

IBM Inter-University Programming Contest 2012 Training

Chapter : Software Analysis with IBM Rational Application Developer



IBM Inter-University **Programming Contest** **2012**

February 11, 2012 (Saturday)
Cliftons, Hong Kong



Agenda

- Introduction to Integrated Development Environment (IDE)
- Introduction to Static Analysis
- Real World Static Analysis Examples
- Lab

What is Integrated Development Environment (IDE) ?

- 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:
 - Source code editor
 - Compiler and/or an interpreter
 - Code visualisation tool
 - Build automation tools
 - Debugger or Profiler
 - Version Control tool or adaptor
 - Runtime test environment integration

Eclipse (Open Source) IDE

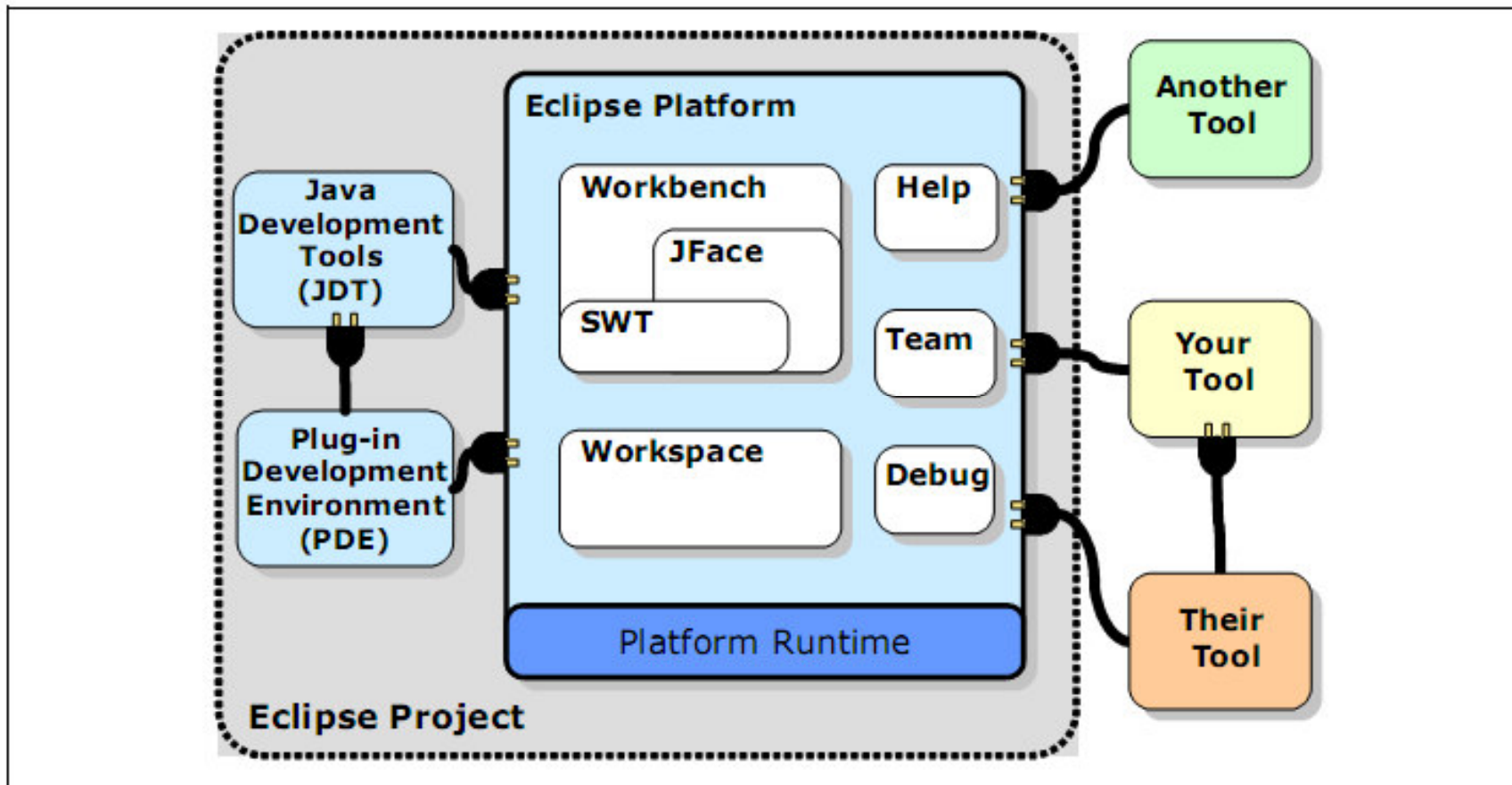
www.eclipse.org (free to download)



- Multi-language software development environment comprising an IDE and a plug-in system to extend it. It is written primarily in Java and can be used to develop applications in Java and, by means of the various plug-ins, in other languages as well, including C, C++, COBOL, Python, Perl, PHP, and others. The IDE is often called Eclipse ADT for Ada, Eclipse CDT for C, Eclipse JDT for Java and Eclipse PDT for PHP.

Release	Date	Platform version
Eclipse 3.0	28 June 2004	3.0
Eclipse 3.1	28 June 2005	3.1
Callisto	30 June 2006	3.2
Europa	29 June 2007	3.3
Ganymede	25 June 2008	3.4
Galileo	24 June 2009	3.5
Helios	23 June 2010	3.6

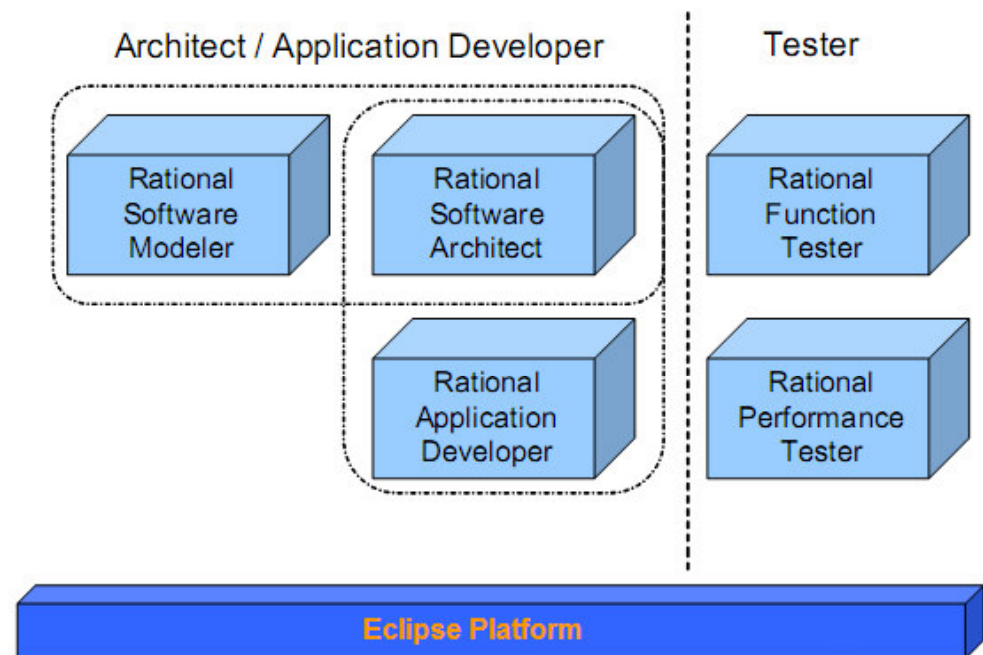
Eclipse Architecture



Commercial Tool: IBM Rational Application Developer







IBM Rational Application Developer for WebSphere Software (RAD) is an integrated development environment (IDE) that extends Eclipse, for:

- Visually designing,
- Constructing,
- Testing, and deploying Web application, web services, portals, and
- Java Enterprise Edition (JEE) applications.



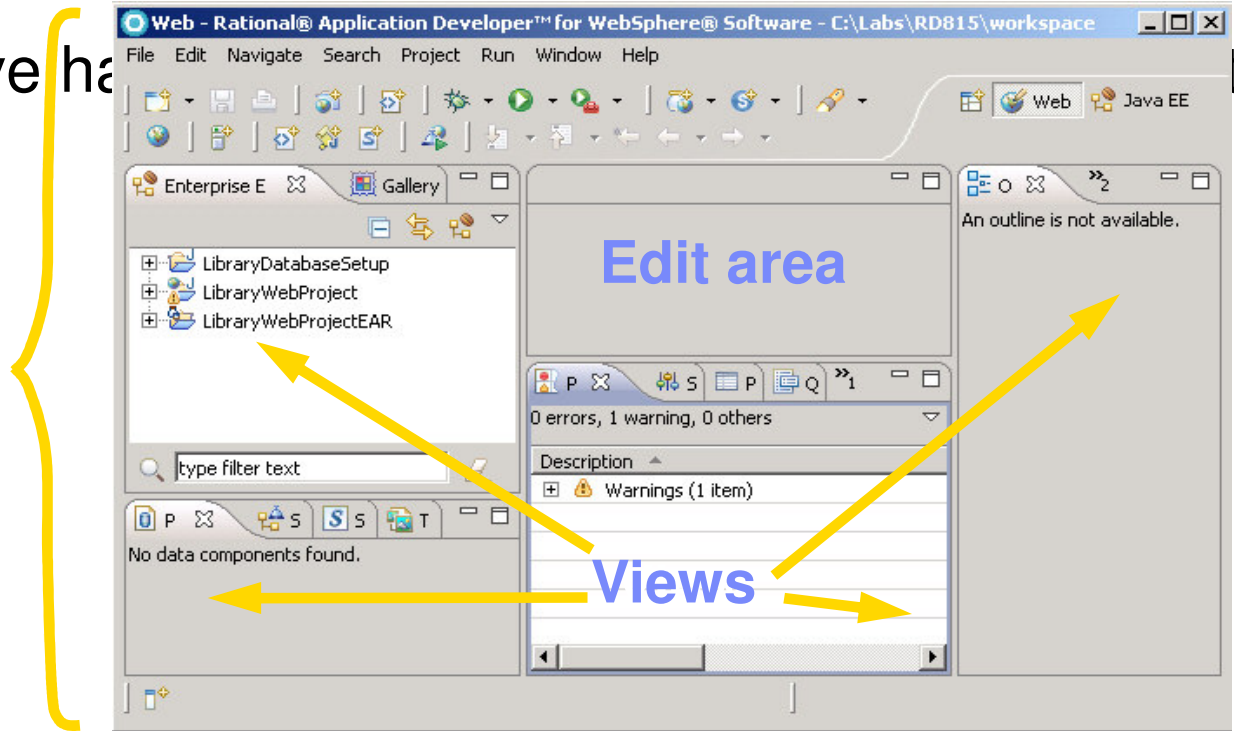
Workbench basic



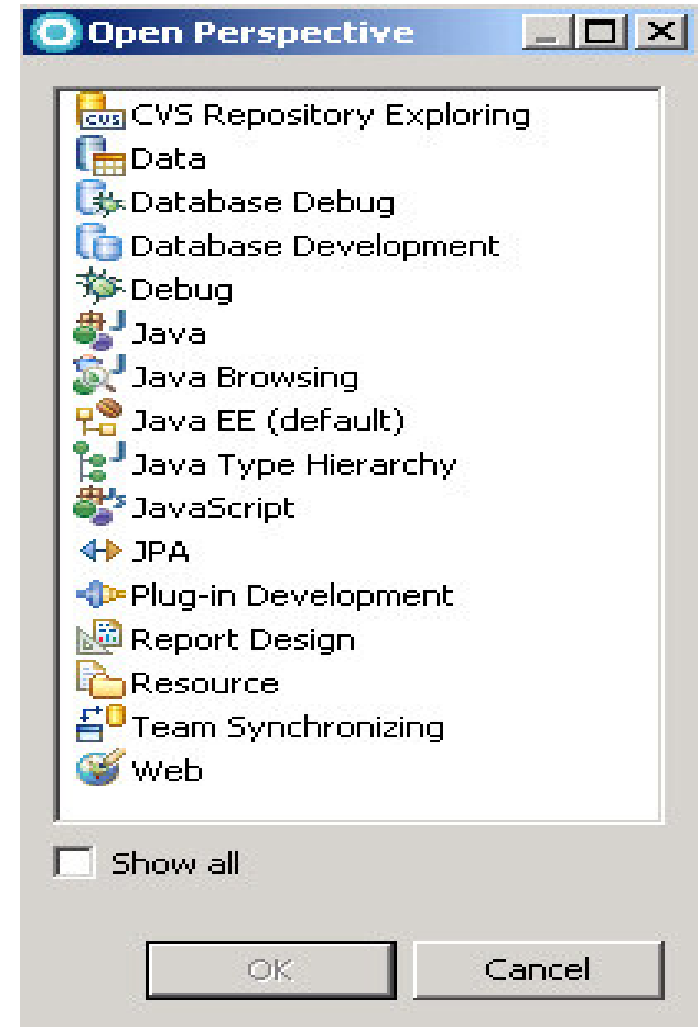
Icon Image	Name	Description
	Overview	An overview of the key functions in Application Developer.
	What's New	A description of the major new features and highlights of the product.
	Tutorials	Tutorial screens to learn how to use key features Application Developer. Provides a link to Tutorials Gallery.
	Samples	Sample code for the user to begin working with "live" examples with minimal assistance. Provides link to Samples Gallery
	First Steps	Step-by-step guidance to help first-time users to perform some key tasks.
	Web Resources	URL links to Web pages where you can find relevant and timely tips, articles, updates, and references to industry standards.

- A **perspective** is a collection of views, toolbar icons, and menus, grouped to accomplish a specific type of work.
- A **view** supports editors and presents information differently.
- **Editors** allow you to modify the code.
- Every perspective has an **edit area**.

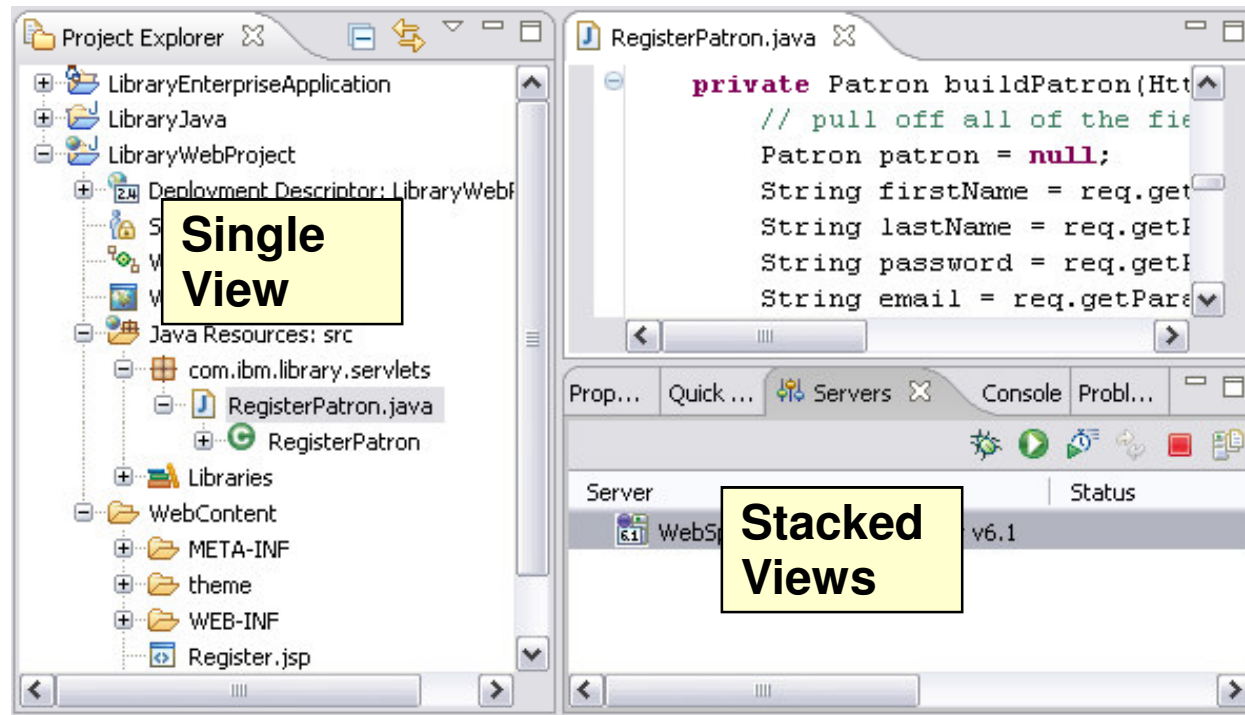
Perspective



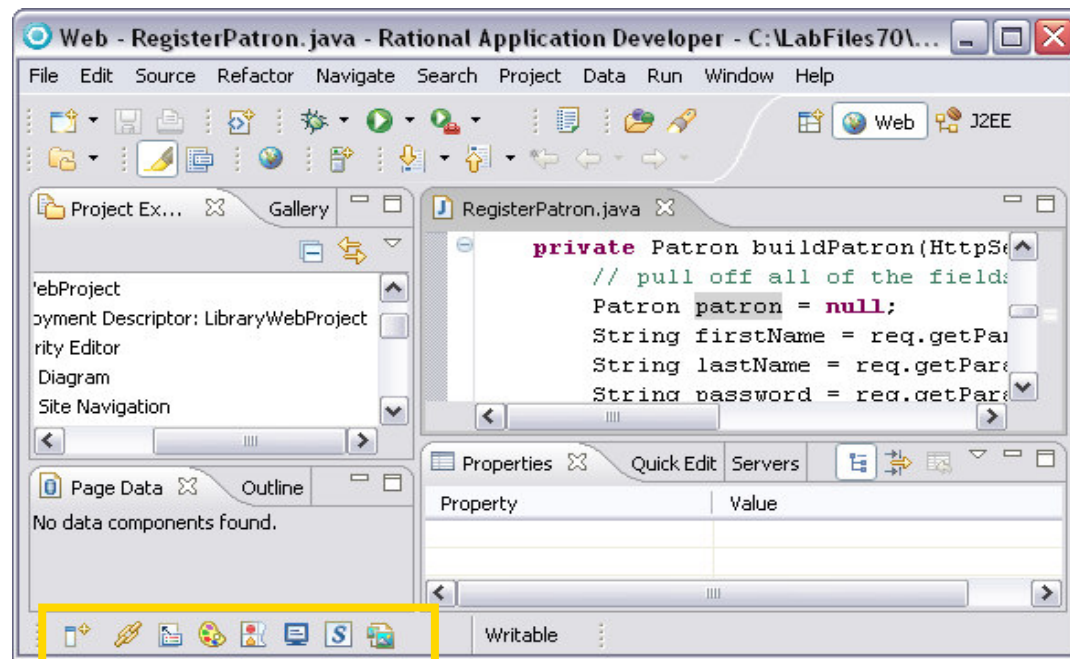
- A perspective defines the initial set and layout of views in the Workbench window.
- Each perspective provides functionality aimed at accomplishing a specific type of task, or works with specific types of resources.
- Perspectives control what appears in certain menus and toolbars.
 - They define visible action sets, which you can change to customize a perspective.



- There are two states for views
 - Single
 - There are no other views at that position in the perspective
 - Stacked
 - There are other available views, represented by tabs at the top of the view pane

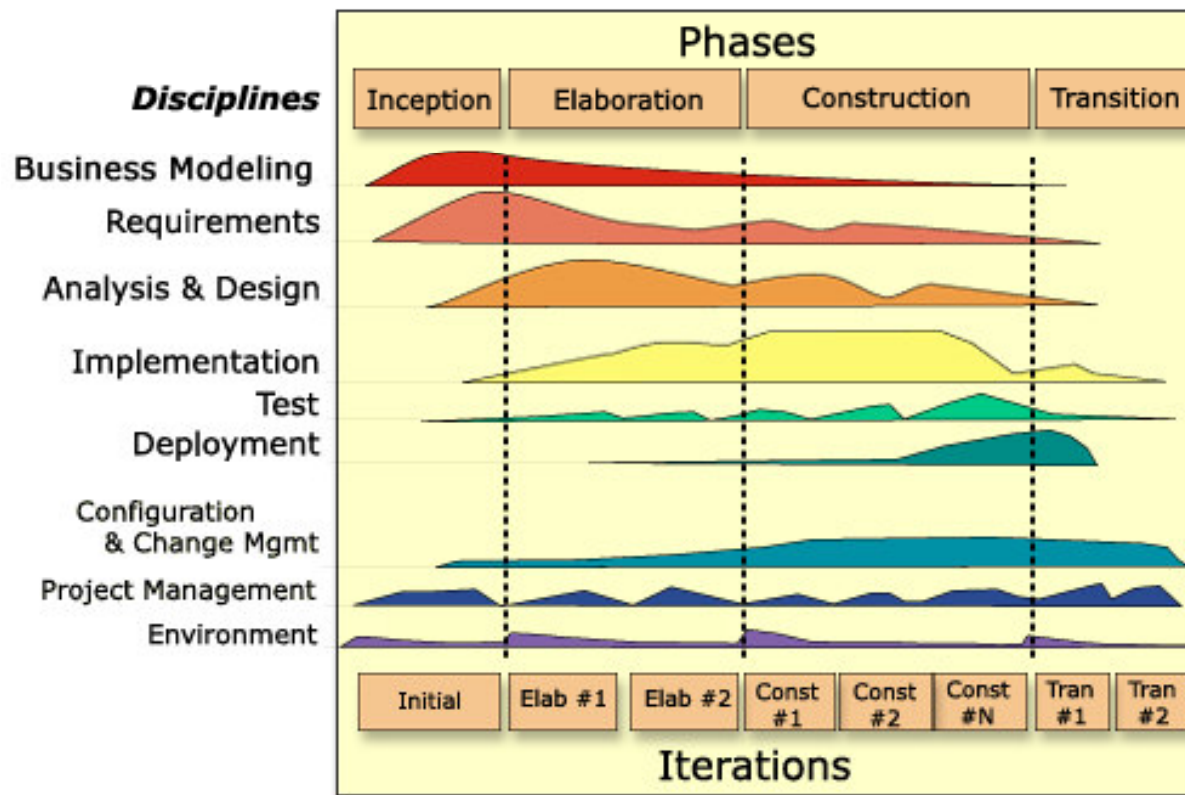


- You can quickly open and close fast views.
 - This functionality is available via the fast view bar
- There are two ways to create a fast view:
 - **Drag** a view onto the shortcut bar
 - **Right-click** the view icon and choose **Fast View**
- To restore a fast view, right-click the icon in the shortcut bar, and clear the **Fast View** option.



Rational Unified Process (RUP) Project Lifecycle

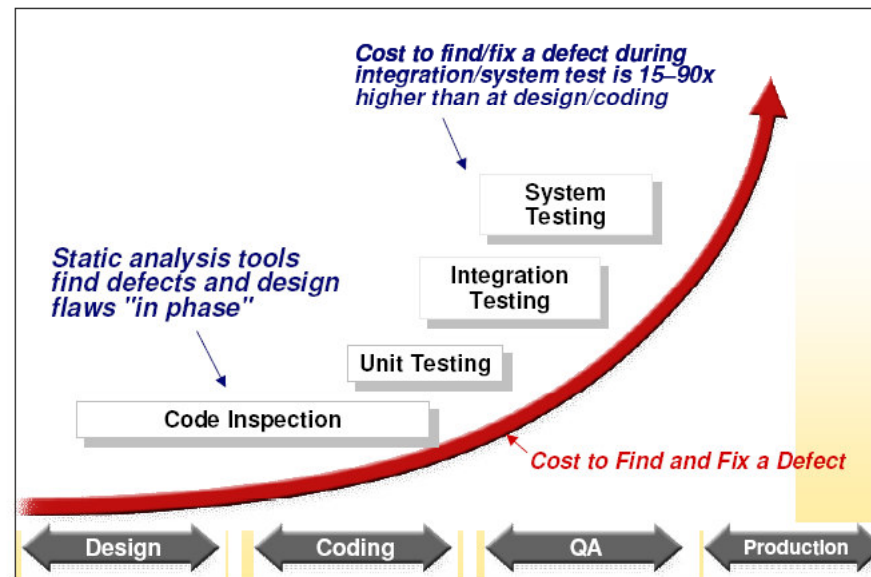
- A typical software development process consists of the following phases and disciplines
 - Each phase is broken into iterations



The high cost of fixing defects



- A single defect can cost between \$12-18K.
- Thousands of potential defects in a large piece of software can cost between \$600K-\$2.7M.



Source: IDC, 2005

A real-world example: Ford recall




- A two-year investigation determined that the problem was system-related.
- Pain includes:
 - costs of the recall itself
 - bad PR
 - customer satisfaction
 - potential safety lawsuits
- Potential cost to manufacturer of **\$54 million dollars.**

[Home](#) > [Money](#)

Ford Recalling 3.6 Million Vehicles

Ford Recalling 3.6 Million Vehicles to Deal With Concerns About



(AP Photo)

By **KEN THOMAS** Associated Press Writer
The Associated Press
WASHINGTON Aug 3, 2007 (AP)

Ford Motor Co. said Friday it is recalling 3.6 million passenger cars, trucks, sport utility vehicles and vans to address concerns about a cruise control switch that has led to previous recalls based on reports of fires.

Font Size
A A A
 E-mail
 Print
 Share

Ford said the recall covered more than a dozen vehicle models built from 1992-2004. The company said it was responding to concerns from owners about the safety of their cars and questions about the speed control deactivation switch in the vehicles that is powered at all times.

Mechanisms for identifying defects early in the lifecycle

- Agile Process
 - Test Driven Development



- Peer Code Review



Agile Process - Test Driven Development



- A technique involving short iterations, where:
 - New test cases covering new functionality are written first.
 - Next, the production code necessary to pass tests is implemented.
 - Finally, the software is refactored to accommodate changes.



- A systematic examination of source code by human experts.
 - Intended to find and fix mistakes overlooked in the initial development phase.
 - Improves both the overall quality of software and the developers' skills.



Adherence to coding guidelines



- **Coding guidelines** are important 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.
 - Adhering to coding guidelines improves the readability of the software, allowing engineers to understand new code more quickly and thoroughly.

```
package java.blah;

import java.blah.blahdy.BlahBlah;

/**
 * Class description goes here.
 *
 * @version    1.82 18 Mar 1999
 * @author     Firstname Lastname
 */
public class Blah extends SomeClass {
    /* A class implementation comment can go here. */

    /** classVar1 documentation comment */
    public static int classVar1;

    /**
     * classVar2 documentation comment that happens to be
     * more than one line long
     */
    private static Object classVar2;

    /** instanceVar1 documentation comment */
    public Object instanceVar1;

    /** instanceVar2 documentation comment */
    protected int instanceVar2;

    /** instanceVar3 documentation comment */
    private Object[] instanceVar3;

    /**
     * ...constructor Blah documentation comment...
     */
    public Blah() {
        // ...implementation goes here...
    }
}
```

Example of Java source code

The static analysis tool

- Processes such as agile development and peer code review are not enough.
- More and more, we need the assistance of automated analysis tools such as **IBM Rational Application Developer** that can ensure adherence to coding guidelines.
- The best time to find problems is to review the source code as it is written.

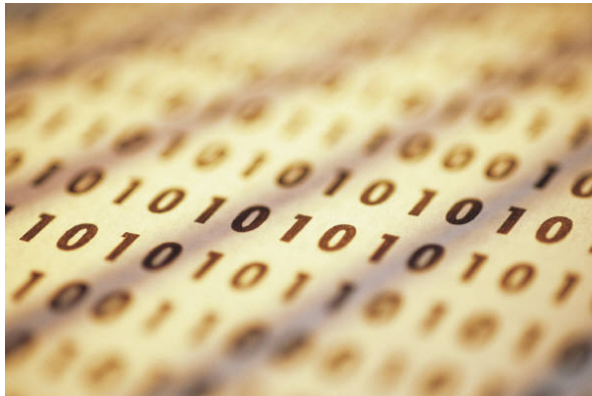
Note that static analysis is simply another tool to improve code quality, it is *not* a complete replacement for manual code reviews.

What is static analysis?



- **Static analysis** is the study of source and/or binary code that is not currently executing.

- Static analysis can:
 - Ensure that the source adheres to a predefined coding standard.
 - Detect common performance problems.
 - Understand the dependencies of the imports of each class.



Static analysis vs. runtime analysis

▪ Static analysis

- The study of source and/or binary code that is **not currently executing**.
- Examines architectural elements of software to find problems relating to dependencies in code (What is the impact of changing a class?)
- Examines the complexity of code (Are there too many paths through the code?)
- Simulates data movement through a system testing for data security problems (does the password get passed outside the system?)

▪ Runtime analysis

- Understanding software component behavior by using data collected **during the execution** of the component.
- Provides information about how the developed component - or the whole application - behaves when it runs.
- Provides explanations for various exposed or potential misbehaviors.

Categories of static analysis – Code Review



- Performs automated code parsing
 - Each source file is loaded and passed through a **parser** that looks for particular code patterns violating a set of established rules.
 - In some languages like C++, rules are built into the compiler or available in external programs like Lint.
 - In other languages like Java, the compiler does little in the way of automated code review.

- Code Review is a good tool to:
 - Enforce coding standards
 - Find basic performance problems
 - Find possible API abuse.



Categories of static analysis – Code Dependency



- Does not examine the format of individual source files.
- Examines the **relationships between** source files (typically classes) to build a map of the overall architecture of a program.
- Commonly used to discover known design patterns (good) or common anti-patterns (bad) in code.



Categories of static analysis – Code Complexity



- Analyzes the program code and compares it to established software metrics
 - Determines if it is unnecessarily complex.
- If a particular piece of code exceeds a given threshold, it can be flagged as a candidate for **refactoring** to help improve maintainability.



- Trend analysis does not use code artifacts directly.
- It is the study of improvements/degradations in code quality based on other forms of analysis.
 - Analyzing the results of analysis.
- Results generated by trend analysis typically appeal to managers and executives.
 - They make a statement about the direction of quality improvements, answering the question “Is the code getting better or worse?”

Real-world Example in Java #1

- **Rule: “Always surround if and loop statements with curly braces”**

```
if( condition )  
    methodCall();  
    anotherMethodCall();
```

- **What was the developer's intended behavior?**
- **Is this a real bug?**

Real-world example in Java #1 **Solution**

- **Both method calls should have been called when the “if” was true**

```
if( condition ) {  
    methodCall();  
    anotherMethodCall();  
}
```

- **This was a real bug!**

Real-world example in Java #2

- **Rule: “Avoid returning null instead of Iterator”**

```
public Iterator myMethod() {  
    List list = getList();  
    if( trueCondition ) {  
        return list.iterator();  
    }  
    return null;  
}
```

- **What' s wrong with this?**

Real-world example in Java #2

- **Inline usage of myMethod()**

```
for( Iterator it = myMethod(); it.hasNext(); ) {  
    // Do something  
}
```

- **Under the right conditions myMethod() can return null resulting in NPE**
- **This is particularly bad because “the right condition” may not happen until a customer executes the code.**

Real-world example in Java #2 Solution

- **Modify myMethod()**

```
public Iterator myMethod() {  
    List list = getList();  
    if( trueCondition ) {  
        return list.iterator();  
    }  
    return new ArrayList(0).iterator();  
}
```

Real-world example in Java #3

- **Rule: “Avoid multiple invocations of the same method”**

```
public static void satisfy( List fullList, IRuleFilter filter ) {
    for( Iterator it = fullList.iterator(); it.hasNext(); ) {
        ASTNode node = (ASTNode)it.next();

        boolean satisfied = filter.satisfies( node );
        if (!filter.isSuccessful()
            || ((filter.isInclusive() && !satisfied)
                || (!filter.isInclusive() && satisfied))) {
            it.remove();
        }
    }
}
```

- **Why does filter.isInclusive() get called twice?**

Real-world example in Java #3 Solution

- **Sometimes a seemingly harmless call is expensive**
- **Add a temporary variable**

```
public static void satisfy( List fullList, IRuleFilter filter ) {
    boolean inclusive = filter.isInclusive();
    for( Iterator it = fullList.iterator(); it.hasNext(); ) {
        ASTNode node = (ASTNode)it.next();

        boolean satisfied = filter.satisfies( node );
        if (!filter.isSuccessful()
            || ((inclusive && !satisfied)
                || (!inclusive && satisfied))) {
            it.remove();
        }
    }
}
```

Real-world example in Java #4

- **Rule: “Consider using HashSet instead of List”**

```
List employees = new ArrayList();  
  
...  
if (!employees.contains(emp)) {  
    employees.add(emp);  
}
```

- **What is wrong with this code?**

Real-world example in Java #4 Solution

- **Sets assure uniqueness without requiring programmers to check for duplicates**

```
Set employees = new HashSet();
```

```
...
```

```
employees.add(emp);
```

```
...
```

- **Better performance since it does not require searching through the entire collection**
- **If order is important use a `LinkedHashSet`**

- Create Rule for Analysis
- Configuration of Analysis
- Analysis the project

Enjoy your lab !



Questions
&
Answers