

复旦大学 计算机学院 研究生 专业课程

高级软件工程

软件构造

彭鑫

pengxin@fudan.edu.cn

www.se.fudan.edu.cn/pengxin

内容提要

- 软件构造概述
- 构造活动中的设计
- 编码
- 错误处理
- 单元测试
- 集成
- 软件构造工具

内容提要

- 软件构造概述
- 构造活动中的设计
- 编码
- 错误处理
- 单元测试
- 集成
- 软件构造工具

软件构造 v.s. 建筑构造

有何相同?

有何不同?



代码是软件开发的最终载体

○ 体现了软件开发的最终价值

- 尽管需求分析、体系结构设计、测试对成功的软件开发不可或缺，但是只有产生满足客户需求的软件产品，才能为客户带来真正的价值

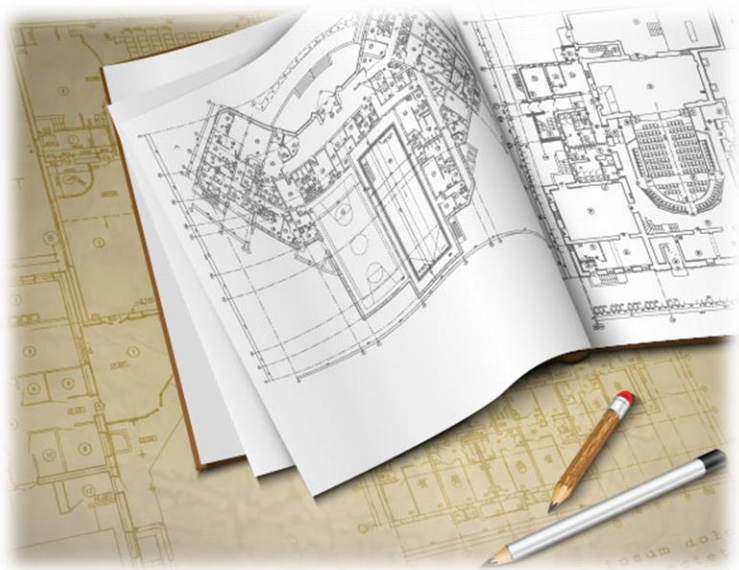
○ 反应了软件开发活动的正确性

- 作为最终载体的代码对期望的行为的表达也更精确和及时，从而更容易获得客户和最终用户的直接反馈。

○ 编码是唯一不可或缺的软件开发活动

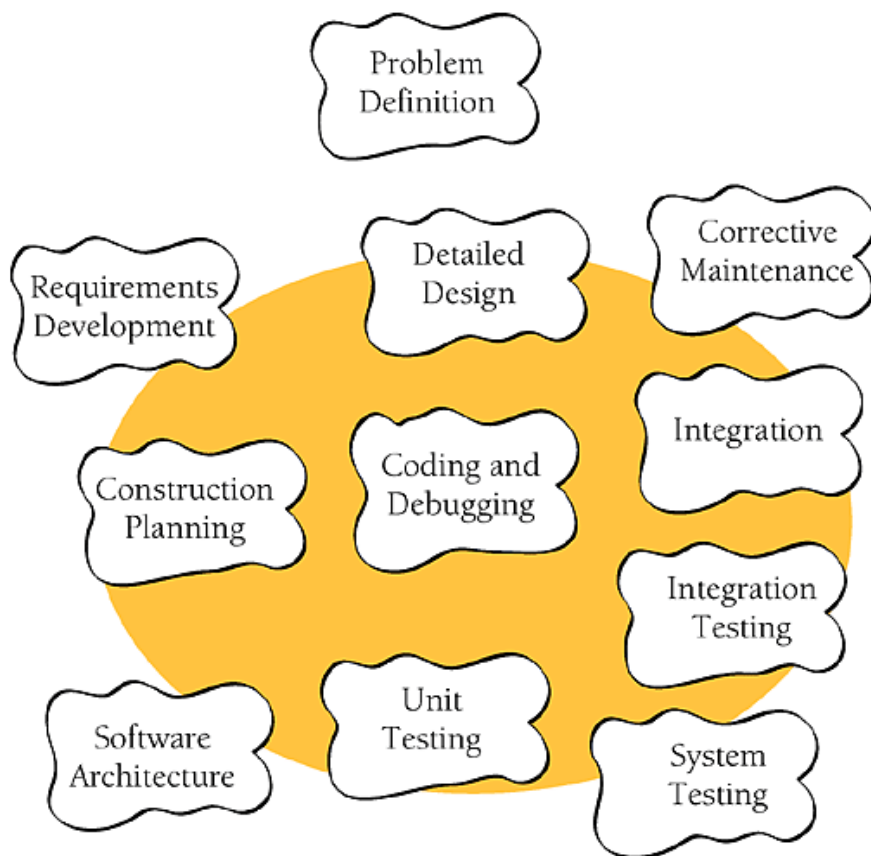
- 规范项目 VS “手工作坊”式项目

设计 VS 构造



软件构造活动不仅仅是编写代码，
更不是对软件设计的机械翻译。

软件构造



详细设计
编码、调试
单元测试
集成

编码不是唯一的构造手段

○ 构件复用与组装

○ 自动代码生成

- 模型驱动的开发（MDA）
- 可视化界面构造器

○ 可执行的模型

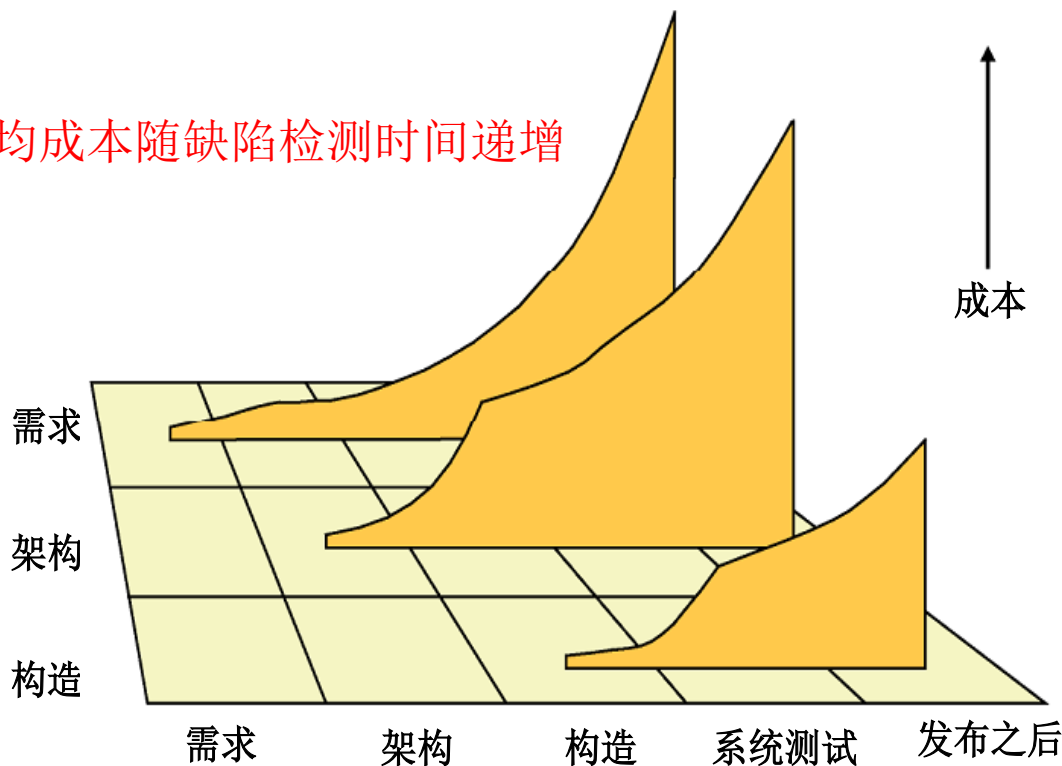
- 容器及中间件支持模型的运行时解释与执行
- 例如BPEL容器+BPEL模型

质量

- 正确性
- 可理解性
- 可复用性
- 可扩展性

正确性与缺陷成本递增律

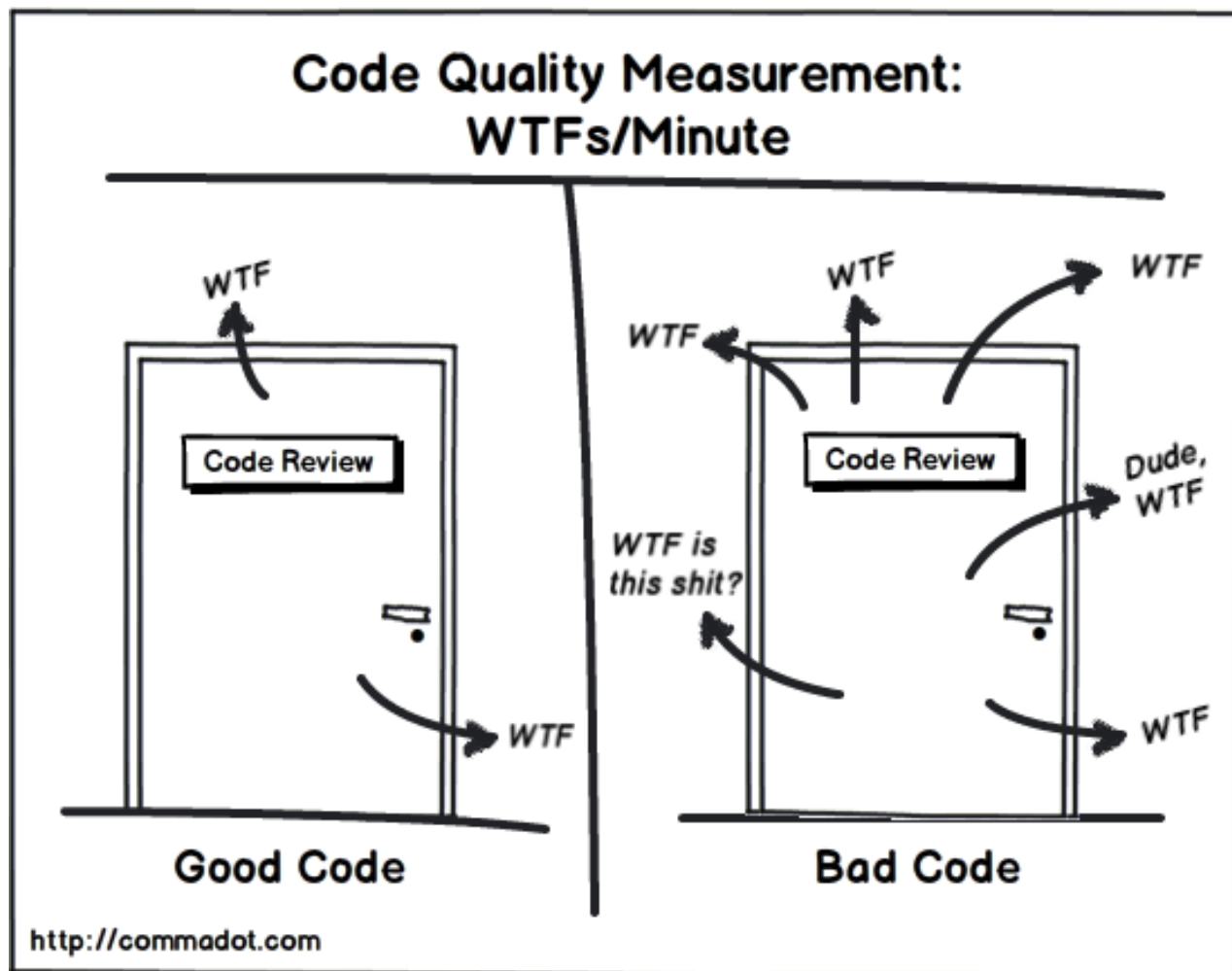
修复缺陷的平均成本随缺陷检测时间递增



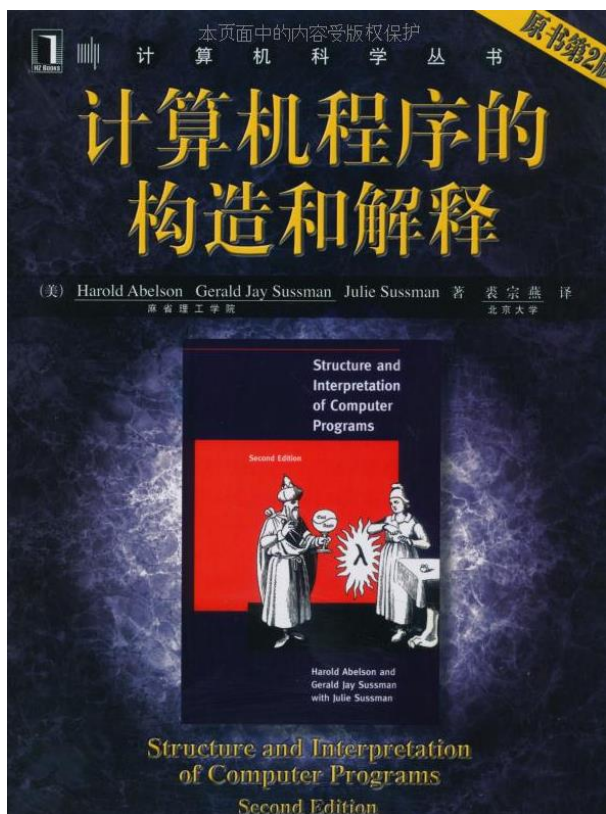
引入缺陷的时间	检测到缺陷的时间				
	需求	架构	构造	系统测试	发布之后
需求	1	3	5-10	10	10-100
架构	-	1	10	15	25-100
构造	-	-	1	10	10-25

惠普、IBM、休斯顿飞机公司等证明：在构造活动开始之前清除一个错误的返工成本仅为开发过程最后阶段做同样事情的1/10或1/100

可理解性



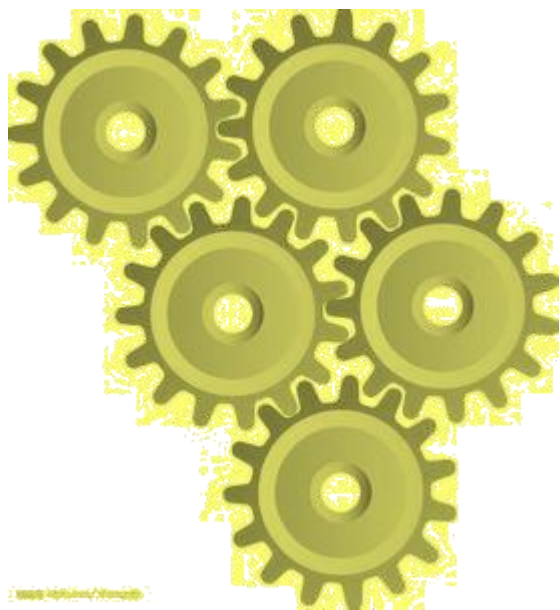
可理解性



程序写出来是给人看的，
只是顺便用作机器执行

——《计算机程序的构造与解释》

可复用性和可扩展性



- 产品需求的变化、新增的需求
- 变化的项目和产品上下文
- 未来的其它项目

内容提要

- 软件构造概述
- 构造活动中的设计
- 编码
- 错误处理
- 单元测试
- 集成
- 软件构造工具

ADT和设计分解

抽象数据类型

- 在较高的逻辑层面对现实世界的实体进行建模,
 - 避免深入该实体的底层细节
- 数据以及对这些数据所作的操作的集合

ADT举例-1

- 这段调整字体大小的代码有什么问题？

```
currentFont.size = 16
```

说明：这段代码将把字号设为小四号字体。小四号字体显示在屏幕上为16个像素。

- 这一段呢？

```
currentFont.size = PointsToPixels(12)
```

ADT 举例-2

○ 上述两段代码的问题

- 在低层次的实现细节上进行思考和描述
- 容易变化、缺乏封装、没有体现问题领域的概念

○ 更好的做法

```
currentFont.setSize(12)
```

更多例子

List

- Initialize list
- Insert item in list
- Remove item from list
- Read next item from list

Light

- Turn on
- Turn off

Stack

- Initialize stack
- Push item onto stack
- Pop item from stack
- Read top of stack

Elevator

- Move up one floor
- Move down one floor
- Move to specific floor
- Report current floor
- Return to home floor

ADT的好处

“首先思考问题的实质，然后思考实现层面”

可以隐藏实现细节，改动不会影响到整个程序

真正意义的接口

更容易提高性能

让程序的正确性更显而易见

更易于修改

易于识别错误

程序更具自我说明性

无需在程序内到处传递数据

ADT和类

- 类是ADT在面向对象领域的实现
- ADT在结构化程序语言中也同样可以实现

类

- 类通常是一组数据和子程序构成的集合，这些数据和子程序共同拥有一组内聚的、明确定义的职责
- 类也可以只是由一组子程序构成的集合，这些子程序提供一组内聚的服务，哪怕其中并未涉及共用的数据

封装

继承

多态

封装

- 内聚：应该把哪些数据和操作放在一起？
 - 内聚性使一个类的代码集中于一个中心目标
 - 体现类的核心概念，进行合理的职责分解
- 信息隐藏：应该让哪些数据和操作对外部可见？
 - 信息隐藏对外部调用者屏蔽了无关的细节
 - 降低外部调用者的负担，便于在未来对内部结构进行调整

封装-内聚性

这段代码是内聚的吗？

```
class Rectangle{  
    public void draw();  
    public double area();  
};
```

- draw () - 绘图相关的职责
- area () - 几何计算相关的职责

假如某Client仅仅需要Rectangle的面积计算，它是否需要了解和绘图相关的内容？如果绘图本身依赖于一个很复杂的库呢？如果绘图库是平台相关的呢？

单一职责原则 (SRP)

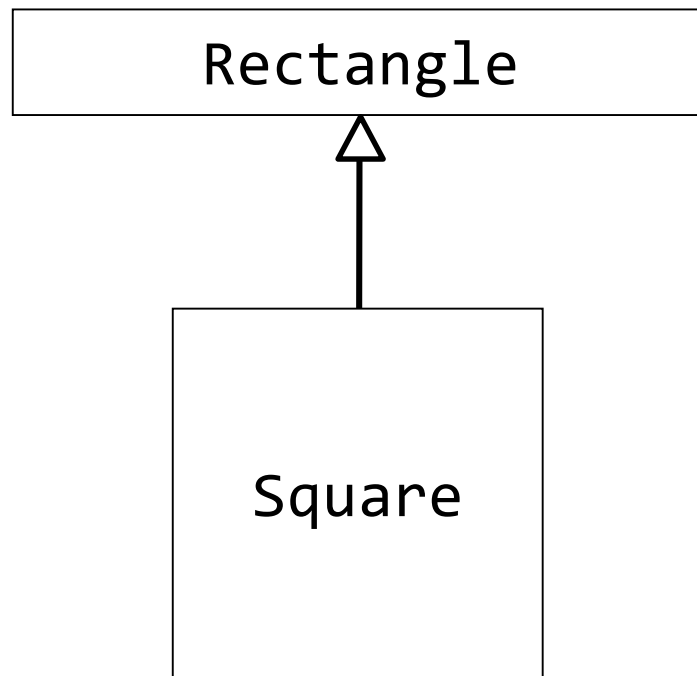
对于一个类而言，应当仅有一个变化的原因



"This drug is very expensive,
but the side effects are worth it."

继承

- 继承是一种“IS-A”的关系，表示子类能够完全继承父类的行为
- 问题：右侧的继承正确吗？



继承-例

```
class Rectangle{  
    public void setWidth(double width);  
    public void setHeight(double height);  
    public double getArea();  
    ...  
};
```

```
Square square = new Square();  
Rectangle rect = square;  
rect.setWidth(3.0);  
rect.setHeight(5.0);
```

getArea的返回值是多少？

LISKOV替换原则

- 任何子类型都应该能够完全替换父类型，即实现父类型所定义的行为
- 慎用继承：继承是一种强耦合关系

多态 (POLYMORPHISM)

- 允许设计人员将子对象的引用设置给父对象，然后父对象就可以根据当前赋值给它的子对象的特性以不同的方式执行

```
interface Shape{  
    public double  
    getArea();  
    ...  
};
```

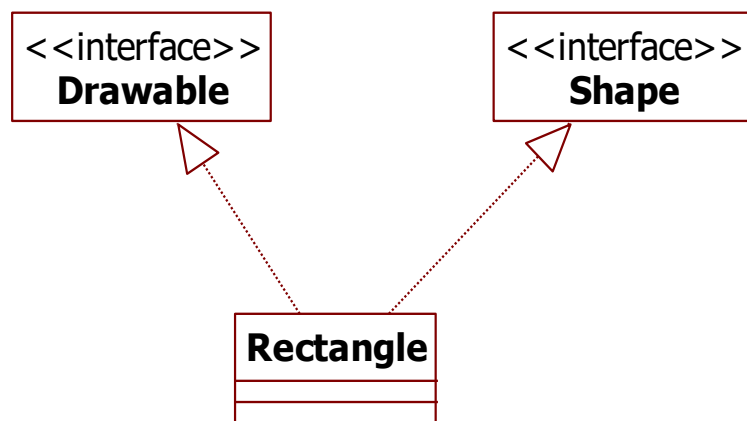
```
class Rectangle implements Shape{  
    public Rectangle(double width,  
    double height){...}  
    public double getArea(){ return  
    _width*_height;}  
};
```

```
class Circle implements Shape{  
    public Circle (double radius){...}  
    public double getArea(){ return  
    PI*_radius*_radius;}  
};
```

依赖

- 面向对象的设计中必然需要类之间相互协作完成系统功能
- 这就给彼此协作的类之间带来了一种依赖关系
- 依赖意味着协作，同时也形成了耦合

接口隔离原则 (ISP)

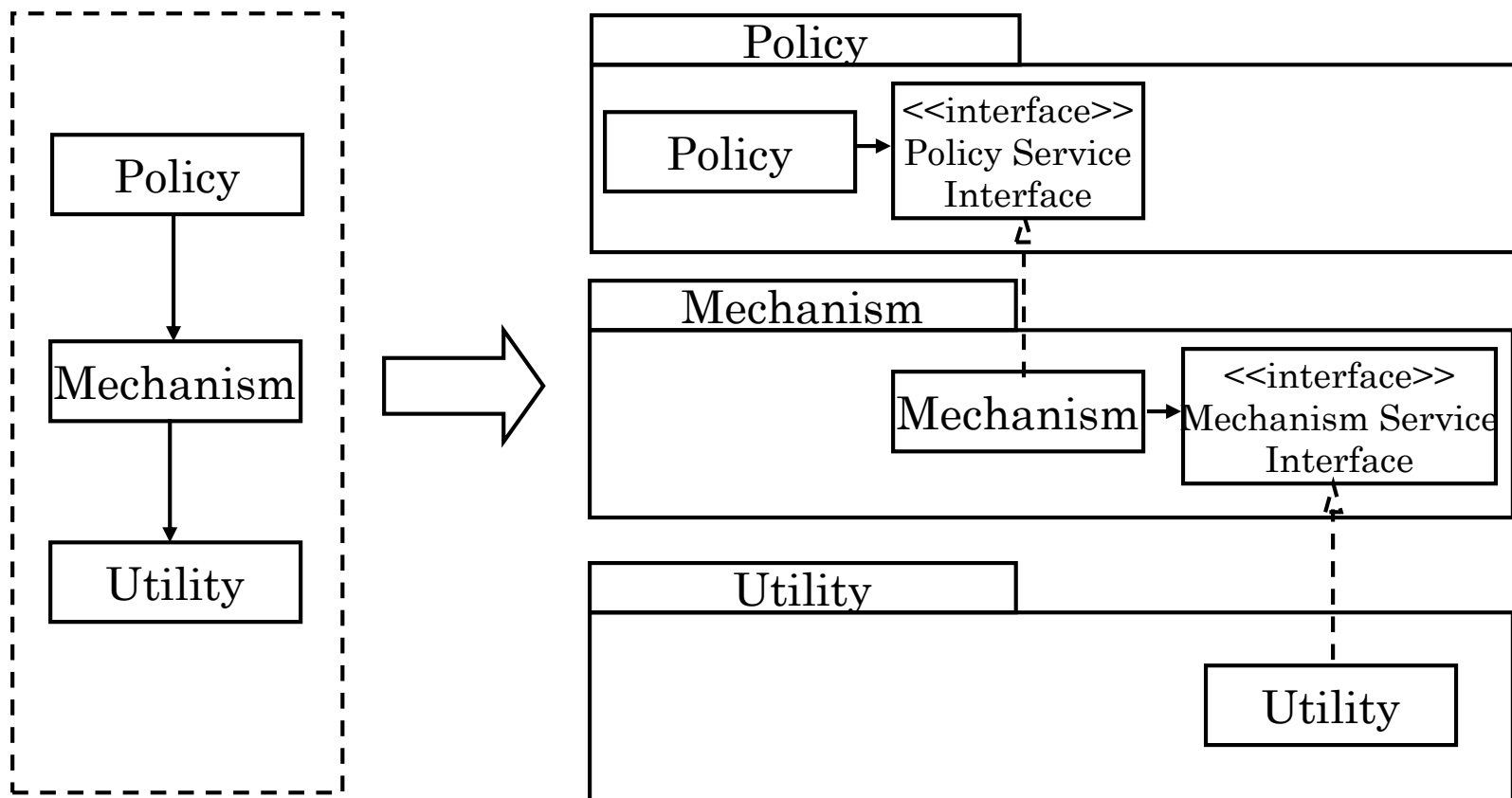


接口隔离:

- (1) 简化了调用方需要理解的概念
- (2) 降低了类间的耦合
- (3) 提高了对变化的适应能力

依赖倒置

让具体依赖于抽象，底层细节依赖于高层接口

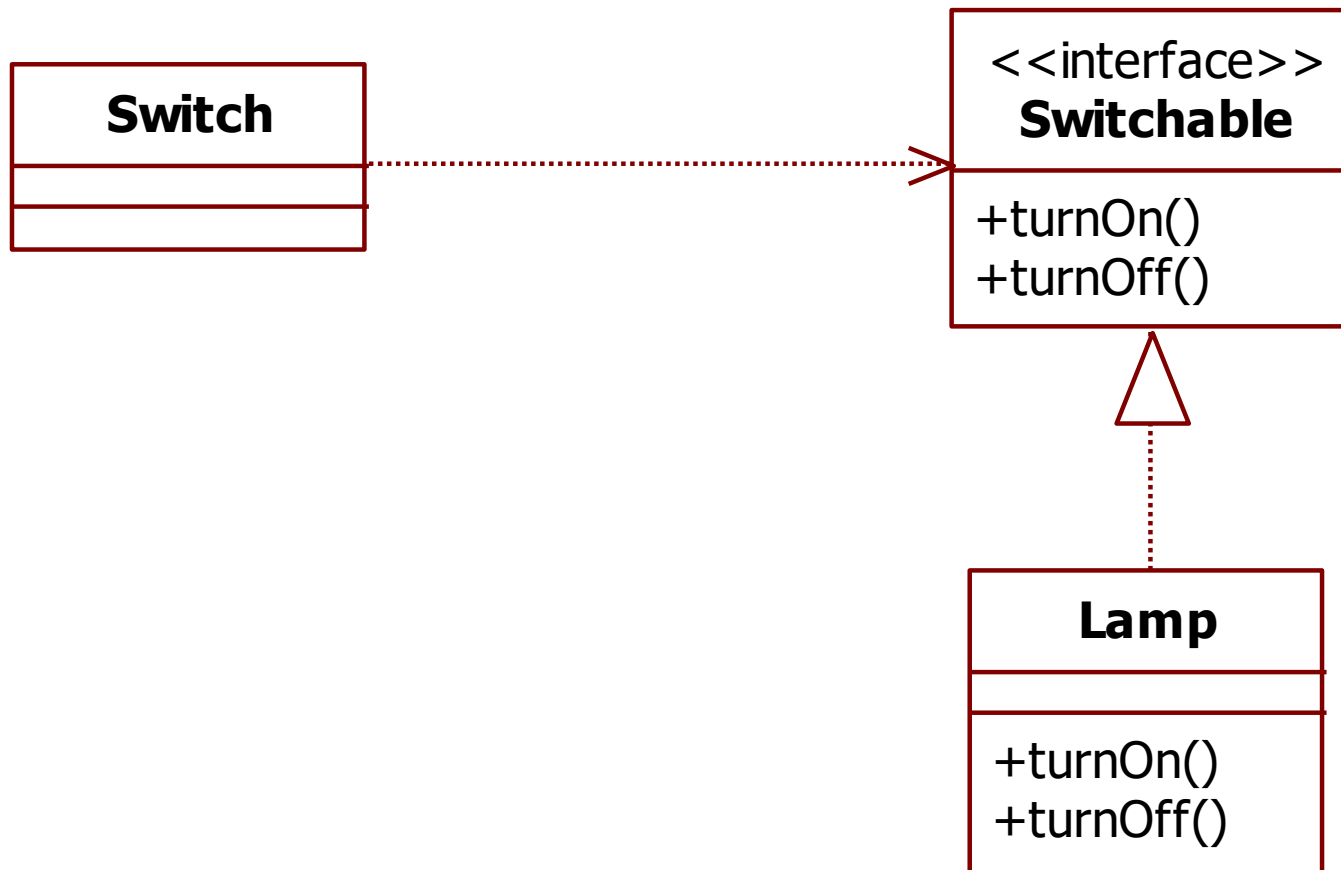


依赖倒置例-1



如果让Switch还可以控制风扇呢？

依赖倒置例-2



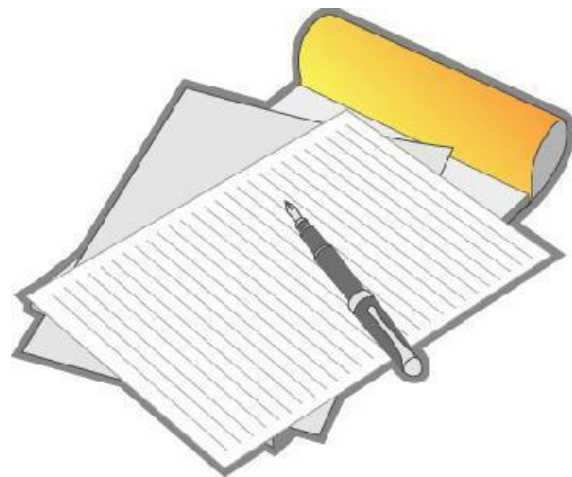
依赖注入

```
public class Switch {  
    public Switch (Switchable switchable) {  
        this.switchable = switchable;  
    }  
    ...  
    private Switchable switchable;  
}  
public class DependencyInjectionExample {  
    public static void main(String[] args) {  
        Switchable lamp = new Lamp();  
        Switch switch = new Switch(lamp);  
    }  
    ...  
}
```

契约式设计 (DBC)

- 契约是一种隐喻，将软件系统的元素之间的协作关系类别为商业活动中的“客户”与“供应商”的合作关系
- 目的：精确描述软件元素的“责任”与“义务”，更好的支持软件系统的元素之间的相互协作

**DBC: Design
By Contract**



“契约” 是一种隐喻

契约（合同）是由双方或多方达成的一个协议，用于约束各方之间的权利和义务



买卖合同		
	客户（甲方）	供应商（乙方）
权利	获得产品	获取费用
义务	支付费用	提供产品



软件系统和人类活动的相似性

软件系统的运行可以看做是**代码元素**（子系统、模块、类、方法等）**之间的协作**



设计契约		
	Client	Server
权利	获得承诺的服务	服务请求必须满足前提条件
义务	仅在允许的前提下调用服务	必须提供承诺的服务

契约的一个原始样例

```
/////////////////////////////////////////////////////////////////
// Function Nam      : Initializer::initPhase1
// Input Parameters  :
// Output Parameters :
// Return Value      :
// Preconditions     :  Pre-condition
// Description       : This function is used to let the initializer object
//                    perform its job : initialization phase 1 of the application.
// Post-conditions   :  Post-condition
// Exceptions        :
// Notes             :
// Examples           :
// See Also          :
//
void initPhase1(void);
```

契约应该明确可检验

- 契约必须显式说明
- 契约必须在代码中明确检验
- 契约式设计要求在设计时通过断言为软件元素定义正式的、精确的、可验证的接口

三种断言

- 先验条件：先验条件规定了调用一个软件元素（如类）的方法（或消息）之前必须为真的条件
- 后验条件：后验条件规定了一个软件元素的方法（或消息）执行完毕之后必须为真的条件
- 不变式：不变式规定了一个软件元素（如类的实例）在调用任何方法前后都必须为真的条件

契约示例

class interface

SIMPLE_STACK[G]

count:INTEGER

-- 栈中的元素数量

is_empty:BOOLEAN

-- 栈是否为空

ensure

consistent_with_count: Result =
(count = 0);

top:INTEGER

-- 查询栈顶的元素

require

stack_not_empty: count>0

push(g: G)

-- 将g压入栈顶

ensure

count_increased: count = old count + 1;

g_on_top: item_at(count) = g

pop

-- 移除栈顶元素

require

stack_not_empty: count > 0

ensure

count_decreased: count = old count - 1;

invariant

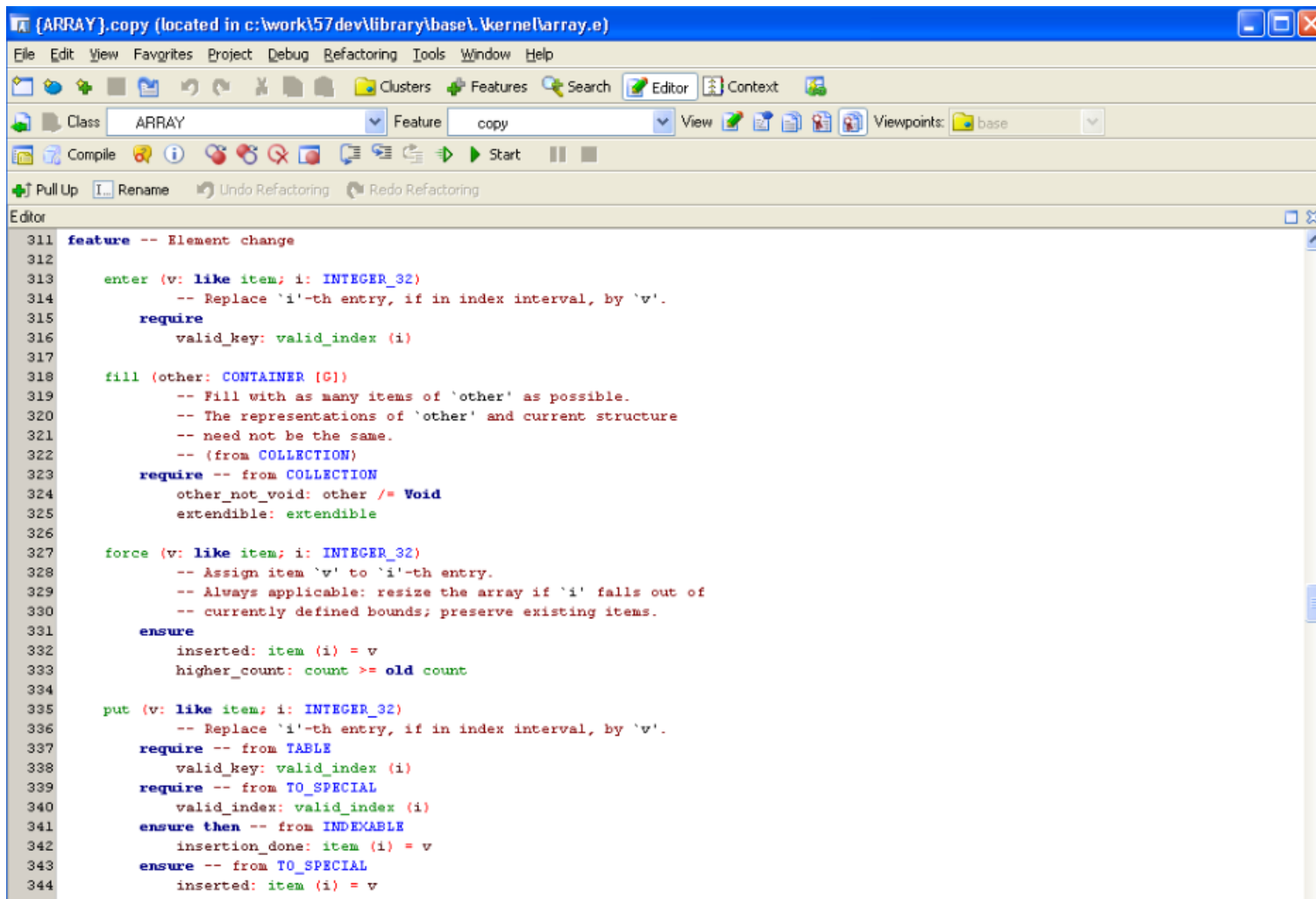
count_is_never_negative: count >= 0

end

使用支持DBC的语言

- Eiffel是一套面向对象编程语言，1985年由Bertrand Meyer所发明
- Eiffel文法类似Pascal，并且将语言本身与软件工程、和工具结合为一
- Eiffel实现契约式设计（Design by Contract）的风格

EIFFEL程序示例



```
[ARRAY].copy (located in c:\work\57dev\library\base\kernel\array.e)
File Edit View Favorites Project Debug Refactoring Tools Window Help
Clusters Features Search Editor Context
Class ARRAY Feature copy View Viewpoints: base
Compile Pull Up Rename Undo Refactoring Redo Refactoring
Editor
311 feature -- Element change
312
313 enter (v: like item; i: INTEGER_32)
314     -- Replace 'i'-th entry, if in index interval, by 'v'.
315     require
316         valid_key: valid_index (i)
317
318     fill (other: CONTAINER (G))
319         -- Fill with as many items of 'other' as possible.
320         -- The representations of 'other' and current structure
321         -- need not be the same.
322         -- (from COLLECTION)
323         require -- from COLLECTION
324             other_not_void: other /= Void
325             extendible: extendible
326
327     force (v: like item; i: INTEGER_32)
328         -- Assign item 'v' to 'i'-th entry.
329         -- Always applicable: resize the array if 'i' falls out of
330         -- currently defined bounds; preserve existing items.
331     ensure
332         inserted: item (i) = v
333         higher_count: count >= old count
334
335     put (v: like item; i: INTEGER_32)
336         -- Replace 'i'-th entry, if in index interval, by 'v'.
337         require -- from TABLE
338             valid_key: valid_index (i)
339         require -- from TO_SPECIAL
340             valid_index: valid_index (i)
341         ensure then -- from INDEXABLE
342             insertion_done: item (i) = v
343         ensure -- from TO_SPECIAL
344             inserted: item (i) = v
```

使用第三方库实现DBC

```
1 import com.google.java.contract.*;
2 public interface Stack<T> {
3     public int size();
4
5     @Requires("size()>=1")
6     public T top();
7
8     @Requires("size()>=1")
9     @Ensures({
10         "size() == old(size()) - 1",
11         "result == old(top())"
12     })
13     public T pop();
14
15     @Ensures({
16         "size() == old(size()) + 1",
17         "top == old(obj)"
18     })
19
20     public void push (T obj);
21 }
22
```

Cofoja: enable to
annotate code with
contracts in the
form of
preconditions,
postconditions and
invariants

原则1：区分命令和查询

- 查询返回一个结果，但是不改变对象的可见性质
- 命令改变对象状态，但是不返回结果

```
class Stack{  
public:  
    virtual int size() const = 0;  
    virtual int top() const = 0;  
    virtual int pop() = 0;  
    virtual void push(int obj) = 0;  
    virtual ~Stack(){};  
};
```

原则2:

将基本查询同派生查询区分开

```
int count() const;  
bool isEmpty() const;
```

如果count为0，则isEmpty就必然为true。

基本查询： 不能被其它查询定义的查询

派生查询： 可以被其它查询定义的查询

原则3-6：如何定义契约

- **原则3：针对每个派生查询，设定一个后验条件，使用一个或多个基本查询的结果来定义它**
 - 只要我们知道基本查询的值，也就能知道派生查询的值
- **原则4：对于每个命令都撰写一个后验条件，以规定每个基本查询的值**
 - 结合“用基本查询定义派生查询”的原则(原则3)，我们现在已经能够知道每个命令的全部可视效果
- **原则5：对于每个查询和命令，采用一个合适的先验条件**
 - 先验条件限定了客户调用查询和命令的时机
- **原则6：撰写不变式来定义对象的恒定特性**
 - 类是某种抽象的体现，应当将注意力集中在最重要的属性上，以建立关于类抽象的正确概念模型

内容提要

- 软件构造概述
- 构造活动中的设计
- 编码
- 错误处理
- 单元测试
- 集成
- 软件构造工具

语言问题

论坛上经常有关于
语言的争辩。
这种争论必要吗？

语言问题

语言选择和生产效率

语言	等效的C语言行数
C	1.0
C++	2.5
Fortran 95	2.0
Java	2.5
Perl	6.0
Python	6.0
Smalltalk	6.0



影响语言选择的要素

- 语言自身的效率：现代软件工程中，首要考虑的是编程效率而不是执行效率
- 语言的普及程度
 - 更容易获得技术资料
 - 更容易找到程序员
- 当前系统所使用的编程语言
 - 何时应当继承？何时应当演进？
- 当前团队成员对语言的熟悉程度
 - 学习成本
 - 编程语言有相当丰富的经验的程序员比几乎没有经验的程序员开发效率要高3倍

语言和编程范式

○ 分类1

- 命令式编程 (Imperative programming)
 - 告诉计算机HOW
 - C, C++, Java, ...
- 声明式编程 (declarative programming)
 - 告诉计算机WHAT
 - SQL, Prolog, ...

○ 分类2

- 结构化编程
- 面向对象编程
- 函数式编程
- 逻辑编程

语言和编程范式（续）

○ 分类3

- 强类型语言：Java, ...
- 弱类型语言：C, JavaScript, ...

○ 分类4

- 静态类型语言：Java, C++, ...
- 动态类型语言：Ruby, Python, ...

编程规范

1. 为开发人员提供了一致的编码风格，有助于提高软件的可读性以及维护的效率
2. 便于有效地推广已被证明行之有效的最佳实践
3. 可为日后的改进提供基线

一个编程规范的示例

Heuristic 2.5: Do not put implementation details such as common-code private functions into the public interface of a class.

Interpretation: In this heuristic, “common-code private functions” refers to functions that are created by “factoring” common code out of several public member functions. These private functions are part of the internal implementation of the class, not the public interface. If you put these functions in the public interface, some users of the class might call them in their code. This will make it more difficult to change the internal class implementation details at a later date.

Example: In the following class, the `recompute_document_size()` function is a common-code private function – it shouldn’t be in the public interface, but it performs an internal operation that several other class operations need:

```
class Word_processor_document {
private:
    std::list<Paragraph> doc_contents; /* contents of the doc */
    int document_size; /* current number of pages */
public:
    void insert_paragraph(Paragraph p, Position insertpoint);
    void insert_text_block(TextBlock b, Position insertpoint);
    void insert_graphic(Graphic g, Position insertpoint);
    void run_spell_check(); /* this may modify the document */
    int get_document_size() const;
private:
    void recompute_document_size();
};
```

In this example, each of the “insert” functions and the `run_spell_check()` function will call `recompute_document_size()` to reset the value of the `document_size` attribute.

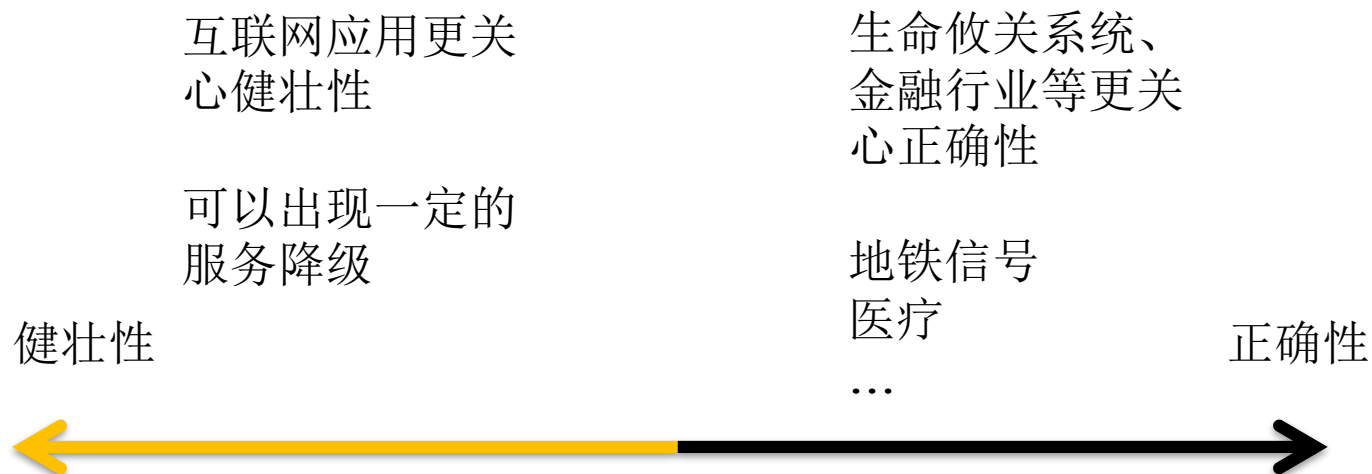
内容提要

- 软件构造概述
- 构造活动中的设计
- 编码
- 错误处理
- 单元测试
- 集成
- 软件构造工具

错误处理

- 软件在运行中可能出现预期的错误和不可预期的错误
- 错误会造成两类影响：
 - 正确性：系统表现出错误的功能
 - 健壮性：系统出现崩溃、宕机等

正确性和健壮性



某些设计模式能够同时兼顾二者。
当然要付出额外的成本。
例如三模块冗余结构。

断言：及时发现错误的方式

- 应用场景1：在契约式设计中，用断言来检查互相协作的模块双方是否遵循了接口所定义的义务
- 应用场景2：用于对模块内部状态的检查，例如指针不应该为空、不应该向数据容器放入超过容量的数据、某个操作的结果应该处于特定状态等
- 断言的使用应该限于那些“不应该发生”的错误

这段断言有没有问题？

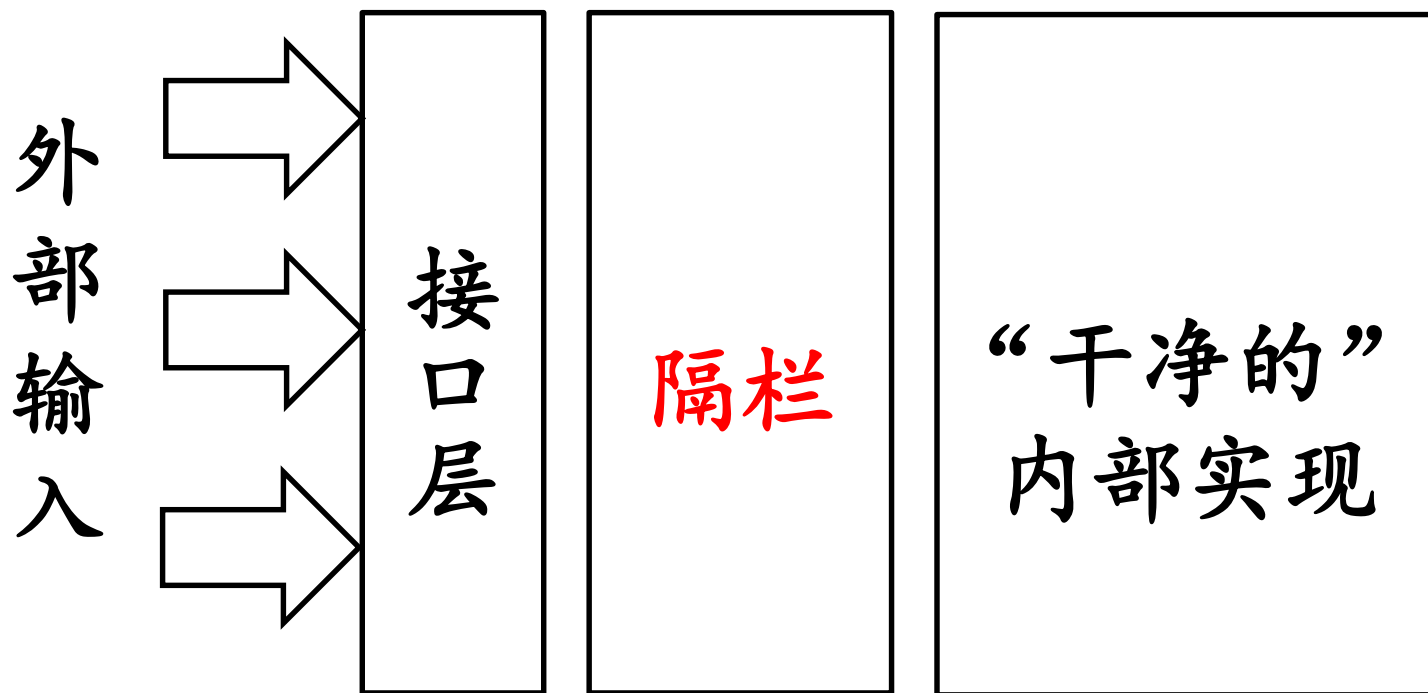
```
while (iteration.hasMoreElements()){  
    Test.ASSERT(iter.nextElement()!=null);  
    Object obj = iter.nextElement();  
    //...  
}
```

断言语句不能有副作用

防御式编程

(DEFENSIVE PROGRAMMING)

原则：子程序应该不因传入错误数据而被破坏，哪怕是由其他子程序产生的错误数据



使用隔栏机制

隔栏如何设计?

- 用户输入：提醒用户重新输入
- 瞬态错误(如传感器数据采集、通讯链路)
 - 返回中立值
 - 换用下一个正确数据
 - 换用最接近的合法值
 - 记录警告信息
 - ...

在不同的层次之间的错误传递

- 返回错误码

- 异常

注意：错误码和异常仅应当用于“真正的异常”！

内容提要

- 软件构造概述
- 构造活动中的设计
- 编码
- 错误处理
- 单元测试
- 集成
- 软件构造工具

单元测试的价值

- 根据缺陷成本递增规律，发现缺陷的时间越晚，修复成本就越高
 - 单元测试比系统测试更靠近开发阶段，能够更快的发现软件中存在的错误，从而避免在后续阶段的浪费。
- 单元测试隔离了代码单元和其它部分的依赖
 - 在测试执行上所用的时间和所需的信息更少，对错误定位的速度也更快

单元测试的一些要点

- **单元测试是开发者测试**
 - 开发人员能够更容易地理解要测试的目标
 - 发现问题时能够更快的修复
- **单元测试应尽可能自动化**
 - 采用自动化的方式，有助于降低单元测试的总体运行成本，提高开发人员运行单元测试的频率
- **单元测试应该隔离依赖**
 - 如果被依赖的部分发生变化，很可能会导致需要测试的代码单元出现错误的行为

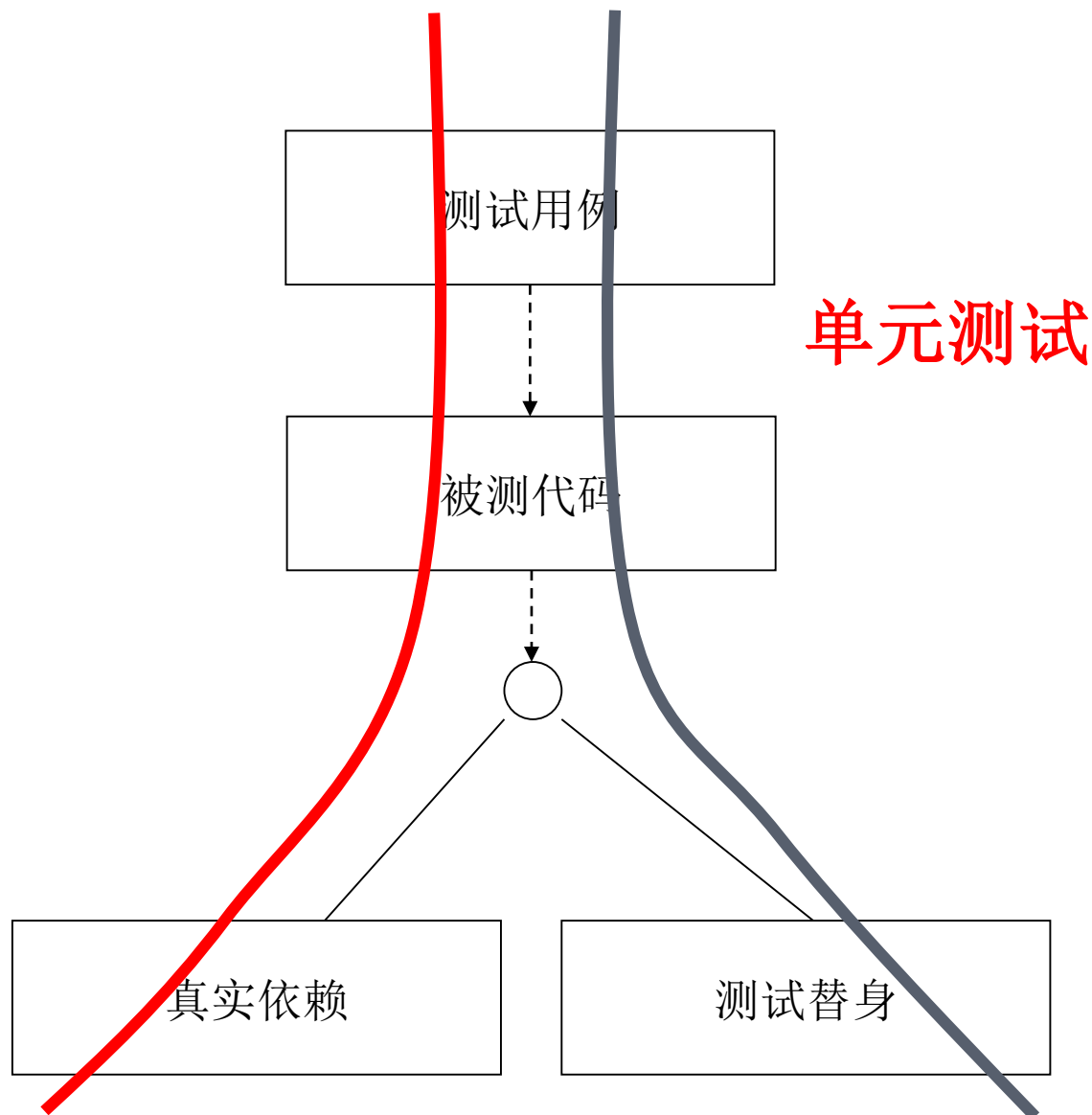
自动化单元测试框架XUNIT

```
Public class TestString {  
  
    @Test  
    Public void test_compare_to_same_string_should_be_zero() {  
        String s_1 = "abc";  
        String s_2 = "abc";  
        assertEquals(0, s_1.compareTo(s_2));  
    }  
  
    @Test  
    Public void test_compare_to_smaller_string_should_be_positive() {  
        String s_1 = "abc";  
        String s_2 = "ab0";  
        assertTrue(s_1.compareTo(s_2) > 0);  
    }  
}
```

四阶段模式

- 建立：建立被测代码的前置条件，从而能够开始进行测试
- 执行：调用被测单元的接口
- 验证：通过断言，确定是否获得了预期的结果，从而判断被测代码的正确性
- 拆卸：拆卸测试夹具，从而将被测目标及环境恢复到测试前的初始状态，避免影响后续的测试

依赖和测试替身



替身类型-1

○ 测试哑元

- 在单元测试中从来不会被真正调用的简单的桩
- 它们存在的目的是使得链接器可以进行链接

○ 测试桩

- 能够在测试用例的指示下，按照要求在特定时刻返回特定的值，从而使得被测代码能够跳转到相应的路径，或者执行特定的行为

○ 测试间谍

- 测试间谍除了能够像测试桩一样返回特定的值，还能够捕获由被测代码传出的参数，以便测试可以校验被测代码所调用的服务或者传出的参数是否正确

替身类型-2

○ 仿制对象

- 仿制对象比测试间谍更为智能
- 除了能像测试间谍一样返回特定的值或者捕获被测代码传出的参数，仿制对象可以校验函数是否被调用、调用顺序是否正确以及参数值是否正确等

○ 仿冒对象

- 仿冒对象比较简单，仅仅是为替代的组件提供一个部分实现。

○ 引爆仿冒

- 引爆仿冒用来模拟“不应当被调用”的场景
- 当被测代码调用到该仿冒对象所仿冒的服务时，仿冒对象将立即触发测试失败。

测试替身示例

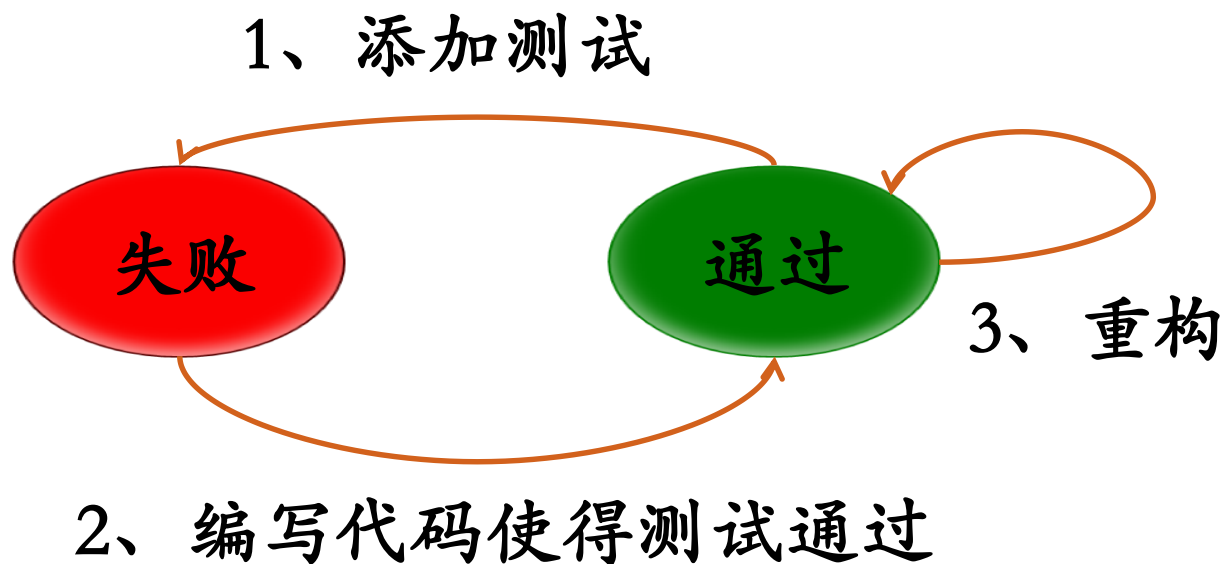
```
public interface ExchangeRate {  
    double getRate(String inputCurrency, String outputCurrency);  
}  
  
public class TestCurrency {  
    @Test  
    public void testToEuros() {  
        Currency one_hundred_yuan = new Currency(100, "CNY");  
        Currency ten_euro = new Currency(10, "EUR");  
  
        ExchangeRate mock = EasyMock.createMock(ExchangeRate.class);  
        EasyMock.expect(mock.getRate("CNY", "EUR")).andReturn(0.1);  
        EasyMock.replay(mock);  
        Currency exchange_result = one_hundred_yuan.toEuros(mock);  
        assertEquals(ten_euro, exchange_result);  
    }  
}
```


测试先行

在设计和编码开始之前就编写测试，这是因为

1. 具备可行性
2. 确保代码的可测试性
3. 使得开发者聚焦于代码所应该提供的行为，而不是具体的实现方式
4. 提高开发效率
5. 方便进度度量

测试驱动开发



从严格意义上说，测试驱动开发并不是一种测试活动，而是一种软件设计活动。

内容提要

- 软件构造概述
- 构造活动中的设计
- 编码
- 错误处理
- 单元测试
- 集成
- 软件构造工具

“大爆炸”(BIG-BANG)集成

- 一种在产品发行阶段一次性进行集成的方式
- 缺点
 - 工作量大、耗时长
 - 时间弹性弱、反馈困难且时间长
 - 涉及模块过多，难以进行错误诊断和定位

“大爆炸” 集成的问题

○ 接口问题

- 模块间接口设计的不一致往往是在基于错误的假设进行了数月的开发之后，在集成阶段才能发现
- 这种外部接口的不一致可能会严重的影响到已经完成的模块实现

○ 模块内部质量问题

- 未在模块测试中发现的内部实现错误会影响集成的进度：本来集成仅仅需要解决模块间的协作问题，而这时还不得不解决模块内部的问题
- 由于集成阶段涉及到的模块数量较多，从而进一步加大问题的解决难度

增量集成

- 软件的集成并非一次性完成，而是通过多个增量，按照一定顺序（例如自底向上）逐步集成为最终的系统
- 例如，在一个大规模的嵌入式系统中
 - 开发者可以首先集成操作系统、硬件平台和硬件适配层
 - 然后增加通信层、应用层的各个模块
 - 最后集成各个应用所对应的操作界面等

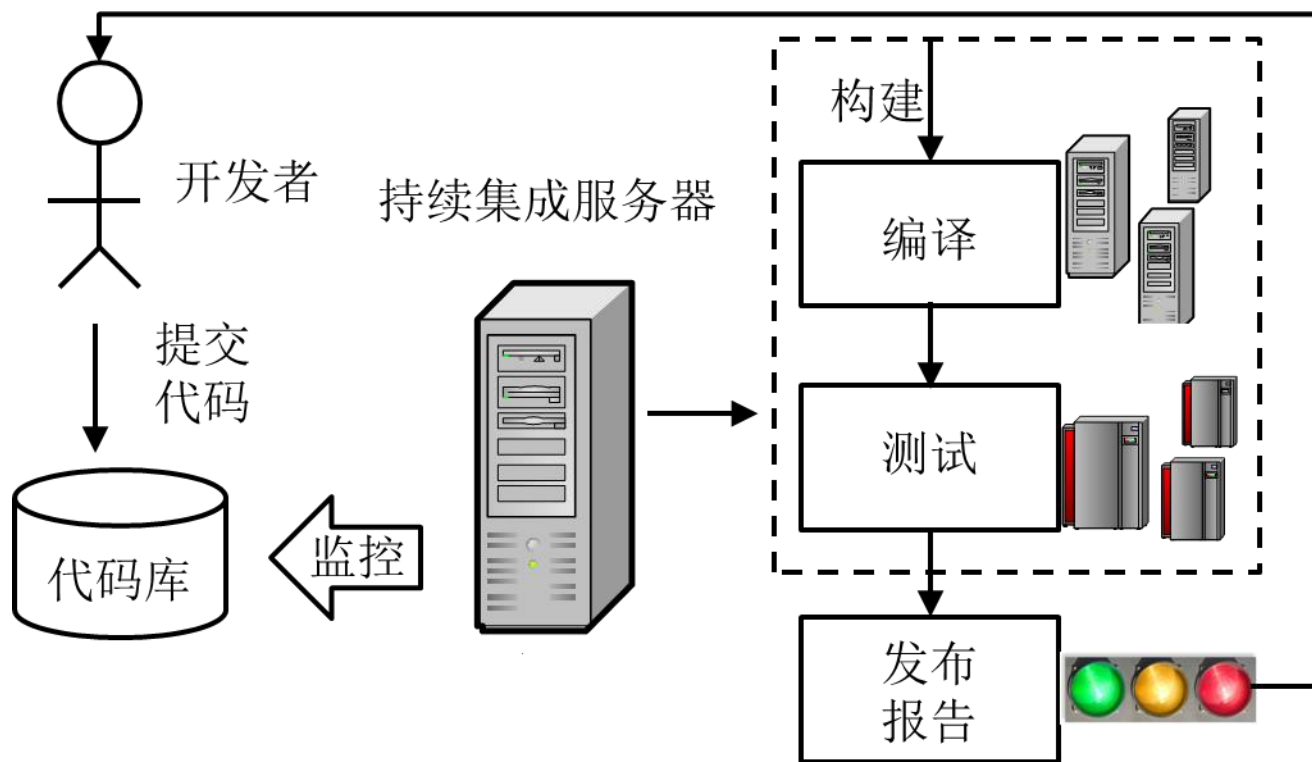
持续集成

- 持续集成是一种软件开发实践，在其中团队成员经常集成他们的工作
 - 每个人至少每天都会有一次集成，从而每天都会有多个集成
 - 每个集成都通过自动构建（包括测试）来验证，以尽快的检测集成错误
- 持续集成的目标：始终保持一个可以工作的系统
- 持续集成要求每次只引入细小变化、缓慢的但是稳健的保持系统增长

反馈是持续集成的核心

- 持续集成本质上是一个反馈系统，以最快的速度发现新编写的代码的问题，避免错误累积
- 保证持续集成有效反馈的方式就是始终保持一个正确的系统基线
- 这样在任何时刻，只要有新的错误发现，可以仅调查最近的代码提交

一个持续集成环境示例



持续集成的工作步骤

- (1)在开始工作前首先检查构建的状态。如果当前构建状态是成功的，基于库中的版本进行新的开发。如果构建是失败的（属于例外情形），修复构建是最高优先级的任务。
- (2)完成一部分开发任务。
- (3)在本地执行和持续集成服务器上相同的构建过程。这一步是为了在提交代码前发现问题，尽量避免造成持续集成服务器的构建失败。
- (4)如果构建失败，回到第(2)步。如果本地构建成功，检查主线上是否还有其他人已经做了更改。如果其他人有更改，转到第(5)步。否则，转到第(6)步。
- (5)合并其他人的更改，重新运行本地构建。如果失败，回到第(2)步。
- (6)将代码提交到代码库。持续集成服务器发现新的代码提交，开始执行构建过程。
- (7)开发者等待持续集成服务器发布构建状态。如果状态失败，应该及时修复构建。或者如果不能快速修复，回滚本次代码提交。

内容提要

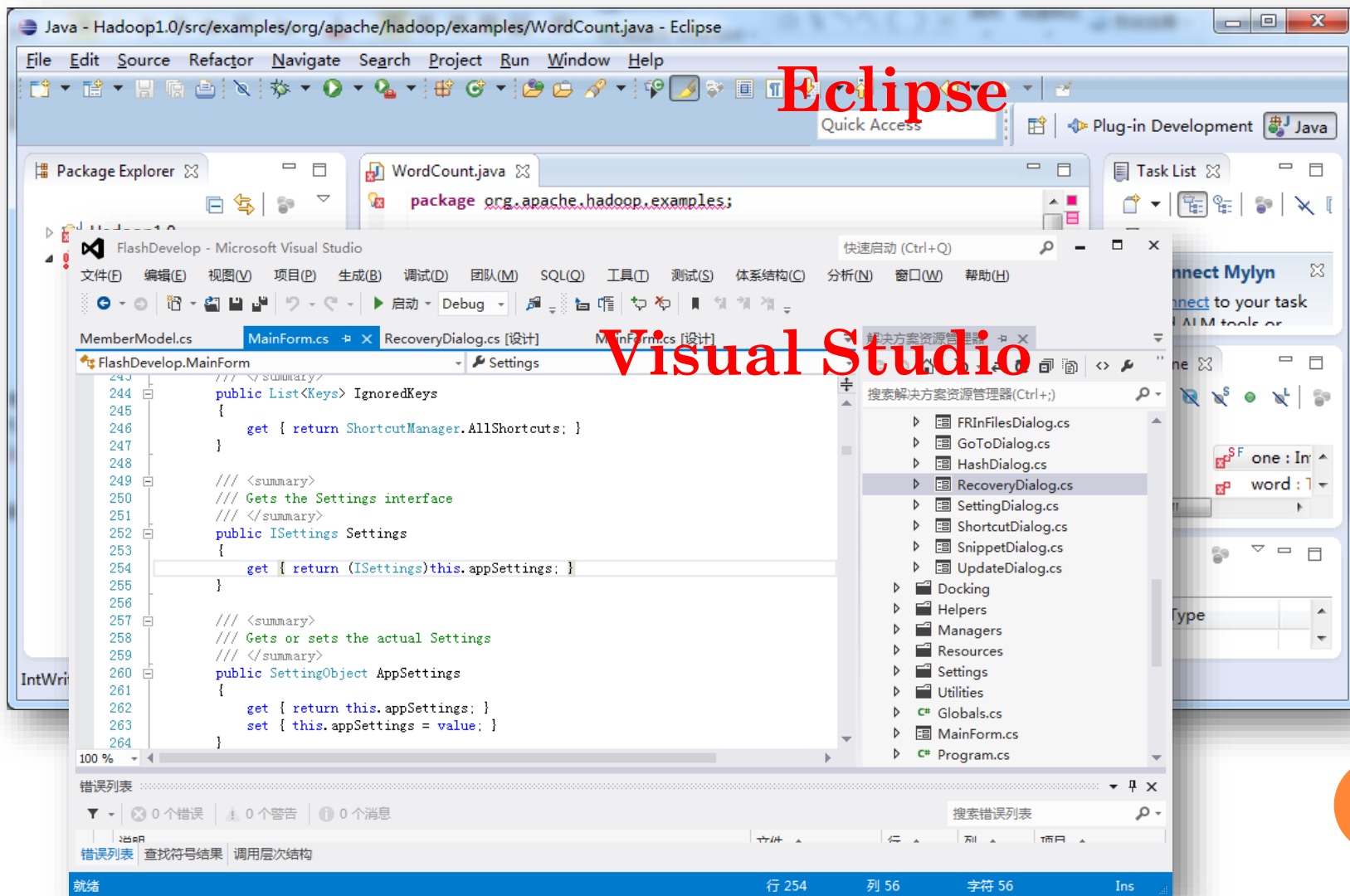
- 软件构造概述
- 构造活动中的设计
- 编码
- 错误处理
- 单元测试
- 集成
- 软件构造工具

集成开发环境的功能

- 代码编辑、自动完成、格式重排...
- 代码导航...
- 重构、逆向工程...
- 质量分析...
- 编译、链接、调试...
- 版本管理、团队协作...

IDE: Integrated Development Environment

常用的IDE



版本管理系统

○ 常用系统：SVN, Git, Clearcase

○ 要求

- 原子化操作：代码提交要么全部成功，要么全部失败
- 基线管理：里程碑版本，可以有效地进行版本的回溯
- 并发支持：多人并发开发支持，例如版本的分支与合并、版本的锁定等

○ 集中式和分布式版本管理系统

- 集中式（服务器保存所有版本）：Clearcase, CVS, SVN
- 分布式（不区分服务器与客户端）：Mercurial, Git

编译和调试工具

- 编译器、连接器或语言解释器
 - 如g++、java
- 调试器
 - 如GDB、DDD
- 构建工具
 - 如Make、Ant
- 性能调优工具
 - 如gprof

编程工具的选择

- 编程工具最重要的不是功能多寡
- 编程工具应该能够支持程序员的每日工作场景，降低单调乏味的细节性工作
- 有效的软件开发团队和个人应该选择或定制最适合自己的上下文的工具环境

高级软件工程

软件构造

The End