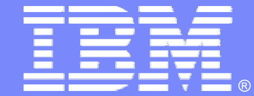


IBM Software Group

面向对象技术和可视化建模

IBM 软件部　李燕生
liyans@cn.ibm.com

e business on demand software

案例：问题描述

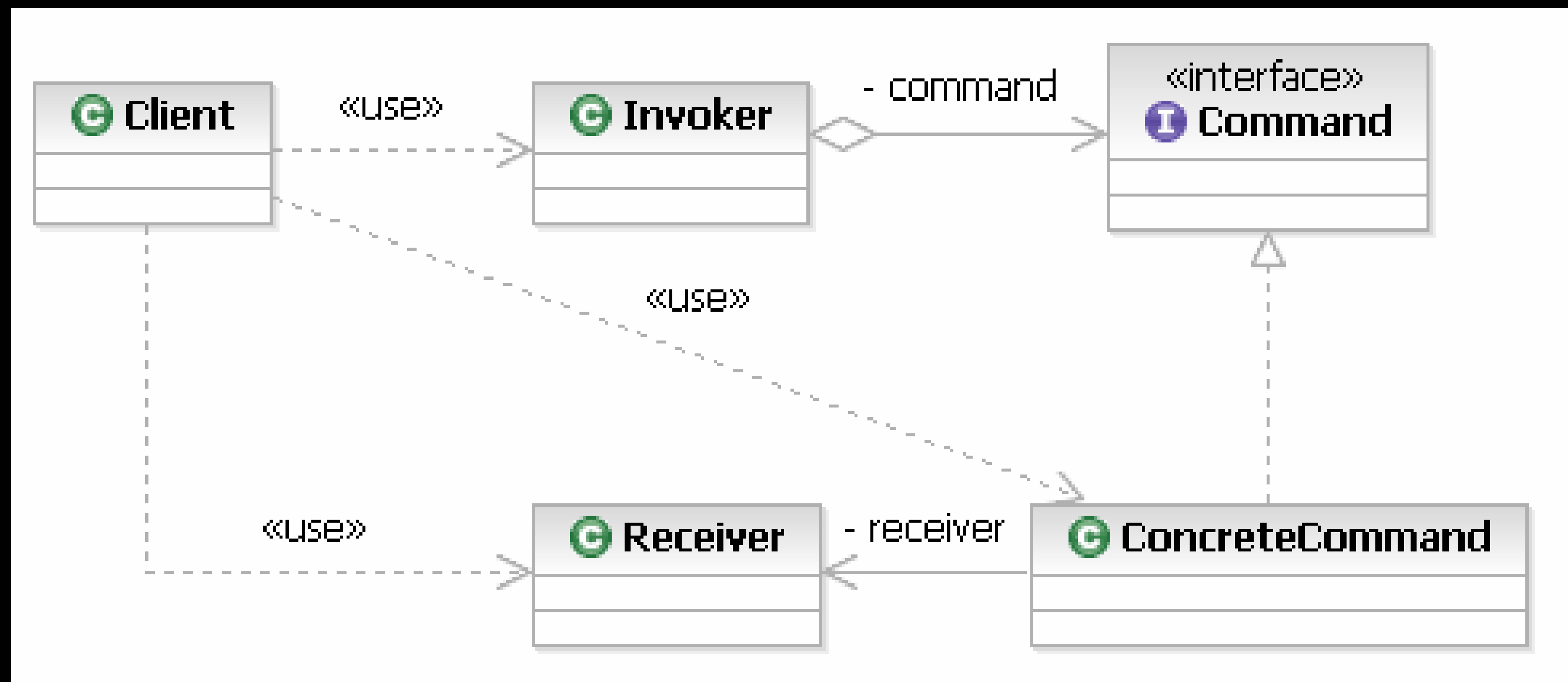
- 一个电信计费系统
- 需要在系统中定时调度一些任务
  - 检查数据缓冲区中是否有交换机传过来的计费数据
  - 启动计费处理过程
  - 其他，如超时(Timeout)检查、生成报告等
- 你会如果设计实现这些调度操作？

传统面向过程的方法

```
public static void main(String[] args) throws
  InterruptedException {
    …
        do {
                Thread.sleep(1000);                     // 等待
  一秒钟

                timer1 = timer1 + 1000;
                timer2 = timer2 + 1000;

                if (timer1 == TIMER_COUNT1) {           // 定时
  器1计时到时
                        timer1 = 0;
        // 定时器1复位
                        call_check_buffer();
        // 检查数据缓冲区
                }

                if (timer2 == TIMER_COUNT2) {
                        timer2 = 0;
                        call_billing_process();
  // 启动计费处理
                }
   …
        } while (true);                                 // 循环
}
```

Command 设计模式

- Client 创建一个 ConcreteCommand 并且指定它的 receiver
- Invoker 对象拥有指向 ConcreteCommand 对象的句柄
- Invoker 调用 Command 的 execute()
- ConcreteCommand 调用 receiver 的 action() 来执行相关的服务请求



Command 设计模式在本例中的应用

«Pattern Instance»

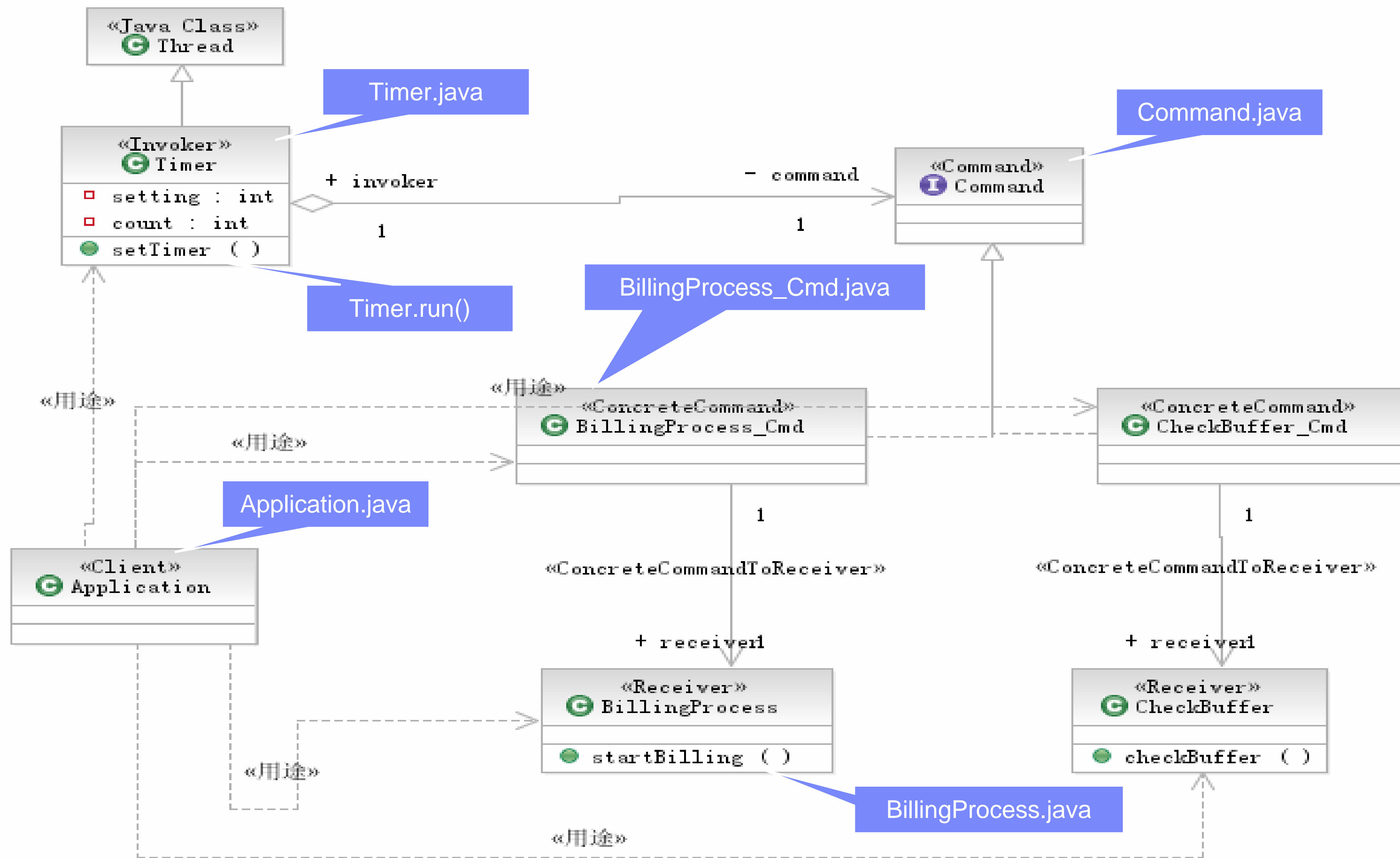
Command

模式参数

Command

Parameter	Value
Command [1] :	Command
ConcreteCommand [1..*] :	BillingProcess Cmd CheckBuffer Cmd
Client [1] :	Application
Invoker [1] :	Timer
Receiver [1..*] :	BillingProcess CheckBuffer
ConcreteCommandReceiver [1..*] :	Unnamed Association Unnamed Association
CommandManager [1] :	

用类图 (Class Diagram) 来表达系统静态结构



时序图 (Sequence Diagram) – 系统初始化

:Application | myBillingProcess:BillingProcess | myBillingProcess_Cmd:BillingProcess_Cmd | billingTimer:Timer

应用初始化

:Application

«create»
1: \调用的操作\

myBillingProcess:BillingProcess

«create»
2: \调用的操作\

myBillingProcess_Cmd:BillingProcess_Cmd

«create»
3: \调用的操作\

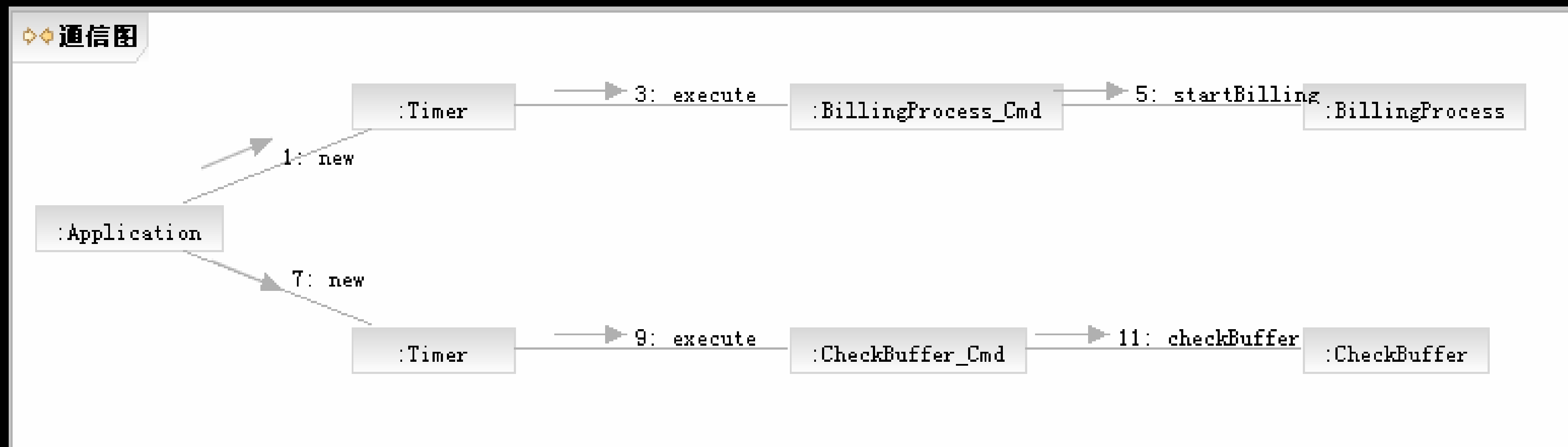
billingTimer:Timer

4: setTimer

«return»
5: setTimer

通信图 (Communication Diagram)

- 面向过程的方法：过程相互调用，实现系统功能
- 面向对象的方法：系统中所有的对象相互协作，完成了系统的功能



面向过程 vs. 面向对象

- 面向过程
  - 所有的功能都是通过主过程来调用
  - 每次增加一种新的定时服务都要修改主循环
  - 如果需要并发处理的话每个定时服务都需要自己创建并维护线程
- 面向对象
  - 增加新的定时服务只需要增加 ConcreteCommand 和新的 Receiver，不需要修改主循环
  - Timer 和 Command 是可重用的组件，并且具有最小耦合度
  - 线程由 Timer 对象创建并维护，新加的定时服务不需要考虑

面向过程的本质

- 以描述数据处理过程为核心
- 适用于数据驱动的应用开发
- 适用于相对单纯的系统或较少的系统交互的系统
- 适用于固定模式的数据处理

面向对象的本质

- 如何解决更为复杂的问题：抽象的种类和质量

例如：汇编语言是对机器语言的抽象
　　　高级语言是对汇编语言的抽象

- 在“方案空间”和”问题空间”之间作映射

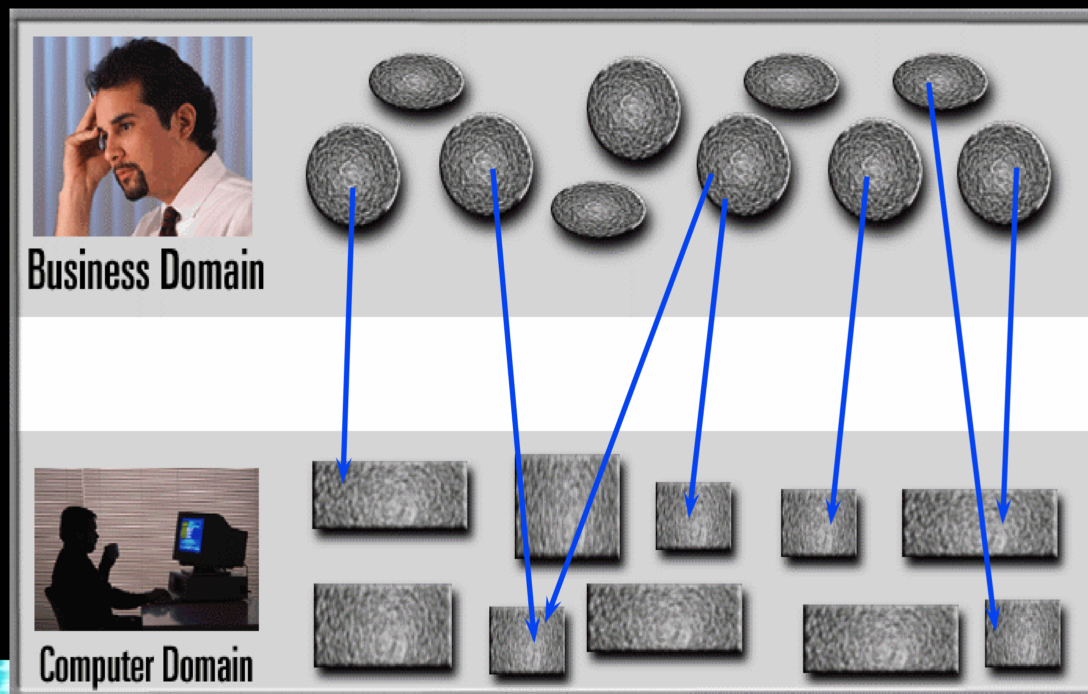
　OOP :根据问题描述问题,而不是根据方案描述问题

- 面向对象的系统的基本特征

　所有东西都是对象
　程序是一大堆对象的组合
　对象的存储空间
　对象的特性
　对象的消息

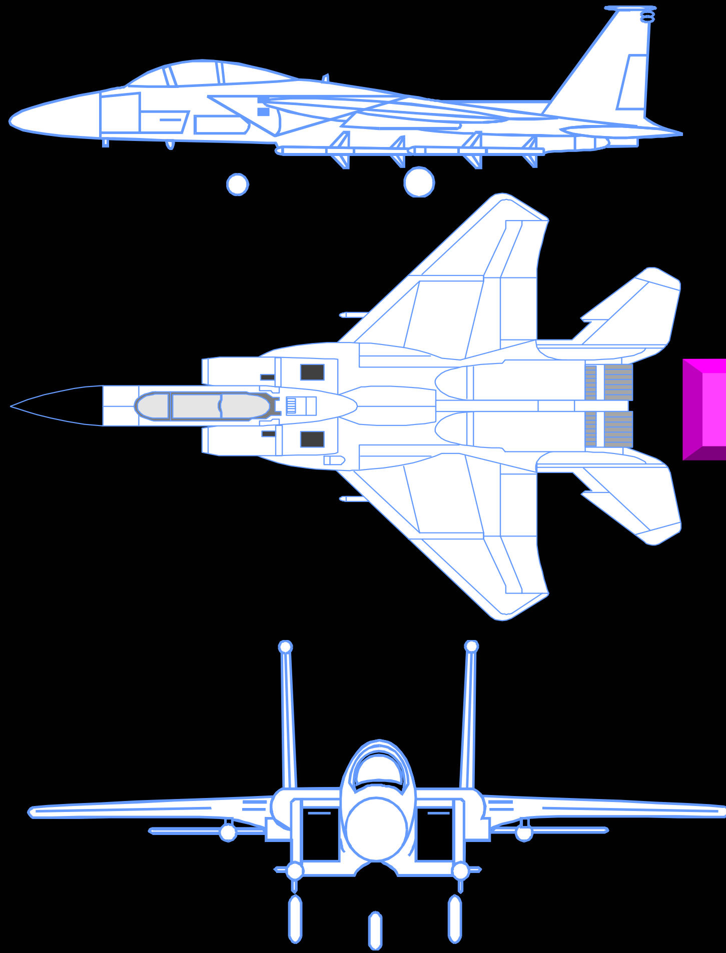
软件开发的本质

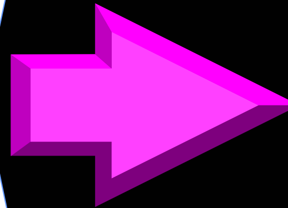
- 建立现实世界和软件世界之间的一个映射
- 面向过程的方法 （关注的是处理过程的描述）
  将现实世界映射成为一个过程
- 面向对象的方法 （关注的是构成过程的对象实体）
  使用UML建立对象模型来映射现实世界



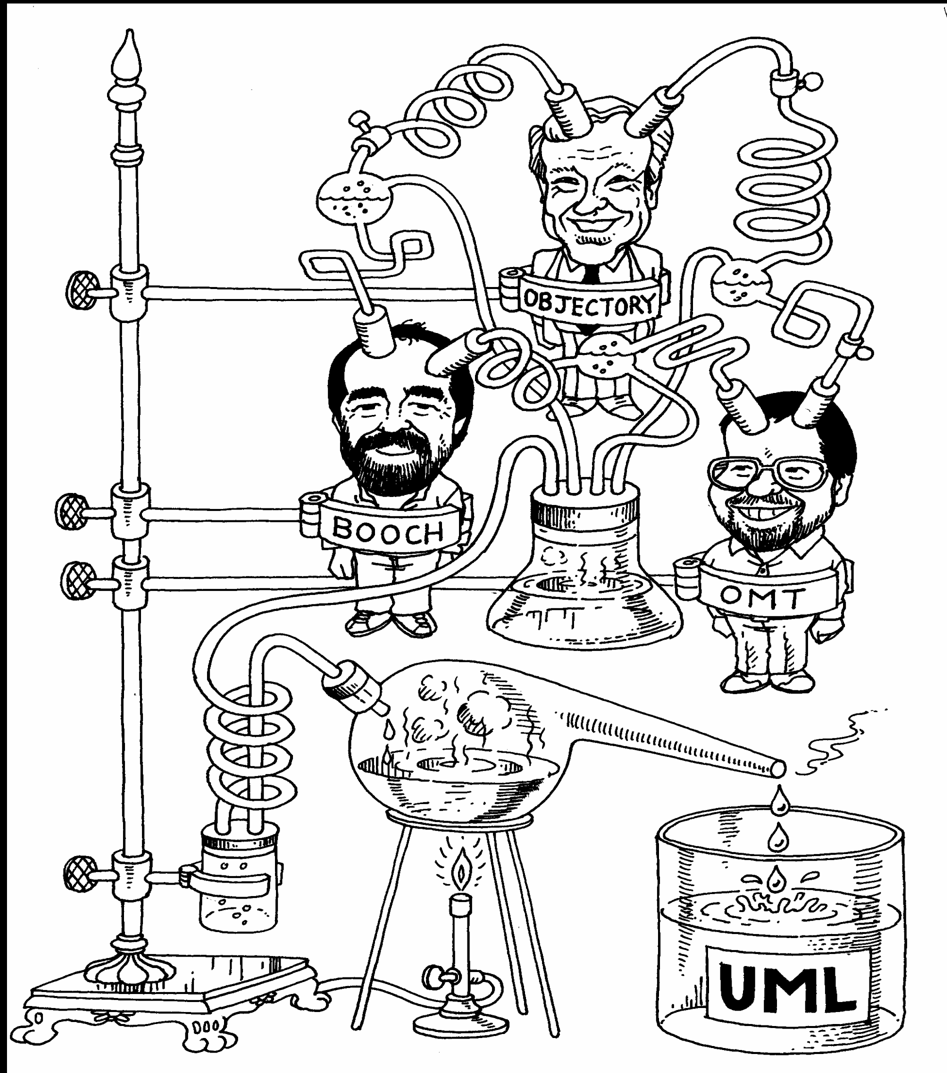
什么是模型?

- 模型是对现实世界的抽象





UML：统一的国际标准建模语言（描述软件系统的方法）



大师思想的结晶

Jim Rumbaugh

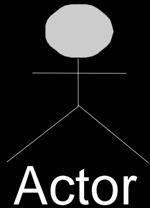
Grady Booch

Ivar Jacobson

基于UML的建模过程

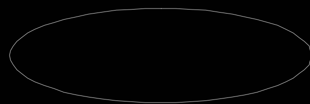
- 用例建模：描述系统需求
- 应用建模：搭建软件架构

参与者和用例



参与者(Actor)

在系统外部与系统发生交互的人或物



用例(Use case)

用例是由系统执行的一系列动作，其目的是针对某一特定的参与者产生一个可观察的结果

用例图

自动提款机(ATM)

银行客户

银行系统

出纳

维护职员

Withdraw Cash

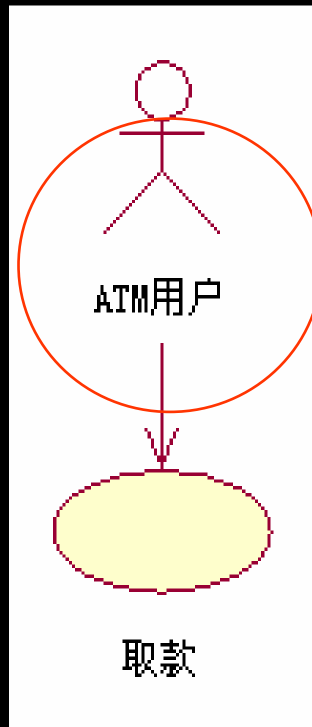
Transfer Funds

Deposit Funds

Collect Deposits

Maintain ATM

用例图



Use Case立足用户视角的描述，为具体的需求提供了充分的上下文信息，是衔接用户和开发者的纽带和沟通方式

用例规约

基本事件序列（ Basic Flow ）

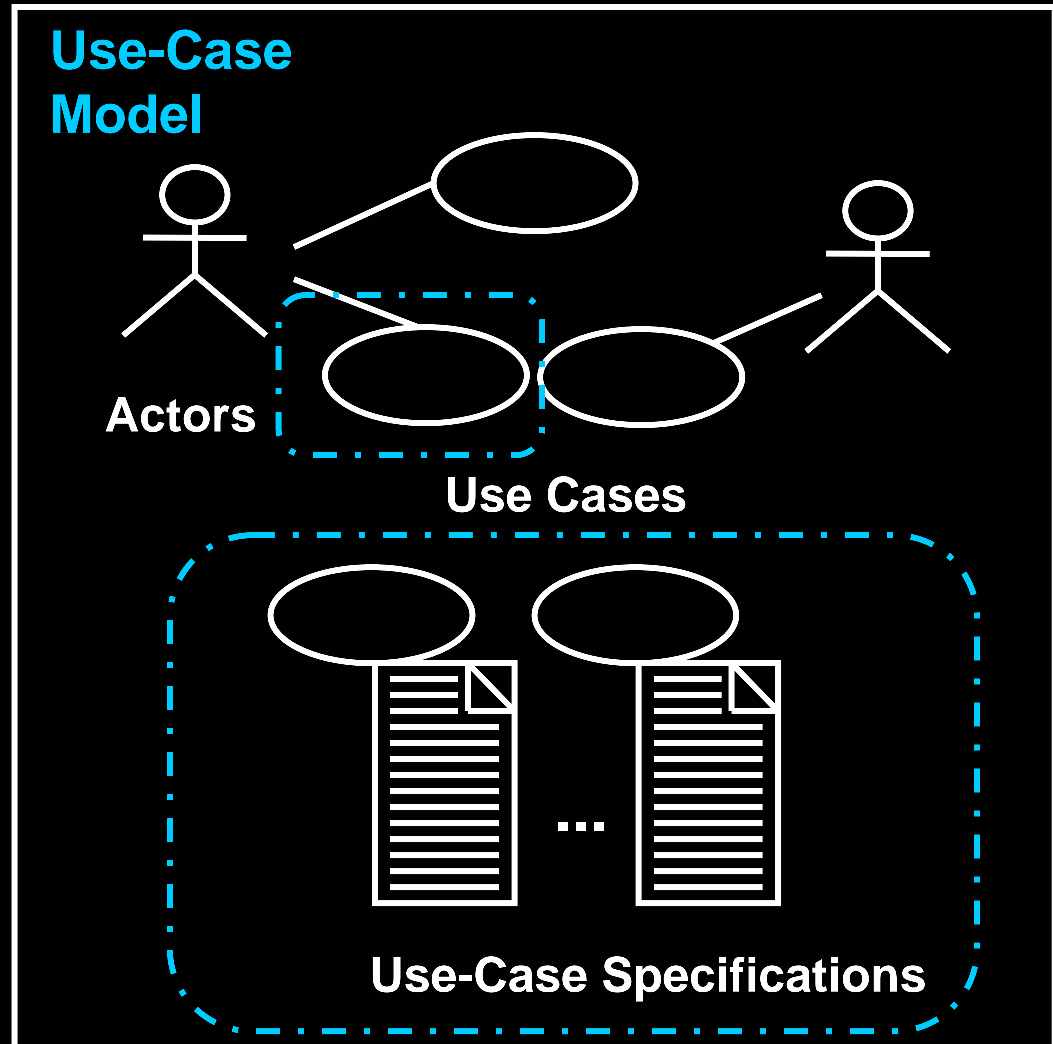
- 用户插入ATM卡
- 系统要求输入合法的密码
- 用户输入正确密码，如果用户输入的密码有误，转至备选事件流A1
- 系统提示用户选择“存款”或者“取款”
- 用户选择“取款”
- 系统提示用户输入取款金额
- 用户输入(合理)取款金额并确认，如果取款金额不合理,转至备选事件序列A2
- 系统从帐户中扣除取款金额，提示用户“打印收据”或者“不打印收据”
- 用户要求不打印收据，如果要求打印收据，转至备选事件序列A3
- 系统显示“交易结束”

备选事件序列（ Alternative Flows ）

- A1. …
- A2. …
- …

用例规约

- 用例名
- 概述
- 事件流
  - 基本流
  - 备选流
- 特殊需求
- 前置条件
- 后置条件



需求是一组文档

业务需求
产品特性
软件需求

功能性需求 (F)

前景文档

非功能性需求 (URPS)

用例模型

补充规约

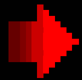
测试规约

设计规约

用户文档

基于UML的建模过程

- 用例建模：描述系统需求



- 应用建模：搭建软件架构

软件架构：“4+1 View” 模型

设计模型

Logical View

Analysts/Designers

Structure

实施模型

Implementation View

Programmers

Software management

Use-Case View

End-user

Functionality

Process View

System integrators

Performance

Scalability

Throughput

进程模型

Deployment View

System engineering

System topology

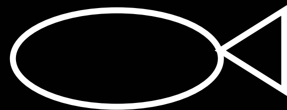
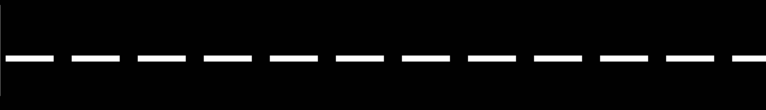
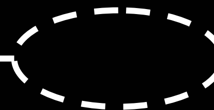
Delivery, installation

communication

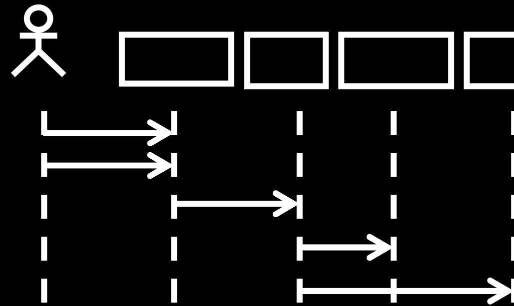
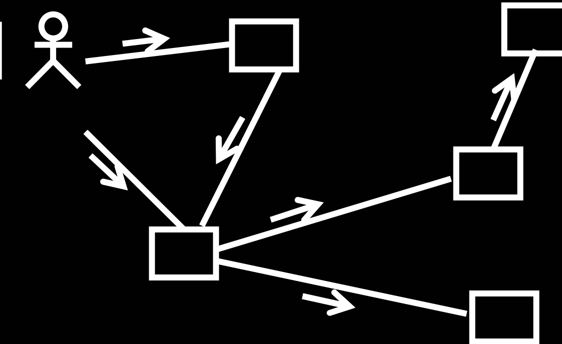
部署模型

需求和设计之间的桥梁：用例实现

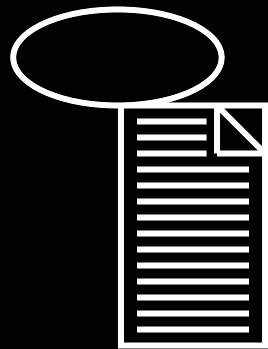
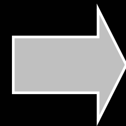
Use-Case Model *Design Model*

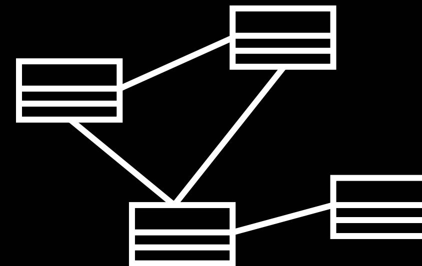
Use Case Use-Case Realization

Sequence Diagrams Collaboration Diagrams

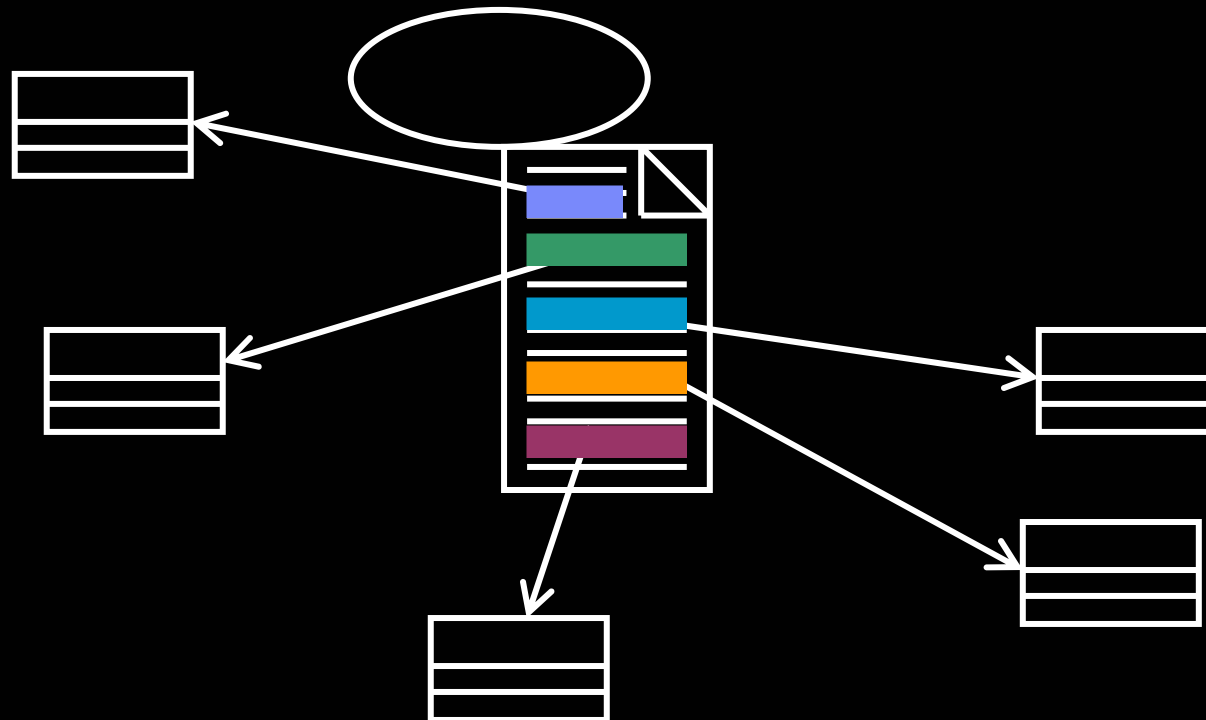
Use Case



Class Diagrams

用例实现的过程

- 把用例所描述的系统行为分配给一组类，通过类的实例对象之间的协作实现用例所描述的功能



使用序列图来描述对象之间的协作

: Student

:RegisterForCoursesForm

:RegistrationController

:CourseCatalogSystem

: Course Catalog

1: create schedule( )

2: get course offerings( )

3: get course offerings(forSemester)

4: get course offerings( )

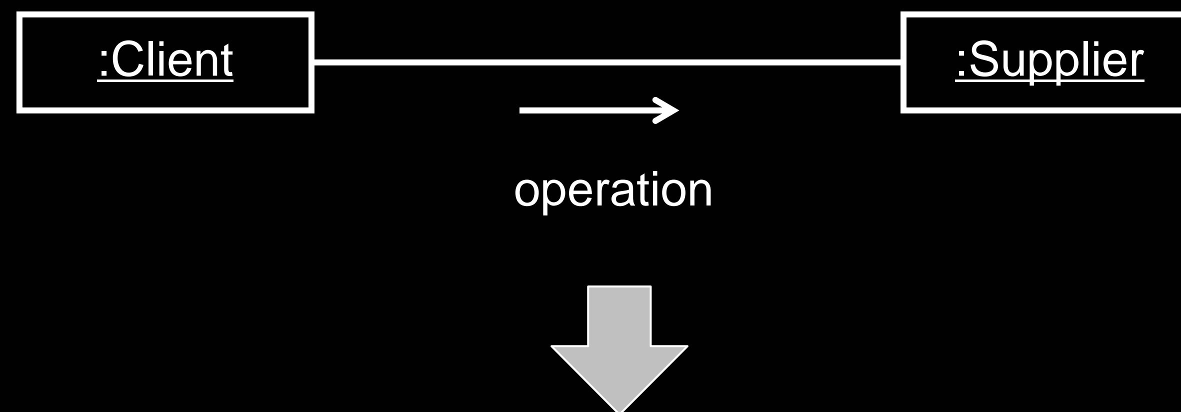
5: display course offerings( )

6: display blank schedule( )

找到类的操作

- 在描述交互图的过程中逐渐将用例行为分配到每一个类，成为类的操作

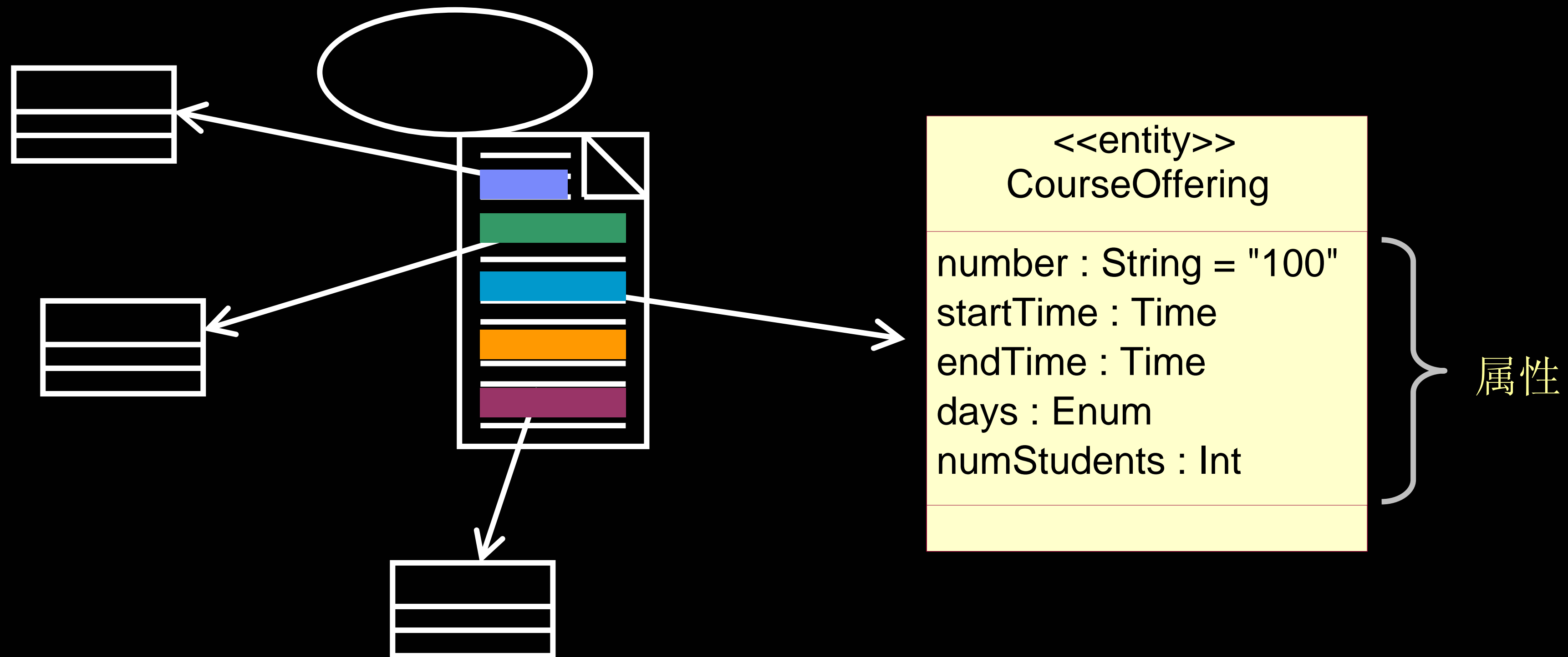
时序图



类图

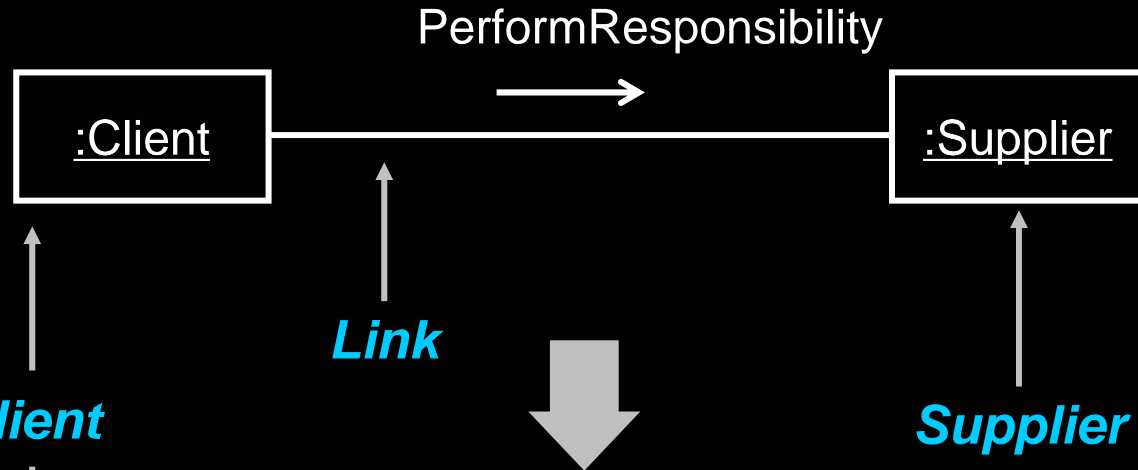
找到类的属性

- 用例行为中所用到的数据和信息被分配到相关的类中，成为类的属性



确定类之间的关系

交互图

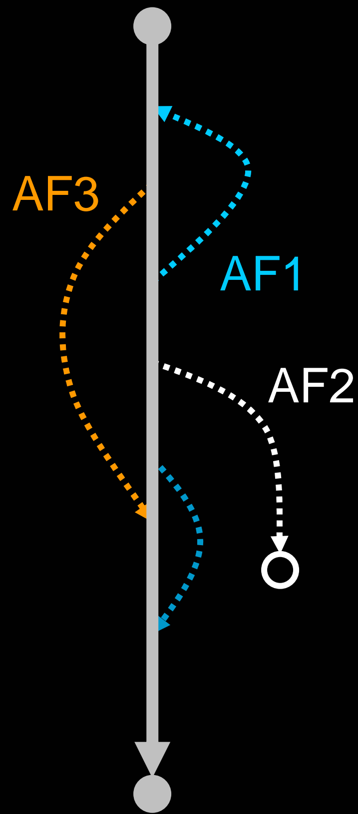


类图

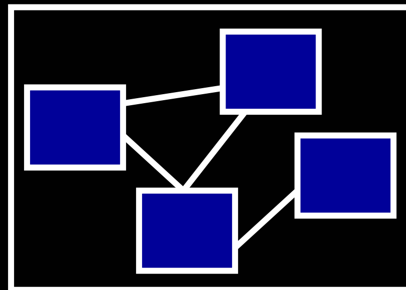
Relationship for every link!

描述用例的每一个场景(Scenario)

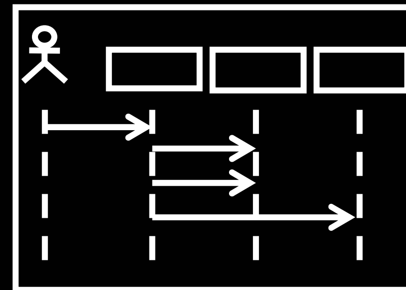
Basic Flow



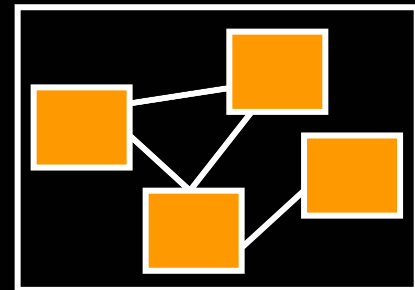
Alternate Flow 1



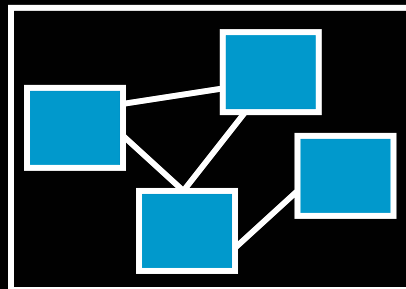
Alternate Flow 2



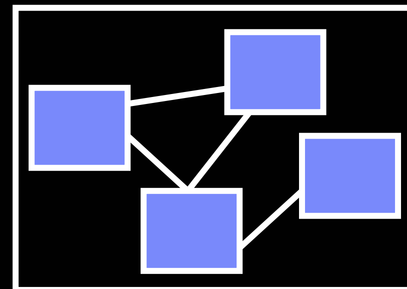
Alternate Flow 3



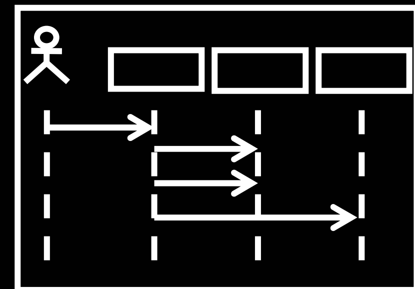
Alternate Flow 4



Alternate Flow 5



Alternate Flow n



描述相关类之间的静态关系

<<boundary>>
MainForm

// select maintain schedule()

1

0..1

<<boundary>>
MaintainScheduleForm

+ // open()
+ // select 4 primary and 2 alternate offerings()

1

1

<<boundary>>
CourseCatalogSystem

// get course offerings()

1

0..*

<<control>>
RegistrationController

// add courses to schedule()
// get course offerings ()

0..1

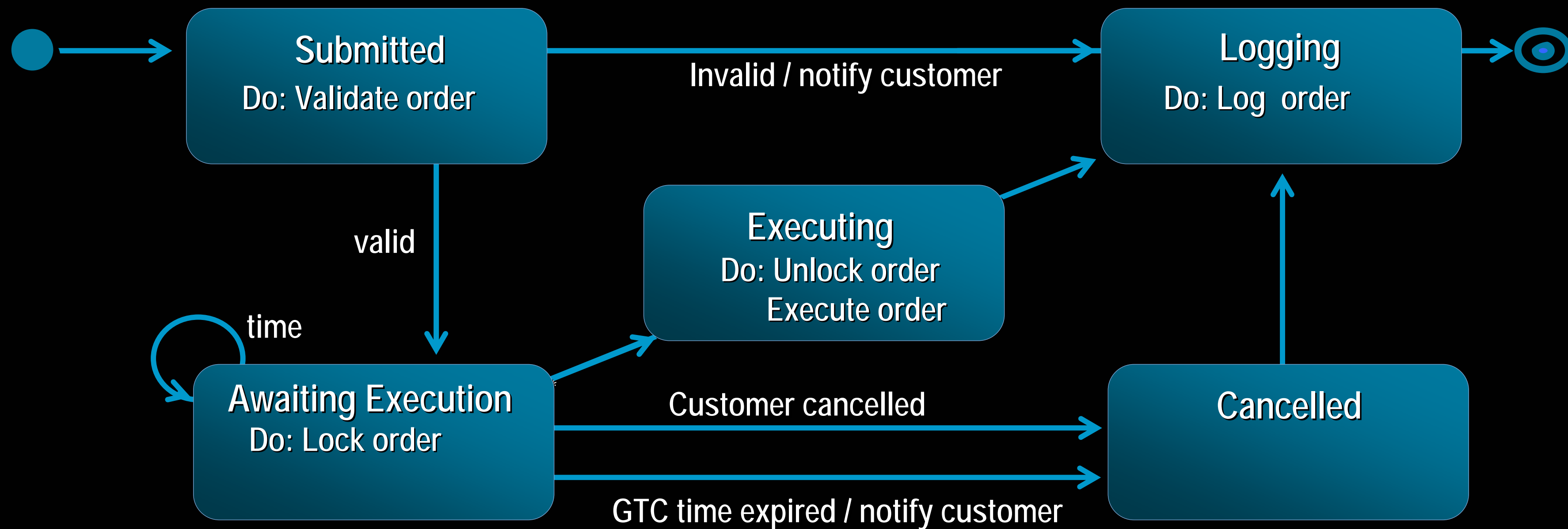
1

<<entity>>
Schedule

// create with offerings()

Logic View – 状态图

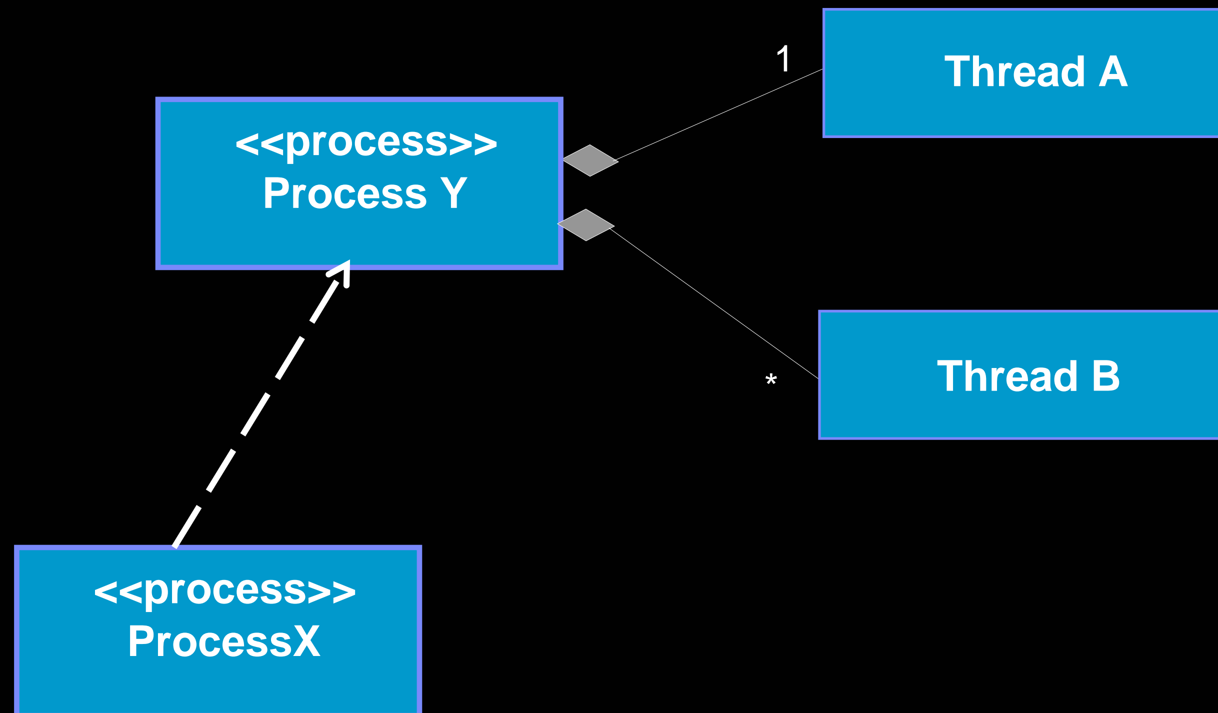
- 描述对象的生命周期



State Diagrams capture dynamic behavior of system

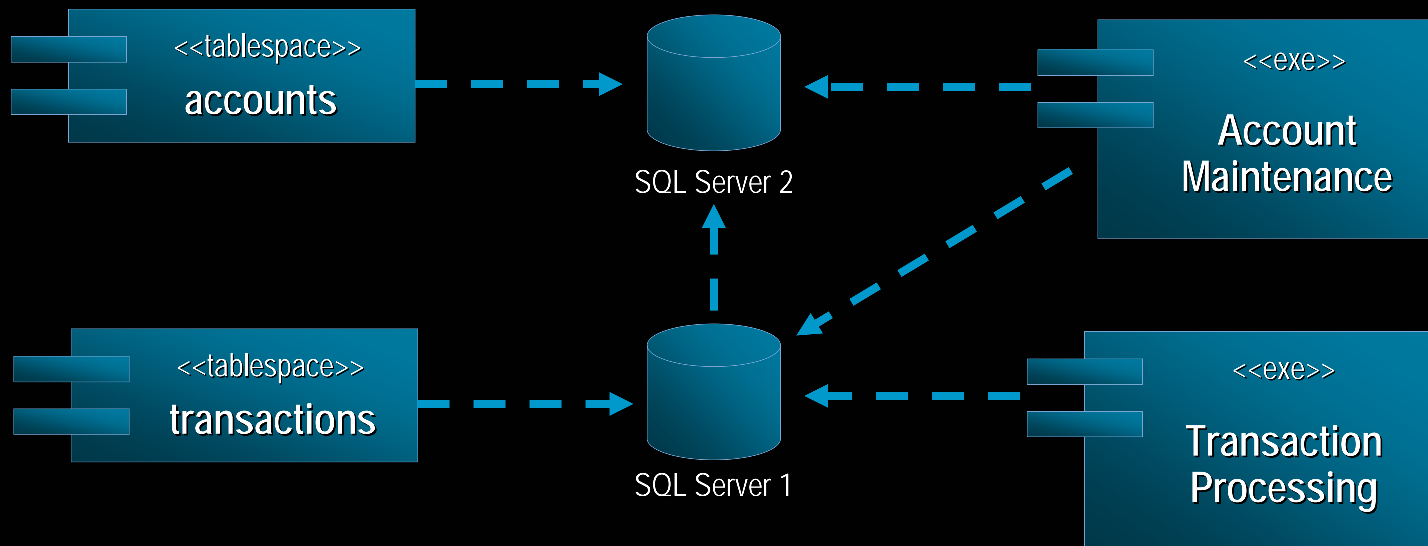
Process View – 进程关系

- 描述系统运行时的动态逻辑



Implement View – 组件图

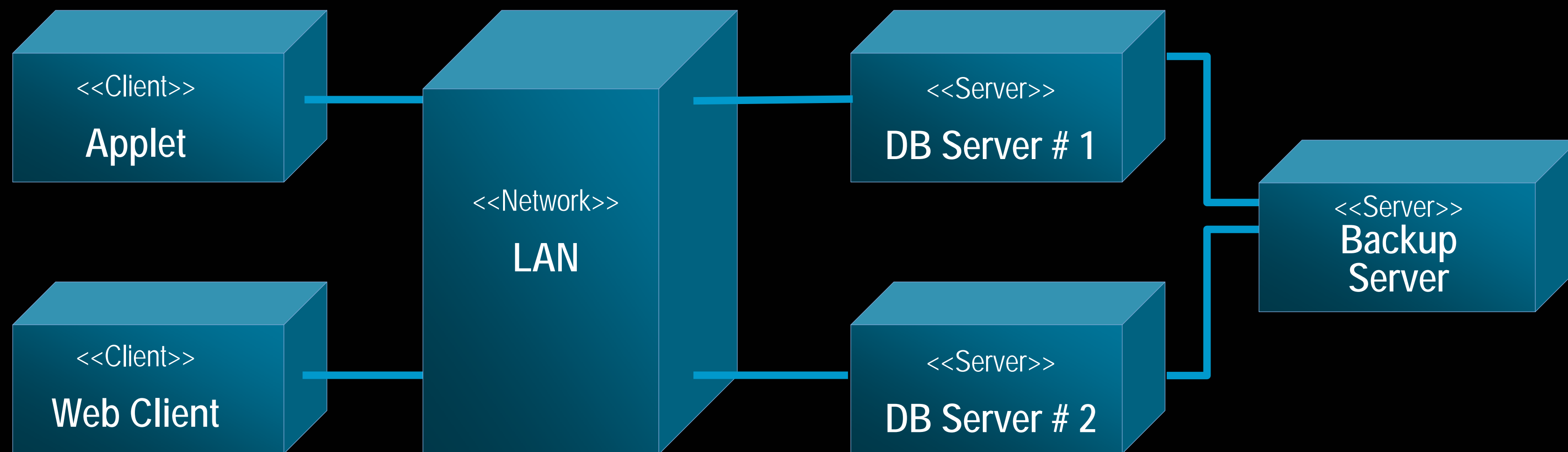
- 将逻辑设计映射到物理实现
- 确定最终的发布版本是由哪些构件组成的



Component Diagrams capture physical dependencies

Deployment View – 部署图

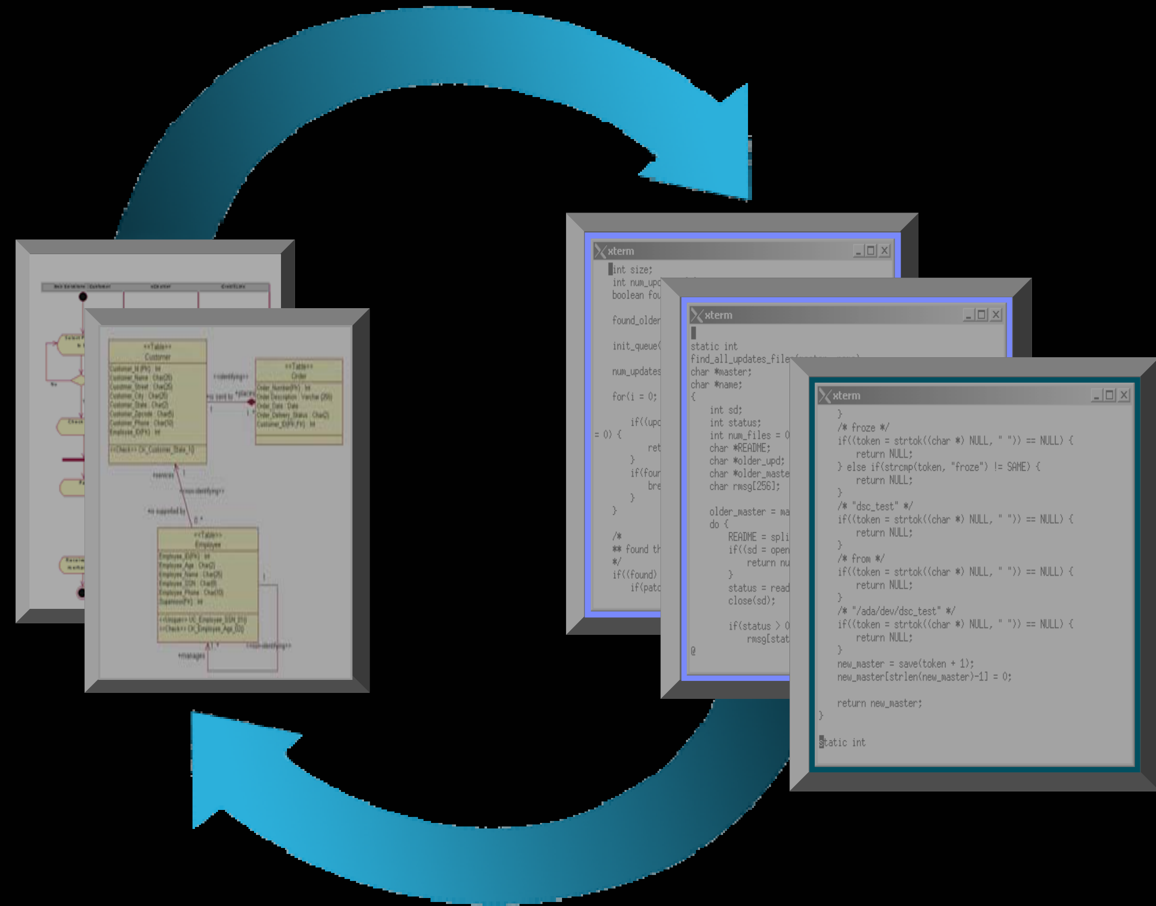
- 描述系统的网络拓扑结构
- 以及各功能组件在不同节点上的分布



Deployment Diagrams capture physical topology

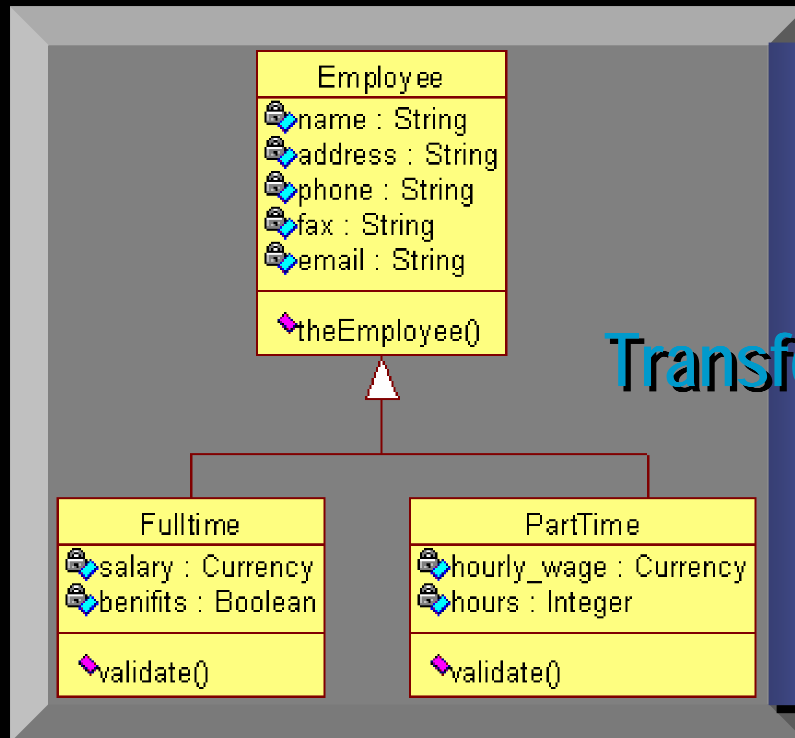
双向工程式 （Round-Trip Engineering)

- 保证模型和代码的一致性
- 模型的改变和代码的改变保持同步

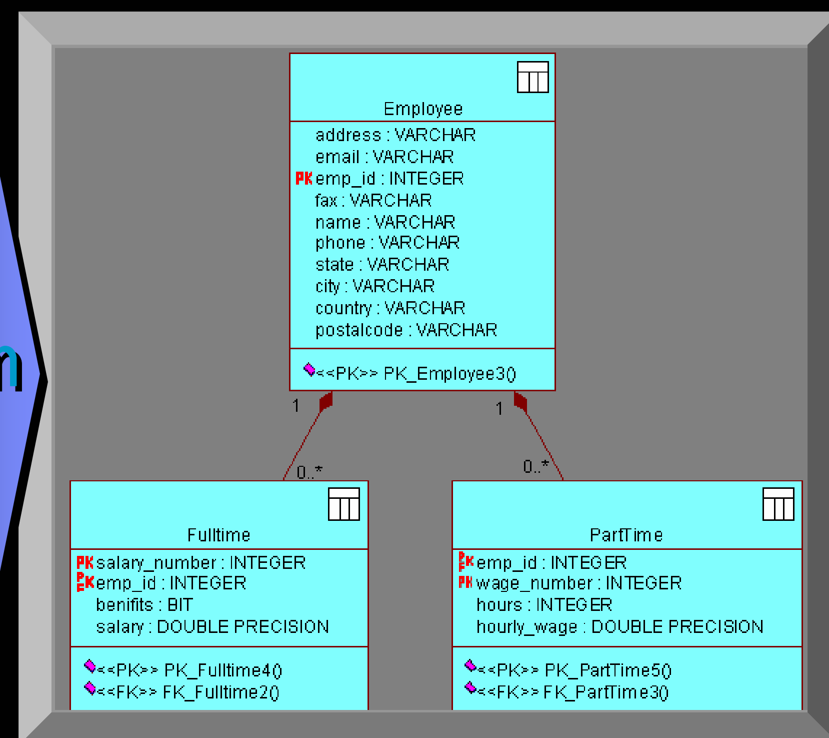


数据建模

Class Diagram



Database Diagram



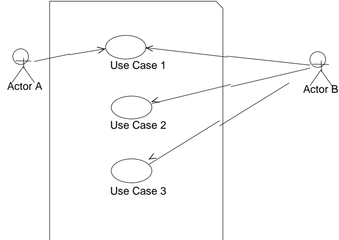
Class models are transformed to create a data model

可视化建模可以帮助我们：

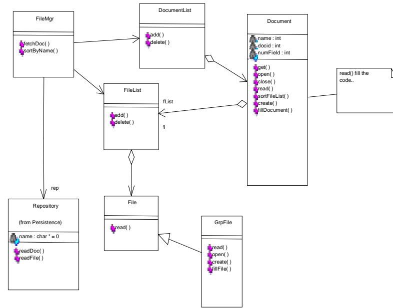
- 展现开发人员的设计思想
- 有效管理软件的复杂度
- 促进软件复用，提高软件生产率和质量
- 统一团队沟通的手段

展现开发人员的设计思想

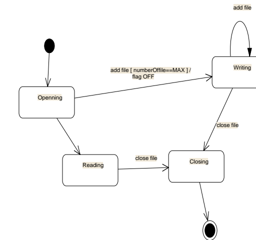
Use-Case Diagram



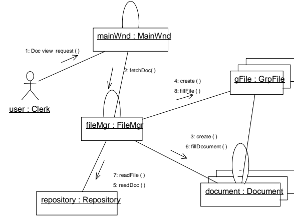
Class Diagram

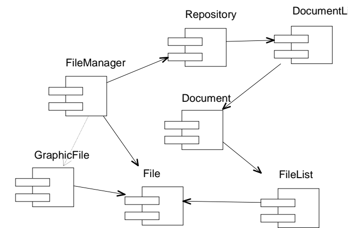


Statechart Diagram



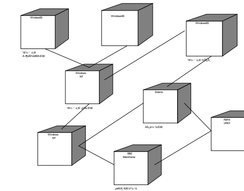
Collaboration Diagram

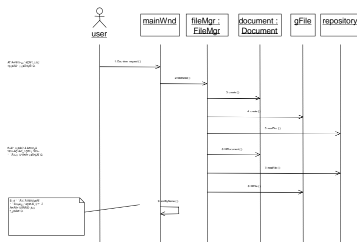




Component Diagram

Deployment Diagram

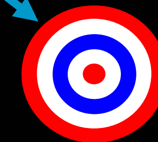




Sequence Diagram

Forward and Reverse Engineering

Target System



想象一下如何来理解如此复杂的图形

一个来自于消费类电子产品公司的实际应用架构

有效管理软件的复杂性

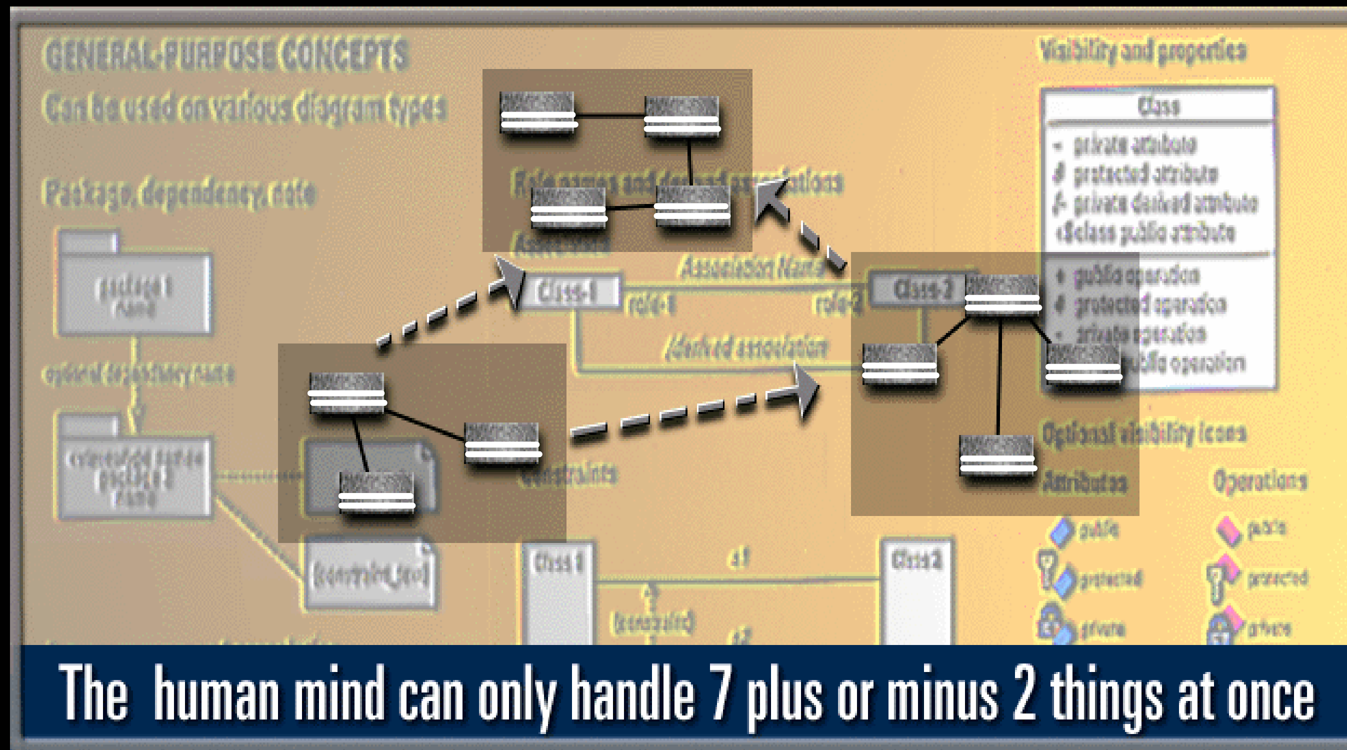
- 软件系统架构

逻辑架构：系统的功能特点，功能方面描述
物理架构：非功能性，可靠性，兼容性，系统分布，资源使用情况等。

软件体系分层(机制)
应用体系架构的思想，层模式，管道和过滤器，代理模式，MVC，微内核，Messagebus等等
子系统划分，component的划分，对象的管理等等

复杂度降低：使得软件质量有保证
软件结构化：使得软件具有更好的复用性

有效管理软件的复杂性



- 将模型划分成不同的视图（结构、行为等）
- 用包（Package）将视图组织成一棵抽象层次渐深的树形结构

有效地提高对软件的把握和分析能力

- 从不同的角度分析软件系统
软件结构的角度，信息流的角度，物理存放的角度，协同工作的角度，状态变化的角度

- 以不同的内容为中心来分析软件结构
- 有效地维护软件资产

软件架构： “4+1 View” 模型

设计模型

Logical View

Analysts/Designers
Structure

实施模型

Implementation View

Programmers
Software management

End-user
Functionality

Use-Case View

Process View

System integrators
Performance
Scalability
Throughput

进程模型

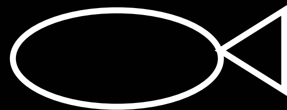
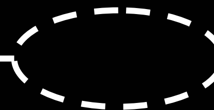
Deployment View

System engineering
System topology
Delivery, installation
communication

部署模型

需求和设计之间的桥梁：用例实现

Use-Case Model *Design Model*

Use Case

Use-Case Realization

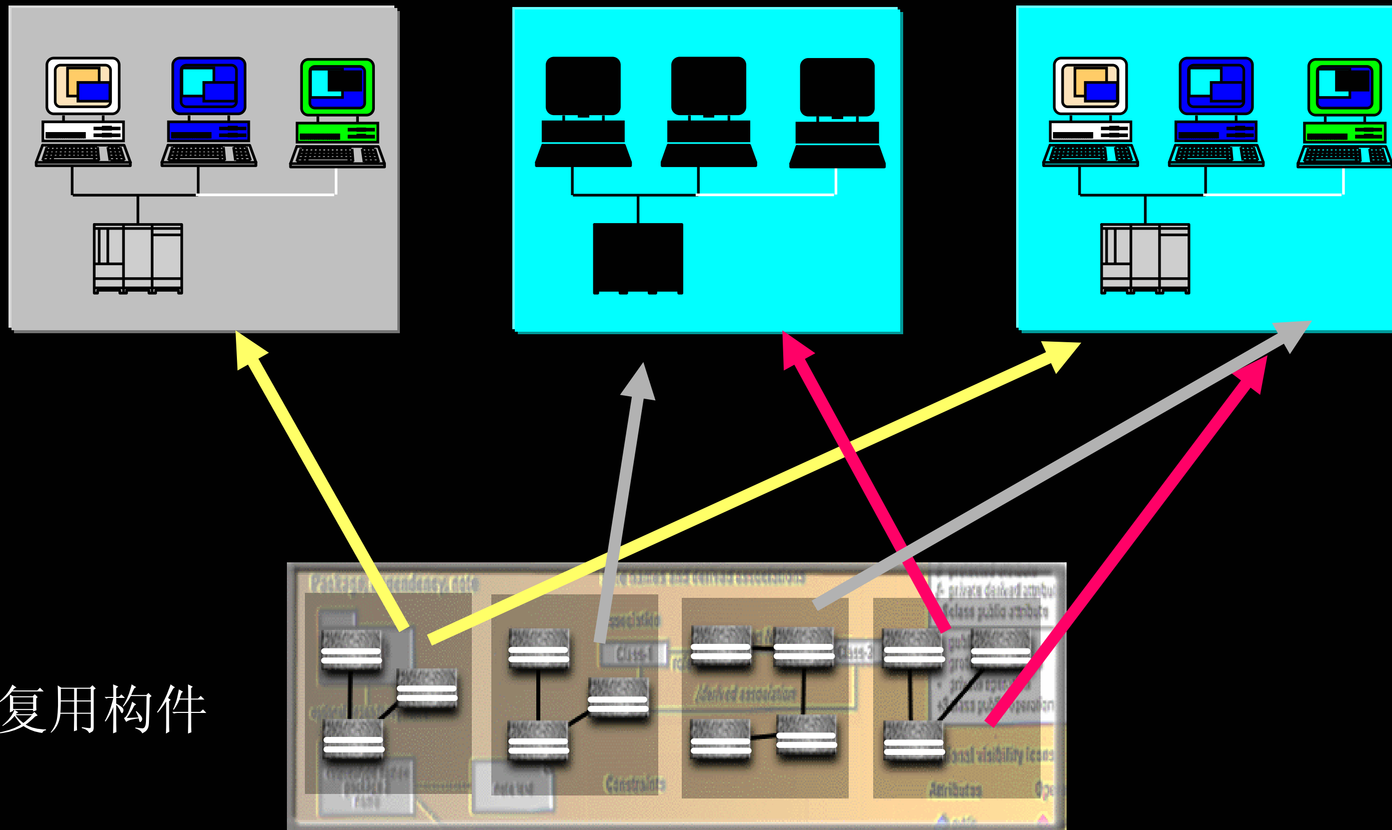
Sequence Diagrams

Communication Diagrams

Use Case

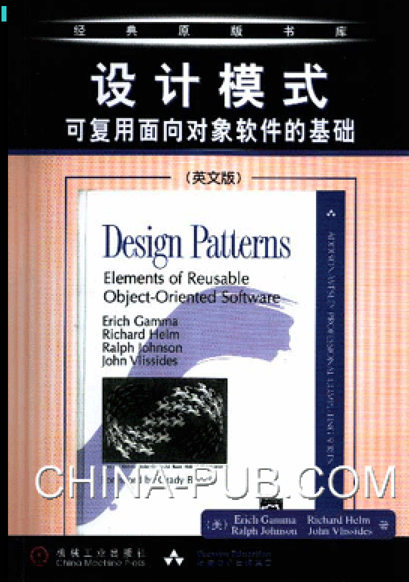
Class Diagrams

促进软件复用，提高软件生产率和质量



GoF 设计模式

- 四人帮 (Gang of Four)
  - Erich Gamma
  - Richard Helm
  - Ralph Johnson
  - John Vlissides



模式资源管理器

- Design Patterns
  - Behavioral
    - Chain of Responsibility
    - Command
      - Command
      - ConcreteCommand
      - Client
      - Invoker
      - Receiver
      - ConcreteCommandReceiver
      - CommandManager
    - Interpreter
    - Iterator
    - Mediator
    - Memento
    - Observer
    - State

Client «use» Invoker - command «interface» Command
«use»
«use» Receiver - receiver ConcreteCommand

概述 | 简短描述

如何理解设计模式

- “设计模式”电视机

Command模式 -> 电视机和遥控器上的所有按键
Adapter模式 ->遥控器和电视机的接口
Abstract Factory -> 实现电视机的大屏幕
Prototype ->可以试用的虚拟电视机
…

统一团队沟通的手段

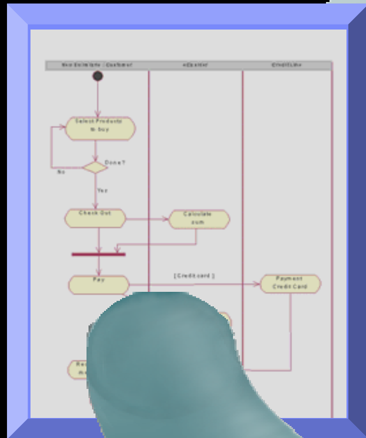
需求建模

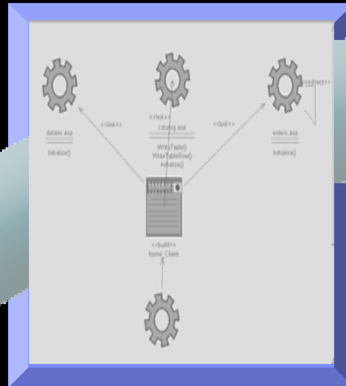
Web建模

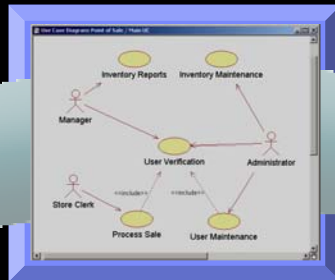
应用建模

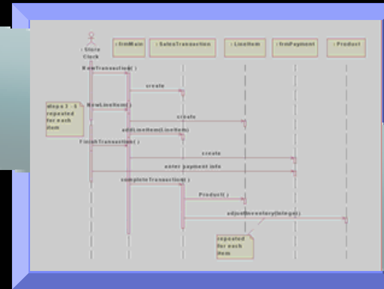
业务建模

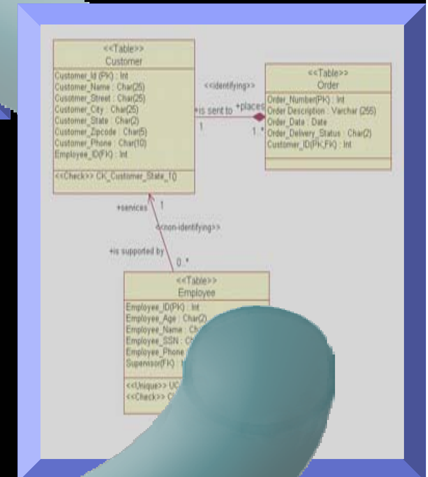
数据建模









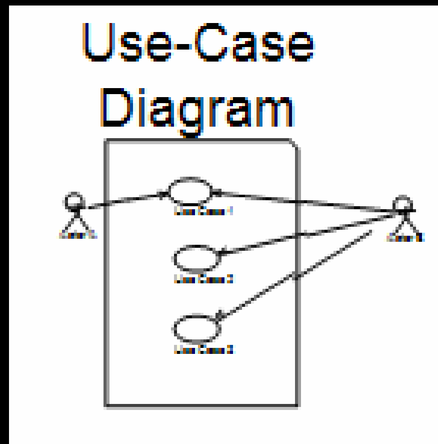


UNIFIED!

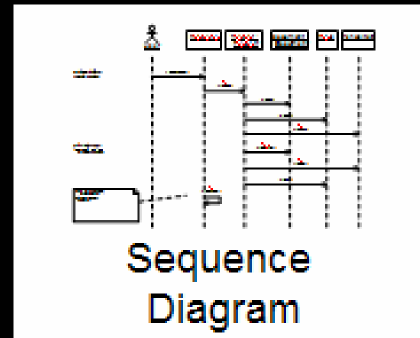
Unified Modeling Language

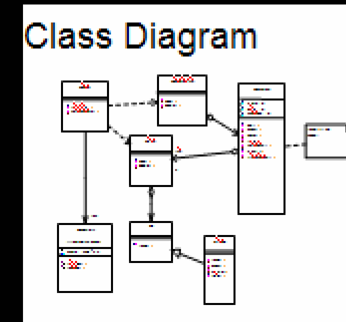
提高团队的沟通能力，沟通效率

业务人员关心的内容
测试人员关心的内容

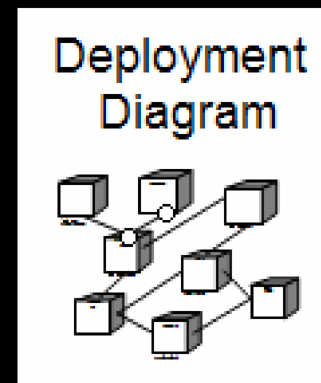


开发人员关心的内容



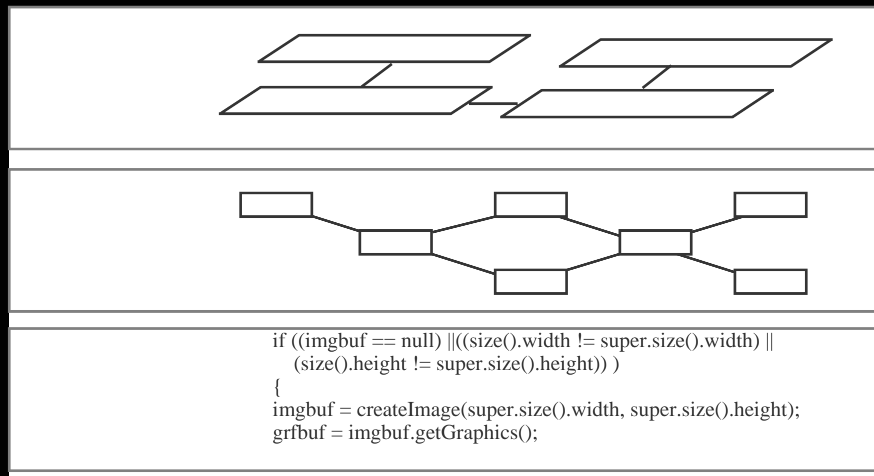


部署人员关心的内容



可视化建模的优点总结

- 表述软件设计
- 管理系统复杂度
- 促进软件重用
- 改善团队沟通



分层的模型适当地隐藏细节，帮助管理复杂度

实例分析及演示

一个具体的实例

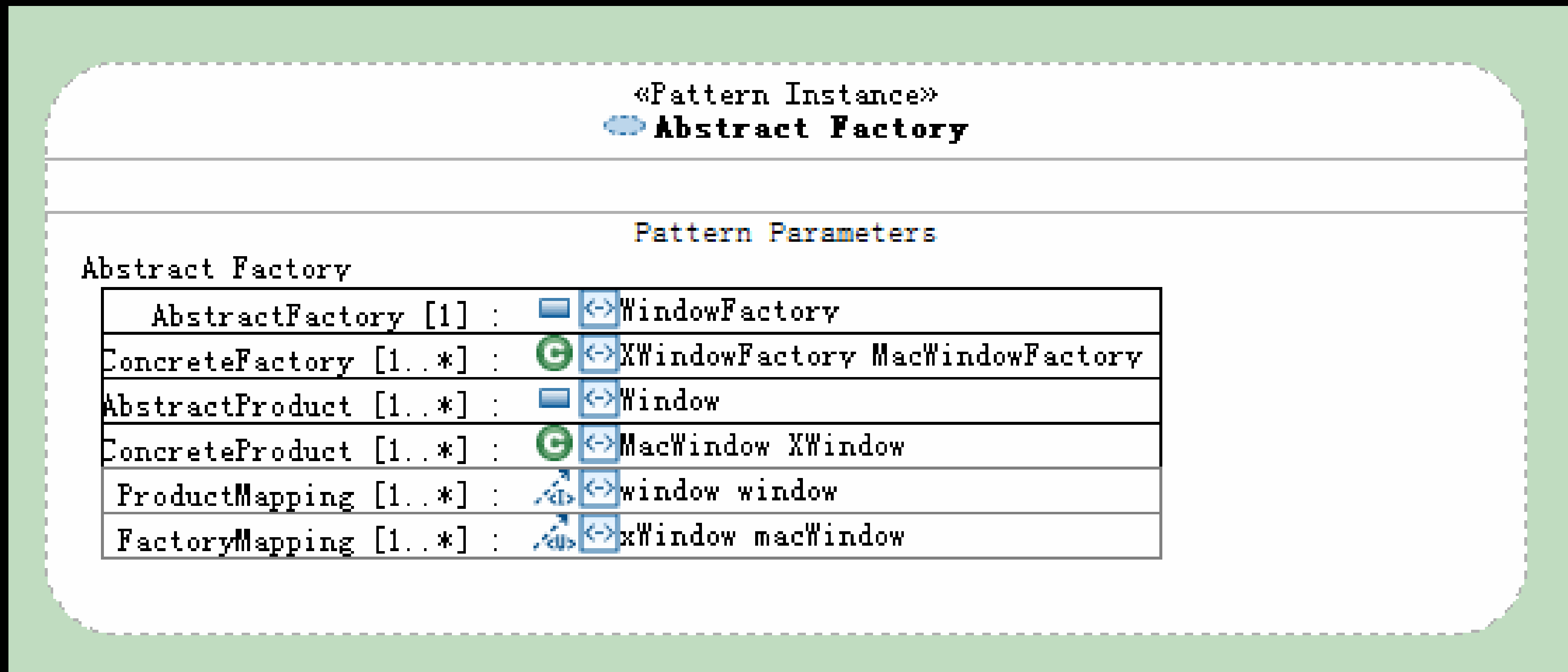
- 需求场景：一个软件系统

1. 要支持多种硬件平台，例如UNIX的XWindow 系统，Machintosh的MachWindow系统
2. Window 系统有大量的复杂的操作行为，例如 Open, Close ,Maximum,Minimum 等等
3. 由于硬件系统的区别，针对每种应用平台，有不同的软件编码。

面向过程的实现方法

```
int openXWindow (…);
Int openMacWindow(…);

int os=UNIX
…
if (os ==UNIX) then
  openXWindow(..);
Else
  if (os==MACHINTOSH) then
    openMacWindow(..);
```

面向对象的实现方法（引入Abstract Factory模式）



面向对象的实现方法（实体之间的关系）

«Java Class»
XWindowFactory
createWindow (

«Java Class»
MacWindowFactory
createWindow (

«use»

«use»

«Java Class»
XWindow
Open ( )
Close ( )
XWindow ( )

«Java Interface»
WindowFactory
createWindow (

«Java Class»
MacWindow
Open ( )
Close ( )
MacWindow (

«use»

«use»

«use»

«Java Interface»
Window
Open ( )
Close ( )

相应的代码

对于XWindows系统:

```
public static void main(String[] args) {
        WindowFactory wf = new XWindowFactory();
        Window w =wf.createWindow();
        w.Open();
```

对于MachintoshWindows系统:

```
public static void main(String[] args) {
       WindowFactory wf = new MacWindowFactory();
        Window w =wf.createWindow();
        w.Open();
```

Thank You