code refactoring is taken to reduce the complexity.

The key software maintenance issues are both managerial and technical. Key management issues are: alignment with customer priorities, staffing, which organization does maintenance, estimating costs. Key technical issues are: limited understanding, impact analysis, testing, maintainability measurement.

Software maintenance processes

This section describes the six software maintenance processes as:

1. The implementation processes contains software preparation and transition activities, such as the conception and creation of the maintenance plan, the preparation for handling problems identified during development, and the follow-up on product configuration management.
2. The problem and modification analysis process, which is executed once the application has become the responsibility of the maintenance group. The maintenance programmer must analyze each request, confirm it (by reproducing the situation) and check its validity, investigate it and propose a solution, document the request and the solution proposal, and, finally, obtain all the required authorizations to apply the modifications.
3. The process considering the implementation of the modification itself.
4. The process acceptance of the modification, by confirming the modified work with the individual who submitted the request in order to make sure the modification provided a solution.
5. The migration process (platform migration, for example) is exceptional, and is not part of daily maintenance tasks. If the software must be ported to another platform without any change in functionality, this process will be used and a maintenance project team is likely to be assigned to this task.
6. Finally, the last maintenance process, also an event which does not occur on a daily basis, is the retirement of a piece of software.

There are a number of processes, activities and practices that are unique to maintainers, for example:

- Transition: a controlled and coordinated sequence of activities during which a system is transferred progressively from the developer to the maintainer;
- Service Level Agreements (SLAs) and specialized (domain-specific) maintenance contracts negotiated by maintainers;

- Modification Request and Problem Report Help Desk: a problem-handling process used by maintainers to prioritize, documents and route the requests they receive;
- Modification Request acceptance/rejection: modification request work over a certain size/effort/complexity may be rejected by maintainers and rerouted to a developer.

Categories of maintenance in ISO/IEC 14764

E.B. Swanson initially identified three categories of maintenance: corrective, adaptive, and perfective [22]. These have since been updated and ISO/IEC 14764 presents:

- Corrective maintenance: Reactive modification of a software product performed after delivery to correct discovered problems.
- Adaptive maintenance: Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment.
- Perfective maintenance: Modification of a software product after delivery to improve performance or maintainability.
- Preventive maintenance: Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.

There is also a notion of pre-delivery/pre-release maintenance which is all the good things you do to lower the total cost of ownership of the software. Things like compliance with coding standards that includes software maintainability goals. The management of coupling and cohesion of the software. The attainment of software supportability goals (SAE JA1004, JA1005 and JA1006 for example). Note also that some academic institutions are carrying out research to quantify the cost to ongoing software maintenance due to the lack of resources such as design documents and system/software comprehension training and resources (multiply costs by approx. 1.5-2.0 where there is no design data available.).

References

1. ↑ "Descriptive Information (DI) Metric Thresholds". *Land Software Engineering Centre*. http://www.lsec.dnd.ca/qsd_current_version/eng_support/di/metrics.htm. Retrieved 19 October 2010.
2. ↑ ***a*** ***b*** Binstock, Andrew. "Integration Watch: Using metrics effectively". *SD Times*. BZ Media. http://www.sdtimes.com/link/34157. Retrieved 19 October 2010.
3. ↑ Kolawa, Adam. "When, Why, and How: Code Analysis". *The Code Project*. http://www.codeproject.com/KB/interviews/Code_Review.aspx. Retrieved 19 October 2010.
4. ↑ Kaner, Dr. Cem, *Software Engineer Metrics: What do they measure and how do we know?*, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1.2542&rep=rep1&type=pdf
5. ↑ ProjectCodeMeter (2010) "ProjectCodeMeter Users Manual" page 65 [5]
6. ↑ Lincke, Rüdiger; Lundberg, Jonas; Löwe, Welf (2008), "Comparing software metrics tools", *International Symposium on Software Testing and Analysis 2008*: pp. 131–142, http://www.arisa.se/files/LLL-08.pdf
7. ↑ DeMarco, Tom. *Controlling Software Projects: Management, Measurement and Estimation*. ISBN 0-13-171711-1.

8. ↑ NASA Metrics Planning and Reporting Working Group (MPARWG) [6]
9. ↑ USC Center for Systems and Software Engineering [7]
10. ↑ (Diehl, 2002; Diehl, 2007; Knight, 2002)
11. ↑ (Marcus et al., 2003; Wettel et al., 2007)
12. ↑ (Staples & Bieman, 1999)
13. ↑ (Lanza, 2004)
14. ↑ (Girba et al., 2005, Lopez et al., 2004; Van Rysselberghe et al., 2004)
15. ↑ (Kuhn et al., 2006, Stasko et al., 1997)
16. ↑ (Keim, 2002; Soukup, 2002).
17. ↑ ***a*** ***b*** Kolawa, Adam; Huizinga, Dorota (2007). *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press. p. 260. ISBN 0470042125. http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470042125.html.
18. ↑ Jones, Capers; Christof, Ebert (April 2009). "Embedded Software: Facts, Figures, and Future". IEEE Computer Society. http://doi.ieeecomputersociety.org/10.1109/MC.2009.118. Retrieved 2010-10-05.
19. ↑ Ganssle, Jack (February 2010). "A Guide to Code Inspections". The Ganssle Group. http://www.ganssle.com/inspections.pdf. Retrieved 2010-10-05.
20. ↑ Jones, Capers (June 2008). "Measuring Defect Potentials and Defect Removal Efficiency". Crosstalk, The Journal of Defense Software Engineering. http://www.stsc.hill.af.mil/crosstalk/2008/06/0806jones.html. Retrieved 2010-10-05.
21. ↑ ISO/IEC 14764:2006 Software Engineering — Software Life Cycle Processes — Maintenance
22. ↑ E. Burt Swanson, The dimensions of maintenance. Proceedings of the 2nd international conference on Software engineering, San Francisco, 1976, pp 492 — 497

Further reading

- April, Alain; Abran, Alain (2008). *Software Maintenance Management*. New York: Wiley-IEEE. ISBN 978-0470-14707-8.
- Gopalaswamy Ramesh; Ramesh Bhattiprolu (2006). *Software maintenance : effective practices for geographically distributed environments*. New Delhi: Tata McGraw-Hill. ISBN 9780070483453.
- Grubb, Penny; Takang, Armstrong (2003). *Software Maintenance*. New Jersey: World Scientific Publishing. ISBN 9789812384256.
- Lehman, M.M.; Belady, L.A. (1985). *Program evolution : processes of software change*. London: Academic Press Inc. ISBN 0-12-442441-4.
- Page-Jones, Meilir (1980). *The Practical Guide to Structured Systems Design*. New York: Yourdon Press. ISBN 0-917072-17-0.

External links

- [12] Software Continuity : a rating methodology for software sustainability.
- Journal of Software Maintenance
- Software Maintenance Maturity Model

Evolution

Software evolution is the term used in software engineering (specifically software maintenance) to refer to the process of developing software initially, then repeatedly updating it for various reasons.

General introduction

Fred Brooks, in his key book *The Mythical Man-Month*,[1] states that over 90% of the costs of a typical system arise in the maintenance phase, and that any successful piece of software will inevitably be maintained.

In fact, Agile methods stem from maintenance like activities in and around web based technologies, where the bulk of the capability comes from frameworks and standards.[*citation needed*]

Software maintenance address bug fixes and minor enhancements and software evolution focus on adaptation and migration.

Types of software maintenance

E.B. Swanson initially identified three categories of maintenance: corrective, adaptive, and perfective. Four categories of software were then catalogued by Lientz and Swanson (1980) [2]. These have since been updated and normalized internationally in the ISO/IEC 14764:2006:[3]

- *Corrective maintenance*: Reactive modification of a software product performed after delivery to correct discovered problems;
- *Adaptive maintenance*: Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment;
- *Perfective maintenance*: Modification of a software product after delivery to improve performance or maintainability;
- *Preventive maintenance*: Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults.

All of the preceding take place when there is a known requirement for change.

Although these categories were supplemented by many authors like Warren et al. (1999) [*citation needed*] and Chapin (2001)[*citation needed*], the ISO/IEC 14764:2006 international standard has kept the basic four categories.

More recently the description of software maintenance and evolution has been done using ontologies (Kitchemham et al. (1999),[*citation needed*] Derider (2002),[*citation needed*] Vizcaíno 2003, [*citation needed*] Dias (2003),[*citation needed*] and Ruiz (2004)),[*citation needed*] which enrich the description of the many evolution activities.

Lehman's Laws of Software Evolution

Prof. Meir M. Lehman, who worked at Imperial College London from 1972 to 2002, and his colleagues have identified a set of behaviours in the evolution of proprietary software. These behaviours (or observations) are known as *Lehman's Laws*, and there are eight of them:

1. Continuing Change
2. Increasing Complexity
3. Large Program Evolution
4. Invariant Work-Rate
5. Conservation of Familiarity
6. Continuing Growth
7. Declining Quality
8. Feedback System

It is worth mentioning that the laws are believed to apply mainly to monolithic, proprietary software. For example, some empirical observations coming from the study of open source software development appear to challenge some of the laws [*citation needed*].

The laws predict that change is inevitable and not a consequence of bad programming and that there are limits to what a software evolution team can achieve in terms of safely implementing changes and new functionality.

Maturity Models specific to software evolution have been developed to help improve processes to ensure continuous rejuvenation of the software evolves iteratively.

The "global process" that is made by the many stakeholders (e.g. developers, users, their managers) has many feedback loops. The evolution speed is a function of the feedback loop structure and other characteristics of the global system. Process simulation techniques, such as system dynamics can be useful in understanding and managing such global process.

Software evolution is not likely to be Darwinian, Lamarckian or Baldwinian, but an important phenomenon on its own. Giving the increasing dependence on software at all levels of society and economy, the successful evolution of software is becoming increasingly critical. This is an important topic of research that hasn't received much attention.

The evolution of software, because of its rapid path in comparison to other man-made entities, was seen by Lehman as the "fruit fly" of the study of the evolution of artificial systems.

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-

7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

Project Management

Introduction

Software project management is the art and science of planning and leading software projects[4]. It is a sub-discipline of project management in which software projects are planned, monitored and controlled.

History

The history of software project management is closely related to the history of software. Software was developed for dedicated purposes for dedicated machines until the concept of object-oriented programming began to become popular in the 1960's, making *repeatable solutions* possible for the software industry. Dedicated systems could be adapted to other uses thanks to component-based software engineering. Companies quickly understood the relative ease of use that software programming had over hardware circuitry, and the software industry grew very quickly in the 1970's and 1980's. To manage new development efforts, companies applied proven project management methods, but project schedules slipped during test runs, especially when confusion occurred in the gray zone between the user specifications and the delivered software. To be able to avoid these problems, software project management methods focused on matching user requirements to delivered products, in a method known now as the waterfall model. Since then, analysis of software project management failures has shown that the following are the most common causes:[5]

1. Unrealistic or unarticulated project goals
2. Inaccurate estimates of needed resources
3. Badly defined system requirements
4. Poor reporting of the project's status
5. Unmanaged risks
6. Poor communication among customers, developers, and users
7. Use of immature technology
8. Inability to handle the project's complexity
9. Sloppy development practices
10. Poor project management
11. Stakeholder politics
12. Commercial pressures

The first three items in the list above show the difficulties articulating the needs of the client in such a way that proper resources can deliver the proper project goals. Specific software project management tools are useful and often necessary, but the true art in software project management is applying the correct method and then using tools to support the method. Without a method, tools are worthless. Since the 1960's, several proprietary software project management methods have been developed by software manufacturers for their own use, while computer consulting firms have also

developed similar methods for their clients. Today software project management methods are still evolving, but the current trend leads away from the waterfall model to a more cyclic project delivery model that imitates a Software release life cycle.

Software development process

A software development process is concerned primarily with the production aspect of software development, as opposed to the technical aspect, such as software tools. These processes exist primarily for supporting the management of software development, and are generally skewed toward addressing business concerns. Many software development processes can be run in a similar way to general project management processes. Examples are:

- Risk management is the process of measuring or assessing risk and then developing strategies to manage the risk. In general, the strategies employed include transferring the risk to another party, avoiding the risk, reducing the negative effect of the risk, and accepting some or all of the consequences of a particular risk. Risk management in software project management begins with the business case for starting the project, which includes a cost-benefit analysis as well as a list of fallback options for project failure, called a contingency plan.
    - A subset of risk management that is gaining more and more attention is "Opportunity Management", which means the same thing, except that the potential risk outcome will have a positive, rather than a negative impact. Though theoretically handled in the same way, using the term "opportunity" rather than the somewhat negative term "risk" helps to keep a team focussed on possible positive outcomes of any given risk register in their projects, such as spin-off projects, windfalls, and free extra resources.
- Requirements management is the process of identifying, eliciting, documenting, analyzing, tracing, prioritizing and agreeing on requirements and then controlling change and communicating to relevant stakeholders. New or altered computer system[4] Requirements management, which includes Requirements analysis, is an important part of the software engineering process; whereby business analysts or software developers identify the needs or requirements of a client; having identified these requirements they are then in a position to design a solution.
- Change management is the process of identifying, documenting, analyzing, prioritizing and agreeing on changes to scope (project management) and then controlling changes and communicating to relevant stakeholders. Change impact analysis of new or altered scope, which includes Requirements analysis at the change level, is an important part of the software engineering process; whereby business analysts or software developers identify the altered needs or requirements of a client; having identified these requirements they are then in a position to re-design or modify a solution. Theoretically, each change can impact the timeline and budget of a software project, and therefore by definition must include risk-benefit analysis before approval.
- Software configuration management is the process of identifying, and documenting the scope itself, which is the software product underway, including all sub-products and changes and enabling communication of these to relevant stakeholders. In general, the processes employed include version control, naming convention (programming), and software archival agreements.

- Release management is the process of identifying, documenting, prioritizing and agreeing on releases of software and then controlling the release schedule and communicating to relevant stakeholders. Most software projects have access to three software environments to which software can be released; Development, Test, and Production. In very large projects, where distributed teams need to integrate their work before release to users, there will often be more environments for testing, called unit testing, system testing, or integration testing, before release to User acceptance testing (UAT).
    - A subset of release management that is gaining more and more attention is Data Management, as obviously the users can only test based on data that they know, and "real" data is only in the software environment called "production". In order to test their work, programmers must therefore also often create "dummy data" or "data stubs". Traditionally, older versions of a production system were once used for this purpose, but as companies rely more and more on outside contributors for software development, company data may not be released to development teams. In complex environments, datasets may be created that are then migrated across test environments according to a test release schedule, much like the overall software release schedule.

Project planning, monitoring and control

The purpose of project planning is to identify the scope of the project, estimate the work involved, and create a project schedule. Project planning begins with requirements that define the software to be developed. The project plan is then developed to describe the tasks that will lead to completion.

The purpose of project monitoring and control is to keep the team and management up to date on the project's progress. If the project deviates from the plan, then the project manager can take action to correct the problem. Project monitoring and control involves status meetings to gather status from the team. When changes need to be made, change control is used to keep the products up to date.

Issue

In computing, the term **issue** is a unit of work to accomplish an improvement in a system. An issue could be a bug, a requested feature, task, missing documentation, and so forth. The word "issue" is popularly misused in lieu of "problem." This usage is probably related. [*citation needed*]

For example, OpenOffice.org used to call their modified version of BugZilla IssueZilla. As of September 2010, they call their system Issue Tracker.

Problems occur from time to time and fixing them in a timely fashion is essential to achieve correctness of a system and avoid delayed deliveries of products.

Severity levels

Issues are often categorized in terms of **severity levels**. Different companies have different definitions of severities, but some of the most common ones are:

- Critical
- High - The bug or issue affects a crucial part of a system, and must be fixed in order for it to resume normal operation.
- Medium - The bug or issue affects a minor part of a system, but has some impact on its operation. This severity level is assigned when a non-central requirement of a system is affected.
- Low - The bug or issue affects a minor part of a system, and has very little impact on its operation. This severity level is assigned when a non-central requirement of a system (and with lower importance) is affected.
- Cosmetic - The system works correctly, but the appearance does not match the expected one. For example: wrong colors, too much or too little spacing between contents, incorrect font sizes, typos, etc. This is the lowest priority issue.

In many software companies, issues are often investigated by Quality Assurance Analysts when they verify a system for correctness, and then assigned to the developer(s) that are responsible for resolving them. They can also be assigned by system users during the User Acceptance Testing (UAT) phase.

Issues are commonly communicated using Issue or Defect Tracking Systems. In some other cases, emails or instant messengers are used.

Philosophy

As a subdiscipline of project management, some regard the management of software development akin to the management of manufacturing, which can be performed by someone with management skills, but no programming skills. John C. Reynolds rebuts this view, and argues that software development is entirely design work, and compares a manager who cannot program to the managing editor of a newspaper who cannot write.[6]

External links

- Resources on Software Project Management from Steve McConnell: http://www.construx.com/Page.aspx?nid=22
- Resources on Software Project Management from Dan Galorath: http://www.galorath.com/wp/category/project-management

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-

7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

- Jalote, Pankaj (2002). *Software project management in practice*. Addison-Wesley. ISBN 0201737213.

Software Estimation

Software development efforts estimation is the process of predicting the most realistic use of effort required to develop or maintain software based on incomplete, uncertain and/or noisy input. Effort estimates may be used as input to project plans, iteration plans, budgets, investment analyses, pricing processes and bidding rounds.

State-of-practice

Published surveys on estimation practice suggest that expert estimation is the dominant strategy when estimating software development effort[7].

Typically, effort estimates are over-optimistic and there is a strong over-confidence in their accuracy. The mean effort overrun seems to be about 30% and not decreasing over time. For a review of effort estimation error surveys, see [8]. However, the measurement of estimation error is not unproblematic, see Assessing and interpreting the accuracy of effort estimates. The strong over-confidence in the accuracy of the effort estimates is illustrated by the finding that, on average, if a software professional is 90% confident or “almost sure” to include the actual effort in a minimum-maximum interval, the observed frequency of including the actual effort is only 60-70% [9].

Currently the term “effort estimate” is used to denote as different concepts as most likely use of effort (modal value), the effort that corresponds to a probability of 50% of not exceeding (median), the planned effort, the budgeted effort or the effort used to propose a bid or price to the client. This is believed to be unfortunate, because communication problems may occur and because the concepts serve different goals [10] [11].

History

Software researchers and practitioners have been addressing the problems of effort estimation for software development projects since at least the 1960s; see, e.g., work by Farr [12] and Nelson [13].

Most of the research has focused on the construction of formal software effort estimation models. The early models were typically based on regression analysis or mathematically derived from theories from other domains. Since then a high number of model building approaches have been evaluated, such as approaches founded on case-based reasoning, classification and regression trees,

simulation, neural networks, Bayesian statistics, lexical analysis of requirement specifications, genetic programming, linear programming, economic production models, soft computing, fuzzy logic modeling, statistical bootstrapping, and combinations of two or more of these models. The perhaps most common estimation products today, e.g., the formal estimation models COCOMO and SLIM have their basis in estimation research conducted in the 1970s and 1980s. The estimation approaches based on functionality-based size measures, e.g., function points, is also based on research conducted in the 1970s and 1980s, but are re-appearing with modified size measures under different labels, such as "use case points" [14] in the 1990s and COSMIC in the 2000s.

Estimation approaches

There are many ways of categorizing estimation approaches, see for example [15][16]. The top level categories are the following:

- Expert estimation: The quantification step, i.e., the step where the estimate is produced based on judgmental processes.
- Formal estimation model: The quantification step is based on mechanical processes, e.g., the use of a formula derived from historical data.
- Combination-based estimation: The quantification step is based on a judgmental or mechanical combination of estimates from different sources.

Below are examples of estimation approaches within each category.

Estimation approach	Category	Examples of support of implementation of estimation approach
Analogy-based estimation	Formal estimation model	ANGEL, Weighted Micro Function Points
WBS-based (bottom up) estimation	Expert estimation	Project management software, company specific activity templates
Parametric models	Formal estimation model	COCOMO, SLIM, SEER-SEM
Size-based estimation models[17]	Formal estimation model	Function Point Analysis[18], Use Case Analysis, Story points-based estimation in Agile software development
Group estimation	Expert estimation	Planning poker, Wideband Delphi
Mechanical combination	Combination-based estimation	Average of an analogy-based and a Work breakdown structure-based effort estimate
Judgmental combination	Combination-based estimation	Expert judgment based on estimates from a parametric model and group estimation

Selection of estimation approach

The evidence on differences in estimation accuracy of different estimation approaches and models suggest that there is no “best approach” and that the relative accuracy of one approach or model in comparison to another depends strongly on the context [19]. This implies that different organizations benefit from different estimation approaches. Findings, summarized in [20], that may support the selection of estimation approach based on the expected accuracy of an approach include:

- Expert estimation is on average at least as accurate as model-based effort estimation. In particular, situations with unstable relationships and information of high importance not included in the model may suggest use of expert estimation. This assumes, of course, that experts with relevant experience are available.
- Formal estimation models not tailored to a particular organization’s own context, may be very inaccurate. Use of own historical data is consequently crucial if one cannot be sure that the estimation model’s core relationships (e.g., formula parameters) are based on similar project contexts.
- Formal estimation models may be particularly useful in situations where the model is tailored to the organization’s context (either through use of own historical data or that the model is derived from similar projects and contexts), and/or it is likely that the experts’ estimates will be subject to a strong degree of wishful thinking.

The most robust finding, in many forecasting domains, is that combination of estimates from independent sources, preferable applying different approaches, will on average improve the estimation accuracy [21] [22] [23].

In addition, other factors such as ease of understanding and communicating the results of an approach, ease of use of an approach, cost of introduction of an approach should be considered in a selection process.

Uncertainty assessment approaches

The uncertainty of an effort estimate can be described through a prediction interval (PI). An effort PI is based on a stated certainty level and contains a minimum and a maximum effort value. For example, a project leader may estimate that the most likely effort of a project is 1000 work-hours and that it is 90% certain that the actual effort will be between 500 and 2000 work-hours. Then, the interval [500, 2000] work-hours is the 90% PI of the effort estimate of 1000 work-hours. Frequently, other terms are used instead of PI, e.g., prediction bounds, prediction limits, interval prediction, prediction region and, unfortunately, confidence interval. An important difference between confidence interval and PI is that PI refers to the uncertainty of an estimate, while confidence interval usually refers to the uncertainty associated with the parameters of an estimation model or distribution, e.g., the uncertainty of the mean value of a distribution of effort values. The confidence level of a PI refers to the expected (or subjective) probability that the real value is within the predicted interval[24].

There are several possible approaches to calculate effort PIs, e.g., formal approaches based on regression or bootstrapping [25], formal or judgmental approaches based on the distribution of previous estimation error [26], and pure expert judgment of minimum-maximum effort for a given

level of confidence. Expert judgments based on the distribution of previous estimation error has been found to systematically lead to more realistic uncertainty assessment than the traditional minimum-maximum effort intervals in several studies, see for example [27].

Assessing and interpreting the accuracy of effort estimates

The most common measures of the average estimation accuracy is the MMRE (Mean Magnitude of Relative Error), where MRE is defined as:

MRE = |actual effort − estimated effort| / |actual effort|

This measure has been criticized [28] [29] [30] and there are several alternative measures, such as more symmetric measures [31] , Weighted Mean of Quartiles of relative errors (WMQ) [32] and Mean Variation from Estimate (MVFE) [33].

A high estimation error cannot automatically be interpreted as an indicator of low estimation ability. Alternative, competing or complementing, reasons include low cost control of project, high complexity of development work, and more delivered functionality than originally estimated. A framework for improved use and interpretation of estimation error measurement is included in [34].

Psychological issues related to effort estimation

There are many psychological factors potentially explaining the strong tendency towards over-optimistic effort estimates that need to be dealt with to increase accuracy of effort estimates. These factors are essential even when using formal estimation models, because much of the input to these models is judgment-based. Factors that have been demonstrated to be important are: Wishful thinking, anchoring, planning fallacy and cognitive dissonance. A discussion on these and other factors can be found in work by Jørgensen and Grimstad [35].

- It's easy to estimate what you know.
- It's hard to estimate what you know you don't know.
- It's very hard to estimate things that you don't know you don't know.

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.

4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

External links

- Industry Productivity data for Input into Software Development Estimates and guidance and tools for Estimation - International Software Benchmarking Standards Group: http://www.isbsg.org
- Free first-order benchmarking utility from Software Benchmarking Organization: http://www.sw-benchmarking.org/report.php
- Special Interest Group on Software Effort Estimation: http://www.forecastingprinciples.com/Software_Estimation/index.html
- General forecasting principles: http://www.forecastingprinciples.com
- Project estimation tools: http://www.projectmanagementguides.com/TOOLS/project_estimation_tools.html
- Downloadable research papers on effort estimation: http://simula.no/research/engineering/projects/best
- Mike Cohn's Estimating With Use Case Points from article from Methods & Tools: http://www.methodsandtools.com/archive/archive.php?id=25
- Resources on Software Estimation from Steve McConnell: http://www.construx.com/Page.aspx?nid=297
- Resources on Software Estimation from Dan Galorath: http://www.galorath.com/wp/

Cost Estimation

The ability to accurately estimate the time and/or cost taken for a project to come in to its successful conclusion is a serious problem for software engineers. The use of a repeatable, clearly defined and well understood software development process has, in recent years, shown itself to be the most effective method of gaining useful historical data that can be used for statistical estimation. In particular, the act of sampling more frequently, coupled with the loosening of constraints between parts of a project, has allowed more accurate estimation and more rapid development times.

Methods

Popular methods for estimation in software engineering include:

- Analysis Effort method
- COCOMO
- COSYSMO
- Evidence-based Scheduling Refinement of typical agile estimating techniques using minimal measurement and total time accounting.
- Function Point Analysis
- Parametric Estimating
- PRICE Systems Founders of Commercial Parametric models that estimates the scope, cost, effort and schedule for software projects.

- Proxy-based estimating (PROBE) (from the Personal Software Process)
- Program Evaluation and Review Technique (PERT)
- SEER-SEM Parametric Estimation of Effort, Schedule, Cost, Risk. Mimimum time and staffing concepts based on Brooks's law
- SLIM
- The Planning Game (from Extreme Programming)
- Weighted Micro Function Points (WMFP)
- Wideband Delphi

External links

- Software Estimation chapter from Applied Software Project Management (O'Reilly)
- Article Estimating With Use Case Points from Methods & Tools
- The Dynamics of Software Projects Estimation
- Resources on Software Estimation from Steve McConnell
- Links on tools and techniques of software estimation
- Article Estimating techniques throughout the SDLC

Development Speed

Introduction to Software Engineering/Project Management/Development Speed

Tools

Introduction

Basically, for every step in the development process there are tools available.

- **Modelling and Case Tools:** StarUML, objectiF, Visio, ArgoUML
- **Writing Code:** IDEs like Eclipse, Netbeans, Visual Studio; Compilers and Debuggers; SourceControl like CVS, Subversion, Git, Mercurial, SourceSafe, Perforce
- **Testing Code:** Testing frameworks like JUnit, FIT, TestNG, HTMLUnit; Coverage with Clover, NCover; Profiling tools like EclipseProfile, Netbean's Profiler, JProf, JProbe
- **Automation:** Build tools: make, Ant, Maven,
- **Documentation:** JavaDoc, Doxygen, NDoc; Wikis
- **Project Management, Bug Tracking,Continuous Integration:** Trac, Bugzilla, Mantis; CruiseControl, Hudson
- **Re-engineering:** Decompiler: JAD; Obfuscators

Some of these tools we have talked about before, but some we still need to learn about.

Modelling and Case Tools

Example of a CASE tool.

Computer-aided software engineering (**CASE**) is the scientific application of a set of tools and methods to a software system which is meant to result in high-quality, defect-free, and maintainable software products.[36] It also refers to methods for the development of information systems together with automated tools that can be used in the software development process.[37]

Overview

The term "computer-aided software engineering" (CASE) can refer to the software used for the automated development of systems software, i.e., computer code. The CASE functions include analysis, design, and programming. CASE tools automate methods for designing, documenting, and producing structured computer code in the desired programming language.

CASE software supports the software process activities such as requirement engineering, design, program development and testing. Therefore, CASE tools include design editors, data dictionaries, compilers, debuggers, system building tools, etc.

CASE also refers to the methods dedicated to an engineering discipline for the development of information system using automated tools.

CASE is mainly used for the development of quality software which will perform effectively.

History

The ISDOS project at the University of Michigan initiated a great deal of interest in the whole concept of using computer systems to help analysts in the very difficult process of analysing requirements and developing systems. Several papers by Daniel Teichroew fired a whole generation of enthusiasts with the potential of automated systems development. His PSL/PSA tool was a CASE tool although it predated the term. His insights into the power of meta-meta-models was inspiring, particularly to a former student, Dr. Hasan Sayani, currently Professor, Program Director at University of Maryland University College.

Another major thread emerged as a logical extension to the DBMS directory. By extending the range of meta-data held, the attributes of an application could be held within a dictionary and used at runtime. This "active dictionary" became the precursor to the more modern "model driven execution" (MDE) capability. However, the active dictionary did not provide a graphical representation of any of the meta-data. It was the linking of the concept of a dictionary holding analysts' meta-data, as derived from the use of an integrated set of techniques, together with the graphical representation of such data that gave rise to the earlier versions of I-CASE.

The term CASE was originally coined by software company Nastec Corporation of Southfield, Michigan in 1982 with their original integrated graphics and text editor GraphiText, which also was the first microcomputer-based system to use hyperlinks to cross-reference text strings in documents —an early forerunner of today's web page link. GraphiText's successor product, DesignAid, was the first microprocessor-based tool to logically and semantically evaluate software and system design diagrams and build a data dictionary.

Under the direction of Albert F. Case, Jr. vice president for product management and consulting, and Vaughn Frick, director of product management, the DesignAid product suite was expanded to support analysis of a wide range of structured analysis and design methodologies, notably Ed Yourdon and Tom DeMarco, Chris Gane & Trish Sarson, Ward-Mellor (real-time) SA/SD and Warnier-Orr (data driven).

The next entrant into the market was Excelerator from Index Technology in Cambridge, Mass. While DesignAid ran on Convergent Technologies and later Burroughs Ngen networked microcomputers, Index launched Excelerator on the IBM PC/AT platform. While, at the time of launch, and for several years, the IBM platform did not support networking or a centralized database as did the Convergent Technologies or Burroughs machines, the allure of IBM was strong, and Excelerator came to prominence. Hot on the heels of Excelerator were a rash of offerings from companies such as Knowledgeware (James Martin, Fran Tarkenton and Don Addington), Texas Instrument's IEF and Accenture's FOUNDATION toolset (METHOD/1, DESIGN/1, INSTALL/1, FCP).

CASE tools were at their peak in the early 1990s. At the time IBM had proposed AD/Cycle, which was an alliance of software vendors centered around IBM's Software repository using IBM DB2 in mainframe and OS/2:

> *The application development tools can be from several sources: from IBM, from vendors, and from the customers themselves. IBM has entered into relationships with Bachman Information Systems, Index Technology Corporation, and Knowledgeware, Inc. wherein selected products from these vendors will be marketed through an IBM complementary marketing program to provide offerings that will help to achieve complete life-cycle coverage*. [38]

With the decline of the mainframe, AD/Cycle and the Big CASE tools died off, opening the market for the mainstream CASE tools of today. Nearly all of the leaders of the CASE market of the early 1990s ended up being purchased by Computer Associates, including IEW, IEF, ADW, Cayenne, and Learmonth & Burchett Management Systems (LBMS).

Supporting software

Alfonso Fuggetta classified CASE into 3 categories:[39]

1. *Tasks* support only specific tasks in the software process.
2. *Workbenches* support only one or a few activities.
3. *Environments* support (a large part of) the software process.

Workbenches and environments are generally built as collections of tools. Tools can therefore be either stand alone products or components of workbenches and environments.

Tools

CASE tools are a class of software that automate many of the activities involved in various life cycle phases. For example, when establishing the functional requirements of a proposed application, prototyping tools can be used to develop graphic models of application screens to assist end users to

visualize how an application will look after development. Subsequently, system designers can use automated design tools to transform the prototyped functional requirements into detailed design documents. Programmers can then use automated code generators to convert the design documents into code. Automated tools can be used collectively, as mentioned, or individually. For example, prototyping tools could be used to define application requirements that get passed to design technicians who convert the requirements into detailed designs in a traditional manner using flowcharts and narrative documents, without the assistance of automated design software.[40]

Existing CASE tools can be classified along 4 different dimensions:

1. Life-cycle support
2. Integration dimension
3. Construction dimension
4. Knowledge-based CASE dimension[41]

Let us take the meaning of these dimensions along with their examples one by one:

Life-Cycle Based CASE Tools

This dimension classifies CASE Tools on the basis of the activities they support in the information systems life cycle. They can be classified as Upper or Lower CASE tools.

- Upper CASE Tools support strategic planning and construction of concept-level products and ignore the design aspect. They support traditional diagrammatic languages such as ER diagrams, Data flow diagram, Structure charts, Decision Trees, Decision tables, etc.
- Lower CASE Tools concentrate on the back end activities of the software life cycle, such as physical design, debugging, construction, testing, component integration, maintenance, reengineering and reverse engineering.

Integration dimension

Three main CASE Integration dimensions have been proposed:[42]

1. CASE Framework
2. ICASE Tools
3. Integrated Project Support Environment(IPSE)

Workbenches

Workbenches integrate several CASE tools into one application to support specific software-process activities. Hence they achieve:

- a homogeneous and consistent interface (presentation integration).
- easy invocation of tools and tool chains (control integration).
- access to a common data set managed in a centralized way (data integration).

CASE workbenches can be further classified into following 8 classes:[39]

1. Business planning and modeling
2. Analysis and design
3. User-interface development
4. Programming
5. Verification and validation
6. Maintenance and reverse engineering
7. Configuration management
8. Project management

Environments

An environment is a collection of CASE tools and workbenches that supports the software process. CASE environments are classified based on the focus/basis of integration[39]

1. Toolkits
2. Language-centered
3. Integrated
4. Fourth generation
5. Process-centered

Toolkits

Toolkits are loosely integrated collections of products easily extended by aggregating different tools and workbenches. Typically, the support provided by a toolkit is limited to programming, configuration management and project management. And the toolkit itself is environments extended from basic sets of operating system tools, for example, the Unix Programmer's Work Bench and the VMS VAX Set. In addition, toolkits' loose integration requires user to activate tools by explicit invocation or simple control mechanisms. The resulting files are unstructured and could be in different format, therefore the access of file from different tools may require explicit file format conversion. However, since the only constraint for adding a new component is the formats of the files, toolkits can be easily and incrementally extended.[39]

Language-centered

The environment itself is written in the programming language for which it was developed, thus enabling users to reuse, customize and extend the environment. Integration of code in different languages is a major issue for language-centered environments. Lack of process and data integration is also a problem. The strengths of these environments include good level of presentation and control integration. Interlisp, Smalltalk, Rational, and KEE are examples of language-centered environments.[39]

Integrated

These environments achieve presentation integration by providing uniform, consistent, and coherent tool and workbench interfaces. Data integration is achieved through the *repository* concept: they have a specialized database managing all information produced and accessed in the environment. Examples of integrated environment are IBM AD/Cycle and DEC Cohesion.[39]

Fourth-generation

Fourth-generation environments were the first integrated environments. They are sets of tools and workbenches supporting the development of a specific class of program: electronic data processing and business-oriented applications. In general, they include programming tools, simple configuration management tools, document handling facilities and, sometimes, a code generator to produce code in lower level languages. Informix 4GL, and Focus fall into this category.[39]

Process-centered

Environments in this category focus on process integration with other integration dimensions as starting points. A process-centered environment operates by interpreting a process model created by specialized tools. They usually consist of tools handling two functions:

- Process-model execution
- Process-model production

Examples are East, Enterprise II, Process Wise, Process Weaver, and Arcadia.[39]

Applications

All aspects of the software development life cycle can be supported by software tools, and so the use of tools from across the spectrum can, arguably, be described as CASE; from project management software through tools for business and functional analysis, system design, code storage, compilers, translation tools, test software, and so on.

However, tools that are concerned with analysis and design, and with using design information to create parts (or all) of the software product, are most frequently thought of as CASE tools. CASE applied, for instance, to a database software product, might normally involve:

- Modeling business / real-world processes and data flow
- Development of data models in the form of entity-relationship diagrams
- Development of process and function descriptions

Risks and associated controls

Common CASE risks and associated controls include:

- *Inadequate standardization*: Linking CASE tools from different vendors (design tool from Company X, programming tool from Company Y) may be difficult if the products do not use standardized code structures and data classifications. File formats can be converted, but usually not economically. Controls include using tools from the same vendor, or using tools based on standard protocols and insisting on demonstrated compatibility. Additionally, if organizations obtain tools for only a portion of the development process, they should consider acquiring them from a vendor that has a full line of products to ensure future compatibility if they add more tools.[40]

- *Unrealistic expectations*: Organizations often implement CASE technologies to reduce development costs. Implementing CASE strategies usually involves high start-up costs. Generally, management must be willing to accept a long-term payback period. Controls include requiring senior managers to define their purpose and strategies for implementing CASE technologies.[40]
- *Slow implementation*: Implementing CASE technologies can involve a significant change from traditional development environments. Typically, organizations should not use CASE tools the first time on critical projects or projects with short deadlines because of the lengthy training process. Additionally, organizations should consider using the tools on smaller, less complex projects and gradually implementing the tools to allow more training time.[40]
- *Weak repository controls*: Failure to adequately control access to CASE repositories may result in security breaches or damage to the work documents, system designs, or code modules stored in the repository. Controls include protecting the repositories with appropriate access, version, and backup controls.[40]

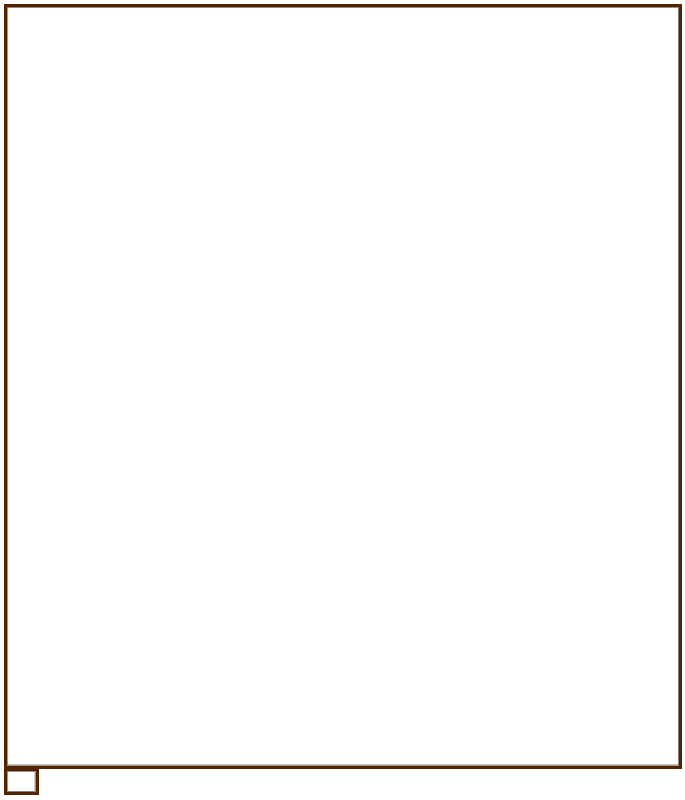
References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

External links

- CASE Tools A CASE tools' community with comments, tags, forums, articles, reviews, etc.
- CASE tool index - A comprehensive list of CASE tools
- UML CASE tools - A comprehensive list of UML CASE tools. Mainly have resources to choose a UML CASE tool and some related to MDA CASE Tools.

Compiler



A diagram of the operation of a typical multi-language, multi-target compiler.

A **compiler** is a computer program (or set of programs) that transforms source code written in a programming language (the *source language*) into another computer language (the *target language*, often having a binary form known as *object code*). The most common reason for wanting to transform source code is to create an executable program.

The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code). If the compiled program can run on a computer whose CPU or operating system is different from the one on which the compiler runs, the compiler is known as a cross-compiler. A program that translates from a low level language to a higher level one is a *decompiler*. A program that translates between high-level languages is usually called a *language translator*, *source to source translator*, or *language converter*. A *language rewriter* is usually a program that translates the form of expressions without a change of language.

A compiler is likely to perform many or all of the following operations: lexical analysis, preprocessing, parsing, semantic analysis (Syntax-directed translation), code generation, and code optimization.

Program faults caused by incorrect compiler behavior can be very difficult to track down and work around; therefore, compiler implementors invest a lot of time ensuring the correctness of their software.

The term compiler-compiler is sometimes used to refer to a parser generator, a tool often used to help create the lexer and parser.

History

Software for early computers was primarily written in assembly language for many years. Higher level programming languages were not invented until the benefits of being able to reuse software on different kinds of CPUs started to become significantly greater than the cost of writing a compiler. The very limited memory capacity of early computers also created many technical problems when implementing a compiler.

Towards the end of the 1950s, machine-independent programming languages were first proposed. Subsequently, several experimental compilers were developed. The first compiler was written by Grace Hopper, in 1952, for the A-0 programming language. The FORTRAN team led by John Backus at IBM is generally credited as having introduced the first complete compiler in 1957. COBOL was an early language to be compiled on multiple architectures, in 1960.[43]

In many application domains the idea of using a higher level language quickly caught on. Because of the expanding functionality supported by newer programming languages and the increasing complexity of computer architectures, compilers have become more and more complex.

Early compilers were written in assembly language. The first *self-hosting* compiler — capable of compiling its own source code in a high-level language — was created for Lisp by Tim Hart and Mike Levin at MIT in 1962.[44] Since the 1970s it has become common practice to implement a compiler in the language it compiles, although both Pascal and C have been popular choices for implementation language. Building a self-hosting compiler is a bootstrapping problem—the first such compiler for a language must be compiled either by a compiler written in a different language, or (as in Hart and Levin's Lisp compiler) compiled by running the compiler in an interpreter.

Compilers in education

Compiler construction and compiler optimization are taught at universities and schools as part of the computer science curriculum. Such courses are usually supplemented with the implementation of a compiler for an educational programming language. A well-documented example is Niklaus Wirth's PL/0 compiler, which Wirth used to teach compiler construction in the 1970s.[45] In spite of its simplicity, the PL/0 compiler introduced several influential concepts to the field:

1. Program development by stepwise refinement (also the title of a 1971 paper by Wirth)[46]
2. The use of a recursive descent parser
3. The use of EBNF to specify the syntax of a language
4. A code generator producing portable P-code
5. The use of T-diagrams[47] in the formal description of the bootstrapping problem

Compilation

Compilers enabled the development of programs that are machine-independent. Before the development of FORTRAN (FORmula TRANslator), the first higher-level language, in the 1950s, machine-dependent assembly language was widely used. While assembly language produces more reusable and relocatable programs than machine code on the same architecture, it has to be modified or rewritten if the program is to be executed on different hardware architecture.

With the advance of high-level programming languages soon followed after FORTRAN, such as COBOL, C, BASIC, programmers can write machine-independent source programs. A compiler translates the high-level source programs into target programs in machine languages for the specific hardwares. Once the target program is generated, the user can execute the program.

The structure of a compiler

Compilers bridge source programs in high-level languages with the underlying hardware. A compiler requires 1) determining the correctness of the syntax of programs, 2) generating correct and efficient object code, 3) run-time organization, and 4) formatting output according to assembler and/or linker conventions. A compiler consists of three main parts: the frontend, the middle-end, and the backend.

The **frontend** checks whether the program is correctly written in terms of the programming language syntax and semantics. Here legal and illegal programs are recognized. Errors are reported, if any, in a useful way. Type checking is also performed by collecting type information. The frontend then generates an *intermediate representation* or *IR* of the source code for processing by the middle-end.

The **middle-end** is where optimization takes place. Typical transformations for optimization are removal of useless or unreachable code, discovery and propagation of constant values, relocation of computation to a less frequently executed place (e.g., out of a loop), or specialization of computation based on the context. The middle-end generates another IR for the following backend. Most optimization efforts are focused on this part.

The **backend** is responsible for translating the IR from the middle-end into assembly code. The target instruction(s) are chosen for each IR instruction. Variables are also selected for the registers. Backend utilizes the hardware by figuring out how to keep parallel FUs busy, filling delay slots, and so on. Although most algorithms for optimization are in NP, heuristic techniques are well-developed.

Compiler output

One classification of compilers is by the platform on which their generated code executes. This is known as the *target platform.*

A *native* or *hosted* compiler is one which output is intended to directly run on the same type of computer and operating system that the compiler itself runs on. The output of a cross compiler is designed to run on a different platform. Cross compilers are often used when developing software for embedded systems that are not intended to support a software development environment.

The output of a compiler that produces code for a virtual machine (VM) may or may not be executed on the same platform as the compiler that produced it. For this reason such compilers are not usually classified as native or cross compilers.

Compiled versus interpreted languages

Higher-level programming languages are generally divided for convenience into compiled languages and interpreted languages. However, in practice there is rarely anything about a language that *requires* it to be exclusively compiled or exclusively interpreted, although it is possible to design languages that rely on re-interpretation at run time. The categorization usually reflects the most popular or widespread implementations of a language — for instance, BASIC is sometimes called an interpreted language, and C a compiled one, despite the existence of BASIC compilers and C interpreters.

Modern trends toward just-in-time compilation and bytecode interpretation at times blur the traditional categorizations of compilers and interpreters.

Some language specifications spell out that implementations *must* include a compilation facility; for example, Common Lisp. However, there is nothing inherent in the definition of Common Lisp that stops it from being interpreted. Other languages have features that are very easy to implement in an interpreter, but make writing a compiler much harder; for example, APL, SNOBOL4, and many scripting languages allow programs to construct arbitrary source code at runtime with regular string operations, and then execute that code by passing it to a special evaluation function. To implement these features in a compiled language, programs must usually be shipped with a runtime library that includes a version of the compiler itself.

Hardware compilation

The output of some compilers may target hardware at a very low level, for example a Field Programmable Gate Array (FPGA) or structured Application-specific integrated circuit (ASIC). Such compilers are said to be *hardware compilers* or synthesis tools because the source code they compile effectively control the final configuration of the hardware and how it operates; the output of the compilation are not instructions that are executed in sequence - only an interconnection of transistors or lookup tables. For example, XST is the Xilinx Synthesis Tool used for configuring FPGAs. Similar tools are available from Altera, Synplicity, Synopsys and other vendors.

Compiler construction

In the early days, the approach taken to compiler design used to be directly affected by the complexity of the processing, the experience of the person(s) designing it, and the resources available.

A compiler for a relatively simple language written by one person might be a single, monolithic piece of software. When the source language is large and complex, and high quality output is required, the design may be split into a number of relatively independent phases. Having separate phases means development can be parceled up into small parts and given to different people. It also becomes much easier to replace a single phase by an improved one, or to insert new phases later (e.g., additional optimizations).

The division of the compilation processes into phases was championed by the Production Quality Compiler-Compiler Project (PQCC) at Carnegie Mellon University. This project introduced the terms *front end*, *middle end*, and *back end*.

All but the smallest of compilers have more than two phases. However, these phases are usually regarded as being part of the front end or the back end. The point at which these two *ends* meet is open to debate. The front end is generally considered to be where syntactic and semantic processing takes place, along with translation to a lower level of representation (than source code).

The middle end is usually designed to perform optimizations on a form other than the source code or machine code. This source code/machine code independence is intended to enable generic optimizations to be shared between versions of the compiler supporting different languages and target processors.

The back end takes the output from the middle. It may perform more analysis, transformations and optimizations that are for a particular computer. Then, it generates code for a particular processor and OS.

This front-end/middle/back-end approach makes it possible to combine front ends for different languages with back ends for different CPUs. Practical examples of this approach are the GNU Compiler Collection, LLVM, and the Amsterdam Compiler Kit, which have multiple front-ends, shared analysis and multiple back-ends.

One-pass versus multi-pass compilers

Classifying compilers by number of passes has its background in the hardware resource limitations of computers. Compiling involves performing lots of work and early computers did not have enough memory to contain one program that did all of this work. So compilers were split up into smaller programs which each made a pass over the source (or some representation of it) performing some of the required analysis and translations.

The ability to compile in a single pass has classically been seen as a benefit because it simplifies the job of writing a compiler and one pass compilers generally compile faster than multi-pass compilers. Thus, partly driven by the resource limitations of early systems, many early languages were specifically designed so that they could be compiled in a single pass (e.g., Pascal).

In some cases the design of a language feature may require a compiler to perform more than one pass over the source. For instance, consider a declaration appearing on line 20 of the source which affects the translation of a statement appearing on line 10. In this case, the first pass needs to gather information about declarations appearing after statements that they affect, with the actual translation happening during a subsequent pass.

The disadvantage of compiling in a single pass is that it is not possible to perform many of the sophisticated optimizations needed to generate high quality code. It can be difficult to count exactly how many passes an optimizing compiler makes. For instance, different phases of optimization may analyse one expression many times but only analyse another expression once.

Splitting a compiler up into small programs is a technique used by researchers interested in producing provably correct compilers. Proving the correctness of a set of small programs often requires less effort than proving the correctness of a larger, single, equivalent program.

While the typical multi-pass compiler outputs machine code from its final pass, there are several other types:

- A "source-to-source compiler" is a type of compiler that takes a high level language as its input and outputs a high level language. For example, an automatic parallelizing compiler will frequently take in a high level language program as an input and then transform the code and annotate it with parallel code annotations (e.g. OpenMP) or language constructs (e.g. Fortran's `DOALL` statements).
- Stage compiler that compiles to assembly language of a theoretical machine, like some Prolog implementations
    - This Prolog machine is also known as the Warren Abstract Machine (or WAM).
    - Bytecode compilers for Java, Python, and many more are also a subtype of this.
- Just-in-time compiler, used by Smalltalk and Java systems, and also by Microsoft .NET's Common Intermediate Language (CIL)
    - Applications are delivered in bytecode, which is compiled to native machine code just prior to execution.

Front end

The front end analyzes the source code to build an internal representation of the program, called the intermediate representation or *IR*. It also manages the symbol table, a data structure mapping each symbol in the source code to associated information such as location, type and scope. This is done over several phases, which includes some of the following:

1. **Line reconstruction**. Languages which strop their keywords or allow arbitrary spaces within identifiers require a phase before parsing, which converts the input character sequence to a canonical form ready for the parser. The top-down, recursive-descent, table-driven parsers used in the 1960s typically read the source one character at a time and did not require a separate tokenizing phase. Atlas Autocode, and Imp (and some implementations of ALGOL and Coral 66) are examples of stropped languages which compilers would have a *Line Reconstruction* phase.
2. Lexical analysis breaks the source code text into small pieces called *tokens*. Each token is a single atomic unit of the language, for instance a keyword, identifier or symbol name. The token syntax is typically a regular language, so a finite state automaton constructed from a regular expression can be used to recognize it. This phase is also called lexing or scanning, and the software doing lexical analysis is called a lexical analyzer or scanner.
3. Preprocessing. Some languages, e.g., C, require a preprocessing phase which supports macro substitution and conditional compilation. Typically the preprocessing phase occurs before syntactic or semantic analysis; e.g. in the case of C, the preprocessor manipulates lexical tokens rather than syntactic forms. However, some languages such as Scheme support macro substitutions based on syntactic forms.

4. Syntax analysis involves parsing the token sequence to identify the syntactic structure of the program. This phase typically builds a parse tree, which replaces the linear sequence of tokens with a tree structure built according to the rules of a formal grammar which define the language's syntax. The parse tree is often analyzed, augmented, and transformed by later phases in the compiler.
5. Semantic analysis is the phase in which the compiler adds semantic information to the parse tree and builds the symbol table. This phase performs semantic checks such as type checking (checking for type errors), or object binding (associating variable and function references with their definitions), or definite assignment (requiring all local variables to be initialized before use), rejecting incorrect programs or issuing warnings. Semantic analysis usually requires a complete parse tree, meaning that this phase logically follows the parsing phase, and logically precedes the code generation phase, though it is often possible to fold multiple phases into one pass over the code in a compiler implementation.

Back end

The term *back end* is sometimes confused with *code generator* because of the overlapped functionality of generating assembly code. Some literature uses *middle end* to distinguish the generic analysis and optimization phases in the back end from the machine-dependent code generators.

The main phases of the back end include the following:

1. Analysis: This is the gathering of program information from the intermediate representation derived from the input. Typical analyses are data flow analysis to build use-define chains, dependence analysis, alias analysis, pointer analysis, escape analysis etc. Accurate analysis is the basis for any compiler optimization. The call graph and control flow graph are usually also built during the analysis phase.
2. Optimization: the intermediate language representation is transformed into functionally equivalent but faster (or smaller) forms. Popular optimizations are inline expansion, dead code elimination, constant propagation, loop transformation, register allocation and even automatic parallelization.
3. Code generation: the transformed intermediate language is translated into the output language, usually the native machine language of the system. This involves resource and storage decisions, such as deciding which variables to fit into registers and memory and the selection and scheduling of appropriate machine instructions along with their associated addressing modes (see also Sethi-Ullman algorithm).

Compiler analysis is the prerequisite for any compiler optimization, and they tightly work together. For example, dependence analysis is crucial for loop transformation.

In addition, the scope of compiler analysis and optimizations vary greatly, from as small as a basic block to the procedure/function level, or even over the whole program (interprocedural optimization). Obviously, a compiler can potentially do a better job using a broader view. But that broad view is not free: large scope analysis and optimizations are very costly in terms of compilation time and memory space; this is especially true for interprocedural analysis and optimizations.

Interprocedural analysis and optimizations are common in modern commercial compilers from HP, IBM, SGI, Intel, Microsoft, and Sun Microsystems. The open source GCC was criticized for a long time for lacking powerful interprocedural optimizations, but it is changing in this respect. Another open source compiler with full analysis and optimization infrastructure is Open64, which is used by many organizations for research and commercial purposes.

Due to the extra time and space needed for compiler analysis and optimizations, some compilers skip them by default. Users have to use compilation options to explicitly tell the compiler which optimizations should be enabled.

Compiler correctness

Compiler correctness is the branch of software engineering that deals with trying to show that a compiler behaves according to its language specification.[*citation needed*] Techniques include developing the compiler using formal methods and using rigorous testing (often called compiler validation) on an existing compiler.

Related techniques

Assembly language is a type of low-level language and a program that compiles it is more commonly known as an *assembler*, with the inverse program known as a *disassembler*.

A program that translates from a low level language to a higher level one is a *decompiler*.

A program that translates between high-level languages is usually called a *language translator*, *source to source translator*, *language converter*, or *language rewriter*. The last term is usually applied to translations that do not involve a change of language.

International conferences and organizations

Every year, the **European Joint Conferences on Theory and Practice of Software** (ETAPS) sponsors the **International Conference on Compiler Construction** (CC), with papers from both the academic and industrial sectors.[48]

Notes

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."

3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

References

- Compiler textbook references A collection of references to mainstream Compiler Construction Textbooks
- Aho, Alfred V.; Sethi, Ravi; and Ullman, Jeffrey D., *Compilers: Principles, Techniques and Tools* (ISBN 0-201-10088-6) link to publisher. Also known as "The Dragon Book."
- Allen, Frances E., "A History of Language Processor Technology in IBM", *IBM Journal of Research and Development*, v.25, no.5, September 1981.
- Allen, Randy; and Kennedy, Ken, *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann Publishers, 2001. ISBN 1-55860-286-0
- Appel, Andrew Wilson
    - *Modern Compiler Implementation in Java*, 2nd edition. Cambridge University Press, 2002. ISBN 0-521-82060-X
    - *Modern Compiler Implementation in ML*, Cambridge University Press, 1998. ISBN 0-521-58274-1
- Bornat, Richard, *Understanding and Writing Compilers: A Do It Yourself Guide*, Macmillan Publishing, 1979. ISBN 0-333-21732-2
- Cooper, Keith D., and Torczon, Linda, *Engineering a Compiler*, Morgan Kaufmann, 2004, ISBN 1-55860-699-8.
- Leverett; Cattel; Hobbs; Newcomer; Reiner; Schatz; Wulf, *An Overview of the Production Quality Compiler-Compiler Project*, in *Computer* 13(8):38-49 (August 1980)
- McKeeman, William Marshall; Horning, James J.; Wortman, David B., *A Compiler Generator*, Englewood Cliffs, N.J. : Prentice-Hall, 1970. ISBN 0-13-155077-2
- Muchnick, Steven, *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, 1997. ISBN 1-55860-320-4
- Scott, Michael Lee, *Programming Language Pragmatics*, Morgan Kaufmann, 2005, 2nd edition, 912 pages. ISBN 0-12-633951-1 (The author's site on this book).
- Srikant, Y. N.; Shankar, Priti, *The Compiler Design Handbook: Optimizations and Machine Code Generation*, CRC Press, 2003. ISBN 0-8493-1240-X
- Terry, Patrick D., *Compilers and Compiler Generators: An Introduction with C++*, International Thomson Computer Press, 1997. ISBN 1-85032-298-8,
- Wirth, Niklaus, *Compiler Construction* (ISBN 0-201-40353-6), Addison-Wesley, 1996, 176 pages. Revised November 2005.

External links

Look up ***compiler*** in Wiktionary, the free dictionary.

Also see the *Compiler Construction* book.

- The comp.compilers newsgroup and RSS feed
- Hardware compilation mailing list
- Practical introduction to compiler construction using flex and yacc
- Book "Basics of Compiler Design" by Torben Ægidius Mogensen

A **debugger** or **debugging tool** is a computer program that is used to test and debug other programs (the "target" program). The code to be examined might alternatively be running on an *instruction set simulator* (ISS), a technique that allows great power in its ability to halt when specific conditions are encountered but which will typically be somewhat slower than executing the code directly on the appropriate (or the same) processor. Some debuggers offer two modes of operation - full or partial simulation, to limit this impact.

A "crash" happens when the program cannot normally continue because of a programming bug. For example, the program might have tried to use an instruction not available on the current version of the CPU or attempted to access unavailable or protected memory. When the program "crashes" or reaches a preset condition, the debugger typically shows the position in the original code if it is a **source-level debugger** or **symbolic debugger**, commonly now seen in integrated development environments. If it is a **low-level debugger** or a **machine-language debugger** it shows the line in the disassembly (unless it also has online access to the original source code and can display the appropriate section of code from the assembly or compilation).

Typically, debuggers also offer more sophisticated functions such as running a program step by step (**single-stepping** or program animation), stopping (**breaking**) (pausing the program to examine the current state) at some event or specified instruction by means of a breakpoint, and tracking the values of some variables. Some debuggers have the ability to modify the state of the program while it is running, rather than merely to observe it. It may also be possible to continue execution at a different location in the program to bypass a crash or logical error.

The importance of a good debugger cannot be overstated. Indeed, the existence and quality of such a tool for a given language and platform can often be the deciding factor in its use, even if another language/platform is better-suited to the task.[*citation needed*]. The absence of a debugger, having once been accustomed to using one, has been said to "make you feel like a blind man in a dark room looking for a black cat that isn't there".[49] However, software can (and often does) behave differently running under a debugger than normally, due to the inevitable changes the presence of a debugger will make to a software program's internal timing. As a result, even with a good debugging tool, it is often very difficult to track down runtime problems in complex multi-threaded or distributed systems.

The same functionality which makes a debugger useful for eliminating bugs allows it to be used as a software cracking tool to evade copy protection, digital rights management, and other software protection features. It often also makes it useful as a general testing verification tool test coverage and performance analyzer, especially if instruction path lengths are shown.

Most current mainstream debugging engines, such as gdb and dbx provide console-based command line interfaces. Debugger front-ends are popular extensions to debugger engines that provide IDE integration, program animation, and visualization features. Some early mainframe debuggers such

as IBM OLIVER (CICS interactive test/debug) and SIMON (Batch Interactive test/debug) provided this same functionality for the IBM System/360 and later operating systems, as long ago as the 1970s.

Language dependency

Some debuggers operate on a single specific language while others can handle multiple languages transparently. For example if the main target program is written in COBOL but CALLs Assembler subroutines and also PL/1 subroutines, the debugger may dynamically switch modes to accommodate the changes in language as they occur.

Memory protection

Some debuggers also incorporate memory protection to avoid storage violations such as buffer overflow. This may be extremely important in transaction processing environments where memory is dynamically allocated from memory 'pools' on a task by task basis.

Hardware support for debugging

Most modern microprocessors have at least one of these features in their CPU design to make debugging easier:

- hardware support for single-stepping a program, such as the trap flag.
- An instruction set that meets the Popek and Goldberg virtualization requirements makes it easier to write debugger software that runs on the same CPU as the software being debugged; such a CPU can execute the inner loops of the program under test at full speed, and still remain under the control of the debugger.
- In-System Programming allows an external hardware debugger to re-program a system under test (for example, adding or removing instruction breakpoints). Many systems with such ISP support also have other hardware debug support.
- Hardware support for code and data breakpoints, such as address comparators and data value comparators or, with considerably more work involved, page fault hardware.
- JTAG access to hardware debug interfaces such as those on ARM architecture processors or using the Nexus command set. Processors used in embedded systems typically have extensive JTAG debug support.
- Microcontrollers with as few as six pins need to use low pin-count substitutes for JTAG, such as BDM, Spy-Bi-Wire, or DebugWire on the Atmel AVR. DebugWire, for example, uses bidirectional signaling on the RESET pin.

List of debuggers

Winpdb debugging itself.

- AppPuncher Debugger — for debugging Rich Internet Applications
- AQtime
- CA/EZTEST — was a CICS interactive test/debug software package
- CharmDebug — a Debugger for Charm++
- CodeView
- DBG — a PHP Debugger and Profiler
- dbx
- DDD (Data Display Debugger)
- Distributed Debugging Tool (Allinea DDT)
- DDTLite — Allinea DDTLite for Visual Studio 2008
- DEBUG — the built-in debugger of DOS and Microsoft Windows
- Debugger for MySQL
- Opera Dragonfly
- Dynamic debugging technique (DDT), and its octal counterpart Octal Debugging Technique
- Eclipse
- Embedded System Debug Plug-in for Eclipse
- FusionDebug
- gDEBugger OpenGL, OpenGL ES and OpenCL Debugger and Profiler. For Windows, Linux, Mac OS X and iPhone
- GNU Debugger (GDB), GNU Binutils
- Intel Debugger (IDB)
- Insight
- Parasoft Insure++
- iSYSTEM — In circuit debugger for Embedded Systems
- Interactive Disassembler (IDA Pro)
- Java Platform Debugger Architecture
- Jinx — a whole-system debugger for heisenbugs. It works transparently as a device driver.
- JSwat — open-source Java debugger
- LLDB
- MacsBug
- Nemiver — graphical C/C++ Debugger for the GNOME desktop environment
- OLIVER (CICS interactive test/debug) - a GUI equipped *instruction set simulator* (ISS)
- OllyDbg

- FlexTracer - shareware debugger for SQL statements
- Omniscient Debugger — Forward and backward debugger for Java
- pydbg
- IBM Rational Purify
- RealView Debugger — Commercial debugger produced for and designed by ARM
- sdb
- SIMMON (Simulation Monitor)
- SIMON (Batch Interactive test/debug) — a GUI equipped Instruction Set Simulator for batch
- SoftICE
- TimeMachine — Forward and backward debugger designed by Green Hills Software
- TotalView
- TRACE32 — In circuit debugger for Embedded Systems
- Turbo Debugger
- Ups — C, Fortran source level debugger
- Valgrind
- VB Watch Debugger — debugger for Visual Basic 6.0
- Microsoft Visual Studio Debugger
- WinDbg
- Xdebug — PHP debugger and profiler

Debugger front-ends

Some of the most capable and popular debuggers only implement a simple command line interface (CLI) — often to maximize portability and minimize resource consumption. Debugging via a graphical user interface (GUI) can be considered easier and more productive though. This is the reason for GUI debugger front-ends, that allow users to monitor and control subservient CLI-only debuggers via graphical user interface. Some GUI debugger front-ends are designed to be compatible with a variety of CLI-only debuggers, while others are targeted at one specific debugger.

List of debugger front-ends

- Many Integrated development environments come with integrated debuggers (or front-ends to standard debuggers).
    - Many Eclipse perspectives, e.g. the Java Development Tools (JDT) [13], provide a debugger front-end.
- DDD is the standard front-end from the GNU Project. It is a complex tool that works with most common debuggers (GDB, jdb, Python debugger, Perl debugger, Tcl, and others) natively or with some external programs (for PHP).
- GDB (the GNU debugger) GUI
    - Emacs — Emacs editor with built in support for the GNU Debugger acts as the frontend.
    - KDbg — Part of the KDE development tools.

- Nemiver — A GDB frontend that integrates well in the GNOME desktop environment.
- xxgdb — X-window frontend for GDB and dbx debugger.
- Qt Creator — multi-platform frontend for GDB (debugging example).
- cgdb — ncurses terminal program that mimics vim key mapping.
- ccdebug— A graphical GDB frontend using the Qt toolkit.
- Padb — has a parallel front-end to GDB allowing it to target parallel applications.
- Allinea Distributed Debugging Tool — a parallel and distributed front-end to a modified version of GDB.
- Xcode — contains a GDB front-end as well.
- SlickEdit — contains a GDB front-end as well.
- Eclipse C/C++ Development Tools (CDT) [14] — includes visual debugging tools based on GDB.

References

- Jonathan B. Rosenberg, *How Debuggers Work: Algorithms, Data Structures, and Architecture*, John Wiley & Sons, ISBN 0-471-14966-7

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

External links

Look up ***debugger*** in Wiktionary, the free dictionary.

- Debugging tools at the Open Directory Project
- Debugging Tools for Windows
- OpenRCE: Various Debugger Resources and Plug-ins
- Parallel computing development and debugging tools at the Open Directory Project

IDE

Anjuta, a C and C++ IDE for the GNOME environment

An **integrated development environment (IDE)** (also known as **integrated design environment** or **integrated debugging environment**) is a software application that provides comprehensive facilities to computer programmers for software development. An IDE normally consists of:

- a source code editor
- a compiler and/or an interpreter
- build automation tools
- a debugger

Sometimes a version control system and various tools are integrated to simplify the construction of a GUI. Many modern IDEs also have a class browser, an object inspector, and a class hierarchy diagram, for use with object-oriented software development.[50]

Overview

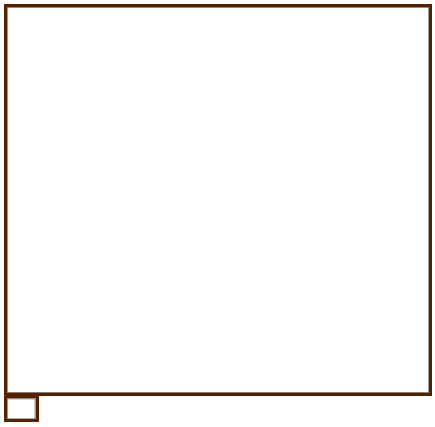
IDEs are designed to maximize programmer productivity by providing tightly-knit components with similar user interfaces. This should mean that the programmer has to do less mode switching versus using discrete development programs. However, because an IDE is a complicated piece of software by its very nature, this higher productivity only occurs after a lengthy learning process.

Typically an IDE is dedicated to a specific programming language, allowing a feature set that most closely matches the programming paradigms of the language. However, there are some multiple-language IDEs, such as Eclipse, ActiveState Komodo, recent versions of NetBeans, Microsoft Visual Studio, WinDev, and Xcode.

IDEs typically present a single program in which all development is done. This program typically provides many features for authoring, modifying, compiling, deploying and debugging software. The aim is to abstract the configuration necessary to piece together command line utilities in a cohesive unit, which theoretically reduces the time to learn a language, and increases developer productivity. It is also thought that the tight integration of development tasks can further increase productivity. For example, code can be compiled while being written, providing instant feedback on syntax errors. While most modern IDEs are graphical, IDEs in use before the advent of windowing

systems (such as Microsoft Windows or X11) were text-based, using function keys or hotkeys to perform various tasks (Turbo Pascal is a common example). This contrasts with software development using unrelated tools, such as vi, GCC or make.

History



GNU Emacs, an extensible editor that is commonly used as an IDE on Unix-like systems

IDEs initially became possible when developing via a console or terminal. Early systems could not support one, since programs were prepared using flowcharts, entering programs with punched cards (or paper tape, etc.) before submitting them to a compiler. Dartmouth BASIC was the first language to be created with an IDE (and was also the first to be designed for use while sitting in front of a console or terminal). Its IDE (part of the Dartmouth Time Sharing System) was command-based, and therefore did not look much like the menu-driven, graphical IDEs prevalent today. However it integrated editing, file management, compilation, debugging and execution in a manner consistent with a modern IDE.

Keyboard Maestro [51]

Maestro I is a product from Softlab Munich and was the world's first integrated development environment[52] 1975 for software. Maestro I was installed for 22,000 programmers worldwide. Until 1989, 6,000 installations existed in the Federal Republic of Germany. Maestro I was arguably the world leader in this field during the 1970s and 1980s. Today one of the last Maestro I can be found in the Museum of Information Technology at Arlington.

One of the first IDEs with a plug-in concept was Softbench. In 1995 Computerwoche commented that the use of an IDE was not well received by developers since it would fence in their creativity.

Topics

Visual programming

Visual programming is a usage scenario in which an IDE is generally required. Visual IDEs allow users to create new applications by moving programming building blocks or code nodes to create flowcharts or structure diagrams that are then compiled or interpreted. These flowcharts often are based on the Unified Modeling Language.

This interface has been popularized with the Lego Mindstorms system, and is being actively pursued by a number of companies wishing to capitalize on the power of custom browsers like those found at Mozilla. KTechlab supports flowcode and is a popular opensource IDE and Simulator for developing software for microcontrollers. Visual programming is also responsible for the power of distributed programming (cf. LabVIEW and EICASLAB software). An early visual programming system, Max, was modelled after analog synthesizer design and has been used to develop real-time music performance software since the 1980s. Another early example was Prograph, a dataflow-based system originally developed for the Macintosh. The graphical programming environment "Grape" is used to program qfix robot kits.

This approach is also used in specialist software such as Openlab, where the end users want the flexibility of a full programming language, without the traditional learning curve associated with one.

An open source visual programming system is Mindscript, which has extended functionality for cryptology, database interfacing,

Language support

Some IDEs support multiple languages, such as Eclipse or NetBeans, both based on Java, or MonoDevelop, based on C#.

Support for alternative languages is often provided by plugins, allowing them to be installed on the same IDE at the same time. For example, Eclipse and Netbeans have plugins for C/C++, Ada, Perl, Python, Ruby, and PHP, among other languages.

Attitudes across different computing platforms

Many Unix programmers argue that traditional command-line POSIX tools constitute an IDE,Template:Who though one with a different style of interface and under the Unix environment. Many programmers still use makefiles and their derivatives. Also, many Unix programmers use Emacs or Vim, which integrates support for many of the standard Unix build tools. Data Display Debugger is intended to be an advanced graphical front-end for many text-based debugger standard tools.

On the various Microsoft Windows platforms, command-line tools for development are seldom used. Accordingly, there are many commercial and non-commercial solutions, however each has a different design commonly creating incompatibilities. Most major compiler vendors for Windows still provide free copies of their command-line tools, including Microsoft (Visual C++, Platform SDK, Microsoft .NET Framework SDK, nmake utility), Embarcadero Technologies (bcc32 compiler, make utility).

Additionally, the free software GNU tools (gcc, gdb, GNU make) are available on many platforms, including Windows etc.

IDEs have always been popular on the Apple Macintosh's Mac OS, dating back to Macintosh Programmer's Workshop, Turbo Pascal, THINK Pascal and THINK C environments in the mid-1980s. Currently Mac OS X programmers can choose between limited IDEs, including native IDEs like Xcode, older IDEs like CodeWarrior, and open-source tools, such as Eclipse and Netbeans. ActiveState Komodo is a proprietary IDE supported on the Mac OS.

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

GUI Builder

A **graphical user interface builder** (or **GUI builder**), also known as **GUI designer**, is a software development tool that simplifies the creation of GUIs by allowing the designer to arrange widgets using a drag-and-drop WYSIWYG editor. Without a GUI builder, a GUI must be built by manually specifying each widget's parameters in code, with no visual feedback until the program is run.

User interfaces are commonly programmed using an event-driven architecture, so GUI builders also simplify creating event-driven code. This supporting code connects widgets with the outgoing and incoming events that trigger the functions providing the application logic.

List of GUI builders

Programs

- AutoIt
- Axure RP
- Cocoa/OpenStep
    - Interface Builder
- Embedded Wizard a commercial development tool focussed on user interface applications for embedded systems.
- Fast, Light Toolkit (FLTK)
    - FLUID
- GNUstep
    - Gorm
- GEM
    - Resource construction set
    - Interface by Shift Computer
    - ORCS (Otto's RCS)
    - K-Resource
    - Resource Master
    - Annabel Junior
    - WERCS by HiSoft
- GTK+
    - Glade Interface Designer
    - Gazpacho
    - Gideon Designer
- GUI Builder
- Intrinsics
- Justinmind Prototyper
    - Motif
        - Builder Xcessory
        - Easymotif
        - ixbuild
        - UIMX
        - X-Designer
- LucidChart
- Object Pascal
    - fpGUI UI Designer (included with fpGUI Toolkit)
- OpenWindows
    - guide (GUI builder)
- Pencil Project
- Qt
    - Qt Designer
- Scaleform
- Tk (framework)

    - GUI Builder
    - ActiveState Komodo
    - Visual Tcl (dead project)
    - PureTkGUI
- Wavemaker open source, browser-based development platform for Ajax development based on Dojo, Spring, Hibernate
- Windows Presentation Foundation
    - Microsoft Expression Blend
- wxWidgets
    - wxGlade
    - wxFormBuilder
    - wxDesigner
- XForms (toolkit)
    - fdesign
- Crank Storyboard Suite
    - Storyboard Designer

IDE Plugins

- NetBeans GUI design tool, formerly known as *Matisse*.
- Visual Editor - A free (Eclipse Public License) plugin for Eclipse on MS Windows and Linux (GTK and Motif).
- Jigloo - A *free for non-commercial use* plugin for Eclipse on MS Windows, Linux (gtk) and Mac OSX.
- WxSmith - A Code::Blocks plug-in for RAD editing of wxWidgets applications.
- Himalia Guilder (Only for Visual Studio 2005; no release since December '06.)

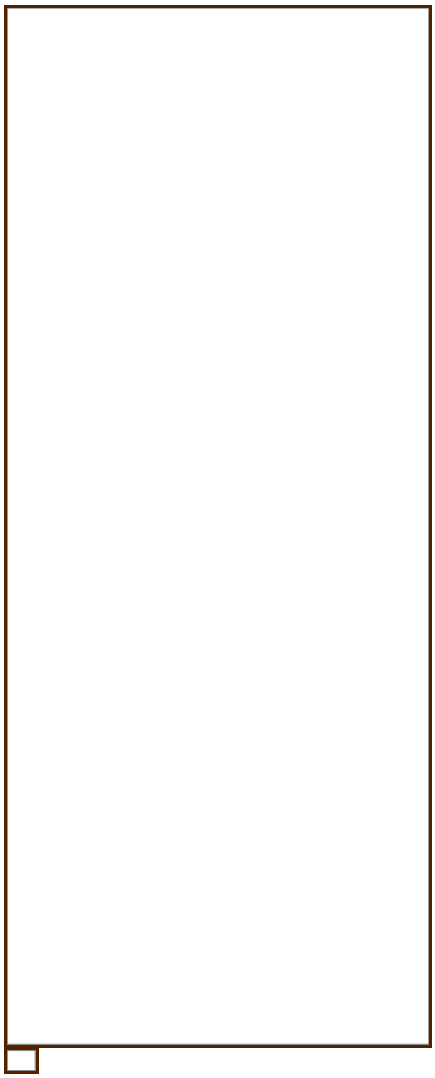
List of development environments

IDEs with GUI builders

- ActiveState Komodo
- Adobe Flash Builder
- Anjuta
- Ares
- CodeGear RAD Studio (former Borland Development Studio)
- Clarion
- Code::Blocks
- Gambas
- Just BASIC/Liberty BASIC
- KDevelop
- Lazarus
- Microsoft Visual Studio
- MonoDevelop

- MSEide+MSEgui
- NetBeans
- Qt Creator
- REALbasic
- SharpDevelop
- Softwell Maker
- WinDev
- wxDev-C++

Source Control



Example history tree of a revision-controlled project.

Revision control, also known as **version control** or **source control** (and an aspect of **software configuration management** or SCM), is the management of changes to documents, programs, and other information stored as computer files. It is most commonly used in software development, where a team of people may change the same files. Changes are usually identified by a number or letter code, termed the "revision number", "revision level", or simply "revision". For example, an

initial set of files is "revision 1". When the first change is made, the resulting set is "revision 2", and so on. Each revision is associated with a timestamp and the person making the change. Revisions can be compared, restored, and with some types of files, merged.

Version control systems (VCSs – singular VCS) most commonly run as stand-alone applications, but revision control is also embedded in various types of software such as word processors (e.g., Microsoft Word, OpenOffice.org Writer, KWord, Pages, etc.), spreadsheets (e.g., Microsoft Excel, OpenOffice.org Calc, KSpread, Numbers, etc.), and in various content management systems (e.g., Drupal, Joomla, WordPress). Integrated revision control is a key feature of wiki software packages such as MediaWiki, DokuWiki, TWiki etc. In wikis, revision control allows for the ability to revert a page to a previous revision, which is critical for allowing editors to track each other's edits, correct mistakes, and defend public wikis against vandalism and spam.

Software tools for revision control are essential for the organization of multi-developer projects.[53]

Overview

Revision control developed from formalized processes based on tracking revisions of early blueprints or bluelines. This system of control implicitly allowed returning to any earlier state of the design, for cases in which an engineering dead-end was reached in the development of the design. Likewise, in computer software engineering, revision control is any practice that tracks and provides control over changes to source code. Software developers sometimes use revision control software to maintain documentation and configuration files as well as source code. Also, version control is widespread in business and law. Indeed, "contract redline" and "legal blackline" are some of the earliest forms of revision control,[*citation needed*] and are still employed with varying degrees of sophistication. An entire industry has emerged to service the document revision control needs of business and other users, and some of the revision control technology employed in these circles is subtle, powerful, and innovative. The most sophisticated techniques are beginning to be used for the electronic tracking of changes to CAD files (see product data management), supplanting the "manual" electronic implementation of traditional revision control.

As teams design, develop and deploy software, it is common for multiple versions of the same software to be deployed in different sites and for the software's developers to be working simultaneously on updates. Bugs or features of the software are often only present in certain versions (because of the fixing of some problems and the introduction of others as the program develops). Therefore, for the purposes of locating and fixing bugs, it is vitally important to be able to retrieve and run different versions of the software to determine in which version(s) the problem occurs. It may also be necessary to develop two versions of the software concurrently (for instance, where one version has bugs fixed, but no new features (branch), while the other version is where new features are worked on (trunk).

At the simplest level, developers could simply retain multiple copies of the different versions of the program, and label them appropriately. This simple approach has been used on many large software projects. While this method can work, it is inefficient as many near-identical copies of the program have to be maintained. This requires a lot of self-discipline on the part of developers, and often leads to mistakes. Consequently, systems to automate some or all of the revision control process have been developed.

Moreover, in software development, legal and business practice and other environments, it has become increasingly common for a single document or snippet of code to be edited by a team, the members of which may be geographically dispersed and may pursue different and even contrary interests. Sophisticated revision control that tracks and accounts for ownership of changes to documents and code may be extremely helpful or even necessary in such situations.

Revision control may also track changes to configuration files, such as those typically stored in `/etc` or `/usr/local/etc` on Unix systems. This gives system administrators another way to easily track changes made and a way to roll back to earlier versions should the need arise.

Source-management models

Traditional revision control systems use a centralized model where all the revision control functions take place on a shared server. If two developers try to change the same file at the same time, without some method of managing access the developers may end up overwriting each other's work. Centralized revision control systems solve this problem in one of two different "source management models": file locking and version merging.

Atomic operations

Computer scientists speak of *atomic* operations if the system is left in a consistent state even if the operation is interrupted. The *commit* operation is usually the most critical in this sense. Commits are operations which tell the revision control system you want to make a group of changes you have been making final and available to all users. Not all revision control systems have atomic commits; notably, the widely-used CVS lacks this feature.

File locking

The simplest method of preventing "concurrent access" problems involves locking files so that only one developer at a time has write access to the central "repository" copies of those files. Once one developer "checks out" a file, others can read that file, but no one else may change that file until that developer "checks in" the updated version (or cancels the checkout).

File locking has both merits and drawbacks. It can provide some protection against difficult merge conflicts when a user is making radical changes to many sections of a large file (or group of files). However, if the files are left exclusively locked for too long, other developers may be tempted to bypass the revision control software and change the files locally, leading to more serious problems.

Version merging

Most version control systems allow multiple developers to edit the same file at the same time. The first developer to "check in" changes to the central repository always succeeds. The system may provide facilities to merge further changes into the central repository, and preserve the changes from the first developer when other developers check in.

Merging two files can be a very delicate operation, and usually possible only if the data structure is simple, as in text files. The result of a merge of two image files might not result in an image file at all. The second developer checking in code will need to take care with the merge, to make sure that the changes are compatible and that the merge operation does not introduce its own logic errors within the files. These problems limit the availability of automatic or semi-automatic merge operations mainly to simple text based documents, unless a specific merge plugin is available for the file types.

The concept of a *reserved edit* can provide an optional means to explicitly lock a file for exclusive write access, even when a merging capability exists.

Baselines, labels and tags

Most revision control tools will use only one of these similar terms (baseline, label, tag) to refer to the action of identifying a snapshot ("label the project") or the record of the snapshot ("try it with baseline *X*"). Typically only one of the terms *baseline*, *label*, or *tag* is used in documentation or discussion[*citation needed*]; they can be considered synonyms.

In most projects some snapshots are more significant than others, such as those used to indicate published releases, branches, or milestones.

When both the term *baseline* and either of *label* or *tag* are used together in the same context, *label* and *tag* usually refer to the mechanism within the tool of identifying or making the record of the snapshot, and *baseline* indicates the increased significance of any given label or tag.

Most formal discussion of configuration management uses the term *baseline*.

Distributed revision control

Distributed revision control (DRCS) takes a peer-to-peer approach, as opposed to the client-server approach of centralized systems. Rather than a single, central repository on which clients synchronize, each peer's working copy of the codebase is a bona-fide repository.[54] Distributed revision control conducts synchronization by exchanging patches (change-sets) from peer to peer. This results in some important differences from a centralized system:

- No canonical, reference copy of the codebase exists by default; only working copies.
- Common operations (such as commits, viewing history, and reverting changes) are fast, because there is no need to communicate with a central server.[55]

Rather, communication is only necessary when pushing or pulling changes to or from other peers.

- Each working copy effectively functions as a remote backup of the codebase and of its change-history, providing natural protection against data loss.[55]

Integration

Some of the more advanced revision-control tools offer many other facilities, allowing deeper integration with other tools and software-engineering processes. Plugins are often available for IDEs such as Oracle JDeveloper, IntelliJ IDEA, Eclipse and Visual Studio. NetBeans IDE and Xcode come with integrated version control support.

Common vocabulary

Terminology can vary from system to system, but some terms in common usage include:[56][57]

Baseline
: An approved revision of a document or source file from which subsequent changes can be made. See baselines, labels and tags.

Branch
: A set of files under version control may be **branched** or **forked** at a point in time so that, from that time forward, two copies of those files may develop at different speeds or in different ways independently of each other.

Change
: A **change** (or **diff**, or **delta**) represents a specific modification to a document under version control. The granularity of the modification considered a change varies between version control systems.

Change list
: On many version control systems with atomic multi-change commits, a **changelist**, **change set**, or **patch** identifies the set of **changes** made in a single commit. This can also represent a sequential view of the source code, allowing the examination of source "as of" any particular changelist ID.

Checkout
: A **check-out** (or **co**) is the act of creating a local working copy from the repository. A user may specify a specific revision or obtain the latest. The term 'checkout' can also used as a noun to describe the working copy.

Commit
: A **commit** (**checkin**, **ci** or, more rarely, **install**, **submit** or **record**) is the action of writing or merging the changes made in the working copy back to the repository. The terms 'commit' and 'checkin' can also used in noun form to describe the new revision that is created as a result of committing.

Conflict
: A conflict occurs when different parties make changes to the same document, and the system is unable to reconcile the changes. A user must **resolve** the conflict by combining the changes, or by selecting one change in favour of the other.

Delta compression
: Most revision control software uses delta compression, which retains only the differences between successive versions of files. This allows for more efficient storage of many different versions of files.

Dynamic stream
: A stream in which some or all file versions are mirrors of the parent stream's versions.

Export
: **exporting** is the act of obtaining the files from the repository. It is similar to **checking-out**

except that it creates a clean directory tree without the version-control metadata used in a working copy. This is often used prior to publishing the contents, for example.

Head
: Also sometime called **tip**, this refers to the most recent commit.

Import
: **importing** is the act of copying a local directory tree (that is not currently a working copy) into the repository for the first time.

Label
: See **tag**.

Mainline
: Similar to *trunk*, but there can be a mainline for each branch.

Merge
: A **merge** or **integration** is an operation in which two sets of changes are applied to a file or set of files. Some sample scenarios are as follows:
  - A user, working on a set of files, **updates** or **syncs** their working copy with changes made, and checked into the repository, by other users.[58]
  - A user tries to **check-in** files that have been updated by others since the files were **checked out**, and the **revision control software** automatically merges the files (typically, after prompting the user if it should proceed with the automatic merge, and in some cases only doing so if the merge can be clearly and reasonably resolved).
  - A set of files is **branched**, a problem that existed before the branching is fixed in one branch, and the fix is then merged into the other branch.
  - A **branch** is created, the code in the files is independently edited, and the updated branch is later incorporated into a single, unified **trunk**.

Promote
: The act of copying file content from a less controlled location into a more controlled location. For example, from a user's workspace into a repository, or from a stream to its parent.[59]

Repository
: The **repository** is where files' current and historical data are stored, often on a server. Sometimes also called a **depot** (for example, by SVK, AccuRev and Perforce).

Resolve
: The act of user intervention to address a conflict between different changes to the same document.

Reverse integration
: The process of merging different team branches into the main trunk of the versioning system.

Revision
: Also **version**: A version is any change in form. In SVK, a Revision is the state at a point in time of the entire tree in the repository.

Ring
: [*citation needed*] See **tag**.

Share
: The act of making one file or folder available in multiple branches at the same time. When a shared file is changed in one branch, it is changed in other branches.

Stream
: A container for branched files that has a known relationship to other such containers. Streams form a hierarchy; each stream can inherit various properties (like versions, namespace, workflow rules, subscribers, etc.) from its parent stream.

Tag

A **tag** or **label** refers to an important snapshot in time, consistent across many files. These files at that point may all be tagged with a user-friendly, meaningful name or revision number. See baselines, labels and tags.

Trunk
: The unique line of development that is not a branch (sometimes also called Baseline or Mainline)

Update
: An **update** (or **sync**) merges changes made in the repository (by other people, for example) into the local **working copy**.[58]

Working copy
: The **working copy** is the local copy of files from a repository, at a specific time or revision. All work done to the files in a repository is initially done on a working copy, hence the name. Conceptually, it is a *sandbox*.

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

External links

- Eric Sink's Source Control HOWTO A primer on the basics of version control
- Visual Guide to Version Control
- Better SCM Initiative: Comparison A useful summary of different systems and their features.

Build Tools

Build automation is the act of scripting or automating a wide variety of tasks that software developers do in their day-to-day activities including things like:

- compiling computer source code into binary code
- packaging binary code
- running tests
- deployment to production systems

- creating documentation and/or release notes

History

Historically, developers used build automation to call compilers and linkers from inside a build script versus attempting to make the compiler calls from the command line. It is simple to use the command line to pass a single source module to a compiler and then to a linker to create the final deployable object. However, when attempting to compile and link many source code modules, in a particular order, using the command line process is not a reasonable solution. The make scripting language offered a better alternative. It allowed a build script to be written to call in a series, the needed compile and link steps to build a software application. GNU Make [60] also offered additional features such as "makedepend" which allowed some source code dependency management as well as incremental build processing. This was the beginning of Build Automation. Its primary focus was on automating the calls to the compilers and linkers. As the build process grew more complex, developers began adding pre and post actions around the calls to the compilers such as a check-out from version control to the copying of deployable objects to a test location. The term "build automation" now includes managing the pre and post compile and link activities as well as the compile and link activities.

New breed of solutions

In recent years, build management solutions have provided even more relief when it comes to automating the build process. Both commercial and open source solutions are available to perform more automated build and workflow processing. Some solutions focus on automating the pre and post steps around the calling of the build scripts, while others go beyond the pre and post build script processing and drive down into streamlining the actual compile and linker calls without much manual scripting. These tools are particularly useful for continuous integration builds where frequent calls to the compile process are required and incremental build processing is needed.

Advanced build automation

Advanced build automation offers remote agent processing for distributed builds and/or distributed processing. The term "distributed builds" means that the actual calls to the compiler and linkers can be served out to multiple locations for improving the speed of the build. This term is often confused with "distributed processing". Distributed processing means that each step in a process or workflow can be sent to a different machine for execution. For example, a post step to the build may require the execution of multiple test scripts on multiple machines. Distributed processing can send the different test scripts to different machines. Distributed processing is not distributed builds. Distributed processing cannot take a make, ant or maven script, break it up and send it to different machines for compiling and linking. The distributed build process must have the machine intelligence to understand the source code dependencies in order to send the different compile and link steps to different machines. A build automation solution must be able to manage these dependencies in order to perform distributed builds. Some build tools can discover these relationships programmatically (Rational ClearMake distributed[61], Electric Cloud ElectricAccelerator[62]), while others depend on user-configured dependencies (Platform LSF lsmake[63]) Build automation that can sort out source code dependency relationships can also be

configured to run the compile and link activities in a parallelized mode. This means that the compiler and linkers can be called in multi-threaded mode using a machine that is configured with more than one core.

Not all build automation tools can perform distributed builds. Most only provide distributed processing support. In addition, most solutions that do support distributed builds can only handle C or C++. Build automation solutions that support distributed processing are often make based and many do not support Maven or Ant.

An example of a distributed build solution is Xoreax's IncrediBuild[64] for the Microsoft Visual Studio platform or the open-source CMake[65]. These may require particular configurations of a product environment so that it can run successfully on a distributed platform—library locations, environment variables, and so forth.

Advantages

- Improve product quality
- Accelerate the compile and link processing
- Eliminate redundant tasks
- Minimize "bad builds"
- Eliminate dependencies on key personnel
- Have history of builds and releases in order to investigate issues
- Save time and money - because of the reasons listed above.[66]

Types

- **On-Demand automation** such as a user running a script at the command line
- **Scheduled automation** such as a continuous integration server running a nightly build
- **Triggered automation** such as a continuous integration server running a build on every commit to a version control system.

Makefile

One specific form of build automation is the automatic generation of Makefiles. This is accomplished by tools like

- GNU Automake
- CMake
- imake
- qmake
- nmake
- wmake
- Apache Ant
- Apache Maven

- OpenMake Meister

Requirements of a build system

Basic requirements:

1. Frequent or overnight builds to catch problems early.[67][68][69]
2. Support for Source Code Dependency Management
3. Incremental build processing
4. Reporting that traces source to binary matching
5. Build acceleration
6. Extraction and reporting on build compile and link usage

Optional requirements:[70]

1. Generate release notes and other documentation such as help pages
2. Build status reporting
3. Test pass or fail reporting
4. Summary of the features added/modified/deleted with each new build

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

Notes

- Mike Clark: *Pragmatic Project Automation*, The Pragmatic Programmers ISBN 0-9745140-3-9

Software Documentation

Software documentation or **source code documentation** is written text that accompanies computer software. It either explains how it operates or how to use it, and may mean different things to people in different roles.

Involvement of people in software life

Documentation is an important part of software engineering. Types of documentation include:

1. Requirements - Statements that identify attributes, capabilities, characteristics, or qualities of a system. This is the foundation for what shall be or has been implemented.
2. Architecture/Design - Overview of softwares. Includes relations to an environment and construction principles to be used in design of software components.
3. Technical - Documentation of code, algorithms, interfaces, and APIs.
4. End User - Manuals for the end-user, system administrators and support staff.
5. Marketing - How to market the product and analysis of the market demand.

Requirements documentation

Requirements documentation is the description of what a particular software does or shall do. It is used throughout development to communicate what the software does or shall do. It is also used as an agreement or as the foundation for agreement on what the software shall do. Requirements are produced and consumed by everyone involved in the production of software: end users, customers, product managers, project managers, sales, marketing, software architects, usability engineers, interaction designers, developers, and testers, to name a few. Thus, requirements documentation has many different purposes.

Requirements come in a variety of styles, notations and formality. Requirements can be goal-like (e.g., *distributed work environment*), close to design (e.g., *builds can be started by right-clicking a configuration file and select the 'build' function*), and anything in between. They can be specified as statements in natural language, as drawn figures, as detailed mathematical formulas, and as a combination of them all.

The variation and complexity of requirements documentation makes it a proven challenge. Requirements may be implicit and hard to uncover. It is difficult to know exactly how much and what kind of documentation is needed and how much can be left to the architecture and design documentation, and it is difficult to know how to document requirements considering the variety of people that shall read and use the documentation. Thus, requirements documentation is often incomplete (or non-existent). Without proper requirements documentation, software changes become more difficult—and therefore more error prone (decreased software quality) and time-consuming (expensive).

The need for requirements documentation is typically related to the complexity of the product, the impact of the product, and the life expectancy of the software. If the software is very complex or developed by many people (e.g., mobile phone software), requirements can help to better communicate what to achieve. If the software is safety-critical and can have negative impact on human life (e.g., nuclear power systems, medical equipment), more formal requirements documentation is often required. If the software is expected to live for only a month or two (e.g., very small mobile phone applications developed specifically for a certain campaign) very little requirements documentation may be needed. If the software is a first release that is later built upon, requirements documentation is very helpful when managing the change of the software and verifying that nothing has been broken in the software when it is modified.

Traditionally, requirements are specified in requirements documents (e.g. using word processing applications and spreadsheet applications). To manage the increased complexity and changing nature of requirements documentation (and software documentation in general), database-centric systems and special-purpose requirements management tools are advocated.

Architecture/Design documentation

Architecture documentation is a special breed of design document. In a way, architecture documents are third derivative from the code (design document being second derivative, and code documents being first). Very little in the architecture documents is specific to the code itself. These documents do not describe how to program a particular routine, or even why that particular routine exists in the form that it does, but instead merely lays out the general requirements that would motivate the existence of such a routine. A good architecture document is short on details but thick on explanation. It may suggest approaches for lower level design, but leave the actual exploration trade studies to other documents.

Another breed of design docs is the comparison document, or trade study. This would often take the form of a *whitepaper*. It focuses on one specific aspect of the system and suggests alternate approaches. It could be at the user interface, code, design, or even architectural level. It will outline what the situation is, describe one or more alternatives, and enumerate the pros and cons of each. A good trade study document is heavy on research, expresses its idea clearly (without relying heavily on obtuse jargon to dazzle the reader), and most importantly is impartial. It should honestly and clearly explain the costs of whatever solution it offers as best. The objective of a trade study is to devise the best solution, rather than to push a particular point of view. It is perfectly acceptable to state no conclusion, or to conclude that none of the alternatives are sufficiently better than the baseline to warrant a change. It should be approached as a scientific endeavor, not as a marketing technique.

A very important part of the design document in enterprise software development is the Database Design Document (DDD). It contains Conceptual, Logical, and Physical Design Elements. The DDD includes the formal information that the people who interact with the database need. The purpose of preparing it is to create a common source to be used by all players within the scene. The potential users are:

- Database Designer
- Database Developer
- Database Administrator
- Application Designer
- Application Developer

When talking about Relational Database Systems, the document should include following parts:

- Entity - Relationship Schema, including following information and their clear definitions:
    - Entity Sets and their attributes
    - Relationships and their attributes
    - Candidate keys for each entity set
    - Attribute and Tuple based constraints
- Relational Schema, including following information:

- Tables, Attributes, and their properties
- Views
- Constraints such as primary keys, foreign keys,
- Cardinality of referential constraints
- Cascading Policy for referential constraints
- Primary keys

It is very important to include all information that is to be used by all actors in the scene. It is also very important to update the documents as any change occurs in the database as well.

Technical documentation

This is what most programmers mean when using the term *software documentation*. When creating software, code alone is insufficient. There must be some text along with it to describe various aspects of its intended operation. It is important for the code documents to be thorough, but not so verbose that it becomes difficult to maintain them. Several How-to and overview documentation are found specific to the software application or software product being documented by API Writers. This documentation may be used by developers, testers and also the end customers or clients using this software application. Today, we see lot of high end applications in the field of power, energy, transportation, networks, aerospace, safety, security, industry automation and a variety of other domains. Technical documentation has become important within such organizations as the basic and advanced level of information may change over a period of time with architecture changes. Hence, technical documentation has gained lot of importance in recent times, especially in the software field.

Often, tools such as Doxygen, NDoc, javadoc, EiffelStudio, Sandcastle, ROBODoc, POD, TwinText, or Universal Report can be used to auto-generate the code documents—that is, they extract the comments and software contracts, where available, from the source code and create reference manuals in such forms as text or HTML files. Code documents are often organized into a *reference guide* style, allowing a programmer to quickly look up an arbitrary function or class.

The idea of auto-generating documentation is attractive to programmers for various reasons. For example, because it is extracted from the source code itself (for example, through comments), the programmer can write it while referring to the code, and use the same tools used to create the source code to make the documentation. This makes it much easier to keep the documentation up-to-date.

Of course, a downside is that only programmers can edit this kind of documentation, and it depends on them to refresh the output (for example, by running a cron job to update the documents nightly). Some would characterize this as a pro rather than a con.

Donald Knuth has insisted on the fact that documentation can be a very difficult afterthought process and has advocated literate programming, writing at the same time and location as the source code and extracted by automatic means.

Elucidative Programming is the result of practical applications of Literate Programming in real programming contexts. The Elucidative paradigm proposes that source code and documentation be stored separately. This paradigm was inspired by the same experimental findings that produced Kelp. Often, software developers need to be able to create and access information that is not going

to be part of the source file itself. Such annotations are usually part of several software development activities, such as code walks and porting, where third party source code is analysed in a functional way. Annotations can therefore help the developer during any stage of software development where a formal documentation system would hinder progress. Kelp stores annotations in separate files, linking the information to the source code dynamically.

User documentation

Unlike code documents, user documents are usually far more diverse with respect to the source code of the program, and instead simply describe how it is used.

In the case of a software library, the code documents and user documents could be effectively equivalent and are worth conjoining, but for a general application this is not often true.

Typically, the user documentation describes each feature of the program, and assists the user in realizing these features. A good user document can also go so far as to provide thorough troubleshooting assistance. It is very important for user documents to not be confusing, and for them to be up to date. User documents need not be organized in any particular way, but it is very important for them to have a thorough index. Consistency and simplicity are also very valuable. User documentation is considered to constitute a contract specifying what the software will do. API Writers are very well accomplished towards writing good user documents as they would be well aware of the software architecture and programming techniques used. See also Technical Writing.

There are three broad ways in which user documentation can be organized.

1. **Tutorial:** A tutorial approach is considered the most useful for a new user, in which they are guided through each step of accomplishing particular tasks [71].
2. **Thematic:** A thematic approach, where chapters or sections concentrate on one particular area of interest, is of more general use to an intermediate user. Some authors prefer to convey their ideas through a knowledge based article to facilitating the user needs. This approach is usually practiced by a dynamic industry, such as Information technology, where the user population is largely correlated with the troubleshooting demands [72], [73].
3. **List or Reference:** The final type of organizing principle is one in which commands or tasks are simply listed alphabetically or logically grouped, often via cross-referenced indexes. This latter approach is of greater use to advanced users who know exactly what sort of information they are looking for.

A common complaint among users regarding software documentation is that only one of these three approaches was taken to the near-exclusion of the other two. It is common to limit provided software documentation for personal computers to online help that give only reference information on commands or menu items. The job of tutoring new users or helping more experienced users get the most out of a program is left to private publishers, who are often given significant assistance by the software developer.

Marketing documentation

For many applications it is necessary to have some promotional materials to encourage casual observers to spend more time learning about the product. This form of documentation has three purposes:-

1. To excite the potential user about the product and instill in them a desire for becoming more involved with it.
2. To inform them about what exactly the product does, so that their expectations are in line with what they will be receiving.
3. To explain the position of this product with respect to other alternatives.

One good marketing technique is to provide clear and memorable *catch phrases* that exemplify the point we wish to convey, and also emphasize the interoperability of the program with anything else provided by the manufacturer.

Notes

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

External links

- kelp - a source code annotation framework for architectural, design and technical documentation.
- ISO documentation standards committee - International Organization for Standardization committee which develops user documentation standards.

Static Code Analysis

This is a list of tools for static code analysis.

Historical products

- Lint — The original static code analyzer of C code.

Open-source or Non-commercial products

Multi-language

- PMD Copy/Paste Detector (CPD) — PMDs duplicate code detection for (e.g.) Java, JSP, C, C++ and PHP code.
- Sonar — A continuous inspection engine to manage the technical debt (unit tests, complexity, duplication, design, comments, coding standards and potential problems). Supported languages are Java, Flex, PHP, PL/SQL, Cobol and Visual Basic 6.
- Yasca — Yet Another Source Code Analyzer, a plugin-based framework for scanning arbitrary file types, with plugins for scanning C/C++, Java, JavaScript, ASP, PHP, HTML/CSS, ColdFusion, COBOL, and other file types. It integrates with other scanners, including FindBugs, JLint, PMD, and Pixy.

.NET (C#, VB.NET and all .NET compatible languages)

- FxCop — Free static analysis for Microsoft .NET programs that compile to CIL. Standalone and integrated in some Microsoft Visual Studio editions. From Microsoft.
- Gendarme — Open-source (MIT License) equivalent to FxCop created by the Mono project. Extensible rule-based tool to find problems in .NET applications and libraries, particularly those that contain code in ECMA CIL format.
- StyleCop — Analyzes C# source code to enforce a set of style and consistency rules. It can be run from inside of Microsoft Visual Studio or integrated into an MSBuild project. Free download from Microsoft.

ActionScript

- Apparat — A language manipulation and optimization framework consisting of intermediate representations for ActionScript.

C

- BLAST (Berkeley Lazy Abstraction Software verification Tool) — A software model checker for C programs based on lazy abstraction.
- Clang — A compiler that includes a static analyzer.
- Frama-C — A static analysis framework for C.
- Lint — The original static code analyzer for C.
- Sparse — A tool designed to find faults in the Linux kernel.
- Splint — An open source evolved version of Lint (for C).

C++

- cppcheck — Open-source tool that checks for several types of errors, including the use of STL.

Java

- Checkstyle — Besides some static code analysis, it can be used to show violations of a configured coding standard.
- FindBugs — An open-source static bytecode analyzer for Java (based on Jakarta BCEL) from the University of Maryland.
- Hammurapi — (Free for non-commercial use only) versatile code review solution.
- PMD — A static ruleset based Java source code analyzer that identifies potential problems.
- Soot — A language manipulation and optimization framework consisting of intermediate languages for Java.
- Squale — A platform to manage software quality (also available for other languages, using commercial analysis tools though).

JavaScript

- Closure Compiler — JavaScript optimizer that rewrites JavaScript code to make it faster and more compact. It also checks your usage of native javascript functions.
- JSLint — JavaScript syntax checker and validator.

Objective-C

- Clang — The free Clang project includes a static analyzer. As of version 3.2, this analyzer is included in Xcode.[74]

Commercial products

Multi-language

- Axivion Bauhaus Suite — A tool for C, C++, C#, Java and Ada code that comprises various analyses such as architecture checking, interface analyses, and clone detection.
- Black Duck Suite — Analyze the composition of software source code and binary files, search for reusable code, manage open source and third-party code approval, honor the legal obligations associated with mixed-origin code, and monitor related security vulnerabilities.
- CAST Application Intelligence Platform — Detailed, audience-specific dashboards to measure quality and productivity. 30+ languages, SAP, Oracle, PeopleSoft, Siebel, .NET, Java, C/C++, Struts, Spring, Hibernate and all major databases.

- Coverity Static Analysis (formerly Coverity Prevent) — Identifies security vulnerabilities and code defects in C, C++, C# and Java code. Complements Coverity Dynamic Code Analysis and Architecture Analysis.
- DMS Software Reengineering Toolkit — Supports custom analysis of C, C++, C#, Java, COBOL, PHP, VisualBasic and many other languages. Also COTS tools for clone analysis, dead code analysis, and style checking.
- Compuware DevEnterprise — Analysis of COBOL, PL/I, JCL, CICS, DB2, IMS and others.
- Fortify — Helps developers identify software security vulnerabilities in C/C++, .NET, Java, JSP, ASP.NET, ColdFusion, "Classic" ASP, PHP, VB6, VBScript, JavaScript, PL/SQL, T-SQL, python and COBOL as well as configuration files.
- GrammaTech CodeSonar — Analyzes C,C++.
- Imagix 4D — Identifies problems in variable usage, task interaction and concurrency, particularly in embedded applications, as part of an overall solution for understanding, improving and documenting C, C++ and Java software.
- Intel - Intel Parallel Studio XE: Contains **Static Security Analysis** (SSA) feature supports C/C++ and Fortran
- JustCode — Code analysis and refactoring productivity tool for JavaScript, C#, Visual Basic.NET, and ASP.NET
- Klocwork Insight — Provides security vulnerability and defect detection as well as architectural and build-over-build trend analysis for C, C++, C# and Java.
- Lattix, Inc. LDM — Architecture and dependency analysis tool for Ada, C/C++, Java, .NET software systems.
- LDRA Testbed — A software analysis and testing tool suite for C, C++, Ada83, Ada95 and Assembler (Intel, Freescale, Texas Instruments).
- Micro Focus (formerly Relativity Technologies) Modernization Workbench — Parsers included for COBOL (multiple variants including IBM, Unisys, MF, ICL, Tandem), PL/I, Natural (inc. ADABAS), Java, Visual Basic, RPG, C & C++ and other legacy languages; Extensible SDK to support 3rd party parsers. Supports automated Metrics (including Function Points), Business Rule Mining, Componentisation and SOA Analysis. Rich ad hoc diagramming, AST search & reporting)
- Ounce Labs (from 2010 IBM Rational Appscan Source) — Automated source code analysis that enables organizations to identify and eliminate software security vulnerabilities in languages including Java, JSP, C/C++, C#, ASP.NET and VB.Net.
- Parasoft — Analyzes Java (Jtest), JSP, C, C++ (C++test), .NET (C#, ASP.NET, VB.NET, etc.) using .TEST, WSDL, XML, HTML, CSS, JavaScript, VBScript/ASP, and configuration files for security[75], compliance[76], and defect prevention.
- Polyspace — Uses abstract interpretation to detect and prove the absence of certain run-time errors in source code for C, C++, and Ada
- Rational Software Analyzer — Supports Java, C/C++ (and others available through extensions)
- SofCheck Inspector — Provides static detection of logic errors, race conditions, and redundant code for Java and Ada. Provides automated extraction of pre/postconditions from code itself.
- Sotoarc/Sotograph — Architecture and quality in-depth analysis and monitoring for Java, C#, C and C++
- Syhunt Sandcat — Detects security flaws in PHP, Classic ASP and ASP.NET web applications.
- Understand — Analyzes C,C++, Java, Ada, Fortran, Jovial, Delphi, VHDL, HTML, CSS, PHP, and JavaScript — reverse engineering of source, code navigation, and metrics tool.

- Veracode — Finds security flaws in application binaries and bytecode without requiring source. Supported languages include C, C++, .NET (C#, C++/CLI, VB.NET, ASP.NET), Java, JSP, ColdFusion, and PHP.
- Visual Studio Team System — Analyzes C++,C# source codes. only available in team suite and development edition.

.NET

Products covering multiple .NET languages.

- CodeIt.Right — Combines Static Code Analysis and automatic Refactoring to best practices which allows automatically correct code errors and violations. Supports both C# and VB.NET.
- CodeRush — A plugin for Visual Studio, it addresses a multitude of short comings with the popular IDE. Including alerting users to violations of best practices by using static code analysis.
- JustCode — Add-on for Visual Studio 2005/2008/2010 for real-time, solution-wide code analysis for C#, VB.NET, ASP.NET, XAML, JavaScript, HTML and multi-language solutions.
- NDepend — Simplifies managing a complex .NET code base by analyzing and visualizing code dependencies, by defining design rules, by doing impact analysis, and by comparing different versions of the code. Integrates into Visual Studio.
- ReSharper — Add-on for Visual Studio 2003/2005/2008/2010 from the creators of IntelliJ IDEA, which also provides static code analysis for C#.
- Kalistick — Mixing from the Cloud: static code analysis with best practice tips and collaborative tools for Agile teams

Ada

- Ada-ASSURED — A tool that offers coding style checks, standards enforcement and pretty printing features.
- AdaCore CodePeer — Automated code review and bug finder for Ada programs that uses control-flow, data-flow, and other advanced static analysis techniques.
- LDRA Testbed — A software analysis and testing tool suite for Ada83/95.
- SofCheck Inspector — Provides static detection of logic errors, race conditions, and redundant code for Ada. Provides automated extraction of pre/postconditions from code itself.

C / C++

- FlexeLint — A multiplatform version of PC-Lint.
- Green Hills Software DoubleCheck — A software analysis tool for C/C++.
- Intel - Intel Parallel Studio XE: Contains **Static Security Analysis** (SSA) feature
- LDRA Testbed — A software analysis and testing tool suite for C/C++.
- Monoidics INFER — A sound tool for C/C++ based on Separation Logic.

- PC-Lint — A software analysis tool for C/C++.
- PVS-Studio — A software analysis tool for C/C++/C++0x.
- QA-C (and QA-C++) — Deep static analysis of C/C++ for quality assurance and guideline enforcement.
- Red Lizard's Goanna — Static analysis for C/C++ in Eclipse and Visual Studio.

Java

- Jtest — Testing and static code analysis product by Parasoft.
- LDRA Testbed — A software analysis and testing tool suite for Java.
- SemmleCode — Object oriented code queries for static program analysis.
- SonarJ — Monitors conformance of code to intended architecture, also computes a wide range of software metrics.
- Kalistick — A Cloud-based platform to manage and optimize code quality for Agile teams with DevOps spirit

Formal methods tools

Tools that use a formal methods approach to static analysis (e.g., using static program assertions):

- ESC/Java and ESC/Java2 — Based on Java Modeling Language, an enriched version of Java.
- Polyspace — Uses abstract interpretation (a formal methods based technique[77]) to detect and prove the absence of certain run-time errors in source code for C, C++, and Ada
- SofCheck Inspector — Statically determines and documents pre- and postconditions for Java methods; statically checks preconditions at all call sites; also supports Ada.
- SPARK Toolset including the SPARK Examiner — Based on the SPARK programming language, a subset of Ada.

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

External links

- Java Static Checkers at the Open Directory Project
- List of Java static code analysis plugins for Eclipse
- List of static source code analysis tools for C
- List of Static Source Code Analysis Tools at CERT
- SAMATE-Source Code Security Analyzers
- SATE - Static Analysis Tool Exposition
- "A Comparison of Bug Finding Tools for Java", by Nick Rutar, Christian Almazan, and Jeff Foster, University of Maryland. Compares Bandera, ESC/Java 2, FindBugs, JLint, and PMD.
- "Mini-review of Java Bug Finders", by Rick Jelliffe, O'Reilly Media.
- Parallel Lint, by Andrey Karpov
- Integrate static analysis into a software development process Explains how one goes about integrating static analysis into a software development process

Profiling

In software engineering, **program profiling**, **software profiling** or simply **profiling**, a form of dynamic program analysis (as opposed to static code analysis), is the investigation of a program's behavior using information gathered as the program executes. The usual purpose of this analysis is to determine which sections of a program to optimize - to increase its overall speed, decrease its memory requirement or sometimes both.

- A **(code) profiler** is a **performance analysis** tool that, most commonly, measures only the frequency and duration of function calls, but there are other specific types of profilers (e.g. memory profilers) in addition to more comprehensive profilers, capable of gathering extensive performance data.
- An instruction set simulator which is also — by necessity — a profiler, can measure the totality of a program's behaviour from invocation to termination.

Gathering program events

Profilers use a wide variety of techniques to collect data, including hardware interrupts, code instrumentation, instruction set simulation, operating system hooks, and performance counters. The usage of profilers is 'called out' in the performance engineering process.

Use of profilers

Program analysis tools are extremely important for understanding program behavior. Computer architects need such tools to evaluate how well programs will perform on new architectures. Software writers need tools to analyze their programs and identify critical sections of code. Compiler writers often use such tools to find out how well their instruction scheduling or branch prediction algorithm is performing... (ATOM, PLDI, '94)

The output of a profiler may be:-

- A statistical *summary* of the events observed (a **profile**)

  Summary profile information is often shown annotated against the source code statements where the events occur, so the size of measurement data is linear to the code size of the program.

```
/* ------------ source------------------------ count */
0001             IF X = "A"                     0055
0002                THEN DO
0003                  ADD 1 to XCOUNT           0032
0004                ELSE
0005             IF X = "B"                     0055
```

- A stream of recorded events (a **trace**)

  For sequential programs, a summary profile is usually sufficient, but performance problems in parallel programs (waiting for messages or synchronization issues) often depend on the time relationship of events, thus requiring a full trace to get an understanding of what is happening. The size of a (full) trace is linear to the program's instruction path length, making it somewhat impractical. A trace may therefore be initiated at one point in a program and terminated at another point to limit the output.

- An ongoing interaction with the hypervisor (continuous or periodic monitoring via on-screen display for instance)

  This provides the opportunity to switch a trace on or off at any desired point during execution in addition to viewing on-going metrics about the (still executing) program. It also provides the opportunity to suspend asynchronous processes at critical points to examine interactions with other parallel processes in more detail.

History

Performance analysis tools existed on IBM/360 and IBM/370 platforms from the early 1970s, usually based on timer interrupts which recorded the Program status word (PSW) at set timer intervals to detect "hot spots" in executing code. This was an early example of sampling (see below). In early 1974, Instruction Set Simulators permitted full trace and other performance monitoring features.

Profiler-driven program analysis on Unix dates back to at least 1979, when Unix systems included a basic tool "prof" that listed each function and how much of program execution time it used. In 1982, gprof extended the concept to a complete call graph analysis [78]

In 1994, Amitabh Srivastava and Alan Eustace of Digital Equipment Corporation published a paper describing ATOM.[79] ATOM is a platform for converting a program into its own profiler. That is, at compile time, it inserts code into the program to be analyzed. That inserted code outputs analysis data. This technique - modifying a program to analyze itself - is known as "instrumentation".

In 2004, both the gprof and ATOM papers appeared on the list of the 50 most influential PLDI papers of all time.[80]

Profiler types based on output

Flat profiler

Flat profilers compute the average call times, from the calls, and do not break down the call times based on the callee or the context.

Call-graph profiler

Call graph profilers show the call times, and frequencies of the functions, and also the call-chains involved based on the callee. However context is not preserved.

Methods of data gathering

Event-based profilers

The programming languages listed here have event-based profilers:

- Java: the JVMTI (JVM Tools Interface) API, formerly JVMPI (JVM Profiling Interface), provides hooks to profilers, for trapping events like calls, class-load, unload, thread enter leave.
- .NET: Can attach a profiling agent as a COM server to the CLR. Like Java, the runtime then provides various callbacks into the agent, for trapping events like method JIT / enter / leave, object creation, etc. Particularly powerful in that the profiling agent can rewrite the target application's bytecode in arbitrary ways.
- Python: Python profiling includes the profile module, hotshot (which is call-graph based), and using the 'sys.setprofile' function to trap events like c_{call,return,exception}, python_{call,return,exception}.
- Ruby: Ruby also uses a similar interface like Python for profiling. Flat-profiler in profile.rb, module, and ruby-prof a C-extension are present.

Statistical profilers

Some profilers operate by sampling. A sampling profiler probes the target program's program counter at regular intervals using operating system interrupts. Sampling profiles are typically less numerically accurate and specific, but allow the target program to run at near full speed.

The resulting data are not exact, but a statistical approximation. *The actual amount of error is usually more than one sampling period. In fact, if a value is n times the sampling period, the expected error in it is the square-root of n sampling periods.* [81]

In practice, sampling profilers can often provide a more accurate picture of the target program's execution than other approaches, as they are not as intrusive to the target program, and thus don't have as many side effects (such as on memory caches or instruction decoding pipelines). Also since they don't affect the execution speed as much, they can detect issues that would otherwise be hidden. They are also relatively immune to over-evaluating the cost of small, frequently called routines or 'tight' loops. They can show the relative amount of time spent in user mode versus interruptible kernel mode such as system call processing.

Still, kernel code to handle the interrupts entails a minor loss of CPU cycles, diverted cache usage, and is unable to distinguish the various tasks occurring in uninterruptible kernel code (microsecond-range activity).

Dedicated hardware can go beyond this: some recent MIPS processors JTAG interface have a PCSAMPLE register, which samples the program counter in a truly undetectable manner.

Some of the most commonly used statistical profilers are AMD CodeAnalyst, Apple Inc. Shark, gprof, Intel VTune and Parallel Amplifier (part of Intel Parallel Studio).

Instrumenting profilers

Some profilers **instrument** the target program with additional instructions to collect the required information.

Instrumenting the program can cause changes in the performance of the program, potentially causing inaccurate results and heisenbugs. Instrumenting will always have some impact on the program execution, typically always slowing it. However, instrumentation can be very specific and be carefully controlled to have a minimal impact. The impact on a particular program depends on the placement of instrumentation points and the mechanism used to capture the trace. Hardware support for trace capture means that on some targets, instrumentation can be on just one machine instruction. The impact of instrumentation can often be deducted (i.e. eliminated by subtraction) from the results.

gprof is an example of a profiler that uses both instrumentation and sampling. Instrumentation is used to gather caller information and the actual timing values are obtained by statistical sampling.

Instrumentation

- **Manual**: Performed by the programmer, e.g. by adding instructions to explicitly calculate runtimes, simply count events or calls to measurement APIs such as the Application Response Measurement standard.
- **Automatic source level**: instrumentation added to the source code by an automatic tool according to an instrumentation policy.
- **Compiler assisted**: Example: "gcc -pg ..." for gprof, "quantify g++ ..." for Quantify
- **Binary translation**: The tool adds instrumentation to a compiled binary. Example: ATOM
- **Runtime instrumentation**: Directly before execution the code is instrumented. The program run is fully supervised and controlled by the tool. Examples: Pin, Valgrind

- **Runtime injection**: More lightweight than runtime instrumentation. Code is modified at runtime to have jumps to helper functions. Example: DynInst

Interpreter instrumentation

- **Interpreter debug** options can enable the collection of performance metrics as the interpreter encounters each target statement. A bytecode, control table or JIT interpreters are three examples that usually have complete control over execution of the target code, thus enabling extremely comprehensive data collection opportunities.

Hypervisor/Simulator

- **Hypervisor**: Data are collected by running the (usually) unmodified program under a hypervisor. Example: SIMMON
- **Simulator** and **Hypervisor**: Data collected interactively and selectively by running the unmodified program under an Instruction Set Simulator. Examples: SIMON (Batch Interactive test/debug) and IBM OLIVER (CICS interactive test/debug).

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

- Dunlavey, “Performance tuning with instruction-level cost derived from call-stack sampling”, ACM SIGPLAN Notices 42, 8 (August, 2007), pp. 4–8.
- Dunlavey, “Performance Tuning: Slugging It Out!”, Dr. Dobb's Journal, Vol 18, #12, November 1993, pp 18–26.

External links

- Article "Need for speed — Eliminating performance bottlenecks" on doing execution time analysis of Java applications using IBM Rational Application Developer.
- Profiling Runtime Generated and Interpreted Code using the VTune™ Performance Analyzer

Code Coverage

Code coverage is a measure used in software testing. It describes the degree to which the source code of a program has been tested. It is a form of testing that inspects the code directly and is therefore a form of white box testing.[82] In time, the use of code coverage has been extended to the field of digital hardware, the contemporary design methodology of which relies on hardware description languages (HDLs).

Code coverage was among the first methods invented for systematic software testing. The first published reference was by Miller and Maloney in *Communications of the ACM* in 1963.

Code coverage is one consideration in the safety certification of avionics equipment. The standard by which avionics gear is certified by the Federal Aviation Administration (FAA) is documented in DO-178B.[83]

Coverage criteria

To measure how well the program is exercised by a test suite, one or more *coverage criteria* are used.

Basic coverage criteria

There are a number of coverage criteria, the main ones being:[84]

- **Function coverage** - Has each function (or subroutine) in the program been called?
- **Statement coverage** - Has each node in the program been executed?
- **Decision coverage** (not the same as **branch coverage** - see Position Paper CAST10.[85]) - Has every edge in the program been executed? For instance, have the requirements of each branch of each control structure (such as in IF and CASE statements) been met as well as not met?
- **Condition coverage** (or predicate coverage) - Has each boolean sub-expression evaluated both to true and false? This does not necessarily imply decision coverage.
- **Condition/decision coverage** - Both decision and condition coverage should be satisfied.

For example, consider the following C++ function:

```
int foo(int x, int y)
{
    int z = 0;
    if ((x>0) && (y>0)) {
        z = x;
    }
    return z;
}
```

Assume this function is a part of some bigger program and this program was run with some test suite.

- If during this execution function 'foo' was called at least once, then *function coverage* for this function is satisfied.
- *Statement coverage* for this function will be satisfied if it was called e.g. as `foo(1,1)`, as in this case, every line in the function is executed including `z = x;`.
- Tests calling `foo(1,1)` and `foo(1,0)` will satisfy *decision coverage*, as in the first case the `if` condition is satisfied and `z = x;` is executed, and in the second it is not.
- *Condition coverage* can be satisfied with tests that call `foo(1,1)`, `foo(1,0)` and `foo(0,0)`. These are necessary as in the first two cases `(x>0)` evaluates to `true` while in the third it evaluates `false`. At the same time, the first case makes `(y>0)` `true` while the second and third make it `false`.

In languages, like Pascal, where standard boolean operations are not short circuited, condition coverage does not necessarily imply decision coverage. For example, consider the following fragment of code:

```
if a and b then
```

Condition coverage can be satisfied by two tests:

- `a=true`, `b=false`
- `a=false`, `b=true`

However, this set of tests does not satisfy decision coverage as in neither case will the `if` condition be met.

Fault injection may be necessary to ensure that all conditions and branches of exception handling code have adequate coverage during testing.

Modified condition/decision coverage

For safety-critical applications (e.g. for avionics software) it is often required that **modified condition/decision coverage (MC/DC)** is satisifed. This criteria extends condition/decision criteria with requirements that each condition should affect the decision outcome independently. For example, consider the following code:

```
if (a or b) and c then
```

The condition/decision criteria will be satisfied by the following set of tests:

- a=true, b=true, c=true
- a=false, b=false, c=false

However, the above tests set will not satisfy modified condition/decision coverage, since in the first test, the value of 'b' and in the second test the value of 'c' would not influence the output. So, the following test set is needed to satisfy MC/DC:

- a=**false**, b=**false**, c=true
- a=**true**, b=false, c=**true**

- a=false, b=**true**, c=**true**
- a=true, b=true, c=**false**

The bold values influence the output, each variable must be present as an influencing value at least once with false and once with true.

Multiple condition coverage

This criteria requires that all combinations of conditions inside each decision are tested. For example, the code fragment from the previous section will require eight tests:

- a=false, b=false, c=false
- a=false, b=false, c=true
- a=false, b=true, c=false
- a=false, b=true, c=true
- a=true, b=false, c=false
- a=true, b=false, c=true
- a=true, b=true, c=false
- a=true, b=true, c=true

Other coverage criteria

There are further coverage criteria, which are used less often:

- **Linear Code Sequence and Jump (LCSAJ) coverage** - has every LCSAJ been executed?
- **JJ-Path coverage** - have all jump to jump paths [86] (aka LCSAJs) been executed?
- **Path coverage** - Has every possible route through a given part of the code been executed?
- **Entry/exit coverage** - Has every possible call and return of the function been executed?
- **Loop coverage** - Has every possible loop been executed zero times, once, and more than once?

Safety-critical applications are often required to demonstrate that testing achieves 100% of some form of code coverage.

Some of the coverage criteria above are connected. For instance, path coverage implies decision, statement and entry/exit coverage. Decision coverage implies statement coverage, because every statement is part of a branch.

Full path coverage, of the type described above, is usually impractical or impossible. Any module with a succession of n decisions in it can have up to 2^n paths within it; loop constructs can result in an infinite number of paths. Many paths may also be infeasible, in that there is no input to the program under test that can cause that particular path to be executed. However, a general-purpose algorithm for identifying infeasible paths has been proven to be impossible (such an algorithm could be used to solve the halting problem).[87] Methods for practical path coverage testing instead attempt to identify classes of code paths that differ only in the number of loop executions, and to achieve "basis path" coverage the tester must cover all the path classes.

In practice

The target software is built with special options or libraries and/or run under a special environment such that every function that is exercised (executed) in the program(s) is mapped back to the function points in the source code. This process allows developers and quality assurance personnel to look for parts of a system that are rarely or never accessed under normal conditions (error handling and the like) and helps reassure test engineers that the most important conditions (function points) have been tested. The resulting output is then analyzed to see what areas of code have not been exercised and the tests are updated to include these areas as necessary. Combined with other code coverage methods, the aim is to develop a rigorous, yet manageable, set of regression tests.

In implementing code coverage policies within a software development environment one must consider the following:

- What are coverage requirements for the end product certification and if so what level of code coverage is required? The typical level of rigor progression is as follows: Statement, Branch/Decision, Modified Condition/Decision Coverage(MC/DC), LCSAJ (Linear Code Sequence and Jump)
- Will code coverage be measured against tests that verify requirements levied on the system under test (DO-178B)?
- Is the object code generated directly traceable to source code statements? Certain certifications, (ie. DO-178B Level A) require coverage at the assembly level if this is not the case: "Then, additional verification should be performed on the object code to establish the correctness of such generated code sequences" (DO-178B) para-6.4.4.2.[88]

Test engineers can look at code coverage test results to help them devise test cases and input or configuration sets that will increase the code coverage over vital functions. Two common forms of code coverage used by testers are statement (or line) coverage and path (or edge) coverage. Line coverage reports on the execution footprint of testing in terms of which lines of code were executed to complete the test. Edge coverage reports which branches or code decision points were executed to complete the test. They both report a coverage metric, measured as a percentage. The meaning of this depends on what form(s) of code coverage have been used, as 67% path coverage is more comprehensive than 67% statement coverage.

Generally, code coverage tools and libraries exact a performance and/or memory or other resource cost which is unacceptable to normal operations of the software. Thus, they are only used in the lab. As one might expect, there are classes of software that cannot be feasibly subjected to these coverage tests, though a degree of coverage mapping can be approximated through analysis rather than direct testing.

There are also some sorts of defects which are affected by such tools. In particular, some race conditions or similar real time sensitive operations can be masked when run under code coverage environments; and conversely, some of these defects may become easier to find as a result of the additional overhead of the testing code.

Software tools

Tools for C / C++

- Cantata++
- Insure++
- IBM Rational Pure Coverage
- Tessy
- Testwell CTC++
- Trucov
- CodeScroll

Tools for C# .NET

- NCover
- Testwell CTC++ (with Java and C# add on)

Tools for Java

- Clover
- Cobertura
- Structure 101
- EMMA
- Jtest
- Serenity
- Testwell CTC++ (with Java and C# add on)

Tools for PHP

- PHPUnit, also need Xdebug to make coverage reports

Hardware tools

- Aldec
- Atrenta
- Cadence Design Systems
- JEDA Technologies
- Mentor Graphics
- Nusym Technology
- Simucad Design Automation
- Synopsys

Notes

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

External links

- Branch Coverage for Arbitrary Languages Made Easy
- Code Coverage Analysis by Steve Cornett
- Code Coverage Introduction
- Comprehensive paper on Code Coverage & tools selection by Vijayan Reddy, Nithya Jayachandran
- Development Tools (Java)/ Introduction to Software Engineering/Print version at the Open Directory Project
- Development Tools (General)/ Introduction to Software Engineering/Print version at the Open Directory Project
- Systematic mistake analysis of digital computer programs
- FAA CAST Position Papers [15]

Project Management

Project management software is a term covering many types of software, including estimation and planning, scheduling, cost control and budget management, resource allocation, collaboration software, communication, quality management and documentation or administration systems, which are used to deal with the complexity of large projects.

Tasks or activities of project management software

Scheduling

One of the most common purposes is to schedule a series of events or tasks and the complexity of the schedule can vary considerably depending on how the tool is used. Some common challenges include:

- Events which depend on one another in different ways or dependencies
- Scheduling people to work on, and resources required by, the various tasks, commonly termed resource scheduling
- Dealing with uncertainties in the estimates of the duration of each task

Providing information

Project planning software can be expected to provide information to various people or stakeholders, and can be used to measure and justify the level of effort required to complete the project(s). Typical requirements might include:

- Tasks lists for people, and allocation schedules for resources
- Overview information on how long tasks will take to complete
- Early warning of any risks to the project
- Information on workload, for planning holidays
- Evidence
- Historical information on how projects have progressed, and in particular, how actual and planned performance are related
- Optimum utilization of available resource

Approaches to project management software

Desktop

Project management software can be implemented as a program that runs on the desktop of each user. This typically gives the most responsive and graphically-intense style of interface.

Desktop applications typically store their data in a file, although some have the ability to collaborate with other users (see below), or to store their data in a central database. Even a file-based project plan can be shared between users if it's on a networked drive and only one user accesses it at a time.

Desktop applications *can* be written to run in a heterogeneous environment of multiple operating systems, although it's unusual.

Web-based

Project management software can be implemented as a Web application, accessed through an intranet, or an extranet using a web browser.

This has all the usual advantages and disadvantages of web applications:

- Can be accessed from any type of computer without installing software on user's computer
- Ease of access-control
- Naturally multi-user

- Only one software version and installation to maintain
- Centralized data repository
- Typically slower to respond than desktop applications
- Project information not available when the user (or server) is offline
- Some solutions allow the user to go offline with a copy of the data

Personal

A personal project management application is one used at home, typically to manage lifestyle or home projects. There is considerable overlap with *single user* systems, although personal project management software typically involves simpler interfaces. See also *non-specialised tools* below.

Single user

A single-user system is programmed with the assumption that only one person will ever need to edit the project plan at once. This may be used in small companies, or ones where only a few people are involved in top-down project planning. Desktop applications generally fall into this category.

Collaborative

A collaborative system is designed to support multiple users modifying different sections of the plan at once; for example, updating the areas they personally are responsible for such that those estimates get integrated into the overall plan. Web-based tools, including extranets, generally fall into this category, but have the limitation that they can only be used when the user has live Internet access. To address this limitation, some software tools using client–server architecture provide a rich client that runs on users' desktop computer and replicate project and task information to other project team members through a central server when users connect periodically to the network. Some tools allow team members to check out their schedules (and others' as read only) to work on them while not on the network. When reconnecting to the database, all changes are synchronized with the other schedules.

Integrated

An integrated system combines project management or project planning, with many other aspects of company life. For example, projects can have bug tracking issues assigned to each project, the list of project customers becomes a customer relationship management module, and each person on the project plan has their own task lists, calendars, and messaging functionality associated with their projects.

Similarly, specialised tools like SourceForge integrate project management software with source control (CVS) software and bug-tracking software, so that each piece of information can be integrated into the same system.

Non-specialised tools

While specialised software may be common, and heavily promoted by each vendor, there are a vast range of other software (and non-software) tools used to plan and schedule projects.

- Calendaring software can often handle scheduling as easily as dedicated software
- Spreadsheets are very versatile, and can be used to calculate things not anticipated by the designers.

Criticisms of project management software

The following may apply in general, or to specific products, or to some specific functions within products.

- May not be derived from a sound project management method. For example, displaying the Gantt chart view by default encourages users to focus on timed task scheduling too early, rather than identifying objectives, deliverables and the imposed logical progress of events (dig the trench first to put in the drain pipe).
- May be inconsistent with the type of project management method. For example, traditional (e.g. Waterfall) vs. agile (e.g. Scrum).
- Focuses primarily on the planning phase and does not offer enough functionality for project tracking, control and in particular plan-adjustment. There may be excessive dependency on the first paper print-out of a project plan, which is simply a snapshot at one moment in time. The plan is dynamic; as the project progresses the plan must change to accommodate tasks that are completed early, late, re-sequenced, etc. Good management software should not only facilitate this, but assist with impact assessment and communication of plan changes.
- Does not make a clear distinction between the planning phase and post planning phase, leading to user confusion and frustration when the software does not behave as expected. For example, shortening the duration of a task when an additional human resource is assigned to it while the project is still being planned.
- Offer complicated features to meet the needs of project management or project scheduling professionals, which must be understood in order to effectively use the product. Additional features may be so complicated as to be of no use to anyone. Complex task prioritization and resource leveling algorithms for example can produce results that make no intuitive sense, and overallocation is often more simply resolved manually.
- Some people may achieve better results using simpler technique, (e.g. pen and paper), yet feel pressured into using project management software by company policy (discussion).
- Similar to PowerPoint, project management software might shield the manager from important interpersonal contact.
- New types of software are challenging the traditional definition of Project Management. Frequently, users of project management software are not actually managing a discrete project. For instance, managing the ongoing marketing for an already-released product is not a "project" in the traditional sense of the term; it does not involve management of discrete resources working on something with a discrete beginning/end. Groupware applications now add "project management" features that directly support this type of workflow-oriented project management. Classically-trained Project Managers may argue whether this is "sound project management." However, the end-users of such tools will refer to it as such, and the de-facto definition of the term Project Management may change.

- When there are multiple larger projects, project management software can be very useful. Nevertheless, one should probably not use management software if only a single small project is involved, as management software incurs a larger time-overhead than is worthwhile.

Books

- Eric Uyttewaal: *Dynamic Scheduling With Microsoft(r) Project 2000: The Book By and For Professionals*, ISBN 0-9708276-0-1
- George Suhanic: *Computer-Aided Project Management*, ISBN 0-19-511591-0
- Richard E. Westney: *Computerized Management of Multiple Small Projects*, ISBN 0-8247-8645-9

Continuous Integration

In software engineering, **continuous integration** (**CI**) implements *continuous* processes of applying quality control — small pieces of effort, applied frequently. Continuous integration aims to improve the quality of software, and to reduce the time taken to deliver it, by replacing the traditional practice of applying quality control *after* completing all development.

Theory

When embarking on a change, a developer takes a copy of the current code base on which to work. As other developers submit changed code to the code repository, this copy gradually ceases to reflect the repository code. When developers submit code to the repository they must first update their code to reflect the changes in the repository since they took their copy. The more changes the repository contains, the more work developers must do before submitting their own changes.

Eventually, the repository may become so different from the developers' baselines that they enter what is sometimes called "integration hell",[89] where the time it takes to integrate exceeds the time it took to make their original changes. In a worst-case scenario, developers may have to discard their changes and completely redo the work.

Continuous integration involves integrating early and often, so as to avoid the pitfalls of "integration hell". The practice aims to reduce rework and thus reduce cost and time.

The rest of this article discusses best practice in how to achieve continuous integration, and how to automate this practice. Automation is a best practice itself.[90][91]

Recommended practices

Continuous integration - as the practice of frequently integrating one's new or changed code with the existing code repository - should occur frequently enough that no intervening window remains between commit and build, and such that no errors can arise without developers noticing them and

correcting them immediately.[92] Normal practice is to trigger these builds by every commit to a repository, rather than a periodically scheduled build. The practicalities of doing this in a multi-developer environment of rapid commits are such that it's usual to trigger a short timer after each commit, then to start a build when either this timer expires, or after a rather longer interval since the last build. Automated tools such as CruiseControl or Hudson offer this scheduling automatically.

Another factor is the need for a version control system that supports atomic commits, i.e. all of a developer's changes may be seen as a single commit operation. There is no point in trying to build from only half of the changed files.

Maintain a code repository

This practice advocates the use of a revision control system for the project's source code. All artifacts required to build the project should be placed in the repository. In this practice and in the revision control community, the convention is that the system should be buildable from a fresh checkout and not require additional dependencies. Extreme Programming advocate Martin Fowler also mentions that where branching is supported by tools, its use should be minimised [*citation needed*]. Instead, it is preferred that changes are integrated rather than creating multiple versions of the software that are maintained simultaneously. The mainline (or trunk) should be the place for the working version of the software.

Automate the build

A single command should have the capability of building the system. Many build-tools, such as make, have existed for many years. Other more recent tools like Ant, Maven, MSBuild or IBM Rational Build Forge are frequently used in continuous integration environments. Automation of the build should include automating the integration, which often includes deployment into a production-like environment. In many cases, the build script not only compiles binaries, but also generates documentation, website pages, statistics and distribution media (such as Windows MSI files, RPM or DEB files).

Make the build self-testing

Once the code is built, all tests should run to confirm that it behaves as the developers expect it to behave.

Everyone commits to the baseline every day

By committing regularly, every committer can reduce the number of conflicting changes. Checking in a week's worth of work runs the risk of conflicting with other features and can be very difficult to resolve. Early, small conflicts in an area of the system cause team members to communicate about the change they are making.

Many programmers recommend committing all changes at least once a day (once per feature built), and in addition performing a nightly build.

Every commit (to baseline) should be built

The system should build commits to the current working version in order to verify that they integrate correctly. A common practice is to use Automated Continuous Integration, although this may be done manually. For many, continuous integration is synonymous with using Automated Continuous Integration where a continuous integration server or daemon monitors the version control system for changes, then automatically runs the build process.

Keep the build fast

The build needs to complete rapidly, so that if there is a problem with integration, it is quickly identified.

Test in a clone of the production environment

Having a test environment can lead to failures in tested systems when they deploy in the production environment, because the production environment may differ from the test environment in a significant way. However, building a replica of a production environment is cost prohibitive. Instead, the pre-production environment should be built to be a scalable version of the actual production environment to both alleviate costs while maintaining technology stack composition and nuances.

Make it easy to get the latest deliverables

Making builds readily available to stakeholders and testers can reduce the amount of rework necessary when rebuilding a feature that doesn't meet requirements. Additionally, early testing reduces the chances that defects survive until deployment. Finding errors earlier also, in some cases, reduces the amount of work necessary to resolve them.

Everyone can see the results of the latest build

It should be easy to find out where/whether the build breaks and who made the relevant change.

Automate deployment

Most CI systems allow the running of scripts after a build finishes. In most situations, it is possible to write a script to deploy the application to a live test server that everyone can look at. A further advance in this way of thinking is Continuous Deployment, which calls for the software to be deployed directly into production, often with additional automation to prevent defects or regressions[93].

History

Continuous Integration emerged in the Extreme Programming (XP) community, and XP advocates Martin Fowler and Kent Beck first wrote about continuous integration circa 1999. Fowler's paper[94] is a popular source of information on the subject. Beck's book *Extreme Programming Explained*[95], the original reference for Extreme Programming, also describes the term.

Advantages and disadvantages

Advantages

Continuous integration has many advantages:

- when unit tests fail or a bug emerges, developers might revert the codebase back to a bug-free state, without wasting time debugging
- developers detect and fix integration problems continuously - avoiding last-minute chaos at release dates, (when everyone tries to check in their slightly incompatible versions).
- early warning of broken/incompatible code
- early warning of conflicting changes
- immediate unit testing of all changes
- constant availability of a "current" build for testing, demo, or release purposes
- immediate feedback to developers on the quality, functionality, or system-wide impact of code they are writing
- frequent code check-in pushes developers to create modular, less complex code[*citation needed*]
- metrics generated from automated testing and CI (such as metrics for code coverage, code complexity, and features complete) focus developers on developing functional, quality code, and help develop momentum in a team[*citation needed*]

Disadvantages

- initial setup time required
- well-developed test-suite required to achieve automated testing advantages
- large-scale refactoring can be troublesome due to continuously changing code base
- hardware costs for build machines can be significant

Many teams using CI report that the advantages of CI well outweigh the disadvantages.[96] The effect of finding and fixing integration bugs early in the development process saves both time and money over the lifespan of a project.

Software

To support continuous integration, software tools such as automated build software can be employed.

Software tools for continuous integration include:

- AnthillPro — continuous integration server by Urbancode
- Apache Continuum — continuous integration server supporting Apache Maven and Apache Ant. Supports CVS, Subversion, Ant, Maven, and shell scripts
- Apache Gump — continuous integration tool by Apache
- Automated Build Studio — proprietary automated build, continuous integration and release management system by AutomatedQA
- Bamboo — proprietary continuous integration server by Atlassian Software Systems
- BuildBot — Python/Twisted-based continuous build system
- BuildMaster — proprietary application lifecycle management and continuous integration tool by Inedo
- CABIE - Continuous Automated Build and Integration Environment — open source, written in Perl; works with CVS, Subversion, AccuRev, Bazaar and Perforce
- Cascade — proprietary continuous integration tool; provides a checkpointing facility to build and test changes before they are committed
- codeBeamer — proprietary collaboration software with built-in continuous integration features
- CruiseControl — Java-based framework for a continuous build process
- CruiseControl.NET — .NET-based automated continuous integration server
- CruiseControl.rb - Lightweight, Ruby-based continuous integration server that can build any codebase, not only Ruby, released under Apache Licence 2.0
- ElectricCommander — proprietary continuous integration and release management solution from Electric Cloud
- FinalBuilder Server — proprietary automated build and continuous integration server by VSoft Technologies
- Go — proprietary agile build and release management software by Thoughtworks
- Jenkins (formerly known as Hudson) — MIT-licensed, written in Java, runs in servlet container, supports CVS, Subversion, Mercurial, Git, StarTeam, Clearcase, Ant, NAnt, Maven, and shell scripts
- Software Configuration and Library Manager — software configuration management system for z/OS by IBM Rational Software
- QuickBuild - proprietary continuous integration server with free community edition featuring build life cycle management and pre-commit verification.
- TeamCity — proprietary continuous-integration server by JetBrains with free professional edition
- Team Foundation Server — proprietary continuous integration server and source code repository by Microsoft
- Tinderbox — Mozilla-based product written in Perl
- Rational Team Concert — proprietary software development collaboration platform with built-in build engine by IBM including Rational Build Forge

See comparison of continuous integration software for a more in depth feature matrix.

Further reading

- Duvall, Paul M. (2007). *Continuous Integration. Improving Software Quality and Reducing Risk*. Addison-Wesley. ISBN 0-321-33638-0.

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

External links

- Continuous integration by Martin Fowler (an introduction)
- Continuous Integration at the Portland Pattern Repository (a collegial discussion)
- Cross platform testing at the Portland Pattern Repository
- Continuous Integration Server Feature Matrix (a guide to tools)
- Continuous Integration: The Cornerstone of a Great Shop by Jared Richardson (an introduction)
- A Recipe for Build Maintainability and Reusability by Jay Flowers
- Continuous Integration anti-patterns by Paul Duvall
- Extreme programming

Bug Tracking

A **bug tracking system** is a software application that is designed to help quality assurance and programmers keep track of reported software bugs in their work. It may be regarded as a type of issue tracking system.

Many bug-tracking systems, such as those used by most open source software projects, allow users to enter bug reports directly. Other systems are used only internally in a company or organization doing software development. Typically bug tracking systems are integrated with other software project management applications.

Having a bug tracking system is extremely valuable in software development, and they are used extensively by companies developing software products. Consistent use of a bug or issue tracking system is considered one of the "hallmarks of a good software team".[97]

Components

A major component of a bug tracking system is a database that records facts about known bugs. Facts may include the time a bug was reported, its severity, the erroneous program behavior, and details on how to reproduce the bug; as well as the identity of the person who reported it and any programmers who may be working on fixing it.[98]

Typical bug tracking systems support the concept of the life cycle for a bug which is tracked through status assigned to the bug. A bug tracking system should allow administrators to configure permissions based on status, move the bug to another status, or delete the bug. The system should also allow administrators to configure the bug statuses and to what status a bug in a particular status can be moved. Some systems will e-mail interested parties, such as the submitter and assigned programmers, when new records are added or the status changes.

Usage

The main benefit of a bug-tracking system is to provide a clear centralized overview of development requests (including both bugs and improvements, the boundary is often fuzzy), and their state. The prioritized list of pending items (often called backlog) provides valuable input when defining the product roadmap, or maybe just "the next release".

In a corporate environment, a bug-tracking system may be used to generate reports on the productivity of programmers at fixing bugs. However, this may sometimes yield inaccurate results because different bugs may have different levels of severity and complexity. The severity of a bug may not be directly related to the complexity of fixing the bug. There may be different opinions among the managers and architects.

A *local bug tracker (LBT)* is usually a computer program used by a team of application support professionals (often a help desk) to keep track of issues communicated to software developers. Using an LBT allows support professionals to track bugs in their "own language" and not the "language of the developers." In addition, a LBT allows a team of support professionals to track specific information about users who have called to complain — this information may not always be needed in the actual development queue. Thus, there are two tracking systems when an LBT is in place.

Bug tracking systems as a part of integrated project management systems

Bug and issue tracking systems are often implemented as a part of integrated project management systems. This approach allows including bug tracking and fixing in a general product development process, fixing bugs in several product versions, automatic generation of a product knowledge base and release notes.

Distributed bug tracking

Some bug trackers are designed to be used with distributed revision control software. These distributed bug trackers allow bug reports to be conveniently read, added to the database or updated while a developer is offline.[99] Distributed bug trackers include Bugs Everywhere, and Fossil.

Recently, commercial bug tracking systems have also begun to integrate with distributed version control. FogBugz, for example, enables this functionality via the source-control tool, Kiln.[100]

Although wikis and bug tracking systems are conventionally viewed as distinct types of software, ikiwiki can also be used as a distributed bug tracker. It can manage documents and code as well, in an integrated distributed manner. However, its query functionality is not as advanced or as user-friendly as some other, non-distributed bug trackers such as Bugzilla.[101] Similar statements can be made about org-mode, although it is not wiki software as such.

Bug tracking and test management

While traditional test management tools such as HP Quality Center and Rational Software come with their own bug tracking systems. Other tools integrate with popular bug tracking systems. [*citation needed*]

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

External links

- Bug Tracking Software at the Open Directory Project
- How to Report Bugs Effectively
- List of distributed bug tracking software

Decompiler

A **decompiler** is the name given to a computer program that performs the reverse operation to that of a compiler. That is, it translates a file containing information at a relatively low level of abstraction (usually designed to be computer readable rather than human readable) into a form having a higher level of abstraction (usually designed to be human readable).

Introduction

The term *decompiler* is most commonly applied to a program which translates executable programs (the output from a compiler) into source code in a (relatively) high level language which, when compiled, will produce an executable whose behavior is the same as the original executable program. By comparison, a disassembler translates an executable program into assembly language (and an assembler could be used to assemble it back into an executable program).

Decompilation is the act of using a decompiler, although the term, when used as a noun, can also refer to the output of a decompiler. It can be used for the recovery of lost source code, and is also useful in some cases for computer security, interoperability and error correction.[102] The success of decompilation depends on the amount of information present in the code being decompiled and the sophistication of the analysis performed on it. The bytecode formats used by many virtual machines (such as the Java Virtual Machine or the .NET Framework Common Language Runtime) often include extensive metadata and high-level features that make decompilation quite feasible. The presence of debug data can make it possible to reproduce the original variable and structure names and even the line numbers. Machine language without such metadata or debug data is much harder to decompile.[103]

Some compilers and post-compilation tools produce obfuscated code (that is, they attempt to produce output that is very difficult to decompile). This is done to make it more difficult to reverse engineer the executable.

Design

Decompilers can be thought of as composed of a series of phases each of which contributes specific aspects of the overall decompilation process.

Loader

The first decompilation phase loads and parses the input machine code or intermediate language program's binary file format. It should be able to discover basic facts about the input program, such as the architecture (Pentium, PowerPC, etc), and the entry point. In many cases, it should be able to find the equivalent of the `main` function of a C program, which is the start of the user written code. This excludes the runtime initialization code, which should not be decompiled if possible. If available the symbol tables and debug data are also loaded. The front end may be able to identify the libraries used even if they are linked with the code, this will provide library interfaces. If it can determine the compiler or compilers used it may provide useful information in identifying code idioms.[104]

Disassembly

The next logical phase is the disassembly of machine code instructions into a machine independent intermediate representation (IR). For example, the Pentium machine instruction

```
mov    eax, [ebx+0x04]
```

might be translated to the IR

```
eax := m[ebx+4];
```

Idioms

Idiomatic machine code sequences are sequences of code whose combined semantics is not immediately apparent from the instructions' individual semantics. Either as part of the disassembly phase, or as part of later analyses, these idiomatic sequences need to be translated into known equivalent IR. For example, the x86 assembly code:

```
cdq    eax             ; edx is set to the sign-extension of eax
xor    eax, edx
sub    eax, edx
```

could be translated to

```
eax := abs(eax);
```

Some idiomatic sequences are machine independent; some involve only one instruction. For example, `xor eax, eax` clears the `eax` register (sets it to zero). This can be implemented with a machine independent simplification rule, such as `a xor a = 0`.

In general, it is best to delay detection of idiomatic sequences if possible, to later stages that are less affected by instruction ordering. For example, the instruction scheduling phase of a compiler may insert other instructions into an idiomatic sequence, or change the ordering of instructions in the sequence. A pattern matching process in the disassembly phase would probably not recognize the altered pattern. Later phases group instruction expressions into more complex expressions, and modify them into a canonical (standardized) form, making it more likely that even the altered idiom will match a higher level pattern later in the decompilation.

It is particularly important to recognize the compiler idioms for subroutine calls, exception handling, and switch statements. Some languages also have extensive support for strings or long integers.

Program analysis

Various program analyses can be applied to the IR. In particular, expression propagation combines the semantics of several instructions into more complex expressions. For example,

```
mov   eax,[ebx+0x04]
add   eax,[ebx+0x08]
sub   [ebx+0x0C],eax
```

could result in the following IR after expression propagation:

```
m[ebx+12] := m[ebx+12] - (m[ebx+4] + m[ebx+8]);
```

The resulting expression is more like high level language, and has also eliminated the use of the machine register `eax` . Later analyses may eliminate the `ebx` register.

Data flow analysis

The places where register contents are defined and used must be traced using data flow analysis. The same analysis can be applied to locations that are used for temporaries and local data. A different name can then be formed for each such connected set of value definitions and uses. It is possible that the same local variable location was used for more than one variable in different parts of the original program. Even worse it is possible for the data flow analysis to identify a path whereby a value may flow between two such uses even though it would never actually happen or matter in reality. This may in bad cases lead to needing to define a location as a union of types. The decompiler may allow the user to explicitly break such unnatural dependencies which will lead to clearer code. This of course means a variable is potentially used without being initialized and so indicates a problem in the original program.

Type analysis

A good machine code decompiler will perform type analysis. Here, the way registers or memory locations are used result in constraints on the possible type of the location. For example, an `and` instruction implies that the operand is an integer; programs do not use such an operation on floating point values (except in special library code) or on pointers. An `add` instruction results in three constraints, since the operands may be both integer, or one integer and one pointer (with integer and pointer results respectively; the third constraint comes from the ordering of the two operands when the types are different).[105]

Various high level expressions can be recognized which trigger recognition of structures or arrays. However, it is difficult to distinguish many of the possibilities, because of the freedom that machine code or even some high level languages such as C allow with casts and pointer arithmetic.

The example from the previous section could result in the following high level code:

```
struct T1 *ebx;
   struct T1 {
       int v0004;
       int v0008;
       int v000C;
   };
 ebx->v000C -= ebx->v0004 + ebx->v0008;
```

Structuring

The penultimate decompilation phase involves structuring of the IR into higher level constructs such as `while` loops and `if/then/else` conditional statements. For example, the machine code

```
   xor eax, eax
l0002:
   or  ebx, ebx
   jge l0003
   add eax,[ebx]
   mov ebx,[ebx+0x4]
   jmp l0002
l0003:
   mov [0x10040000],eax
```

could be translated into:

```
   eax = 0;
   while (ebx < 0) {
       eax += ebx->v0000;
       ebx = ebx->v0004;
   }
   v10040000 = eax;
```

Unstructured code is more difficult to translate into structured code than already structured code. Solutions include replicating some code, or adding boolean variables.[106]

Code generation

The final phase is the generation of the high level code in the back end of the decompiler. Just as a compiler may have several back ends for generating machine code for different architectures, a decompiler may have several back ends for generating high level code in different high level languages.

Just before code generation, it may be desirable to allow an interactive editing of the IR, perhaps using some form of graphical user interface. This would allow the user to enter comments, and non-generic variable and function names. However, these are almost as easily entered in a post decompilation edit. The user may want to change structural aspects, such as converting a `while` loop to a `for` loop. These are less readily modified with a simple text editor, although source code refactoring tools may assist with this process. The user may need to enter information that failed to be identified during the type analysis phase, e.g. modifying a memory expression to an array or structure expression. Finally, incorrect IR may need to be corrected, or changes made to cause the output code to be more readable.

Legality

The majority of computer programs are covered by copyright laws. Although the precise scope of what is covered by copyright differs from region to region, copyright law generally provides the author (the programmer(s) or employer) with a collection of exclusive rights to the program.[107] These rights include the right to make copies, including copies made into the computer's RAM[*citation needed*]. Since the decompilation process involves making multiple such copies, it is generally prohibited without the authorization of the copyright holder. However, because decompilation is often a necessary step in achieving software interoperability, copyright laws in both the United States and Europe permit decompilation to a limited extent.

In the United States, the copyright fair use defense has been successfully invoked in decompilation cases. For example, in *Sega v. Accolade*, the court held that Accolade could lawfully engage in decompilation in order to circumvent the software locking mechanism used by Sega's game consoles.[108]

In Europe, the 1991 Software Directive explicitly provides for a right to decompile in order to achieve interoperability. The result of a heated debate between, on the one side, software protectionists, and, on the other, academics as well as independent software developers, Article 6 permits decompilation only if a number of conditions are met:

- First, a person or entity must have a license to use the program to be decompiled.
- Second, decompilation must be necessary to achieve interoperability with the target program or other programs. Interoperability information should therefore not be readily available, such as through manuals or API documentation. This is an important limitation. The necessity must be proven by the decompiler. The purpose of this important limitation is primarily to provide an incentive for developers to document and disclose their products' interoperability information.[109]
- Third, the decompilation process must, if possible, be confined to the parts of the target program relevant to interoperability. Since one of the purposes of decompilation is to gain an understanding of the program structure, this third limitation may be difficult to meet. Again, the burden of proof is on the decompiler.

In addition, Article 6 prescribes that the information obtained through decompilation may not be used for other purposes and that it may not be given to others.

Overall, the decompilation right provided by Article 6 codifies what is claimed to be common practice in the software industry. Few European lawsuits are known to have emerged from the decompilation right. This could be interpreted as meaning either one of two things: 1) the decompilation right is not used frequently and the decompilation right may therefore have been unnecessary, or 2) the decompilation right functions well and provides sufficient legal certainty not to give rise to legal disputes. In a recent report regarding implementation of the Software Directive by the European member states, the European Commission seems to support the second interpretation.

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

External links

- Decompilers and Disassemblers at the Open Directory Project
- Legality of Decompilation
- Simple Step by Step Java Decompilation Example
- Article on decompilation
- Reverse Engineering Resources
- A decompiler
- The commercial Hex-Rays decompiler for x86 binaries
- DJ Java Decompiler

Obfuscation

Obfuscated code is source or machine code that has been made difficult to understand for humans. Programmers may deliberately obfuscate code to conceal its purpose (security through obscurity) or its logic to prevent tampering, deter reverse engineering, or as a puzzle or recreational challenge for someone reading the source code. Programs known as *obfuscators* transform readable code into obfuscated code using various techniques. Code obfuscation is different in essence from hardware obfuscation, where description and/or structure of a circuit is modified to hide its functionality.

Overview

Some languages may be more prone to obfuscation than others.[110][111] C,[112] C++,[113] and Perl[114] are some examples.

Recreational obfuscation

Writing and reading obfuscated source code can be a brain teaser for programmers. A number of programming contests reward the most creatively obfuscated code: the International Obfuscated C Code Contest, Obfuscated Perl Contest, and International Obfuscated Ruby Code Contest.

Types of obfuscations include simple keyword substitution, use or non-use of whitespace to create artistic effects, and self-generating or heavily compressed programs.

Short obfuscated Perl programs may be used in signatures of Perl programmers. These are JAPHs ("Just another Perl hacker").[*citation needed*]

Examples

This is a winning entry from the International Obfuscated C Code Contest[115] written by Ian Phillipps in 1988[116] and subsequently reverse engineered by Thomas Ball.[117]

```
/*
  LEAST LIKELY TO COMPILE SUCCESSFULLY:
  Ian Phillipps, Cambridge Consultants Ltd., Cambridge, England
*/

#include <stdio.h>
main(t,_,a)
char
*
a;
{
        return!

0<t?
t<3?

main(-79,-13,a+
main(-87,1-_,
main(-86, 0, a+1 )

+a)):

1,
t<_?
main(t+1, _, a )
:3,

main ( -94, -27+t, a )
&&t == 2 ?_
<13 ?

main ( 2, _+1, "%s %d %d\n" )

:9:16:
t<0?
t<-72?
main( _, t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+,/+#n+,/#;\
#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l q#'+d'K#!/+k#;\
q#'r}eKK#}w'r}eKK{nl]'/#;#q#n')()#}w'){){nl]'/+#n';d}rw' i;# ){nl]!/n{n#'; \
r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#\
\
n'wk nw' iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c ;;\
{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;\
#'rdq#w! nr'/ ') }+}{rl#'{n' ')# }'+}##(!!/")
:
t<-50?
```

```
_==*a ?
putchar(31[a]):

main(-65,_,a+1)
:
main((*a == '/') + t, _, a + 1 )
:

0<t?

main ( 2, 2 , "%s")
:*a=='/'||

main(0,

main(-61,*a, "!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry")

,a+1);}
```

It is a C program that when compiled and run will generate the 12 verses of *The 12 Days of Christmas*. It contains all the strings required for the poem in an encoded form within the code.

A non-winning entry from the same year, the next example illustrates creative use of whitespace; it generates mazes of arbitrary length [118]:

```
char*M,A,Z,E=40,J[40],T[40];main(C){for(*J=A=scanf(M="%d",&C);
--           E;          J[           E]          =T
[E   ]=  E)   printf("._");  for(;(A-=Z=!Z)  ||  (printf("\n|"
)    ,   A    =           39           ,C          --
)    ;   Z    ||    printf   (M   ))M[Z]=Z[A-(E   =A[J-Z])&&!C
&    A   ==          T[                          A]
|6<<27<rand()||!C&!Z?J[T[E]=T[A]]=E,J[T[A]=A-Z]=A,"_.":" |"];}
```

Modern C compilers don't allow constant strings to be overwritten, which can be avoided by changing "*M" to "M[3]" and omitting "M=".

An example of a JAPH:

```
@P=split//,".URRUU\c8R";@d=split//,"\nrekcah xinU / lreP rehtona tsuJ";sub p{
@p{"r$p","u$p"}=(P,P);pipe"r$p","u$p";++$p;($q*=2)+=$f=!fork;map{$P=$P[$f^ord
($p{$_})&6];$p{$_}=/ ^$P/ix?$P:close$_}keys%p}p;p;p;p;p;map{$p{$_}=~/^[P.]/&&
close$_}%p;wait until$?;map{/^r/&&<$_>}%p;$_=$d[$q];sleep rand(2)if/\S/;print
```

This slowly displays the text "Just another Perl / Unix hacker", multiple characters at a time, with delays. An explanation can be found here.

Some Python examples can be found in the official Python programming FAQ.

Disadvantages of obfuscation

At best, obfuscation merely makes it time-consuming, but not impossible, to reverse engineer a program. [119]

Obfuscating software

A variety of tools exists to perform or assist with code obfuscation. These include experimental research tools created by academics, hobbyist tools, commercial products written by professionals, and open-source software. There also exist deobfuscation tools that attempt to perform the reverse transformation.

Although the majority of commercial obfuscation solutions work by transforming either program source code[120][121], or platform-independent bytecode as used by Java[122] and .NET[123], there are also some that work with C and C++[124][125] - languages that are typically compiled to native code.

Notes

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

References

- B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan and K. Yang. "On the (Im)possibility of Obfuscating Programs". *21st Annual International Cryptology Conference*, Santa Barbara, California, USA. Springer Verlag LNCS Volume 2139, 2001.
- Mateas, Michael; Nick Montfort. "A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics". *Proceedings of the 6th Digital Arts and Culture Conference, IT University of Copenhagen, 1–3 December 2005*. pp. 144–153. http://nickm.com/cis/a_box_darkly.pdf.

External links

- The International Obfuscated C Code Contest
- Protecting Java Code Via Code Obfuscation, ACM Crossroads, Spring 1998 issue
- Protect Your Java Code - Through Obfuscators And Beyond, April 2009
- Dotfuscator in Visual Studio on MSDN resource page — Visual Studio 2008 documentation for built-in .NET obfuscation
- Obfuscation tools for .NET, on MSDN — Obfuscation resources for .NET, on the Microsoft Developer Center.

- Can we obfuscate programs?
- Yury Lifshits. Lecture Notes on Program Obfuscation (Spring'2005)
- Obfuscate member names in .NET code
- Java obfuscators at the Open Directory Project
- Analysis of the 12 days program
- Analysis of the obfuscated maze generating program
- Obfuscated Perl program with explanation
- Making C compiler generate obfuscated code
- Protect php code via code obfuscation

Re-engineering

Introduction

The **reengineering** of software was described by Chikofsky and Cross in their 1990 paper[126], as **"The examination and alteration of a system to reconstitute it in a new form"**. Less formally, reengineering is the modification of a software system that takes place after it has been reverse engineered, generally to add new functionality, or to correct errors.

This entire process is often erroneously referred to as reverse engineering; however, it is more accurate to say that reverse engineering is the initial examination of the system, and reengineering is the subsequent modification.

Re-engineering is mostly used in the context where a legacy system is involved[127]. Software systems are evolving on high rate because there more research to make the better so therefore software system in most cases, legacy software needs to operate on a new computing platform. '**Re-engineering'** is a set of activities that are carried out to re-structure a legacy system to a new system with better functionalities and conform to the hardware and software quality constraint.

See also

- Code refactoring
- Rewrite (programming)
- Program transformation
- DMS Software Reengineering Toolkit

References

1. ↑ Fred Brooks, *The Mythical Man-Month*. Addison-Wesley, 1975 & 1995. ISBN 0-201-00650-2 & ISBN 0-201-83595-9.
2. ↑ Lientz, B.P. and Swanson, E.B., *Software Maintenance Management, A Study Of The Maintenance Of Computer Application Software In 487 Data Processing Organizations*. Addison-Wesley, Reading MA, 1980. ISBN 0201042053
3. ↑ ISO/IEC 14764:2006, 2006.
4. ↑ ***a*** ***b*** Stellman, Andrew; Greene, Jennifer (2005). *Applied Software Project Management*. O'Reilly Media. ISBN 978-0-596-00948-9. http://www.stellman-greene.com/aspm/.
5. ↑ IEEE magazine article "Why Software Fails"
6. ↑ John C. Reynolds, *Some thoughts on teaching programming and programming languages*, SIGPLAN Notices, Volume 43, Issue 11, November 2008, p.108: "Some argue that one can manage software production without the ability to program. This belief seems to arise from the mistaken view that software production is a form of manufacturing. But manufacturing is the repeated construction of identical objects, while software production is the construction

of unique objects, i.e., the entire process is a form of design. As such it is closer to the production of a newspaper — so that a software manager who cannot program is akin to a managing editor who cannot write."
7. ↑ Jørgensen, M.. "A Review of Studies on Expert Estimation of Software Development Effort". http://simula.no/research/engineering/publications/SE.4.Joergensen.2004.c.
8. ↑ Molokken, K. Jorgensen, M.. "A review of software surveys on software effort estimation". http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1237981.
9. ↑ Jørgensen, M. Teigen, K.H. Ribu, K.. "Better sure than safe? Over-confidence in judgement based software development effort prediction intervals". http://www.sciencedirect.com/science?_ob=ArticleURL&_udi=B6V0N-49N06GS-5&_user=674998&_rdoc=1&_fmt=&_orig=search&_sort=d&view=c&_acct=C000036598&_version=1&_urlVersion=0&_userid=674998&md5=36c6383445cf481447d06cb30c1ccb63.
10. ↑ Edwards, J.S. Moores, T.T. (1994), "A conflict between the use of estimating and planning tools in the management of information systems.". European Journal of Information Systems 3(2): 139-147.
11. ↑ Goodwin, P. (1998). Enhancing judgmental sales forecasting: The role of laboratory research. Forecasting with judgment. G. Wright and P. Goodwin. New York, John Wiley & Sons: 91-112.
12. ↑ Farr, L. Nanus, B.. "Factors that affect the cost of computer programming". http://stinet.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=AD0603707.
13. ↑ Nelson, E. A. (1966). Management Handbook for the Estimation of Computer Programming Costs. AD-A648750, Systems Development Corp.
14. ↑ Anda, B. Angelvik, E. Ribu, K.. "Improving Estimation Practices by Applying Use Case Models". http://www.springerlink.com/content/7lpyel912m5cr654/.
15. ↑ Briand, L. C. and I. Wieczorek (2002). Resource estimation in software engineering. Encyclopedia of software engineering. J. J. Marcinak. New York, John Wiley & Sons: 1160-1196.
16. ↑ Jørgensen, M. Shepperd, M.. "A Systematic Review of Software Development Cost Estimation Studies". http://simula.no/research/engineering/publications/Jorgensen.2007.1.
17. ↑ Hill Peter (ISBSG) - Estimation Workbook 2 - published by International Software Benchmarking Standards Group ISBSG - Estimation and Benchmarking Resource Centre
18. ↑ Morris Pam - Overview of Function Point Analysis Total Metrics - Function Point Resource Centre
19. ↑ Shepperd, M. Kadoda, G.. "Comparing software prediction techniques using simulation". http://ieeexplore.ieee.org/Xplore/login.jsp?url=/iel5/32/20846/00965341.pdf?arnumber=965341.
20. ↑ Jørgensen, M.. "Estimation of Software Development Work Effort:Evidence on Expert Judgment and Formal Models". http://simula.no/research/engineering/publications/Jorgensen.2007.2.
21. ↑ Winkler, R.L.. "Combining forecasts: A philosophical basis and some current issues Manager". http://www.sciencedirect.com/science/article/B6V92-45P4G7H-2B/2/d05dc6c369ab173c5792a05ea1be21d9.
22. ↑ Blattberg, R.C. Hoch, S.J.. "Database Models and Managerial Intuition: 50% Model + 50% Manager". http://www.jstor.org/pss/2632364.
23. ↑ Jørgensen, M.. "Estimation of Software Development Work Effort:Evidence on Expert Judgment and Formal Models". http://simula.no/research/engineering/publications/Jorgensen.2007.2.

24.↑ Armstrong, J. S.. "Principles of forecasting: A handbook for researchers and practitioners". http://www.forecastingprinciples.com.
25.↑ Angelis, L. Stamelos, I.. "A simulation tool for efficient analogy based cost estimation". http://portal.acm.org/citation.cfm?id=594467&dl=ACM&coll=portal.
26.↑ Jørgensen, M. Sjøberg, D.I.K.. "An effort prediction interval approach based on the empirical distribution of previous estimation accuracy". http://www.sciencedirect.com/science?_ob=ArticleURL&_udi=B6V0B-47HC6S5-1&_user=674998&_rdoc=1&_fmt=&_orig=search&_sort=d&view=c&_acct=C000036598&_version=1&_urlVersion=0&_userid=674998&md5=6cb917a379855c79eebe9f18ca9ac424.
27.↑ Jørgensen, M.. "Realism in assessment of effort estimation uncertainty: It matters how you ask". http://simula.no/research/engineering/publications/SE.4.Joergensen.2004.e.
28.↑ Shepperd, M. Cartwright, M. Kadoda, G.. "On Building Prediction Systems for Software Engineers". http://www.ingentaconnect.com/content/klu/emse/2000/00000005/00000003/00278191.
29.↑ Kitchenham, B. Pickard, L.M. MacDonell, S.G. Shepperd,. "What accuracy statistics really measure". http://scitation.aip.org/getabs/servlet/GetabsServlet?prog=normal&id=IPSEFU000148000003000081000001&idtype=cvips&gifs=yes.
30.↑ Foss, T. Stensrud, E. Kitchenham, B. Myrtveit, I.. "A Simulation Study of the Model Evaluation Criterion MMRE". IEEE. http://portal.acm.org/citation.cfm?id=951936.
31.↑ Miyazaki, Y. Terakado, M. Ozaki, K. Nozaki, H.. "Robust regression for developing software estimation models". http://portal.acm.org/citation.cfm?id=198684.
32.↑ Lo, B. Gao, X.. "Assessing Software Cost Estimation Models: criteria for accuracy, consistency and regression". http://dl.acs.org.au/index.php/ajis/article/view/348.
33.↑ Hughes, R.T. Cunliffe, A. Young-Martos, F.. "Evaluating software development effort model-building techniquesfor application in a real-time telecommunications environment". http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=689296.
34.↑ Grimstad, S. Jørgensen, M.. "A Framework for the Analysis of Software Cost Estimation Accuracy". http://simula.no/research/engineering/publications/Grimstad.2006.2/simula_pdf_file.
35.↑ Jørgensen, M. Grimstad, S.. "How to Avoid Impact from Irrelevant and Misleading Information When Estimating Software Development Effort". http://simula.no/research/engineering/publications/Simula.SE.112.
36.↑ Kuhn, D.L (1989). "Selecting and effectively using a computer aided software engineering tool". Annual Westinghouse computer symposium; 6–7 Nov 1989; Pittsburgh, PA (U.S.); DOE Project.
37.↑ P. Loucopoulos and V. Karakostas (1995). *System Requirements Engineering*. McGraw-Hill.
38.↑ "AD/Cycle strategy and architecture", IBM Systems Journal, Vol 29, NO 2, 1990; p. 172.
39.↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** Alfonso Fuggetta (December 1993). "A classification of CASE technology". *Computer* **26** (12): 25–38. doi:10.1109/2.247645. http://www2.computer.org/portal/web/csdl/abs/mags/co/1993/12/rz025abs.htm. Retrieved 2009-03-14.
40.↑ ***a*** ***b*** ***c*** ***d*** ***e*** Software Development Techniques. In: *FFIEC InfoBase*. Retrieved 26 Oct 2008.
41.↑ Software Engineering: Tools, Principles and Techniques by Sangeeta Sabharwal, Umesh Publications
42.↑ Evans R. Rock. *Case Analyst Workbenches: A Detailed Product Evaluation*. Volume 1, pp. 229–242 by
43.↑ "IP: The World's First COBOL Compilers". interesting-people.org. 12 June 1997. http://www.interesting-people.org/archives/interesting-people/199706/msg00011.html.

44.↑ T. Hart and M. Levin. "The New Compiler, AIM-39 - CSAIL Digital Archive - Artificial Intelligence Laboratory Series". publications.ai.mit.edu. ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-039.pdf.
45.↑ "The PL/0 compiler/interpreter". http://www.246.dk/pl0.html.
46.↑ "The ACM Digital Library". http://www.acm.org/classics/dec95/.
47.↑ T diagrams were first introduced for describing bootstrapping and cross-compiling compilers in McKeeman et al. *A Compiler Generator* (1971). Conway described the broader concept before that with his UNCOL in 1958, to which Bratman added in 1961: H. Bratman, "An alternate form of the ´UNCOL diagram´", Comm. ACM 4 (March 1961) 3, p. 142. Later on, others, including P.D. Terry, gave an explanation and usage of T-diagrams in their textbooks on the topic of compiler construction. Cf. Terry, 1997, Chapter 3. T-diagrams are also now used to describe client-server interconnectivity on the World Wide Web: cf. Patrick Closhen, et al. 1997: *T-Diagrams as Visual Language to Illustrate WWW Technology*, Darmstadt University of Technology, Darmstadt, Germany
48.↑ ETAPS - European Joint Conferences on Theory and Practice of Software. Cf. "CC" (Compiler Construction) subsection.
49.↑ http://www.berniecode.com/blog/2007/03/08/how-to-debug-javascript-with-visual-web-developer-express/
50.↑ Dana Nourie (2005-03-24). "Getting Started with an Integrated Development Environment". Sun Microsystems, Inc.. http://java.sun.com/developer/technicalArticles/tools/intro.html. Retrieved 2008-09-09.
51.↑ Image credit: Museum of Information Technology at Arlington
52.↑ "Interaktives Programmieren als Systems-Schlager" from *Computerwoche* (German)
53.↑ "Rapid Subversion Adoption Validates Enterprise Readiness and Challenges Traditional Software Configuration Management Leaders". Collabnet. May 15, 2007. http://www.open.collab.net/news/press/2007/svn_momentum.html. Retrieved October 27, 2010. "Version management is essential to software development and is considered the most critical component of any development environment."
54.↑ Wheeler, David. "Comments on Open Source Software / Free Software (OSS/FS) Software Configuration Management (SCM) Systems". http://www.dwheeler.com/essays/scm.html. Retrieved May 8, 2007.
55.↑ ***a*** ***b*** O'Sullivan, Bryan. "Distributed revision control with Mercurial". http://hgbook.red-bean.com/hgbook.html. Retrieved July 13, 2007.
56.↑ Collins-Sussman, Ben; Fitzpatrick, B.W. and Pilato, C.M. (2004). *Version Control with Subversion*. O'Reilly. ISBN 0-596-00448-6. http://svnbook.red-bean.com/.
57.↑ Wingerd, Laura (2005). *Practical Perforce*. O'Reilly. ISBN 0-596-10185-6. http://safari.oreilly.com/0596101856.
58.↑ ***a*** ***b*** Collins-Sussman, Ben; Brian W. Fitpatrick, and C. Michael Pilato. "Version Control with Subversion". http://svnbook.red-bean.com/en/1.5/svn.tour.cycle.html#svn.tour.cycle.resolve. Retrieved 8 June 2010. "The G stands for merGed, which means that the file had local changes to begin with, but the changes coming from the repository didn't overlap with the local changes."
59.↑ *Accurev Concepts Manual, Version 4.7*. Accurev, Inc.. July, 2008.
60.↑ http://www.gnu.org/software/make/
61.↑ *Dr. Dobb's Distributed Loadbuilds*, http://www.ddj.com/architect/184405385, retrieved 2009-04-13
62.↑ *Dr. Dobb's Take My Build, Please*, http://www.ddj.com/architect/184415472
63.↑ *LSF User's Guide - Using lsmake*, http://www.lle.rochester.edu/pub/support/lsf/10-lsmake.html, retrieved 2009-04-13

64.↑ *Distributed Visual Studio Builds*, http://www.xoreax.com/solutions_vs.htm, retrieved 2009-04-08
65.↑ *CMake - Cross platform make*, http://www.cmake.org/, retrieved 2010-03-27
66.↑ http://www.denverjug.org/meetings/files/200410_automation.pdf
67.↑ http://freshmeat.net/articles/view/392/
68.↑ http://www.ibm.com/developerworks/java/library/j-junitmail/
69.↑ http://buildbot.net/trac
70.↑ http://www.cmcrossroads.com/content/view/12525/120/
71.↑ Woelz, Carlos. "The KDE Documentation Primer". http://i18n.kde.org/docs/doc-primer/index.html. Retrieved 15 June 2009.
72.↑ Microsoft. "Knowledge Base Articles for Driver Development". http://www.microsoft.com/whdc/driver/kernel/kb-drv.mspx. Retrieved 15 June 2009. Template:Dead link
73.↑ Prekaski, Todd. "Building web and Adobe AIR applications from a shared Flex code base". http://www.adobe.com/devnet/air/flex/articles/flex_air_codebase.html. Retrieved 15 June 2009.
74.↑ "Static Analysis in Xcode". Apple. http://developer.apple.com/mac/library/featuredarticles/StaticAnalysis/index.html. Retrieved 2009-09-03.
75.↑ Parasoft Application Security Solution
76.↑ Parasoft Compliance Solution
77.↑ Cousot, Patrick (2007). "The Role of Abstract Interpretation in Formal Methods". IEEE International Conference on Software Engineering and Formal Methods. http://ieeexplore.ieee.org/Xplore/login.jsp?url=http%3A%2F%2Fieeexplore.ieee.org%2Fiel5%2F4343908%2F4343909%2F04343930.pdf%3Farnumber%3D4343930&authDecision=-203. Retrieved 2010-11-08.
78.↑ *gprof: a Call Graph Execution Profiler*
79.↑ Atom: A system for building customized program analysis tools, Amitabh Srivastava and Alan Eustace, 1994 (download)
80.↑ 20 Years of PLDI (1979 - 1999): A Selection, Kathryn S. McKinley, Editor
81.↑ Statistical Inaccuracy of `gprof` Output
82.↑ Kolawa, Adam; Huizinga, Dorota (2007). *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press. p. 254. ISBN 0470042125. http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470042125.html.
83.↑ RTCA/DO-178(b), *Software Considerations in Airborne Systems and Equipment Certification, Radio Technical Commission for Aeronautics*, December 1, 1992.
84.↑ Glenford J. Myers (2004). *The Art of Software Testing, 2nd edition*. Wiley. ISBN 0471469122.
85.↑ [8]
86.↑ M. R. Woodward, M. A. Hennell, "On the relationship between two control-flow coverage criteria: all JJ-paths and MCDC", Information and Software Technology 48 (2006) pp. 433-440
87.↑ Dorf, Richard C.: *Computers, Software Engineering, and Digital Devices*, Chapter 12, pg. 15. CRC Press, 2006. ISBN 0849373409, 9780849373404; via Google Book Search
88.↑ Software Considerations in Airborne System and Equipment Certification-RTCA/DO-178B, RTCA Inc., Washington D.C., December 1992
89.↑ Cunningham, Ward (05 Aug 2009). "Integration Hell". *WikiWikiWeb*. http://c2.com/cgi/wiki?IntegrationHell. Retrieved 19 Sept 2009.

90.↑ Brauneis, David (01 January 2010). "[OSLC Possible new Working Group - Automation"]. *open-services.net Community mailing list*. http://open-services.net/pipermail/community_open-services.net/2010-January/000214.html. Retrieved 16 February 2010.
91.↑ Taylor, Bradley. "Rails Deployment and Automation with ShadowPuppet and Capistrano". http://blog.railsmachine.com/articles/2009/02/10/rails-deployment-and-automation-with-shadowpuppet-and-capistrano/.
92.↑ Fowler, Martin. "Continuous Integration". http://martinfowler.com/articles/continuousIntegration.html#PracticesOfContinuousIntegration. Retrieved 2009-11-11.
93.↑ See http://radar.oreilly.com/2009/03/continuous-deployment-5-eas.html and http://timothyfitz.wordpress.com/2009/02/10/continuous-deployment-at-imvu-doing-the-impossible-fifty-times-a-day/
94.↑ Fowler, Martin. "Continuous Integration". http://www.martinfowler.com/articles/continuousIntegration.html.
95.↑ Beck, Kent (1999). *Extreme Programming Explained*. ISBN 0-201-61641-6.
96.↑ Richardson, Jared (September 2008). "Agile Testing Strategies at No Fluff Just Stuff Conference". Boston, Massachusetts. http://www.nofluffjuststuff.com.
97.↑ Joel Spolsky (November 08 2000). "Painless Bug Tracking". http://www.joelonsoftware.com/articles/fog0000000029.html. Retrieved 29 October 2010.
98.↑ Multiple (wiki). "Bug report". *Docforge*. http://docforge.com/wiki/Bug_report. Retrieved 2010-03-09.
99.↑ Jonathan Corbet (May 14 2008). "Distributed bug tracking". *LWN.net*. http://lwn.net/Articles/281849/. Retrieved 7 January 2009.
100.↑ "FogBugz Features". *Fogbugz.com*. http://www.fogcreek.com/FogBugz/learnmore.html. Retrieved 2010-10-29.
101.↑ Joey Hess (6 April 2007). "Integrated issue tracking with Ikiwiki". *LinuxWorld.com*. IDG. http://www.linuxworld.com/news/2007/040607-integrated-issue-tracking-ikiwiki.html. Retrieved 7 January 2009.
102.↑ "Why Decompilation". Program-transformation.org. 2005-04-11. http://www.program-transformation.org/Transform/WhyDecompilation. Retrieved 2010-09-15.
103.↑ Miecznikowski, Jerome (2002). "Decompiling Java Bytecode: Problems, Traps and Pitfalls". in R Nigel Horspool. *Compiler Construction: 11th International Conference, proceedings / CC 2002*. Springer-Verlag. pp. 111–127. ISBN 3-540-43369-4.
104.↑ Cifuentes, Cristina; Gough, K. John (July 1995). "Decompilation of Binary Programs". *Software Practice and Experience* **25** (7): 811–829. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.8073&rep=rep1&type=pdf.
105.↑ Mycroft, Alan (1999). "Type-Based Decompilation". in S. Doaitse Swierstra. *Programming languages and systems: 8th European Symposium on Programming Languages and Systems*. Springer-Verlag. pp. 208–223. ISBN 3-540-65699-5.
106.↑ C. Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994. (available as compressed postscript Chapter 6)
107.↑ Rowland, Diane (2005). *Informtion technology law* (3rd ed.). Cavendish. ISBN 1-85941-756-6.
108.↑ "The Legality of Decompilation". Program-transformation.org. 2004-12-03. http://www.program-transformation.org/Transform/LegalityOfDecompilation. Retrieved 2010-09-15.
109.↑ B. Czarnota and R.J. Hart, *Legal protection of computer programs in Europe: a guide to the EC directive*. 1991, London: Butterworths.
110.↑ Obfuscation: Cloaking your Code from Prying Eyes
111.↑ Jeff Atwood, May 15 2005

112.↑ Obfuscation
113.↑ C++ Tutorials - Obfuscated Code - A Simple Introduction | DreamInCode.net
114.↑ Pe(a)rls in line noise
115.↑ [The International Obfuscated C Code Contest]
116.↑ "International Obfuscated C Code Winners 1988 - Least likely to compile successfully"
117.↑ "Reverse Engineering the Twelve Days of Christmas" by Thomas Ball
118.↑ Don Libes, *Obfuscated C and Other Mysteries*, John Wiley & Sons, 1993, pp 425. ISBN 0-471-57805-3
119.↑ "Can We Obfuscate Programs?" by Boaz Barak
120.↑ Open Directory - Computers: Programming: Languages: JavaScript: Tools: Obfuscators
121.↑ Open Directory - Computers: Programming: Languages: PHP: Development Tools: Obfuscation and Encryption
122.↑ Open Directory - Computers: Programming: Languages: Java: Development Tools: Obfuscators
123.↑ Open Directory - Computers: Programming: Component Frameworks: .NET: Tools: Obfuscators
124.↑ Cloakware Application Security
125.↑ Morpher - Compiler Driven Obfuscation
126.↑ Chikofsky, E. and Cross, J., 1990. Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software, 7(1):13-18.
127.↑ Asit Kumar Gahalaut et. al. / International Journal of Engineering Science and Technology Vol. 2(06), 2010, 2296-2303

- Robert S.Arnold: "Software reengineering", IEEE Computer Society Press, 1993
- Object Management Group, Knowledge Discovery Metamodel (KDM) 1.0 specification, 2007

External links

- The Program Transformation Wiki
- The Architecture-Driven Modernization website at OMG
- Re-Engineer SAP Implementation, with Developments, Configuration, Roles and DDIC

Reverse Engineering

Reverse engineering is the process of discovering the technological principles of a human made device, object or system through analysis of its structure, function and operation. It often involves taking something (e.g., a mechanical device, electronic component, or software program) apart and analyzing its workings in detail to be used in maintenance, or to try to make a new device or program that does the same thing without using or simply duplicating (without understanding) any part of the original.

Reverse engineering has its origins in the analysis of hardware for commercial or military advantage.[1] The purpose is to deduce design decisions from end products with little or no additional knowledge about the procedures involved in the original production. The same

techniques are subsequently being researched for application to legacy software systems, not for industrial or defence ends, but rather to replace incorrect, incomplete, or otherwise unavailable documentation.[2]

Motivation

Reasons for reverse engineering:

- Interoperability.
- Lost documentation: Reverse engineering often is done because the documentation of a particular device has been lost (or was never written), and the person who built it is no longer available. Integrated circuits often seem to have been designed on obsolete, proprietary systems, which means that the only way to incorporate the functionality into new technology is to reverse-engineer the existing chip and then re-design it.
- Product analysis. To examine how a product works, what components it consists of, estimate costs, and identify potential patent infringement.
- Digital update/correction. To update the digital version (e.g. CAD model) of an object to match an "as-built" condition.
- Security auditing.
- Acquiring sensitive data by disassembling and analysing the design of a system component.[3]
- Military or commercial espionage. Learning about an enemy's or competitor's latest research by stealing or capturing a prototype and dismantling it.
- Removal of copy protection, circumvention of access restrictions.
- Creation of unlicensed/unapproved duplicates.
- Materials harvesting, sorting, or scrapping.[4]
- Academic/learning purposes.
- Curiosity.
- Competitive technical intelligence (understand what your competitor is actually doing versus what they say they are doing).
- Learning: learn from others' mistakes. Do not make the same mistakes that others have already made and subsequently corrected.

Reverse engineering of machines

As computer-aided design (CAD) has become more popular, reverse engineering has become a viable method to create a 3D virtual model of an existing physical part for use in 3D CAD, CAM, CAE or other software.[5] The reverse-engineering process involves measuring an object and then reconstructing it as a 3D model. The physical object can be measured using 3D scanning technologies like CMMs, laser scanners, structured light digitizers or Industrial CT Scanning (computed tomography). The measured data alone, usually represented as a point cloud, lacks topological information and is therefore often processed and modeled into a more usable format such as a triangular-faced mesh, a set of NURBS surfaces or a CAD model.

Reverse engineering is also used by businesses to bring existing physical geometry into digital product development environments, to make a digital 3D record of their own products or to assess competitors' products. It is used to analyse, for instance, how a product works, what it does, and what components it consists of, estimate costs, and identify potential patent infringement, etc.

Value engineering is a related activity also used by businesses. It involves de-constructing and analysing products, but the objective is to find opportunities for cost cutting.

Reverse engineering of software

The term *reverse engineering* as applied to software means different things to different people, prompting Chikofsky and Cross to write a paper researching the various uses and defining a taxonomy. From their paper, they state, "Reverse engineering is the process of analyzing a subject system to create representations of the system at a higher level of abstraction."[6] It can also be seen as "going backwards through the development cycle".[7] In this model, the output of the implementation phase (in source code form) is reverse-engineered back to the analysis phase, in an inversion of the traditional waterfall model. Reverse engineering is a process of examination only: the software system under consideration is not modified (which would make it re-engineering). Software anti-tamper technology is used to deter both reverse engineering and re-engineering of proprietary software and software-powered systems. In practice, two main types of reverse engineering emerge. In the first case, source code is already available for the software, but higher-level aspects of the program, perhaps poorly documented or documented but no longer valid, are discovered. In the second case, there is no source code available for the software, and any efforts towards discovering one possible source code for the software are regarded as reverse engineering. This second usage of the term is the one most people are familiar with. Reverse engineering of software can make use of the clean room design technique to avoid copyright infringement.

On a related note, black box testing in software engineering has a lot in common with reverse engineering. The tester usually has the API, but their goals are to find bugs and undocumented features by bashing the product from outside.

Other purposes of reverse engineering include security auditing, removal of copy protection ("cracking"), circumvention of access restrictions often present in consumer electronics, customization of embedded systems (such as engine management systems), in-house repairs or retrofits, enabling of additional features on low-cost "crippled" hardware (such as some graphics card chip-sets), or even mere satisfaction of curiosity.

The Certified Reverse Engineering Analyst (CREA) is a certification provided by the IACRB that certifies candidates are proficient in reverse engineering software.

Binary software

This process is sometimes termed *Reverse Code Engineering*, or RCE.[8] As an example, decompilation of binaries for the Java platform can be accomplished using Jad. One famous case of reverse engineering was the first non-IBM implementation of the PC BIOS which launched the historic IBM PC compatible industry that has been the overwhelmingly dominant computer hardware platform for many years. An example of a group that reverse-engineers software for enjoyment (and to distribute registration cracks) is CORE which stands for "Challenge Of Reverse

Engineering". Reverse engineering of software is protected in the U.S. by the fair use exception in copyright law.[9] The Samba software, which allows systems that are not running Microsoft Windows systems to share files with systems that are, is a classic example of software reverse engineering,[10] since the Samba project had to reverse-engineer unpublished information about how Windows file sharing worked, so that non-Windows computers could emulate it. The Wine project does the same thing for the Windows API, and OpenOffice.org is one party doing this for the Microsoft Office file formats. The ReactOS project is even more ambitious in its goals, as it strives to provide binary (ABI and API) compatibility with the current Windows OSes of the NT branch, allowing software and drivers written for Windows to run on a clean-room reverse-engineered GPL free software or open-source counterpart.

Binary software techniques

Reverse engineering of software can be accomplished by various methods. The three main groups of software reverse engineering are

1. Analysis through observation of information exchange, most prevalent in protocol reverse engineering, which involves using bus analyzers and packet sniffers, for example, for accessing a computer bus or computer network connection and revealing the traffic data thereon. Bus or network behavior can then be analyzed to produce a stand-alone implementation that mimics that behavior. This is especially useful for reverse engineering device drivers. Sometimes, reverse engineering on embedded systems is greatly assisted by tools deliberately introduced by the manufacturer, such as JTAG ports or other debugging means. In Microsoft Windows, low-level debuggers such as SoftICE are popular.
2. Disassembly using a disassembler, meaning the raw machine language of the program is read and understood in its own terms, only with the aid of machine-language mnemonics. This works on any computer program but can take quite some time, especially for someone not used to machine code. The Interactive Disassembler is a particularly popular tool.
3. Decompilation using a decompiler, a process that tries, with varying results, to recreate the source code in some high-level language for a program only available in machine code or bytecode.

Source code

A number of UML tools refer to the process of importing and analysing source code to generate UML diagrams as "reverse engineering". See List of UML tools.

Reverse engineering of protocols

Protocols are sets of rules that describe message formats and how messages are exchanged (i.e., the protocol state-machine). Accordingly, the problem of protocol reverse-engineering can be partitioned into two subproblems; message format and state-machine reverse-engineering.

The message formats have traditionally been reverse-engineered through a tedious manual process, which involved analysis of how protocol implementations process messages, but recent research proposed a number of automatic solutions [11][12][13]. Typically, these automatic approaches either group observed messages into clusters using various clustering analyses, or emulate the protocol implementation tracing the message processing.

There has been less work on reverse-engineering of state-machines of protocols. In general, the protocol state-machines can be learned either through a process of offline learning, which passively observes communication and attempts to build the most general state-machine accepting all observed sequnces of messages, and online learning, which allows interactive generation of probing sequences of messages and listening to responses to those probing sequences. In general, offline learning of small state-machines is known to be NP-complete [14], while online learning can be done in polynomial time [15]. An automatic offline approach has been demonstrated by Comparetti at al.[13]. and an online approach very recently by Cho et al.[16].

Other components of typical protocols, like encryption and hash functions, can be reverse-engineered automatically as well. Typically, the automatic approaches trace the execution of protocol implementations and try to detect buffers in memory holding unencrpyted packets [17].

Reverse engineering of integrated circuits/smart cards

Reverse engineering is an invasive and destructive form of analyzing a smart card. The attacker grinds away layer by layer of the smart card and takes pictures with an electron microscope. With this technique, it is possible to reveal the complete hardware and software part of the smart card. The major problem for the attacker is to bring everything into the right order to find out how everything works. Engineers try to hide keys and operations by mixing up memory positions, for example, busscrambling.[18][19] In some cases, it is even possible to attach a probe to measure voltages while the smart card is still operational. Engineers employ sensors to detect and prevent this attack.[20] This attack is not very common because it requires a large investment in effort and special equipment that is generally only available to large chip manufacturers. Furthermore, the payoff from this attack is low since other security techniques are often employed such as shadow accounts.

Reverse engineering for military applications

Reverse engineering is often used by militaries in order to copy other nations' technologies, devices or information that have been obtained by regular troops in the fields or by intelligence operations. It was often used during the Second World War and the Cold War. Well-known examples from WWII and later include

- Jerry can: British and American forces noticed that the Germans had gasoline cans with an excellent design. They reverse-engineered copies of those cans. The cans were popularly known as "Jerry cans".
- Tupolev Tu-4: Three American B-29 bombers on missions over Japan were forced to land in the USSR. The Soviets, who did not have a similar strategic bomber, decided to copy the B-29. Within a few years, they had developed the Tu-4, a near-perfect copy.

- V2 Rocket: Technical documents for the V2 and related technologies were captured by the Western Allies at the end of the war. Soviet and captured German engineers had to reproduce technical documents and plans, working from captured hardware, in order to make their clone of the rocket, the R-1, which began the postwar Soviet rocket program that led to the R-7 and the beginning of the space race.
- Vympel K-13/R-3S missile (NATO reporting name **AA-2 Atoll**), a Soviet reverse-engineered copy of the AIM-9 Sidewinder, made possible after a Taiwanese AIM-9B hit a Chinese MiG-17 without exploding; amazingly, the missile became lodged within the airframe, the pilot returning to base with what Russian scientists would describe as a university course in missile development.
- BGM-71 TOW Missile: In May 1975, negotiations between Iran and Hughes Missile Systems on co-production of the TOW and Maverick missiles stalled over disagreements in the pricing structure, the subsequent 1979 revolution ending all plans for such co-production. Iran was later successful in reverse-engineering the missile and are currently producing their own copy: the Toophan.
- China has reversed engineered many examples of Western and Russian hardware, from fighter aircraft to missiles and HMMWV cars.

Legality

In the United States even if an artifact or process is protected by trade secrets, reverse-engineering the artifact or process is often lawful as long as it is obtained legitimately.[21] Patents, on the other hand, need a public disclosure of an invention, and therefore, patented items do not necessarily have to be reverse-engineered to be studied. (However, an item produced under one or more patents could also include other technology that is not patented and not disclosed.) One common motivation of reverse engineers is to determine whether a competitor's product contains patent infringements or copyright infringements.

The reverse engineering of software in the US is generally illegal because most EULA prohibit it, and courts have found such contractual prohibitions to override the copyright law; see Bowers v. Baystate Technologies.[22][23] Article 6 of the 1991 EU Computer Programs Directive allows reverse engineering for the purposes of interoperability, but prohibits it for the purposes of creating a competing product, and also prohibits the public release of information obtained through reverse engineering of software.[24][25][26]

See also

- Antikythera mechanism
- Benchmarking
- Bus analyzer
- Chonda
- Clean room design
- Code morphing
- Connectix Virtual Game Station
- Decompiler
- Digital Millennium Copyright Act (DMCA)

- Forensic engineering
- Interactive Disassembler
- Knowledge Discovery Metamodel
- List of production topics
- Logic analyzer
- Paycheck (film)
- Value engineering
- Cryptanalysis
- Software archaeology

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

Further reading

- Eilam, Eldad (2005). *Reversing: Secrets of Reverse Engineering*. Wiley Publishing. pp. 595. ISBN 0764574817.
- James, Dick (January 19, 2006). "Reverse Engineering Delivers Product Knowledge; Aids Technology Spread". *Electronic Design*. Penton Media, Inc. http://electronicdesign.com/Articles/Index.cfm?AD=1&ArticleID=11966. Retrieved 2009-02-03.
- Raja, Vinesh; Fernandes, Kiran J. (2008). *Reverse Engineering - An Industrial Perspective*. Springer. pp. 242. ISBN 978-1-84628-855-5.
- Thumm, Mike (2007). "Talking Tactics". *IEEE 2007 Custom Integrated Circuits Conference (CICC)*. IEEE, Inc. http://ewh.ieee.org/r5/denver/sscs/References/2007_09_Torrance.pdf. Retrieved 2009-02-03.
- Cipresso, Teodoro (2009). "Software Reverse Engineering Education". *SJSU Master's Thesis*. ProQuest UML. http://www.reversingproject.info. Retrieved 2009-08-22.

External links

- What Is Reverse Engineering define more nicely here

- Java Call Trace to UML Sequence Diagram A reverse engineering tool for Java. This tool helps you to reverse engineer UML Sequence Diagram for your java program at runtime. It works well with both complex java programs (that have multiple threads) and J2EE applications deployed on Application Servers.
- CASE Tools for Reverse Code Engineering
- The Reverse Code Engineering Community

Round-trip Engineering

Round-trip engineering (RTE) is a functionality of software development tools that synchronizes two or more related software artifacts, such as, source code, models, configuration files, and other documents. The need for round-trip engineering arises when the same information is present in multiple artifacts and therefore an inconsistency may occur if not all artifacts are consistently updated to reflect a given change. For example, some piece of information was added to/changed in only one artifact and, as a result, it became missing in/inconsistent with the other artifacts.

Round-trip engineering is closely related to traditional software engineering disciplines: forward engineering (creating software from specifications), reverse engineering (creating specifications from existing software), and reengieering (understanding existing software and modifying it). Round-trip engineering is often wrongly defined as simply supporting both forward and reverse engineering. In fact, the key characteristic of round-trip engineering that distinguishes it from forward and reverse engineering is the ability to synchronize **existing** artifacts that evolved **concurrently** by **incrementally** updating each artifact to reflect changes made to the other artifacts. Furthermore, forward engineering can be seen as a special instance of RTE in which only the specification is present and reverse engineering can be seen as a special instance of RTE in which only the software is present. Many reengineering activities can also be understood as RTE when the software is updated to reflect changes made to the previously reverse engineered specification.

Another characteristic of round-trip engineering is **automatic** update of the artifacts in response to **automatically** detected inconsistencies. In that sense, it is different from forward- and reverse engineering which can be both manual (traditionally) and automatic (via automatic generation or analysis of the artifacts). The automatic update can be either **instantaneous** or **on-demand**. In instantaneous RTE, all related artifacts are immediately updated after each change made to one of them. In on-demand RTE, authors of the artifacts may concurrently evolve the artifacts (even in a distributed setting) and at some point choose to execute matching to identify inconsistencies and choose to propagate some of them and reconcile potential conflicts.

Examples of round-trip engineering

Perhaps the most common form of round-trip engineering is synchronization between UML (Unified Modeling Language) models and the corresponding source code. Many commercial tools and research prototypes (e.g., FUJABA) support this form of RTE. Usually, UML class diagrams are supported to some degree; however, certain UML concepts, such as *associations* and *containment* do not have straightforward representations in many programming languages which limits the usability of the created code and accuracy of code analysis (e.g., containment is hard to recognize in the code). Behavioral parts of UML impose even more challenges for RTE.

A more tractable form of round-trip engineering is implemented in the context of framework application programming interfaces (APIs), whereby a model describing the usage of a framework API by an application is synchronized with that application's code. In this setting, the API **prescribes** all correct ways the framework can be used in applications, which allows precise and complete detection of API usages in the code as well as creation of useful code implementing correct API usages. Two prominent RTE implementations in this category are framework-specific modeling languages and Spring Roo.

Round-trip engineering is critical for maintaining consistency among multiple models and between the models and the code in Object Management Group's (OMG) Model-driven architecture. OMG proposed the QVT (query/view/transformation) standard to handle model transformations required for MDA. To date, a few implementations of the standard have been created. (Need to present practical experiences with MDA in relation to RTE).

References

1. ↑ ***a*** ***b*** ***c*** ***d*** ***e*** ***f*** ***g*** ***h*** *Systems Engineering Fundamentals.* Defense Acquisition University Press, 2001
2. ↑ Executive editors: Alain Abran, James W. Moore; editors Pierre Bourque, Robert Dupuis, ed (March 2005). "Chapter 2: Software Requirements". *Guide to the software engineering body of knowledge* (2004 ed.). Los Alamitos, CA: IEEE Computer Society Press. ISBN 0-7695-2330-7. http://www.computer.org/portal/web/swebok/html/ch2. Retrieved 2007-02-08. "It is widely acknowledged within the software industry that software engineering projects are critically vulnerable when these activities are performed poorly."
3. ↑ Wiegers, Karl E. (2003). *Software Requirements* (2nd ed.). Redmond, WA: Microsoft Press. ISBN 0-7356-1879-8. http://www.processimpact.com.
4. ↑ Phillip A. Laplante (2007) *What Every Engineer Should Know about Software Engineering*. Page 44.

External links

- UML Round-trip Engineering with Altova

Authors

This book is a major collaboration effort. It is based on Wikipedia articles that have been authored over the years by many individuals. We have been trying to keep the editing histories (which was not always possible), and from those we have generated the lists below. Please note, that IP addresses have been removed. If you feel that your name is missing in the list, please add it, using the 'Edit' button.

Primary Authors (more than 50 edits)

Adrignola; Andreas Kaufmann; Derek farn; Frecklefoot; Kuru; Mdd; Michig; MikeDunlavey; MrOllie; Nigelj; Ptrb; Rplano; SimonTrew; SmackBot; Walter Görlitz;

Major Authors (between 10 and 49 edits)

.digamma; Aagtbdfoua; AliveFreeHappy; Allan McInnes; AllanBz; Andy Dingley; AntiVandalBot; Antonielly; Architectchao; Ash; Beetstra; Beland; Bluebot; Cander0000; Chobot; ClueBot; ClueBot NG; Conan; Cybercobra; Dalvizu; Daniel.Cardenas; Derek Ross; Dmcq; DRogers; DSParillo; Dwchin; Ed Poor; Edward; Edward Z. Yang; Erkan Yilmaz; Everything counts; Favonian; FlashSheridan; Fnegroni; Forgotten gentleman; Fredrik; Furrykef; Gandalfgeek; GopiTaylor; Greenrd; Gwern; Haakon; Hariharan wiki; Hervegirod; Hu12; Iterator12n; Jamelan; JAnDbot; JDBravo; Jfire; JillFine; JLaTondre; Jmabel; Jpbowen; Kayau; Kdakin; Khalid hassani; Kku; Kubanczyk; Lakeworks; Liao; Ligulem; Littlesal; Lprichar; Luckas-bot; M4gnum0n; Malleus Fatuorum; Mark Renier; Marudubshinki; Michael Hardy; MickeyWiki; Mike Van Emmerik; Milkfish; Mkoval; Mmeijeri; Moa3333; Neilc; Normxxx; Oicumayberight; Ojw; On5deu; Pcap; Pinecar; PJTraill; Pm master; PradeepArya1109; Raul654; Renesis; Rich Farmbrough; Ripe; Rjwilmsi; Robbot; Ronz; Ruud Koot; Rwwww; S.K.; Stemcd; SteveLoughran; Stumps; Swtechwr; TakuyaMurata; Technobadger; Tedickey; The Anome; Thumperward; Thv; Tobias Bergemann; Tracyragan; Tromp; Van der Hoorn; Vedatcoskun; VolkovBot; Wei.cs; Wernher; Wikid77; William Pietri; XLinkBot; Yobot; YurikBot;

Minor Authors (less than 10 edits)

100110100; (; 16@r; 16x9; 1exec1; 1ForTheMoney; 1sraghavan; 2004-12-29T22:45Z; 2fort5r; 4johnny; 4twenty42o; 5 albert square; 6birc; 777sms; 9Nak; A plague of rainbows; A. B.; A.R.; A5b; Aacool; Aalvarez; Aaron Schulz; Aaronbrick; Aasch; Abdull; Abednigo; Abridge; AbsolutDan; Abtris; Acagney; Ace Coder; AceCalihan; Achalmeena; Achorny; Acockrum; Acroterion; ActiveState; Ad88110; Adair2324; Adam messinger; Adamdaley; Adamleggett; Adashiel; Addere; Addshore; Adityasinghhh; Adler.fa; Adri Timp; Adriatikus; Aeonx; Aesculaepius; Afbcasejr; Agasta; Age Happens; Agencius; Agentbla; Ahc; Ahoerstemeier; Ahy1; Aidinnz; Aij; Aislingdonnelly; Aitias; Aivosto; Ajcheng; Akamad; AKGhetto; Akhristov; Akumiszcza; Alai; Alaibot; Alainr345; Alan ffm; Alan McBeth; Alan Peakall; Alanmossman; AlannY; Alanpc1; Alansohn; ALargeElk; AlarmTripper; Alastaird; Alberto.scarpa; AlephGamma; Alerante; Alex; Alex K. Angelopoulos; Alex Nadtoka; Alexbot; Alexf; AlexMorozov; AlexR; Alextelea; Alfio; Algomaster; Alksentrs; Alksub; Alla tedesca; AlleborgoBot; Allen Moore; AllenDowney; Almabot; ALMGuru; AlnoktaBOT; Alopez6; Alpha0; Altenmann; Alternamke;

Altonbr; Aludstartups; AMackenzie; Amakuha; Amire80; Amirobot; Amitch; Amp wpg; AmphBot; Anaxial; Ancheta Wis; Anchor Link Bot; Anclation; Andareed; Anderbubble; Anders.Warga; AndersBot; Anderswiki; Andika; Andmatt; Andre Engels; Andrea105; Andreas Toth; Andrei Stroe; Andresmlinar; Andrewpmk; AndrewStellman; AndrewWPhillips; AndriuZ; Andrpere; AndyDent; AndyGavin; Andyops; Andypandy.UK; Andysimple; Anetode; AngelaMartin2008; AngeloCoppola; Angusgr; AnhadSingh; AnomieBOT; Anorthup; Anrie Nord; Antaeus Feldspar; Antandrus; Anthony Appleyard; Anthony Fok; AnthonySteele; AntiSpamBot; Antonmind; Anujgoyal; Anwar saadat; Apanait; Apolitano; Aponar Kestrel; Appljax123; Arauzo; ArchDC; ArchonMagnus; Ardonik; Ariadnadionisos; Ariconte; Armadillo-eleven; ArmadilloFromHell; Arminius; ArmixZ; Arnadí; Arneeiri; Arny; Aron.gombas; ArroLu; Art Carlson; Arthena; ArthurBot; Artw; Artyom; Arvin Schnell; Arxantech; AS; Asavoia; Ascánder; Asgeirn; Ashe the Cyborg; Ashishlohorung; Ashwinstudying; Asmitford; Aspects; Astaines; Astronouth7303; Aternity; AThing; Athought; Atreys; Atroche; Attilios; Atul1612; Auntof6; Auric; AussieScribe; Austinm; Autarch; Auteurs; AutumnSnow; AVand; Avochelm; AWendt; AxelBusch; Aymatth2; Ayudante; Babynus; Bakersg13; Bapim; Barcelova; Bardiax; Barkeep; Barneca; BarretBonden; Barticus88; Batsonjay; BBB; Bc510862; Bcwhite; Bdijkstra; Beao; Bebero22; Beccus; Beefman; BeL1EveR; Belma06; BenAveling; BenBaker; BenediktG; Benfellows; BenFrantzDale; Benjaminevans82; BenjaminTsai; BenLiyanage; Beno1000; Benoit.dechateauvieux; Benzbpolo; BenzolBot; Berland; Bernard François; Bernd in Japan; Bernopedia; Berrinam; Bertport; Bet Bass; Betacommand; Bevo; Bfalese; Bgarcia1; Bheron; BigBen212; Bigbluefish; BigMikeW; BigNate37; BillGosset; BillyColl; Billyoneal; Bingbangbong; BioPupil; BIS Ondrej; Bit; Bjorn Elenfors; Bkil; Black-Velvet; Blacklily; BlackMamba; Blaisorblade; Blanchardb; Blathnaid; Blaxthos; Blowdart; Bluemoose; BlueNovember; Bnmike; Bobatwiki; Bobblehead; Bobianite; Bobo192; BoBrandt; BoD; BodhisattvaBot; Bokaal; Boly38; Bonadea; Bonatto; Bones000sw; Bookofjude; Booyabazooka; Borgx; Bota47; BOTarate; BotMultichill; Bovlb; Boxplot; Bpp198; Bradkittenbrink; BrainyBabe; Brandon; Breandandalton; Brent Gulanowski; Brentwills; Brettright; Brian Geppert; Brian R Hunter; Brian.edmond; Brick Thrower; BritishWatcher; Brockert; Brownout; Bruce89; Brucevdk; Bruno Unna; Brunoton; Brusselsshrek; Bryan Derksen; Bryan.dollery; Bryankennedy; Bryanws; Bsadowski1; Bsonderg; Btyner; Bubba73; Bugaware; BullRangifer; Bunyk; BurntSky; BuSchu; Buttonius; Bwefler; C xong; Caerwine; CaliforniaAliBaba; Callidior; Calor; Calréfa Wéná; Calton; Calum MacÙisdean; CambridgeBayWeather; Cameltrader; Camw; Can't sleep, clown will eat me; Canderra; CanisRufus; Canterbury Tail; Caporaletti; Capricorn42; Captone; Caputt; Carlesso; Carlkoppel; CarlManaster; Carlo.milanesi; Carmencr; Cassbeth; Cerrol; CesarB; Cetinsert; Cferrero; CFMWiki1; Cgs; Chandresa; ChangChienFu; Chaos5023; Charan.thinkahead; Charivari; Charles Merriam; Charles T. Betz; CharlesC; Charon.sk; Chatfecter; Chealer; Checkshirt; CheeseSucker; Chelseafan528; ChenzwBot; Cherry blossom tree; Chiefwhite; Chininazu12; ChipX86; ChopMonkey; Chowbok; Chris G; Chris Howard; Chris Pickett; Chris Q; Chris Roy; Chris the speller; Chrislk02; ChrisLoosley; Chrispfister; ChrisRuvolo; ChrisSteinbach; ChristianEdwardGruber; Christoofar; Chronodm; ChuckEsterbrook; Chwu; Chymb; Ciaran H; CiaranG; Citation bot; Citation bot 1; CJLL Wright; CLAES; ClamDip; Clappingsimon; Clausen; Clayoquot; Closeapple; Closedmouth; Cma; CmdrObot; CodeCaster; Coder Dan; Coffee and TV; Coffeehood; Colin Marquardt; Collect; Collinjc; Colonel Warden; Colonies Chris; CometGuru; Cometstyles; Commander Keane bot; ComputerGeezer; Connelly; Conortodd; Conskeptical; Contact2gohar; Conti; Contributor124; Conversion script; Convex hull; CortezK; Corvi; Cotttho; CouchTurnip; CounterVandalismBot; Coveragemeter; Craig Schneiderwent; Craig Stuntz; Craig t moore; Craigwb; Crawdad1960; Creacon; Creando; CRGreathouse; CrinklyCrunk; Crowdes; Crowfeather; Cryptic; Csabo; Csanjay; Ctkeene; Ctrager; Cubslvr; Curtlee2002; CUTKD; Cvanhasselt; Cverlinden; CWenger; Cwolfsheep; CWY2190; Cyberdiablo; CYD; Cyde; Cydebot; CynicalMe; Czambra; D-Rock; D'oh!; D'ohBot; D1ma5ad; D6; Da monster under your bed;

DagErlingSmørgrav; Daio; Dally Horton; Daltenty; Damian Yerrick; Damien Cassou; Damiens.rf; Damir Zakiev; Danarmak; DancingHacker; Danh; Danielanderson; Danielhegglin; DanielVale; Danlev; DanMS; Danny beaudoin; Darkfight; Darklama; Darklilac; DarkseidX; Darrel francis; DARTH SIDIOUS 2; DASHBotAV; DatabACE; Dav4is; Dave31870; Davetron5000; David-Sarah Hopwood; David.alex.lamb; David.Monniaux; DavidBiesack; DavidCary; Davidfstr; DavidLevinson; Davidlow; Dawnseeker2000; Dawynn; Daydreamer302000; Dbelhumeur02; Dbmsview9; DC; Dcfleck; Dcoetzee; Dcouzin; Ddenise; Debresser; Deccico; Deckiller; Dedalus; Deemery; Deepankgupta; Dekart; Dekimasu; Dekisugi; Delenburg; Delirium; Demiane; Denderick; Denis.dallaire; Dennis714; DennisDaniels; Deon Steyn; Depython; Derbeth; Descender; Deuxpi; Deville; Devourer09; Dguglielmi; Dhapp; Dharmabum420; Dharris; Dhdblues; DHGarrette; DHN-bot; Dhollm; Di Stroppo; Diametriks Consulting; Diberri; Diddyzelda; Didimos; Diego Moya; Digantorama; Digsav; Dillard421; Dinamik-bot; Diomidis Spinellis; Discospinster; Dishayloo; DJ Clayworth; Dj stone; Djgandy; DLoom; Długosz; DMacks; Dmharvey; DmitTrix; Dmulter; Dmyersturnbull; Dnas; Docdrum; Docu; Dodji.seketeli; Dodoïste; Dogslovesalsa; DOI bot; Dominus; Donarreiskoffer; Donsez; DonWells; DorganBot; Dotxp; Doug Bell; Doug.hoffman; Douga; Dougluce; DougsTech; Download; Downsize43; Dr ecksk; Dr.Pigosky; Dragon 280; DragonBot; DrDorkus; Dreadstar; Dreftymac; DrFO.Tn.Bot; Drhipp; DrilBot; Dritzer; Drm mills; Drmies; Dsavalia; DSP-user; Dtmilano; DugDownDeep; DuLithgow; DumZiBoT; Duplicity; Dvansant; Dvavasour; Dwayne; Dwheeler; Dwi Secundus; Dyfrgi; Dylanfromthenorth; Dysprosia; E946; EagleFan; Eags; Earlypsychosis; Earthengine; Eastlaw; Ebde; Ebelular; Ebrambot; Echecero; Echoray; Econrad; Ed Brey; Edaelon; EdC; Edcolins; Eddiehu; Edgewalker81; EdHubertson; EdiTor; Edudobay; Eelvex; Eestolano; Eewild; Efitu; Egbsystem; Egil; Ehajiyev; Ehheh; Ejrrjs; El bot de la dieta; El C; El T; Electrobins; Elena1234; Elendal; Elifarley; Elilo; ElinneaG; Eliyak; Elizaveta Revyakina; Elkman; Ellengott; Ellissound; Elonka; Eloquence; Elpecek; Elvis untot; Elwikipedista; EmausBot; Emperorbma; Empiric; Emurphy42; Enchanter; Ency; EngineerScotty; Engology; Enigmasoldier; Eññe; Enochlau; EoGuy; Epbr123; EpicSystems; Epim; EPM Enthusiast; Equilibrioception; Erechtheus; Eric B. and Rakim; Erik9; Erik9bot; Ermey; Erncrts; Esap; Escarbot; Esfdsfvfbfd; Eskimbot; ESkog; Estirabot; Etoastw; Etrigan; Ettrig; Eubulides; Euchiasmus; Euphoria; Eurleif; Euzuncaova; EverLearning; Everyking; Evil saltine; Evildictaitor; Evo rob; Ewlyahoocom; Excirial; Extransit; Fabrictramp; Fabrikant; Fæ; Fafner; FairuseBot; Fake0Name; Falcon8765; Faltenin; Fan-1967; Fanghong; Fasten; Fastily; FatalError; Fatespeaks; Faught; Faulknerck2; Faulty; Fauxpoefoes; Fbahr; Fbax; Fbeppler; Fderepas; Fdp; Feezo; Feigling; Feinoha; Felagund; Felyduw; Fenevad; Ferengi; Fernandopabon; FF2010; Ffangs; FightBoard; FilippoGioachin; Filnik; Finlay McWalter; Firebrandck; Firetrap9254; Firien; Firsfron; Fisherjs; Fishpi; Fishtron; FlaBot; Flamurai; Flcelloguy; Fleminra; Fleshgrinder; FlinkBaum; Flowanda; FlowRate; Fluffykryptonite; FlyHigh; Foobar; Footwarrior; Fpradier; Fragglet; Frankie1969; Franklin90210; FrankTobia; Frap; Frau K; Freakofnurture; Fred Bradstadt; FredCassidy; Freddy.mallet; Free Software Knight; Freeformer; Freejason; FreplySpang; FrescoBot; Fresheneesz; Friday; Friendlydata; FritzSolms; Fsmoura; Ft1; Fubar Obfusco; Full-date unlinking bot; Func; FunnyMan3595; Future Perfect at Sunrise; Futureobservatory; Futurix; Fuzheado; Fx21av; G0rn; G7shihao; Gadfium; Gahrons; GainLine; Gaius Cornelius; Gakrivas; Galorath; Galoubet; Gameboyguy13; Ganesan Janarthanam; Ganjuror; Gargaj; Garrett Albright; Gary King; Gaudol; GB fan; Gbleem; Gbolton; Gdavidp; Gecko06; Geehbee; Gef05; GembaKaizen; GenezypKapen; Geni; Gennaro Prota; Geofflane; Geometry.steve; George Schmidt; GerardM; Gerdemb; Geschichte; GESICC; Getsw; Ggeldenhuys; Ghettoblaster; Gholson; Gibber blot; Giftlite; Gigi fire; Gigs; Gingerbits; Ginkgo100; GiorgioMoroder; Gioto; Gishu Pillai; Glapu; Glenn4pr; Glennsc; Glloq; Gmarinp; Gmcrews; Gmt767; Gnowor; Gnusbiz; Gnyus; Goa103; GoBuck76; Goffrie; Gogo Dodo; Gökhan; Goobergunch; Goodrone; Gorgan almighty; GorillaWarfare; Gorona; Gorpik; Goswamivijay; Gracefool; GraemeL; Grafen; GRAHAMUK; Grammarbot; Grandmasterkush; Granite07; Graue; Grawity; GreatWhiteNortherner;

Greenknight04; GreenReaper; Greensburger; Greg Tyler; Gregbard; Greghc; GregJackP; GregorB; Greyskinnedboy; Gritchka; Gronky; GrouchoBot; Ground Zero; Grstain; Gruber76; Grue; Gsaup; GTBacchus; Guanabot; Guille.hoardings; Gumpu; Guoguo12; Guppie; Gutworth; Guy Harris; Guy M; Guybrush1979; Guyjohnston; Gwernol; GwynnBD; GyroMagician; H; H3llBot; Hadal; HaeB; Haeleth; Hagai Cibulski; Hairy Dude; Ham Pastrami; HamburgerRadio; Hanacy; Hankwang; Hans Adler; Hao2lian; Happysailor; Harburg; Hari Surendran; Harold f; HarrivBOT; Harrym; Hasenstr; Hashar; Hawaiian717; Hayne; HCJK; Hede2000; Heirpixel; HelloAnnyong; Henk Langeveld; Hennessey, Patrick; Henon; Heron; Hertzsprung; Hetar; Hexie; Hfastedge; Hga; Hgfernan; Hilgerdenaar; Hirzel; Hmains; Hob Gadling; HobbesLeviathan; HolgerK; Holizz; Homerjay; Hongguo; Hongooi; HonoluluMan; Hooperbloob; Hooverbag; Hornsofthebull; Hqb; Hritcu; Hsingh77; HumboldtKritiker; Hut 8.5; Huygensvector; Hyad; Hydrargyrum; Hypersonic12; Hzhbcl; Ianb1469; IanOsgood; Ibbn; Icairns; Ice Ardor; Ickettpe; Idioma-bot; Ienumerable; Ignacio Icke; Ikchennai; Ike-bana; Ikiwpedia; Ilikeverin; Ilja Preuß; Illusionx; Ilya; ImageRemovalBot; Imeshev; ImperatorExercitus; Improv; Imroy; IMSoP; InaTonchevaToncheva; Inc ru; Incnis Mrsi; Influent1; Informup; Infovoria; Inquam; Int19h; Intangir; Interversal; Intgr; InTheCastle; Intray; Introvert; Inwind; IO Device; Ipsign; Iri.nobody; Iridescent; Irishguy; Isaacdealey; Ishu76; Isilanes; Iskander s; Israelof; ItsProgrammable; Ivan Pozdeev; Ivanko 98; IvanLanin; Ivpen; Iwat; Ixfd64; J.delanoy; J04n; Jaapvstr; Jabraham mw; Jacek Kendysz; Jack Merridew; Jackie; Jacktruman; Jacob Robertson; Jacob1207; Jacobolus; JacobProffitt; Jaeger48917; James pic; JamesAM; JamesBWatson; Jamesnoe; Jamesontai; Jan Hoeve; Jan Winnicki; Jan1nad; Janbenes; Janiejones56; Janna Isabot; Jasonfrye; Jasonm23; JASpencer; JavaTenor; Jaxl; Jay; Jayaram10g; Jbellwiki; Jbernal37; Jblum; Jbolden1517; JBsupreme; Jcarroll; Jchyip; JCLately; JCRansom; Jcronen1; Jdm64; Jdpipe; Jebus989; Jedimike; Jeepday; Jeff.foster; Jeff.fry; Jeff3000; Jeffhudsonbush; Jehnavi; Jehoshua22; Jeltz; Jengelh; Jensgb; Jeremyharmon; JeremyMcCracken; Jerroleth; Jerryobject; JerryTed; Jéské Couriano; Jesselong; Jezmck; Jfernandez-ramil; Jfmole; JForget; Jglynn43; Jgrahamc; Jgrahn; Jhaniotis; JHFTC; JHP; JHunterJ; Jim.parshall; Jin.etics; Jisunjang; Jittat; Jjamison; Jjdawson7; JJMax; Jjsladek; Jkeen; Jkl; JL-Bot; Jleedev; Jlin; Jm34harvey; Jmath666; Jmlk17; Jncraton; Jni; JnRouvignac; JNW; Jo9100; Joanjoc; Jodurazo; Joeblakesley; JoeBot; Joeggi; JoelSherrill; Joerg.Rech; Jogloran; Johan Natt och Dag; Johannes Simon; Johayek; John; John Vandenberg; John5K35; JohnCD; Johnmc; JohnMcCrory; JohnMcDonnell; JohnOwens; Johnshue; Jon-ecm; Jon513; Jóna Þórunn; Jonadab; Jonas AGX; Jonasmike; JonathonReinhart; Jonb ee; Jondel; Jonel; Jonelo; Jonhanson; JonHarder; Jonik; Jonkpa; Jonon; JordanSamuels; Jorend; Jorgon; JorisvS; Jorunn; José María Sola; Josepant; Josh Parris; JoshDuffMan; JotaEme73; Joy; Joyous!; Jp361; Jpalm 98; JPats; Jpo; Jpvinall; JRose22033; Jrsopka; Jrvz; Jschwa1; Jsub; Jtheires; Jtigger; Juanco; Judgeking; Jujutacular; Julesd; JulesH; JulianMummery; Julias.shaw; Jumbuck; Jumper32; Jusdafax; JustinH; Jutiphan; Jvale; Jverlaan; Jvhertum; Jwalling; Jwarnier; Jwbecher; Jwfearn; Jwoodger; Jyankus; JzG; K.Nevelsteen; K7.india; Kaihsu; KaiserbBot; Kaizendenki; Kal-El-Bot; Kameronmf; KamikazeBot; Kamots; Kampuger; Kaniabi; Kappa; KaragouniS; Karam.Anthony.K; Karamba10; Karl Dickman; Karnesky; Kaster; Kat; Kate; Kazvorpal; Kbdank71; Kbdankbot; Kbh3rd; Kchampcal; Keilana; Kekedada; Kellen`; KellyCoinGuy; Kelvinclayne; KenBoyer; KenFehling; Kenjioba; Kent Beck; Kerrsys; KerryBuckley; Ketansevekari; Ketiltrout; Kevin Rector; Kevin.lee; Kevinalewis; Kevinmtrowbridge; Kewlchops; Kewlito; KeyStroke; KGasso; Kha0sK1d; Khargett dk; Khatchad; KHLehmann; KidOblivious; Kikos; Kim Dent-Brown; KimBecker; Kimchi.sg; Kimleonard; King of Hearts; Kirian; Kispa; KitchM; Kizor; KJRehberg; Kkailas; KKong; Klenod; KMWCHI; Knippı; KnowledgeOfSelf; Knutux; Kocio; Kokoala; Kompas; Konman72; Konstable; Korpo; Korruski; Kostmo; Krallja; Krischik; Krishami; Kristjan Wager; Kristof vt; Krubin; Krzyk2; Krzysfr; Ks0stm; Kskyj; Kslotte; Ktpenrose; Kumud hi; Kuppuz; Kurokaze204; Kurt Jansson; Kurykh; Kuteni; Kuzaar; Kvdveer; Kwhittingham; Kyellan; Kyle the bot; Kylemew; Kyokpae; Kyz; L Kensington; L.W.C. Nirosh; La goutte de pluie; LaaknorBot; Lalamax; Lance.black; Lanjack11; Larry2342;

Lars112; Lars12; LarsHolmberg; Larsw; Laserpointer1; Laterality; Latiligence; LaurensvanLieshout; Lauri.pirttiaho; Lcarscad; Lcgrowth; LDRA; Leafcat; Leandrod; LeaveSleaves; Lectonar; LedgendGamer; Lee Carre; Lee Daniel Crocker; LeeG; LeeHunter; Legobot; Legolost; Leibniz; Lemmie; Lenin1991; Lenoxus; Leonard G.; LeonardoRob0t; Lerdsuwa; LesmanaZimmer; Leuko; Levin; Lewis; Lews Therin; Lexspoon; LFaraone; Liahocho; Libragopi; Libsoc; Lichen0426; LiDaobing; Lightbot; Lightmouse; Ligulembot; LilHelpa; LinDrug; LinguistAtLarge; Linka Pralitz; LinkFA-Bot; Linkspamremover; Linuxbeak; Linuxboygenius; LirazSiri; Little Mountain 5; Littlealien182; LittleDan; Liuti; Livewireo; LizardJr8; Lliberto; Lmerwin; Lo2u; Localh77; Locke Cole; Locobot; Logan; Logicalgregory; Logixoul; LOL; Lomacar; Longhorn72; Looxix; Louislong; LouScheffer; Lowellian; Lpod100; Lt-wiki-bot; Lucian.voinea; Luis Dantas; Luiscolorado; Luk; Lumberjake; Luna Santin; Lupo; Luzian; LVC; Lwoodyiii; Lycurgus; Lzur; M ajith; M-le-mot-dit; M0llusk; Ma1colm; Mac; Maccoy; Macronyx; MadDreamChant; Madduck; Madir; Madjidi; Magioladitis; Magnus Manske; MainFrame; Majorclanger; Mal4mac; Mal7798; Malcohol; Malcolma; Man It's So Loud In Here; Mandarax; MandyOwens; Manop; Manta7; MAntkiewicz; Manzee; Marasmusine; Marcinjeske; Margosbot; Maria C Mosak; Marianocecowski; Marijn; Mark Foskey; MarkDilley; Markhurd; MarkMLl; Markusaachen; MarkyGoldstein; Martarius; Martial75; Martijn Hoekstra; Martin Blazek; Martin Majlis; MartinBot; Martinig; Martyb33333; Marx Gomes; Mashkells; Masonb986; Massysett; Master of Puppets; MastiBot; Maszanchi; Matchups; Materialscientist; Math1337; Mathew Roberson; MathijsM; Mathmo; Mati22081979; Matithyahu; Mato; Matsumuraseito; Matt Crypto; Matt Schwartz; MattGiuca; MattiasAndersson; MattOConnor; Matusz; MauritsBot; Maurreen; Maxamegalon2000; MaxHund; Maximaximax; Maximkr; MaxSem; Mbell; Mberteig; MBisanz; Mboverload; MBPetersen; Mbvlist; Mcamposrocha; MCB; McGeddon; Mckoss; McM.bot; Mcorazao; Mcpatnaik; Mcsee; Medinoc; Meirdart; Membury; MementoVivere; Memset; Mentifisto; Mephistophelian; MER-C; Merbabu; Merenta; MerlLinkBot; MessedRobot; Methossant; MeUser42; Meznaric; Mfdavies; Mhaitham.shammaa; Mhartl; Mheusser; Mhodder; Mhsrinivasan; Michael Daly; Michael Drüing; MichaelBillington; Michaelbusch; Michedav; Michiel Helvensteijn; Michigangold; Microtony; Middayexpress; Mikachu42; Mikatron; Mike Field; Mike.nicholaides; Mike1234; Mikeblas; MikeDogma; MikeLynch; Mikeo; Miker@sundialservices.com; Mild Bill Hiccup; Millerlyte87; Minesweeper; Minghong; Minimac; Mintleaf; Mipadi; Miracleworker5263; Mircea.Vutcovici; Mirror Vax; Misteraznkid; Mistercupcake; MIT Trekkie; Mitch Ames; MithrandirMage; Miyako; Mj1000; Mjchonoles; Mkarlesky; Mkksingha; Mkmcconn; ML02; MLetterle; MLRoach; Mmcdougall; Mmenal; Mmernex; Mmpubs; Mnorbury; Modbear; Mogigoma; Mondalaci; MONGO; Monkey Bounce; Montana; Moondyne; Morgajel; Morlach01; Morphex; Morrillonline; Mortense; Mosheler; Mosquitopsu; Mpandrews; Mpeisenbr; Mpeylo; Mpntod; Mr. Disguise; Mr.Muffet; Mr.Z-man; Mr2001; Mratzloff; Mreftel; Mrhericus; MrJones; Mrlongleg; Mrs; MrTree; Mrwojo; Msabramo; Mschlindwein; Mshonle; Mskeel; Msteffen@interneer.com; Mstucke1; Msulis; Mtomczak; Muijz; Mulad; Muro Bot; Muro de Aguas; Mushroom; MuthuKutty; Mutilin; Mwanner; Myasuda; Mydogategodshat; Mysdaao; MystBot; MywikiaccountSA; N8mills; NAHID; Nainil; Nakednous; Nakon; Nallimbot; Nancy; NantucketNoon; NapoliRoma; Nastajus; Nat hillary; Nate Silva; NateEag; Naterice; Natkeeran; NattyBumppo; Nav102; Navalg; NawlinWiki; Nbarth; Nbryant; Nczempin; Neelix; Neerajsangal; Neetij; NeoChaosX; Neognomic; NerdyScienceDude; Nesmojtar; Netkinetic; Netsnipe; Neurogeek; NeutralPoint; Neverquick; Nevware; New England; NewSkool; Nibblus; Nicholas Drayer; Nicholas Lativy; Nick; Nick UA; Nickmalik; Nicoguaro; Nicolapedia; NigelR; Nightreaver; Nigosh; Nikdo; Nimowy; Ninja247; Ninly; Niteowlneils; Nitromaster101; Nitzanms; Nixdorf; Nixeagle; NjardarBot; Nlfiedler; Nneonneo; No1lakersfan; Noctibus; Nohat; NonDucor; Nonky; Nono64; Nopetro; Norm mit; Northox; Northsimi; Nosbig; Notinasnaid; Notnoisy; Nposs; Nsaa; NSK Nikolaos S. Karastathis; Ntalamai; NTBot; NuclearWarfare; Nuggetboy; Numbo3-bot; Nuno Tavares; Nwalya; Nzd; Nzeemin; O.sharov; O18; Oashi;

Obersachsebot; Obina; Occono; Oege; Ohnoitsjamie; Ohthelameness; Ojcit; OKBot; Okivekas; Olathe; Oleg Alexandrov; OlEnglish; Olexandr Kravchuk; Oli Filth; Oligomous; Olilo; Olinga; Oliver; Oliver55; Omegatron; Omicronpersei8; OMouse; OmriSegal; Onceler; Oneilius; Oneiros; Oni king; Open-collar; OpenToppedBus; Optikos; OrangUtanUK; Orborde; Orderud; OrgasGirl; Orie0505; Orimosenzon; Oriondown; Orphan Wiki; OrphanBot; Ortolan88; Osmodiar; OsoLeon; Ossias; Ottawa4ever; Outback the koala; Owain.davies; Owain.wilson; OwenBlacker; Oxinabox; P.taylor@dotcomsoftwaresolutions.co.uk; P.Y.Python; P3net; Pablasso; Paddy3118; Paddyslacker; Pafcu; Pak21; Pako; Paladinwannabe2; Paling Alchemist; Pam.morris; Panoramix; Pantosys; Panzi; Papercutbiology; Paperfork; Paquitotrek; Parasoft-pl; Parklandspanaway; Patrick; Patrickdepinguin; Paul A; Paul August; Paul Bassin; Paul W; Paul.klinger; Paulgiron; Paulocheque; Pavel Vozenilek; Pbb; Pcb21; Pdemb; Pdmitry; Pearle; Peashy; Pejman47; Pellicci; Pelock; Pengo; Penumbra2000; Perfecto; Peripitus; Personjerry; Personne1212; Peteforsyth; Peterdjones; Peterl; PeterNuernberg; Pexib; Pezra; Pgan002; Pgr94; PGSONIC; PGWG; Pharaoh of the Wizards; Phase Theory; Phatom87; Phelfe; Philip Trueman; PhilipO; PhilipR; PhilKnight; Philwiki; Phlip2005; Phoenix80; Pi is 3.14159; Piano non troppo; Picaroon; Pickerill; Piet Delport; Pietrodn; Pik0; Pindakaas; Pinguin.tk; Pinkadelica; Piotrus; PipepBot; PixelBot; PJY; Plasticup; Platonides; Plouin; Plugwash; Plustgarten; Pm expert; Pmarshal; Pmatu; Pmauriciocosta; Pmcollins; Pmerson; Pmtoolbox; Pne; Pnm; Poeloq; Pol098; Polonyman; Polyparadigm; Pomoxis; Poor Yorick; Postdlf; Prabin60; Prestonmag; Primetime; PrimroseGuy; Professional; Programming Research; Project2501a; Promoa1; Pronob; PS2pcGAMER; Pseudopanax; PSmacchia; Psmcguin; Psychonaut; Ptbotgourou; Pth81; PTSE; Purfection; Pvlasov; Pweemeeuw; PWhittle; Q Chris; Qaiassist; Quadell; Quadra23; Quantum7; QuantumEngineer; QuantumG; QueenCake; Quiddity; Quietust; Quinntaylor; Quintote; Quux; Quuxplusone; Qwertyus; R r245; R. S. Shaw; R'n'B; R3dux; R3m0t; Raanoo; RabbleRouser; Rabit gti; Radagast3; Radagast83; Radak; Radiobeam; RadoDobb; Raevel; Rafiko77; Raghunathan.george; Rahulchic; RainbowCrane; RainbowOfLight; Raise exception; Rajesh1981; Rajeshd; Rajnikant it; Ram.nivas; Ramsyam; Randomalious; RandyKolb; RanjithVenkatesh; Rannpháirtí anaithnid; Raoulduke47; Ratemonth; Rau J; Ravialluru; Ravinag; Ravinder.kadiyan; Ravindrag82; Ravindrat; RayGates; Raymondwinn; Rbalacha; Rbsjrx; Rbt0; Rdh0930; Rdleon; Rebroad; RedBot; Rediahs; Redrocket; RedWolf; RedWordSmith; Reedy; ReformatMe; Régis Décamps; Regregex; Rei-bot; Reinderien; Reintjan1234; Reisio; Remember the dot; Remi; Remy B; Renato Primavera; ReneS; RenniePet; Renox; Retinoblastoma; Retired username; RexNL; ReyBrujo; Rfortner; RHaworth; Rholton; RibotBOT; Richard Katz; Richard R White; Richard@lbrc.org; Richard2Me; Richardelainechambers; Richardgush; Richardkmiller; RichardVeryard; RichMorin; Richwales; RickBeton; RickClements; Rickyp; Ringlen; Rintrah; Rizome; RJBurkhart3; RJFJR; RJL Hartmans; RjwilmsiBot; Rmallins; Rmeier; Rmp; Rnickel; Robaczek; Roadbiker53; Roadrunner; Robbak; RobCheng; Robert Horning; Robert Merkel; Robina Fox; Robinshine; Roblu; Robofish; RobotE; RobotG; Roboto de Ajvol; Rocastelo; Rocketrye12; Rodasmith; Rodrigob; Rodrigobartels; Rogerb67; Rogerborg; Ron Richard; Roneny1; Ronjouch; Rookkey; Rootbeer; Rory096; Ross banter; Rossinglish; Rossj81; RossPatterson; Rotem.E; Roy.clarke; RoyOsherove; RoySmith; Rozek19; Rp; Rpm; Rrburke; Rrobason; Rror; Rs rams; RSaunders; Rschakraborty; Rscottfree; Rtc; RU.Siriuz; Rubinbot; RuggeroB; Rugops; Ruijoel; Rulesdoc; Runderwo; Rupl; Rursus; RussBot; RuudAlthuizen; Rwgreen1173; Ryandaum; RyanGerbil10; Ryans.ryu; Ryguasu; RzR; S0crates9; S3000; S7solutions; Sabine Kreidl; Sabri76; SAE1962; Saif always; Sakurambo; Salamurai; Salix alba; Sam Hocevar; Sam1064; Samansouri; Samaritan; Samiroy; Samrolken; Samw; Samwashburn3; San chako; Sanchom; Sandrarossi; Santryl; Sarah; Sarah.cartwright; Sardanaphalus; Sashakir; SashatoBot; SassoBot; Saturnine42; Sbosklop; ScastlePM; SchfiftyThree; Schmloof; Schol-R-LEA; Schultkl; SchuminWeb; Schwallex; Schzmo; Scientus; Scjnsn; Scmdn; Scope creep; Scottb1978; ScottDavis; Scottmackay; Seajay; Sean William; Seanblanton; SeanLegassick; Seb at oceg; Seba5618; Sebastian Schmied; Sebastian.Dietrich; SebastianHelm; Secdio; Sega381; Segv11; SEI Publications; Semicolons;

SensuiShinobu1234; Seren-dipper; Serge Stinckwich; Servel333; SethTisue; SexHex; Sfdan; Sgasson; Shambhaviroy; Shandris; Shane Lawrence; Shanes; Shangrula; Shannock9; Shant.d; SharpeSoft; SharShar; Shawn wiki; ShelfSkewed; Shenme; Shepard; Shiggity; Shimei; Shimky; Shnout; Shyam 48; SieBot; Sigma 7; Sigmundpetersen; Signalhead; Silicon plains; Silly rabbit; Silverhelm; SilvonenBot; Simeon; Simetrical; SimonKagstrom; SimonP; Simontrumpet; Sinisterstuf; SiobhanHansa; Sjakkalle; Sjc; Sketch051; SkipMcCormick; Skittleys; Skorpion87; SkpVwls; Skwa; SkyWalker; Slady; Slakr; Slashme; SlaterDeterminant; Slayman; Sleepyhead81; Sligocki; SlipperyHippo; Sliwers; Smack; Smalljim; Smartassembly; Smartbear; Smharr4; Smithveg; Smjg; Smountcastle; Snarius; Snigbrook; Snowolf; Snoyes; So God created Manchester; SocioPhobic; Sodium; Sofmlb; SoftComplete; Softtest123; SoftwareDeveloper; Softy; Solde; Some Color Mage; Someonesdad363616; Somercet; Somewherepurple; Soumyasch; South Bay; SoxBot III; Sozin; Sp; SpaceFlight89; Spatarel; SpBot; Specious; Speedplane; SpikeTorontoRCP; Splash; Spokeninsanskrit; Spoon!; Spragc; SpuriousQ; Square87; Squirepants101; SreejithInfo; Srice13; Srikant.sharma; Srinaveen; Srinicenthala; Srittau; Srleffler; Srogers74; SRyll; Ssaymssik; Ssd; Sspiro; St.General; ST47; Staceyeschneider; Stan Shebs; Staniuk; Stansult; STarry; Starrymessenger; Starwiz; Stassats; StaticCast; StaticGull; STBot; STBotD; Ste4k; StefanVanDerWalt; SteinbDJ; Steleki; Stemonitis; Stephan Leclercq; Stephan Leeds; Stephanakib; Stephen; Stephen e nelson; Stephen Gilbert; Stephenb; Stephenbooth uk; Stevage; Steve Grant; Steveluo; SteveMerrick; Steven Zhang; Stevertigo; Stevietheman; Stewartadcock; Stherrmann; Stijn Vermeeren; Stillnotelf; Stimpy; Stormie; Stratadrake; Sttaft; StuffOfInterest; Subgurim; Sujayg; Sullivan.t; Sunil1234b; SunSw0rd; SurfAndSwim; Surfinrhino; Suruena; Susurrus; SuurMyy; Svante1; Sverdrup; SvGeloven; Svick; Swasden; Swasical; Sweetmoose6; Sybersnake; Sylverspyder; SymlynX; Synthebot; Szquirrel; Szwejkc; Taarkshya; TaBOT-zerem; Tachyon01; Taejo; Taeshadow; Tagishsimon; Tagus; Taka; Takdavid; Talkaboutquality; Tamas Szabo; Tannin; Tarinth; Tarquin; Tarret; TastyPoutine; Taw; Tawker; Tawkerbot2; Tcncv; Tdelchiaro; Teapotgeorge; Techdoode; Technoparkcorp; Techtonik; Ted Longstaffe; TedBaker88; Tedernst; Tedwardo2; Teemu Maki; Tekavec; Tekkaman; Tekkenfreak3; Teleomatic; Teles; Tengai; TennysonXII; Teohaik; Teryx; Tesi1700; Test-tools; Testcocoon; TeunSpaans; Tevirselrahc; Texture; TFriesen; Tgruwell; ThaddeusB; Thanassis avg; That Guy, From That Show!; The Evil IP address; The Thing That Should Not Be; The Wild Falcon; Thebeginning; Thecerealpoet; TheFlow; Theinfo; Theleftorium; Themacboy; Themfromspace; Themillofkeytone; Themshow; THEN WHO WAS PHONE?; Thenickdude; Theo10011; Theonlysf; TheParanoidOne; THF; Thijs!bot; Thingg; Thomas Brandstetter; Thomas.uhl; Thowa; Thr3ddy; ThurnerRupert; Thushara tk; Tiagofassoni; Tide rolls; TigerShark; Tigrisek; Tijuana Brass; Tikiwont; Timbatron; Timichal; Timo Honkasalo; Timrollpickering; Timshah; Timwi; Tinucherian; Tinus74; Tivedshambo; Tjarrett; TJRC; Tlaresch; Tlogmer; Tlroche; Tmaufer; Tnxman307; ToastieIL; TobeBot; Tobias Hoevekamp; TobiasPersson; Tobych; Toddst1; Tom harrison; Tom-; Tomb; Tomjenkins52; Tommens; Tommy2010; Tomrbj; Tomtheeditor; Tony Morris; Tony Sidaway; Tonyshan; Tooto; Topaz; Topping; Torc2; TorLillqvist; Torneco; ToSter; TotoBaggins; Totty3478; Tqbf; Tracyragan10; Tranzid; TraumaPony; Travis32; Travis99; Tree Biting Conspiracy; Tregoweth; Tribaal; TrisDG; Troddel; Trout001; Trum123; Trusilver; Truthbro; Trylks; Tsca.bot; Tsilb; Tudor.girba; TUF-KAT; TutterMouse; TuukkaH; Tvarnoe; TWFred; Twsx; TXiKiBoT; Txomin; Tyler Oderkirk; U2perkunas; Ujamba; Ukexpat; Ukkuru; Ultimus; Umar420e; Uncle G; Underpants; Unforgettableid; Unfree; Uni-qdocs; Unittester123; Unixguy; Unomi; Untill; Uriyan; Urs.Waefler; User77764; Utcursch; Uucp; Uzume; V6Zi34; Valafar; Valbaca; Van helsing; Vary; Vaucouleur; Vaughan Pratt; Vbgamer45; Vbose; VegaDark; Veghead; Veralift; Verloren; VernoWhitney; Versageek; Versus22; VFrick; Victorwss; Vihangvk; Vijairaj; Vikerbandt; Viking67; Villeez; Vina; Vina-iwbot; Vincehk; ViniGodoy; Virek; Virgiltrasca; Viridae; Virtuald; Vishnava; Visor; Vivio Testarossa; Vkuncak; Vladimirkondratyev; VMS Mosaic; VoABot II; Vonkje; Vortexrealm; Vp; Vparaschiv; Vreddy498; Vrenator; VVVBot; W1k1th1nk; W3bbo; Wafulz; Wageslave; WalterGR;

Wangi; Wapcaplet; Waratah; Warren; Watcher; Watsonqai; Watts52; Waveclaw; Wavelength; WaysToEscape; Wdyoung; Webmaestro; Webreloaded; WeggeBot; Weidenrinde; Welsh; Werdna; Weregerbil; Wesley; WETaylor; Wettel; Wgdominic; Wgiezeman; Wgoetsch; Whilding87; Whiner01; Whiteknox; Whpq; Whytehorse1413; Whywhenwhohow; Wickity; Wik; WikHead; Wiki alf; Wiki contribs; Wikibot; Wikidrone; Wikijens; Wikilolo; WiKiMan3L; Wikimike2007; WikiSolved; WikitanvirBot; Wikitect; Wikke41; Willem-Paul; William Avery; William M. Connolley; Williamborg; Williampfeifer; Willking1979; WimdeValk; Wimt; Win32Coder; Wing gundam; Winhunter; Winterst; Wissons; Witchinghour; Witten rules; Wizard191; Wjhonson; Wknight8111; Wlievens; WLU; Wmahan; Wmwmurray; WolfgangFahl; Woohookitty; WookieInHeat; Worldtraveller; Woz2; Wrathchild; Wrelwser43; WRK; Wulvengar; WWHenderson; Wx8; Wyatt Riot; X.; X746e; Xagronaut; Xcasejet; Xedaf; Xezbeth; Xfeldman; Xiong Chiamiov; Xompanthy; Xqbot; XqRG; Xredor; Xyb; XZeroBot; Yadyn; Yaegor; Yamla; Yan Kuligin; Yangon; Yaninass2; Yaris678; Yath; Ymalik; Ynhockey; Yonkie; YordanGeorgiev; Yoshm; Yottzumm; Yrithinnd; YUL89YYZ; Yurez; Yuuki Mayuki; Yworo; Yym; ZacBowling; Zachlipton; Zain Ebrahim111; ZamorakO o; ZanderZ; ZappyGun; Zazpot; Zed toocool; Zedlik; Zer0431; ZéroBot; Zeroindent; Zerokitsune; ZeroOne; Zetawoof; Zink Dawg; Ziroby; Zoicon5; Zombiespokebadgers; Zondor; Zorgon7; Zorrobot; Zsinj; Zumbo; Zundark; Zwobot; Олександр Кравчук; ערן;

GNU Free Documentation License

Version 1.2, November 2002

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other

conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same

cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Alphabetical Index