

L6 质量管理(3)

——初探测试驱动开发与解耦设计

大纲

- 什么是TDD
- 设计可测试的软件
- TDD的相关技术

大纲

- **什么是TDD**
 - TDD概述
 - Junit简介
 - 整洁的单元测试
- 设计可测试的软件
- TDD的相关技术

什么是TDD

- **TDD概述**
- JUnit简介
- 整洁的单元测试

什么是TDD

- 先写单元测试用例，再写代码
- 由测试来决定需要什么样的代码
- 由程序员编写和维护完整的测试用例集
- 仅当代码有了相应的测试代码，该代码才能作为成品代码

TDD与自动测试技术

- 测试驱动开发可以达到自动测试的效果，但**本质上测试驱动的开发并不是自动测试技术。**
- 测试驱动开发本质上可以被看成是一种**设计的技术。**
- 自动化验收测试：在TDD方式创建了足够的单元测试的基础上对系统交付的进一步保护
 - 证明软件为客户提供他所期望的价值
 - 对大规模修改进行保护
 - 更好地发现用户场景中的缺陷

演示一

编写一个能够执行加法和减法的计算器类

- 如何使用这个计算器？
 - 静态类 or 实例化？
- 具体如何进行加减操作？
 - 按数字加 or 累加 or 计算算式？
-

演示二

- 账单打印模块的重构与自动化单元测试
 - 代码评审及根据评审识别出来的问题重构代码
 - 过长的代码
 - 职责不清晰
 - 有switch/case语句
 - ...
 - 在测试保障下重构代码带来安全感

演示二：业务背景

- 计算影片租赁费用以及所获得的积分
- 租赁费用规则
 - 普通影片：不超过2天租金2元，超过2天每天增加1.5元
 - 新片速递：每天3元
 - 儿童影片：不超过3天租金1.5元，超过3天每天增加1.5元
- 积分规则
 - 每部影片租赁基本分1分
 - 如果是新片，且租赁2天或2天以上，则额外积1分

演示二：目标功能

- 为客户打印租赁账单(statement), 形如:

Rental Record for Tom

The Wandering Earth 9.00

Man In Black 5.00

Kung Fu Panda 4.50

Amount owed is 18.50

You earned 4 frequent renter points

演示二：目标功能

- 客户租赁账单
 - 如果 客户名为 Tom
 - The Wandering Earth是新片，租赁 3天
 - Man In Black是普通片，租赁4天
 - Kung Fu Panda是儿童片，租赁5天
 - 那么账单信息应该是：

Rental Record for Tom

\tThe Wandering Earth\t9.00

\tMan In Black\t5.00

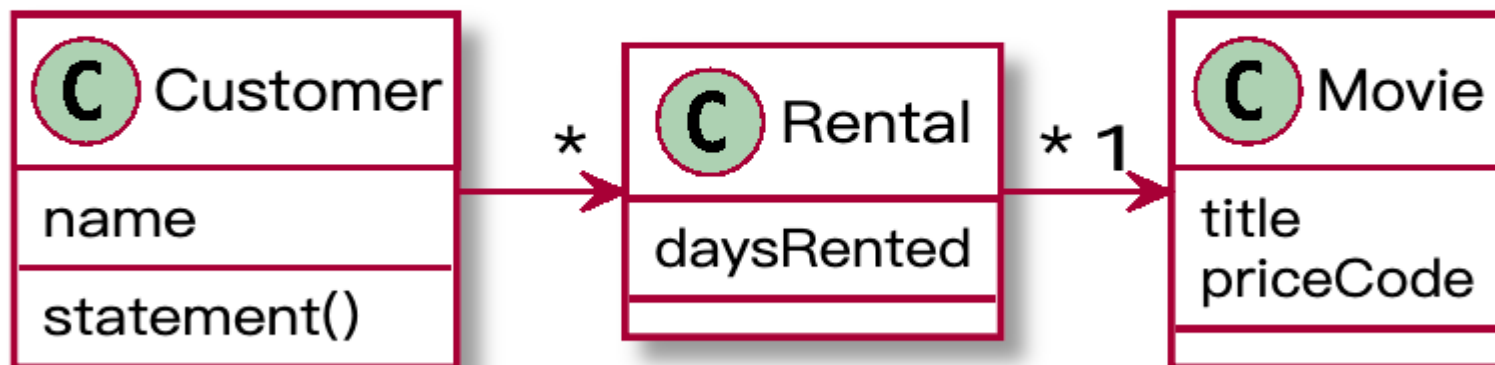
\tKung Fu Panda\t4.50

Amount owned is 18.50

You earned 4 frequent renter points

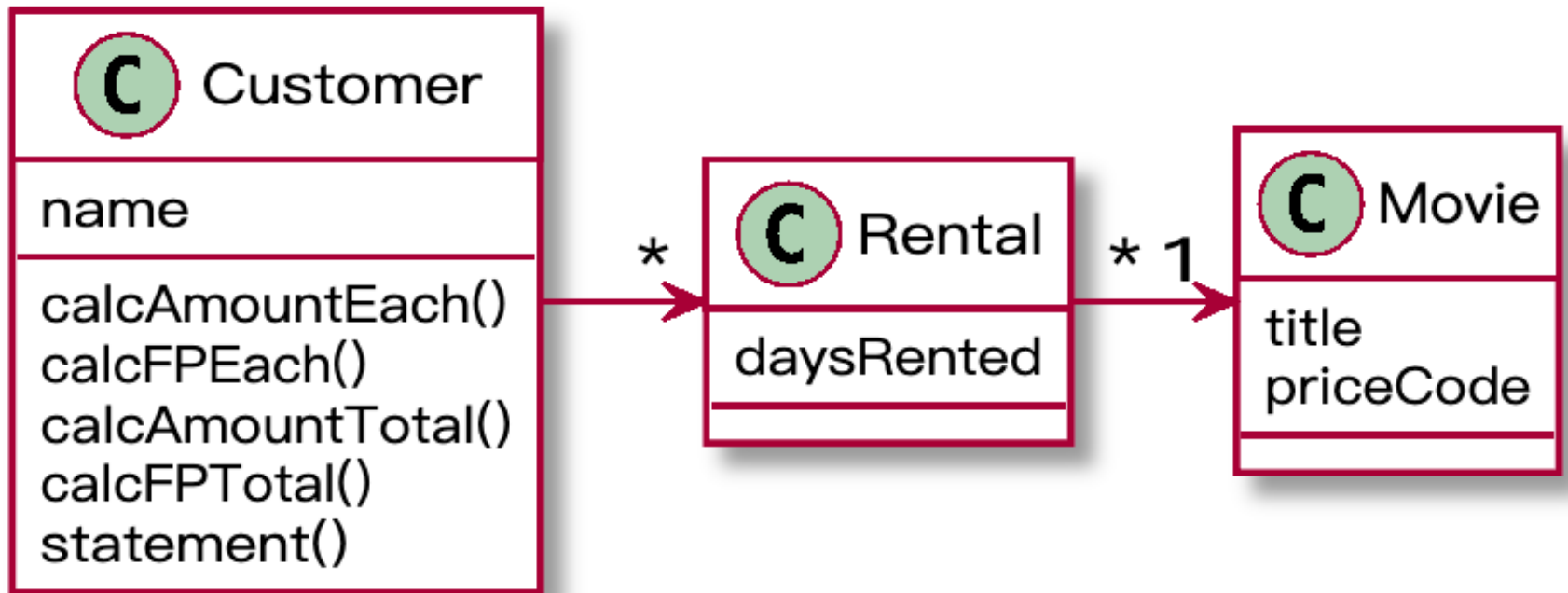
原始设计

Before refactor



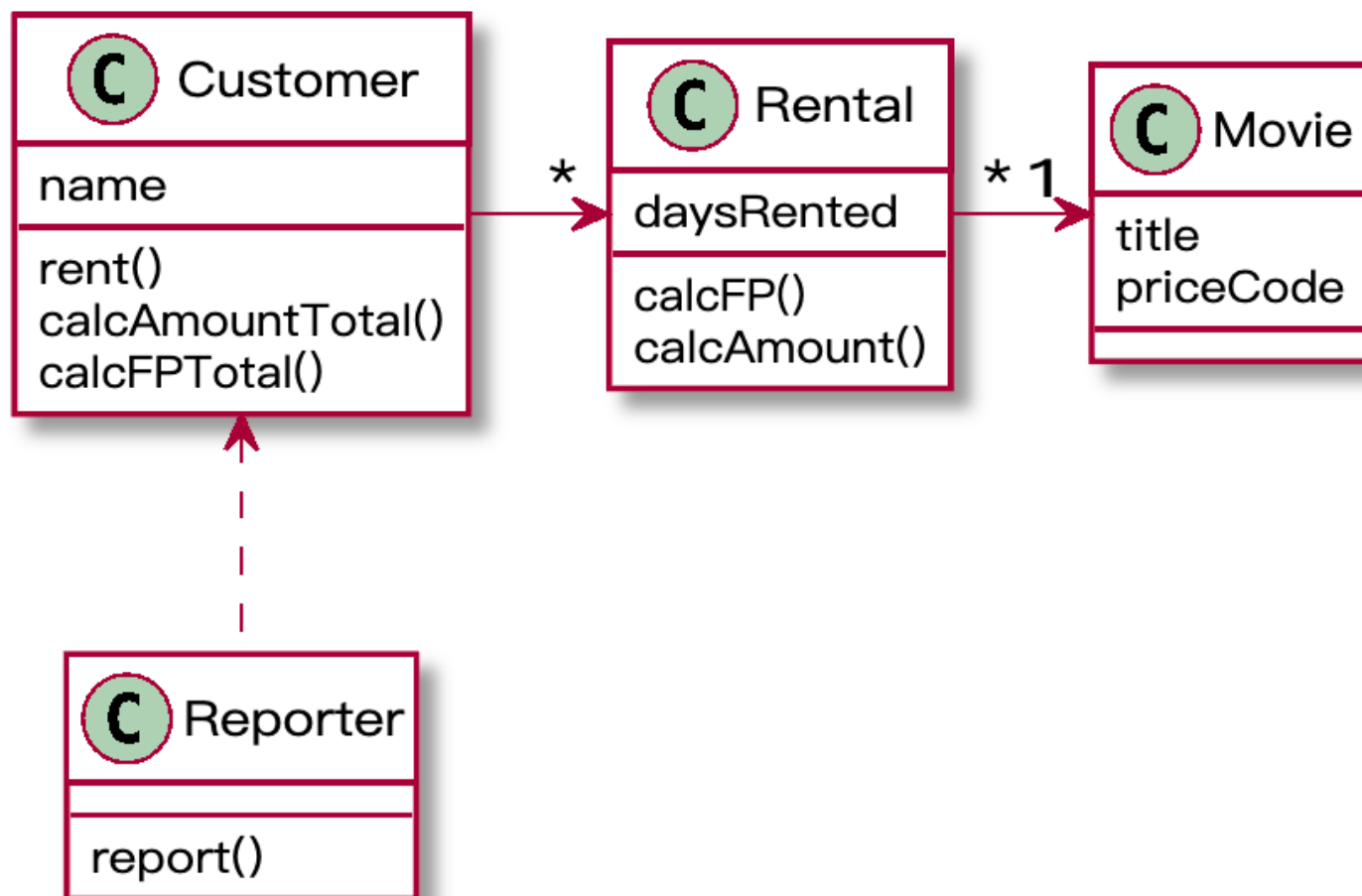
重构步骤1

Refactor phase 1



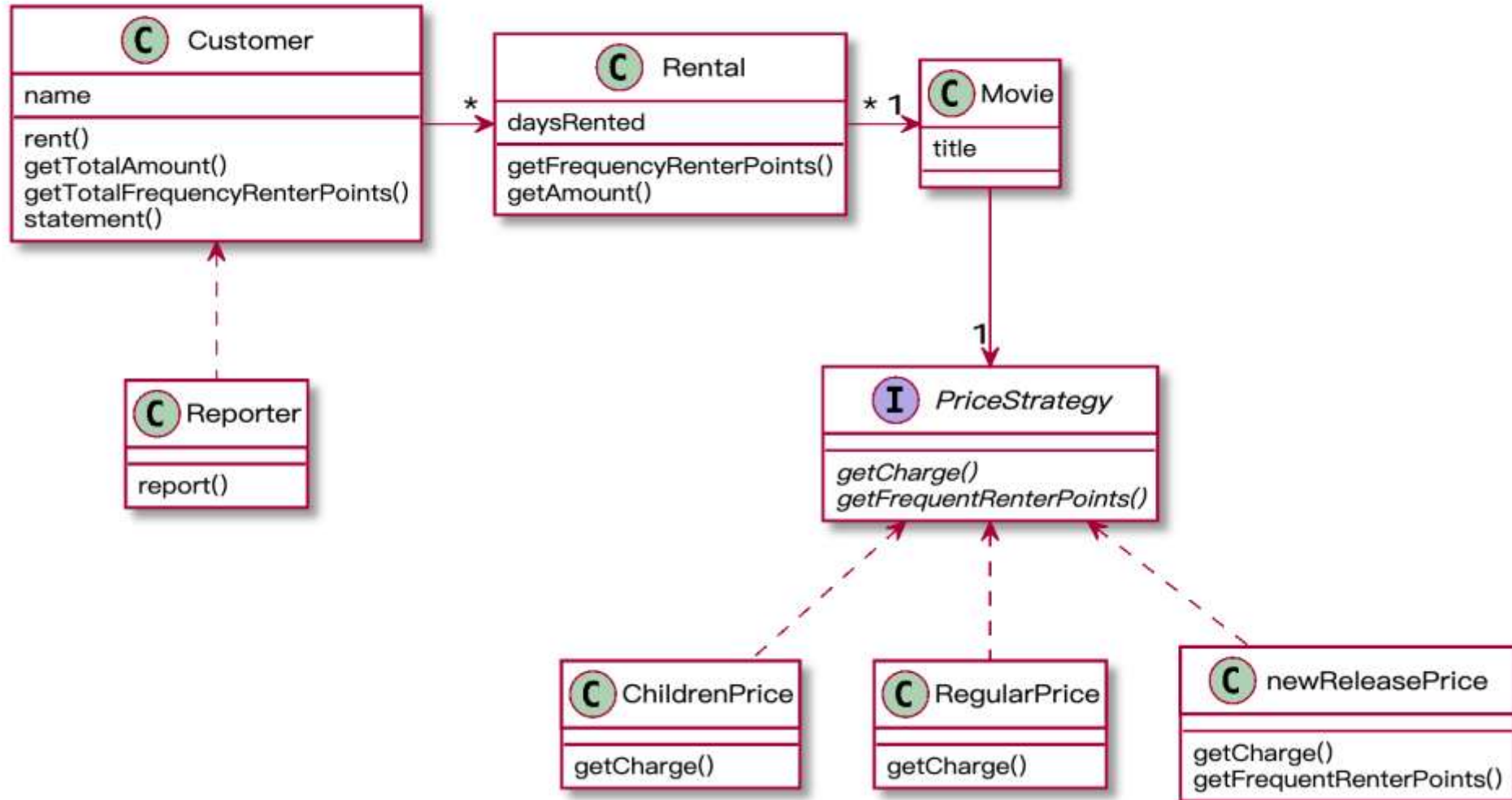
重构步骤2

Refactor phase 2



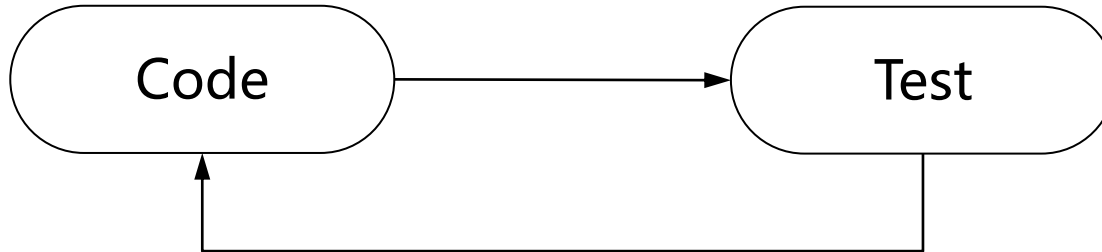
重构步骤3

Apply Strategy Pattern



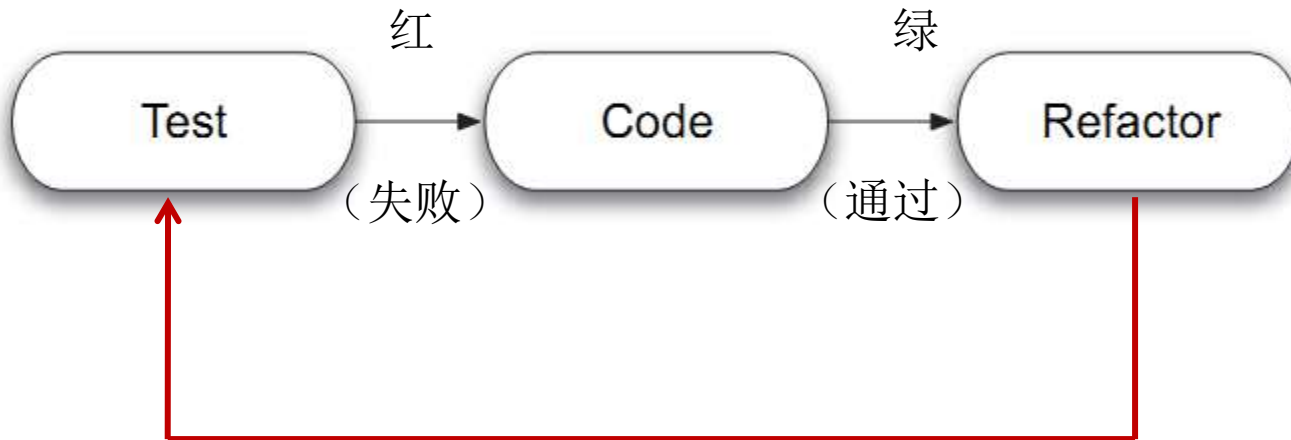
开发过程的变化

- 原来



- 采用测试驱动的开发后：

红/绿/重构



改变习惯

- 通过写测试代码整理设计思路
- 不必害怕错误（包括编译错误的测试错误）
 - 让计算机帮你记住
 - 需要实现哪些类、接口和方法
 - 哪些方法还没有实现或者还存在问题

TDD的优点

- 能够保证编写单元测试
- 使得程序员获得满足感，从而始终如一地坚持编写测试
- 有助于澄清接口和行为的细节
- 可证明、可再现、可自动验证
- 改变事物（改程序、改设计）的信心

什么是TDD

- TDD概述
- **JUnit简介**
- 整洁的单元测试

JUnit历史



- Kent Beck 在上世纪90年代中期在Smalltalk上开发了xUnit的初始版本
- Beck and Gamma(设计模式的第一作者) 在瑞士到华盛顿的航班上完成了JUnit的开发.
- Martin Fowler: “在软件开发领域，从来没有一个软件，受益于它代码行如此之多，而其本身代码行却如此之少”
- JUnit成为Java领域测试启动开发的基础。现在，对于绝大多数的程序设计语言和框架，都有类似的单元测试框架，他们统称为xUnit。

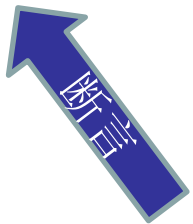



JUnit及其应用

- JUnit是一个测试框架，它的目标是简化单元测试的开发、运行和报告，主要包含以下的功能：
 - 测试类和测试集(suites)
 - 断言(assertions)
 - 测试运行
 - 测试结果报告

测试用例和断言

```
class SpreadsheetTests {  
    @Test  
    public void testCellChangePropagates() {  
        Spreadsheet sheet =  
            new Spreadsheet();  
        sheet.put("A1", "5");  
        sheet.put("A2", "=A1");  
        sheet.put("A1", "10");  
        assertEquals("10", sheet.get("A2"));  
    }  
    //.....  
}
```



测试集

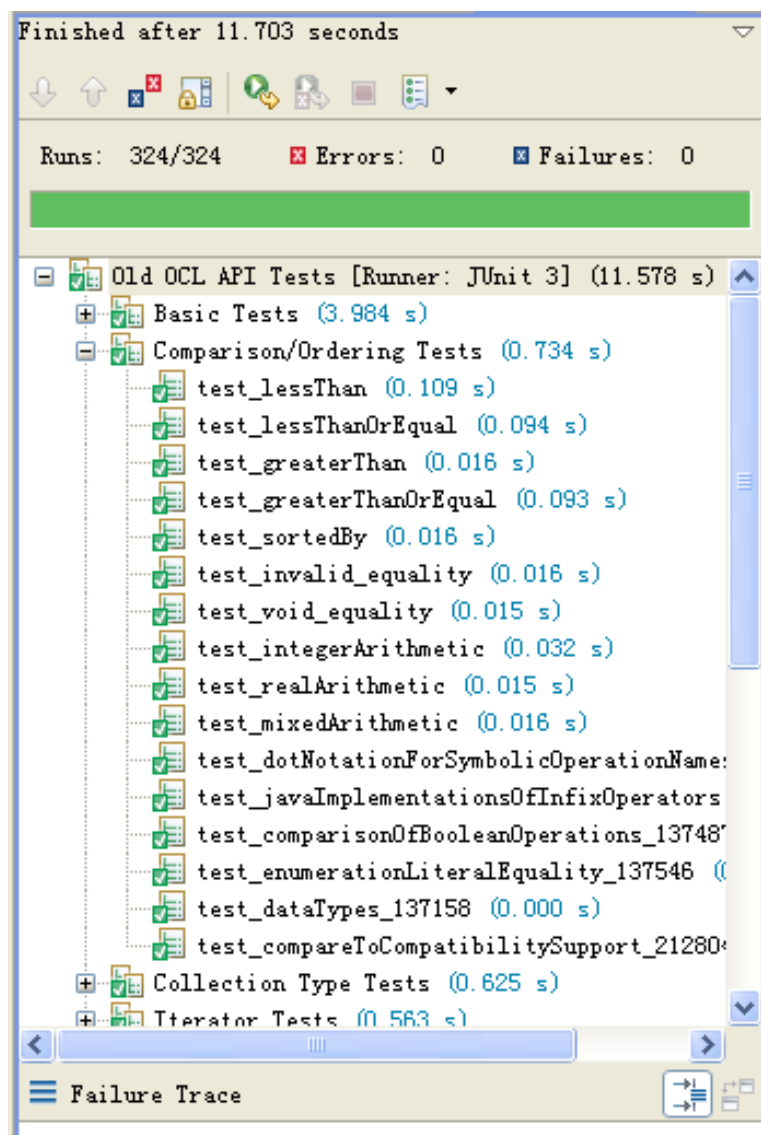
- 多个包含测试用例的类可以组成一个测试集

```
@Suite.SuiteClasses({  
    TestJUnit1.class,  
    TestJUnit2.class  
})  
public class JunitTestSuite {  
}
```



测试集

测试运行及报告：Eclipse IDE



测试运行及报告：VS Code

The screenshot displays the VS Code interface with two main panels. The left panel shows the source code of `CalculatorTest.java` in the `junitdemo` package. The code includes imports for `org.junit.Assert` and `org.junit.Test`, and defines a `testAdd()` method that creates a `Calculator` instance, performs an addition, and asserts the result. The right panel shows the 'Java Test Report' for `junitdemo.CalculatorTest`, indicating that the `testAdd` test passed successfully in 0 seconds.

```
junitdemo > src > test > java > junitdemo > CalculatorTest.java
1  package junitdemo;
2
3  import static org.junit.Assert.assertEquals;
4
5  import org.junit.Test;
6
7  Run Test | Debug Test | ✓
8  public class CalculatorTest {
9      @Test
10     Run Test | Debug Test | ✓
11     public void testAdd(){
12         Calculator calc = new Calculator();
13         int result = calc.add(1,2);
14         assertEquals(3, result);
15     }
16 }
```

Java Test Report

All 1 Passed 1

junitdemo.CalculatorTest

> testAdd Passed 0s

与自动构造工具集成：Maven

```
mvn install
```

```
[INFO]
[INFO] -----
[INFO] Building demo 1.0.0
[INFO] -----
.....
[INFO] -----
[INFO] T E S T S
[INFO] -----
[INFO] Running demo.AllTests
[INFO] Tests run: 33, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 13.252 sec
[INFO] Results :
[INFO] Tests run: 33, Failures: 0, Errors: 0, Skipped: 0
.....
[INFO]
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ demo ---
[INFO] Building jar:demo-1.0.0.jar
[INFO]
.....
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 22.871 s
[INFO] Finished at: 2020-04-04T09:10:38+08:00
[INFO] Final Memory: 20M/334M
[INFO] -----
```

什么是TDD

- TDD概述
- JUnit简介
- **整洁的单元测试**

编写测试用例时的注意事项

- 测试用例也是程序，需要仔细地设计，比如恰当地使用继承、组合关系，提取公共函数等。
- 整洁的测试用例需要遵循以下五条原则
 - 快速 (Fast)
 - 独立 (Independent)
 - 可重复 (Repeatable)
 - 自验证 (Self-Validating)
 - 及时 (Timely)
- 简称为FIRST

编写测试用例时的注意事项

- 确保测试没有副作用，也就是说测试用例运行前后系统的状态保持不变
 - 确保测试用例间没有依赖关系，每一个测试用例都能够**独立**运行
 - 先后多次运行不会产生相互影响，**可重复**执行
 - 例如
 - 测试数据入库，那么入库完成后要恢复测试数据库的原有状态

大纲

- 什么是TDD
- **设计可测试的软件**
 - 什么样的软件是可测试的
 - Mock Object
 - 依赖注入
- TDD的相关技术

设计可测试的软件

- 什么样的软件是可测试的
- Mock Object
- 依赖注入

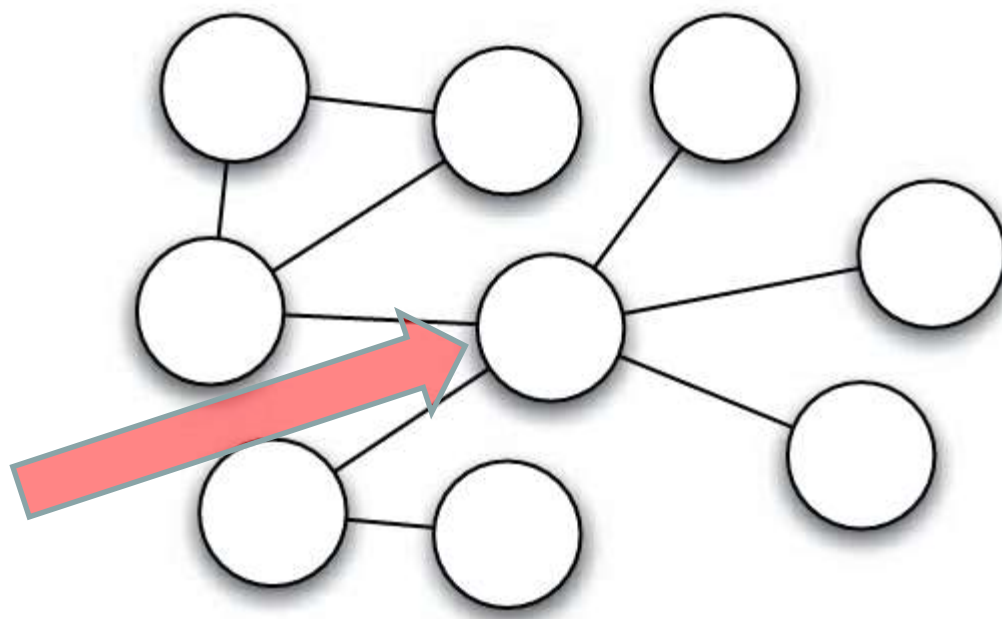
可测试的代码

- TDD的支持工具并不复杂，但让一段代码能够支持单元测试有**其他的挑战**
- 例如
 1. 待测试代码中调用了随机数发生器。
 2. 待测试代码中调用的控制外部设备的API。
 3. 测试一个很长的流程中的末端环节。
- 如果待测试代码中有我们无法控制的部分，我们不能预测结果[1,2]
- 如果一个组件对其他部分的依赖度很高，初始化该组件就会非常复杂，导致测试代码非常冗长 [3]

为测试而设计

- 为了能够测试一个组件，我们需要对该组件以及其依赖的组件的输入进行控制，并能够预测他们的输出。

耦合了无法I/O控制的组件，或者过多**耦合**了其他组件，都会导致难以写单元测试。



为测试而设计

- 保持**较低的耦合度**
 - 这不止是为了测试，实际上较低的耦合度是度量系统质量的一个重要的指标。
 - 在面向对象的方法中，**使用接口**是一种解耦的方法。
- 如果你的软件很难或者无法做单元测试，那往往意味着设计不良

为测试而设计

- 分离接口和实现
 - 识别出需要解耦的部分，将相应的功能抽象到一个接口中。
 - 提供一个或多个生产系统需要的针对该接口的实现。
 - 在测试程序中使用该接口针对测试的一个实现，这个实现通常称为Mock Objects

设计可测试的软件

- 什么样的软件是可测试的
- **Mock Object**
- 依赖注入

为测试而设计：Mock Objects

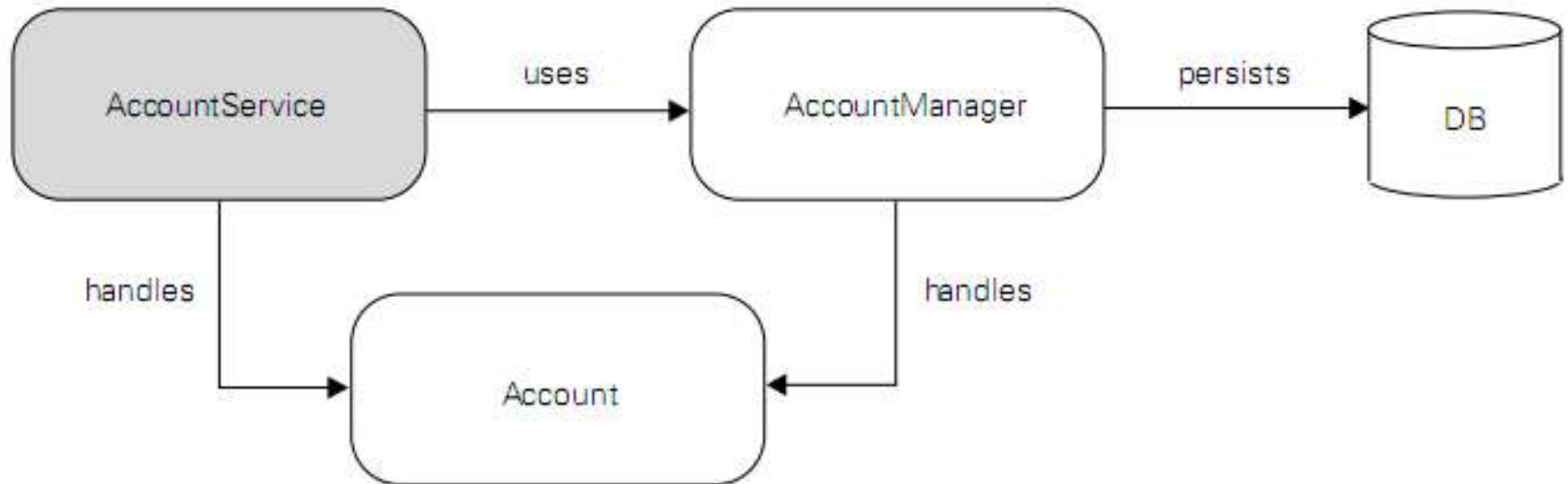
- 有时代码中会需要依赖一些资源，这些资源在测试的时候是无法获得的：
 - 外部系统或者生产数据库
 - 硬件设备
- Mock对象可以认为是工厂模式的一种应用：它可以认为是一个桩(Stub)，用户模拟返回一次调用期望返回的值。

什么是Mock对象

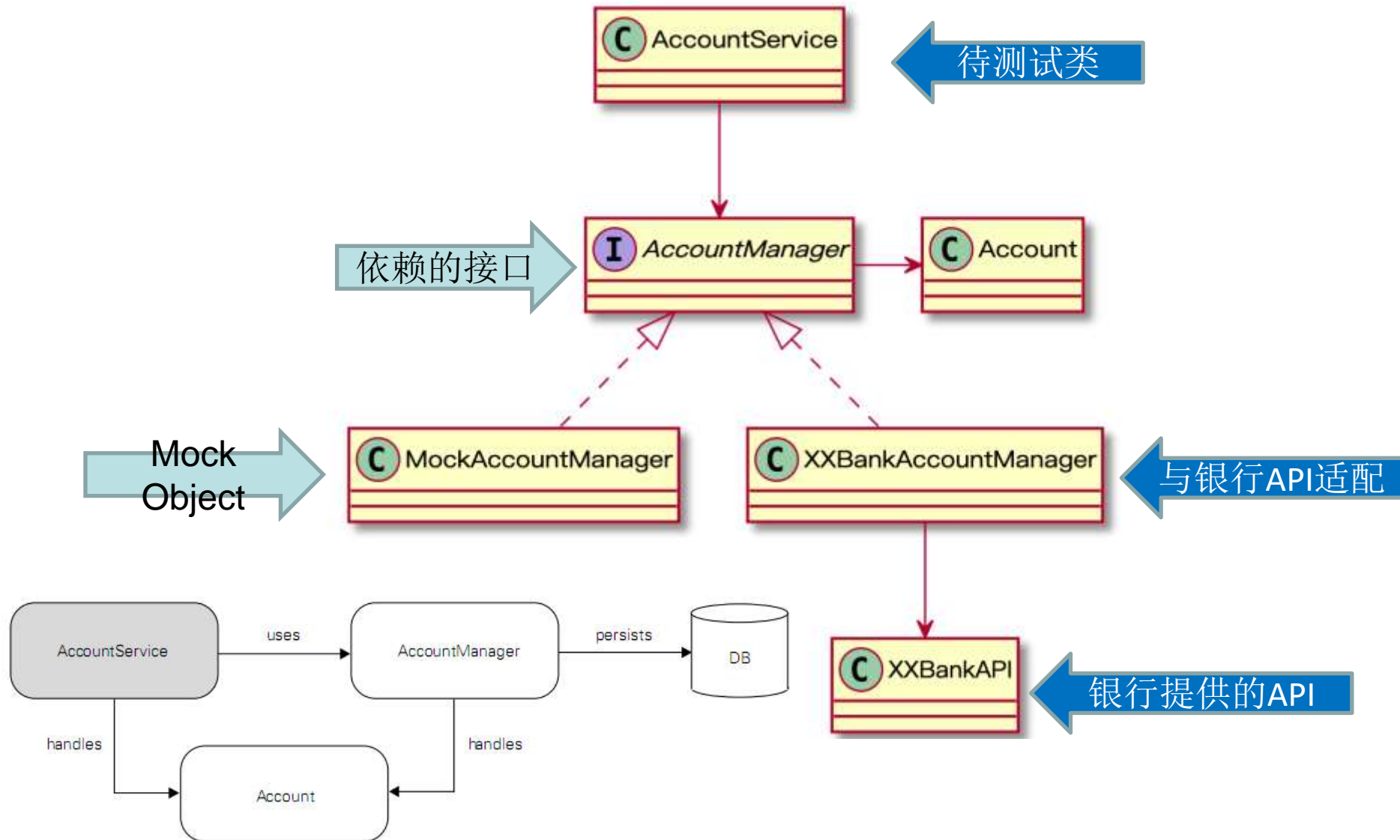
- 定义
 - Mock对象是模拟创建的一个对象，用于替代代码依赖的对象。代码可以调用模拟对象上的方法，这些方法将返回为测试而设置的结果。
- Mock对象是用来替代代码中的部分逻辑，可以将不可控的代码从待测试逻辑中剥离。

Mock Object: 一个简单的示例

- 使用银行提供的接口完成账户间的转账操作



Mock Object: 一个简单的示例



待测试的类

```
public class AccountService
{
    private AccountManager accountManager;

    public void setAccountManager(AccountManager manager)
    {
        this.accountManager = manager;
    }

    public void transfer(String senderId, String beneficiaryId,
        long amount)
    {
        Account sender =
            this.accountManager.findAccountForUser(senderId);
        Account beneficiary =
            this.accountManager.findAccountForUser(beneficiaryId);

        sender.debit(amount);
        beneficiary.credit(amount);

        this.accountManager.updateAccount(sender);
        this.accountManager.updateAccount(beneficiary);
    }
}
```

作为接口的AccountManager

```
package junitbook.fine.tasting;  
  
public interface AccountManager  
{  
    Account findAccountForUser(String userId);  
    void updateAccount(Account account);  
}
```

AccountService的单元测试

```
public class TestAccountService extends TestCase
{
    public void testTransferOk()
    {

        Account senderAccount = new Account("1", 200);
        Account beneficiaryAccount = new Account("2", 100);

        AccountService accountService = new AccountService();

        accountService.transfer("1", "2", 50);

        assertEquals(150, senderAccount.getBalance());
        assertEquals(150, beneficiaryAccount.getBalance());
    }
}
```

1

2

3

MockAccountManager

```
public class MockAccountManager implements AccountManager
{
    private Hashtable accounts = new Hashtable();

    public void addAccount(String userId, Account account)
    {
        this.accounts.put(userId, account);
    }

    public Account findAccountForUser(String userId)
    {
        return (Account) this.accounts.get(userId);
    }

    public void updateAccount(Account account)
    {
        // do nothing
    }
}
```

使用Mock后AccountService的测试用例

```
public class TestAccountService extends TestCase
{
    public void testTransferOk()
    {
        MockAccountManager mockAccountManager =
            new MockAccountManager();
        Account senderAccount = new Account("1", 200);
        Account beneficiaryAccount = new Account("2", 100);

        mockAccountManager.addAccount("1", senderAccount);
        mockAccountManager.addAccount("2", beneficiaryAccount);

        AccountService accountService = new AccountService();
        accountService.setAccountManager(mockAccountManager);

        accountService.transfer("1", "2", 50);

        assertEquals(150, senderAccount.getBalance());
        assertEquals(150, beneficiaryAccount.getBalance());
    }
}
}
```

1

2

3

Mock Object与重构

- 可以使用Mock Object作为重构技术
 - 判断下面代码中的问题

```
public class DefaultAccountManager implements AccountManager
{
    private static final Log LOGGER =
        LoggerFactory.getLog(AccountManager.class);

    public Account findAccountForUser(String userId)
    {
        LOGGER.debug("Getting account for user [" + userId + "]");
        ResourceBundle bundle =
            PropertyResourceBundle.getBundle("technical");
        String sql = bundle.getString("FIND_ACCOUNT_FOR_USER");

        // Some code logic to load a user account using JDBC
        [...]
    }
    [...]
}
```

1

2

Mock Object与重构

- 该抽象出一个能够直接表达设计意图的接口
- 下面的形式既清晰(一致的抽象层次), 又便于测试和扩展。

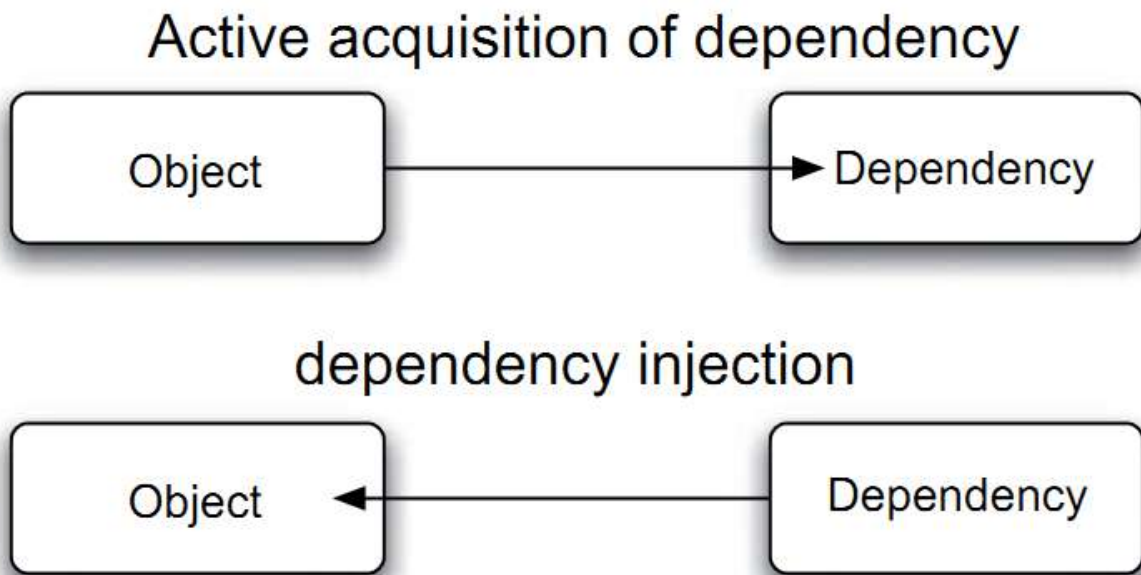
```
public Account findAccountForUser(String userId)
{
    this.logger.debug("Getting account for user ["
        + userId + "]");
    this.configuration.getSQL("FIND_ACCOUNT_FOR_USER");
    // Some code logic to load a user account using JDBC
    [...]
}
```

设计可测试的软件

- 什么样的软件是可测试的
- Mock Object
- **依赖注入**

依赖注入

- 使用接口的情况下，代码中无法直接构造接口的实例，必须由第三方将接口的实例根据应用的场景“注入”，这使得获取实例的方式与传统方式不同：



依赖注入

- 有时也成为控制翻转(Inversion of Control, IoC)
 - 应用IoC模式意味着将创建对象的职责从需要该对象的类中剥离出来，由其他类来提供并组装

```
public void testFindAccountByUser()
{
    MockLog logger = new MockLog();           ❶
    MockConfiguration configuration = new MockConfiguration();
    configuration.setSQL("SELECT * [...]");    ❷

    DefaultAccountManager am =
        new DefaultAccountManager(logger, configuration);  ❸

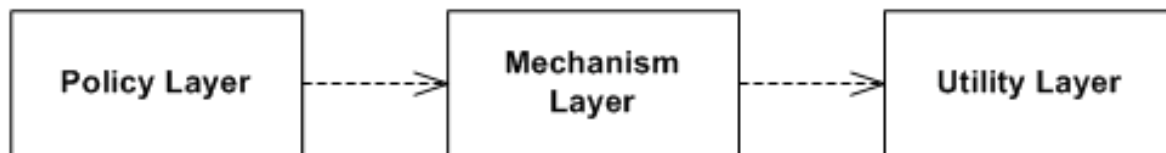
    Account account = am.findAccountForUser("1234");

    // Perform asserts here
    [...]
}
```

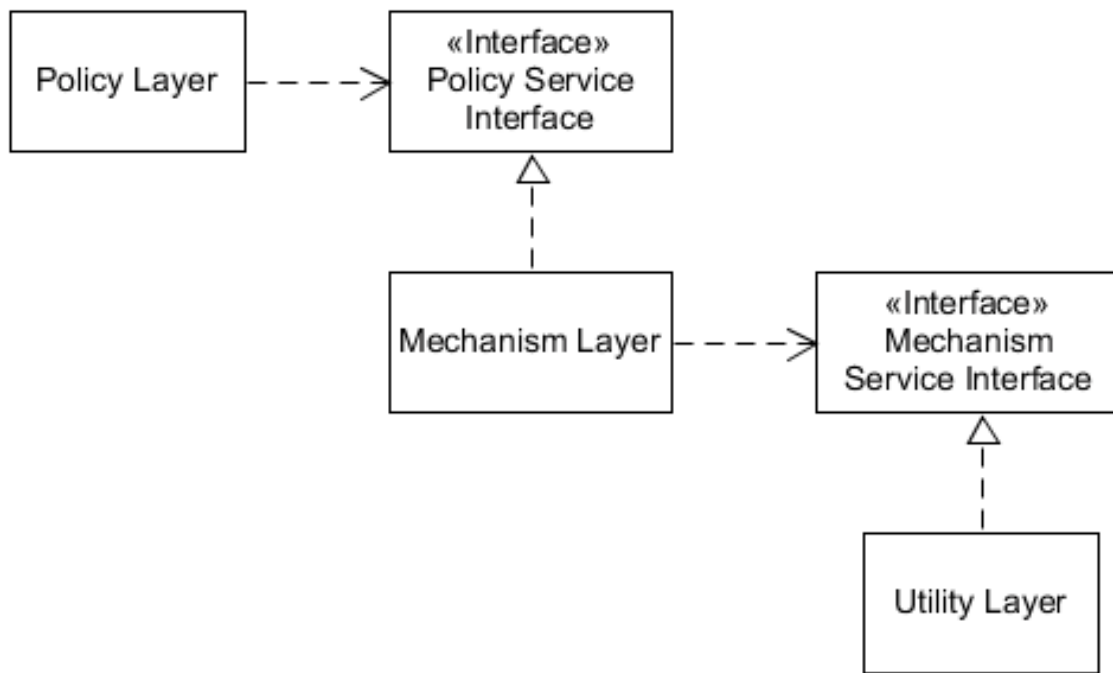
依赖注入

- 基于接口的依赖注入增加了程序的复杂性

传统方式的依赖关系



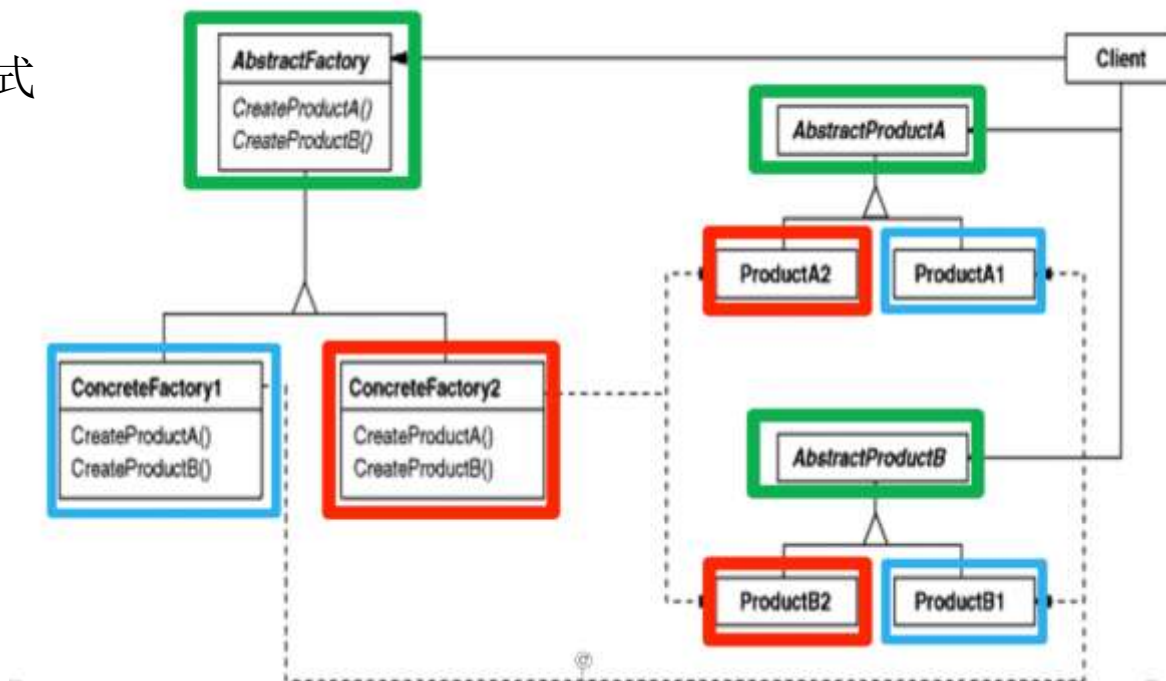
基于接口的、依赖
反转后的依赖关系



依赖注入(实现方式1)

- 可以使用设计模式中的工厂模式(简单抽象工厂或者抽象工厂)来解决这个问题。

抽象工厂是个复杂的模式



依赖注入(实现方式2)

- 使用依赖注入框架
 - Spring Framework
 - Google Guice

演示三

- 使用SpringFramework实现依赖反转，将测试代码与产品代码分离。

Mock Framework

- 通常情况下，为了定义一个Mock对象我们需要定义一个实现了某种接口的类。
 - 对于大量的测试场景，这会比较繁琐
- Mock Framework
 - 允许我们快速地定义Mock Object。
 - 允许开发人员检查Mock Object中方法被调用的顺序和次数。
 - 允许开发人员定义Mock Object上的某个方法被调用后的返回值。

Mock Framework: Easy Mock

- 我们以EasyMock为例来介绍Mock Framework的使用

```
public class SupportDelegationTest {  
  
    private EasyMockSupport support = new EasyMockSupport();  
  
    private Collaborator collaborator;  
  
    private ClassTested classUnderTest;  
  
    @Before  
    public void setup() {  
        classUnderTest = new ClassTested();  
    }  
}
```


Easy Mock

- 定义Mock对象
- 检查方法是否被调用

```
@Test
public void addDocument() {
    collaborator = support.mock(Collaborator.class);
    classUnderTest.setListener(collaborator);
    collaborator.documentAdded("New Document");
    support.replayAll();
    classUnderTest.addDocument("New Document", "content");
    support.verifyAll();
}
```

Easy Mock

- 定义方法调用的返回值：

```
@Test
public void voteForRemovals() {

    IMocksControl ctrl = support.createControl();
    collaborator = ctrl.createMock(Collaborator.class);
    classUnderTest.setListener(collaborator);

    collaborator.documentAdded("Document 1");

    expect(collaborator.voteForRemovals("Document 1")).andReturn((byte) 20);

    collaborator.documentRemoved("Document 1");

    support.replayAll();

    classUnderTest.addDocument("Document 1", "content");
    assertTrue(classUnderTest.removeDocuments("Document 1"));

    support.verifyAll();
}
```

Easy Mock

- 这样，我们就不需要定义一个对Collaborator接口的Mock 类。
- 类似的Mock Framework
 - Mockito
 - JMock

TDD其他相关技术

- **UI测试框架**
- BDD
 - TDD向用户需求方向的延伸

UI 测试

- 用户界面测试是测试产品的用户界面（UI）以确保其符合其规范的过程。
- 相对来说，UI的测试比较有挑战性。
 - UI的变更比较频繁
 - UI框架越来越复杂
 - 往往会比较费时



UI 测试

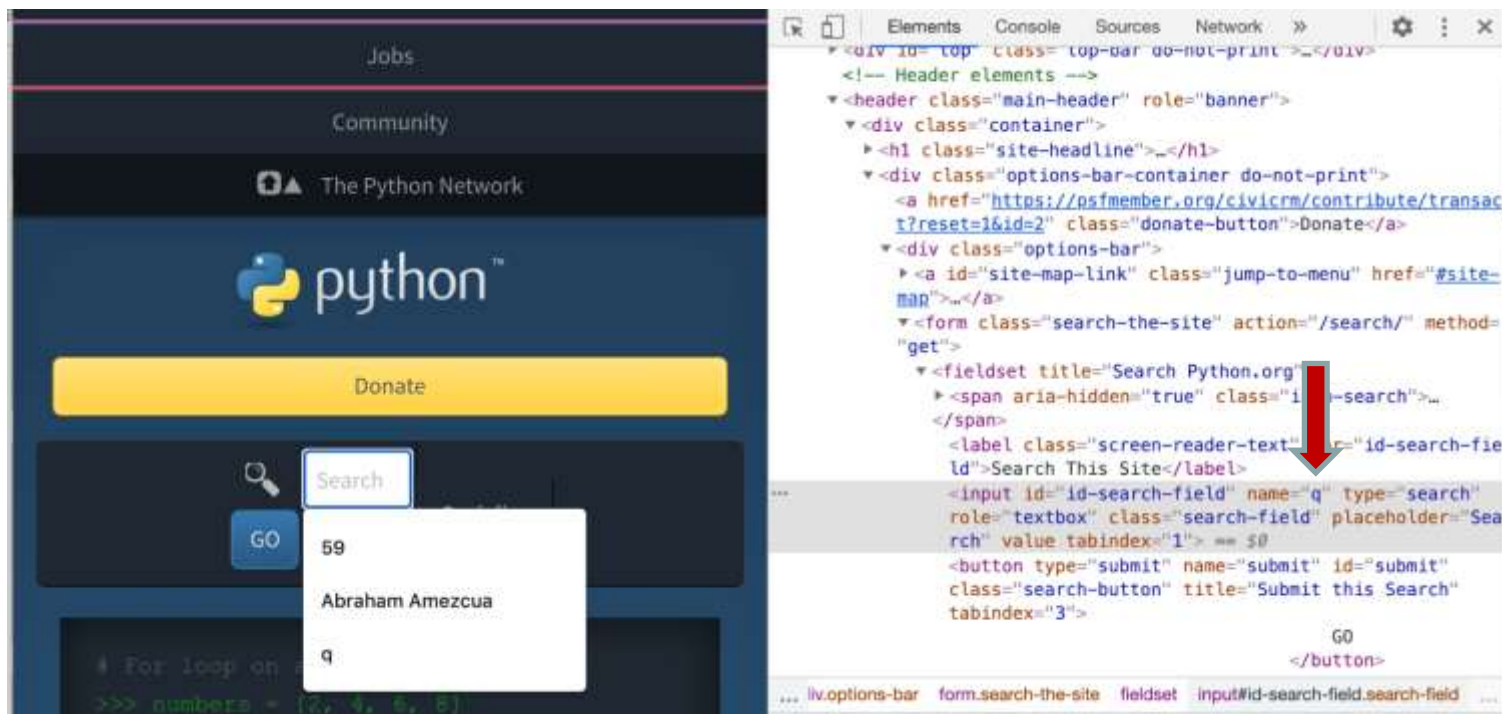
- 由于UI测试本身的挑战，为了最大化可测试代码的比例，在设计中要尽可能将业务逻辑的部分放到业务层，即比较方便做测试的模块中。

UI测试框架: Selenium

- Selenium 是一个用于Web应用程序自动化测试的工具
- Selenium 测试直接在浏览器中运行，就像真实用户所做的一样。Selenium 测试可以在 Windows、Linux 和 Macintosh 上的 Internet Explorer、Chrome 和 Firefox 中运行。

UI测试框架：Selenium

- 使用Selenium模式在<http://python.org>上做一
次查询



UI测试框架：Selenium

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

driver = webdriver.Firefox()
driver.get("http://www.python.org")
assert "Python" in driver.title
elem = driver.find_element_by_name("q")
elem.clear()
elem.send_keys("pycon")
elem.send_keys(Keys.RETURN)
assert "No results found." not in driver.page_source
driver.close()
```

TDD其他相关技术

- Mock Framework
 - 快速定义Mock Object
- UI测试框架
- **BDD**
 - TDD向用户需求方向的延伸

BDD

- 行为驱动的开发
 - 由用户来编写（验收）测试用例

Feature: Pay with credit

Credit can be used for payment. The redeem rate is: every 100 credits equal to 0.1 yuan discount.

Background:

* Create an order with 1 order item of price 20.0

Scenario Outline: Normal pay with credit

Given a user with <ownedCredit> credits

When the user uses <useCredit> credits

Then the user should pay <realAmount> yuan

And the user should have <creditRemain> credits

Examples:

ownedCredit	useCredit	realAmount	creditRemain
200	200	19.8	0
21000	20000	0.0	1000
210	210	19.8	10
21000	21000	0.0	1000
220	220	19.8	20