Perfiientt

测试驱动开发(TDD)

博克软件（杭州）有限公司
Perficient China GDC

日程

- 该课程包括：
  - 单元测试与测试驱动开发基础
  - xUnit系列工具简介
  - 运用模拟对象(Mock Object)测试驱动开发
  - 单元测试的原则、模式及最佳实践
  - 单元测试自动化实践

Perficient

单元测试和测试驱动开发基础

什么是测试?

- 无聊的
- 不断重复的
- 不得不做的
- 容易出错（对人而言）
- 最好由你来做（而不是丢给用户）

无聊的?

Perficient

- 让我们先看一个例子（二分查找）

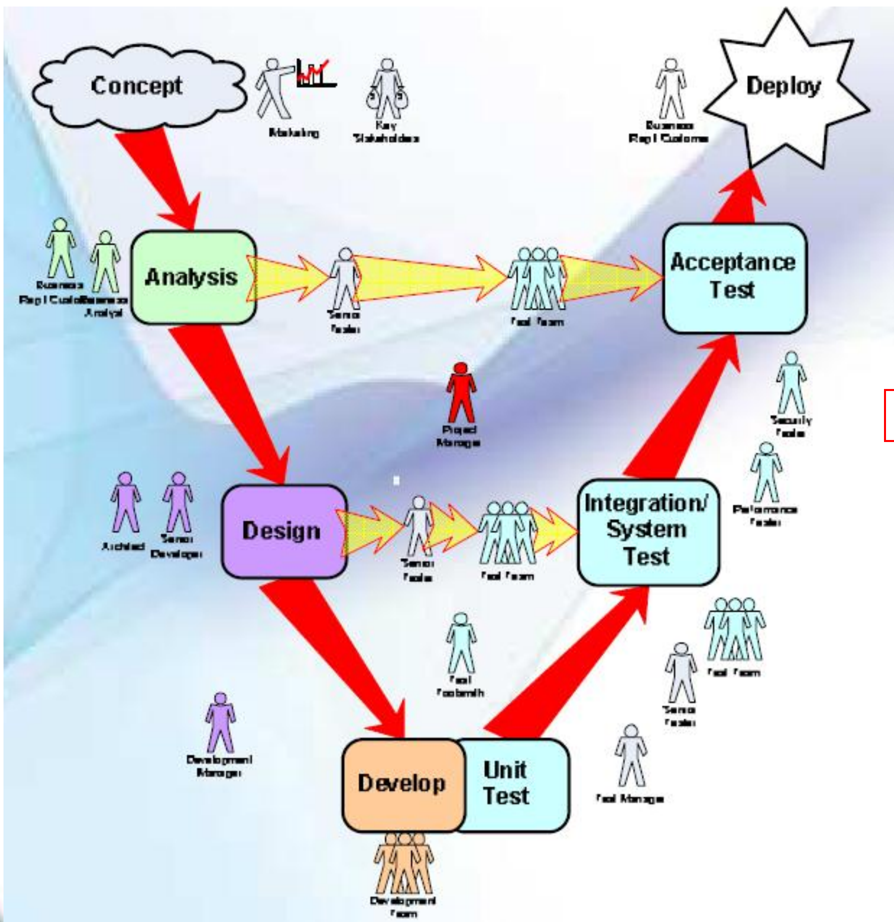
```
public static int buggyBinarySearch(int[] a, int target) {
  int low = 0;
  int high = a.length - 1;
  while (low <= high) {
        int mid = (low + high) / 2;
        int midVal = a[mid];
        if (midVal < target)
        low = mid + 1;
        else if (midVal > target)
          high = mid - 1;
        else
          return mid;
  }
  return -1;
}
```

- 你能在上述代码中发现缺陷么？

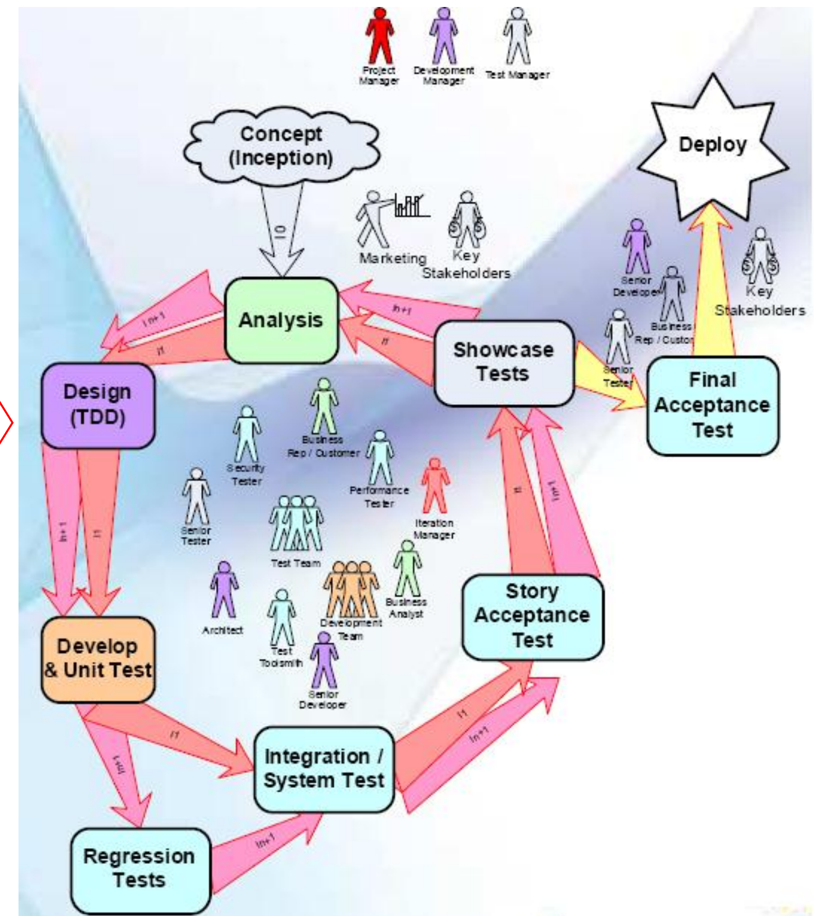
Perficient

测试类别

- 传统测试



- 敏捷测试



什么是单元测试?

- 在计算机编程中，单元测试（又称为模块测试）是针对程序模块（软件设计的最小单位）来进行正确性检验的测试工作。
  - http://zh.wikipedia.org/zh-cn/单元测试
    - ✓ 你所测试的是盖房用的砖，而不是房子本身。
    - ✓ 所谓单元，是指应用系统中最小的可测试的部分。

什么是测试驱动开发？

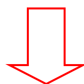
- 从原理上说，它是在编程实现系统功能之前先编写测试。
- 但结果是，它会引导开发人员：
  - 首先考虑“如何使用”
  - 然后才是“如何实现”
- 所以，这不仅仅是测试技术，同样也是**设计**技术。
  - 它能引导程序员开发出易于测试的组件
  - 它能产生易于优化且可适应性更强的组件

测试与设计

Perficient

- 使用测试驱动开发：测试改善设计。
  - 测试将会使用到你在代码中定义的接口，所以你会做出更好用的接口
  - 小的测试更易于编写与维护
  - 你会编写出可测试性高的（模块化的）代码
- 使我们的代码更具可读性，并显著提高设计质量。

```
int mid = (low + high) >>> 1;
```



```
int mid = calculateMidpoint(low, high);
static int calculateMidpoint(int low, int high) {
    return (low + high) >>> 1;
}
```

如何测试驱动开发?

Perficient

- Red
  - 先写一个不能工作的小测试。
- Green
  - 然后迅速让它能工作起来。
- Refactor
  - 接着开始重构，消除代码重复及运用模式提高质量。

Write a failing unit test

Make the test pass

Refactor

Write a failing acceptance test

Write a failing unit test

Make the test pass

Refactor

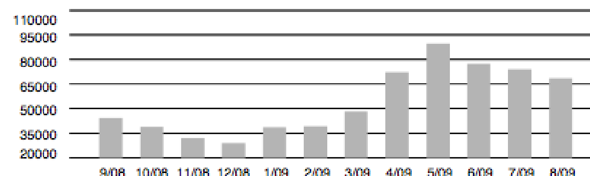
案例：经纪账户(Brokerage Account)

Perficient

- 作为客户，我想买入股票，这样当它们上涨时我可以卖出赚钱。
- 作为客户，我想卖掉股票，这样上涨时我可以赚钱，下跌时我可以避免蒙受更大的损失。

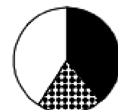
Change in Account Value	This Period	Year to Date
Starting Value	**$ 74,033.79**	**$ 29,224.71**
Cash Value of Purchases & Sales	(4,674.78)	58,595.32
Investments Purchased/Sold	4,674.78	(58,595.32)
Deposits & Withdrawals	(12,000.00)	(807.27)
Dividends & Interest	0.27	1,379.78
Fees & Charges	0.00	(1,103.00)
Transfers	0.00	0.00
Income Reinvested	0.00	(374.55)
Change in Value of Investments	6,786.72	40,501.11
Ending Value on 08/31/2009	**$ 68,820.78**	**$ 68,820.78**
Total Change in Account Value (Totals include Deposits & Withdrawals)	**$ (5,213.01) (7.04)%**	**$ 39,596.07 135.49%**

Account Value ($) Over Last 12 Months



Asset Composition	Market Value	% of Account Assets
Cash	$ 26,534.28	39%
Equities	13,777.50	20%
Options	28,509.00	41%
Total Assets Long	**$ 68,820.78**	
Margin Loan Balance	0.00	
Total Account Value	**$ 68,820.78**	**100%**

Overview



39% Cash
20% Equities
41% Options

Gain or (Loss) Summary

Realized Gain or (Loss) This Period

Short Term	$2,295.87
Long Term	$0.00

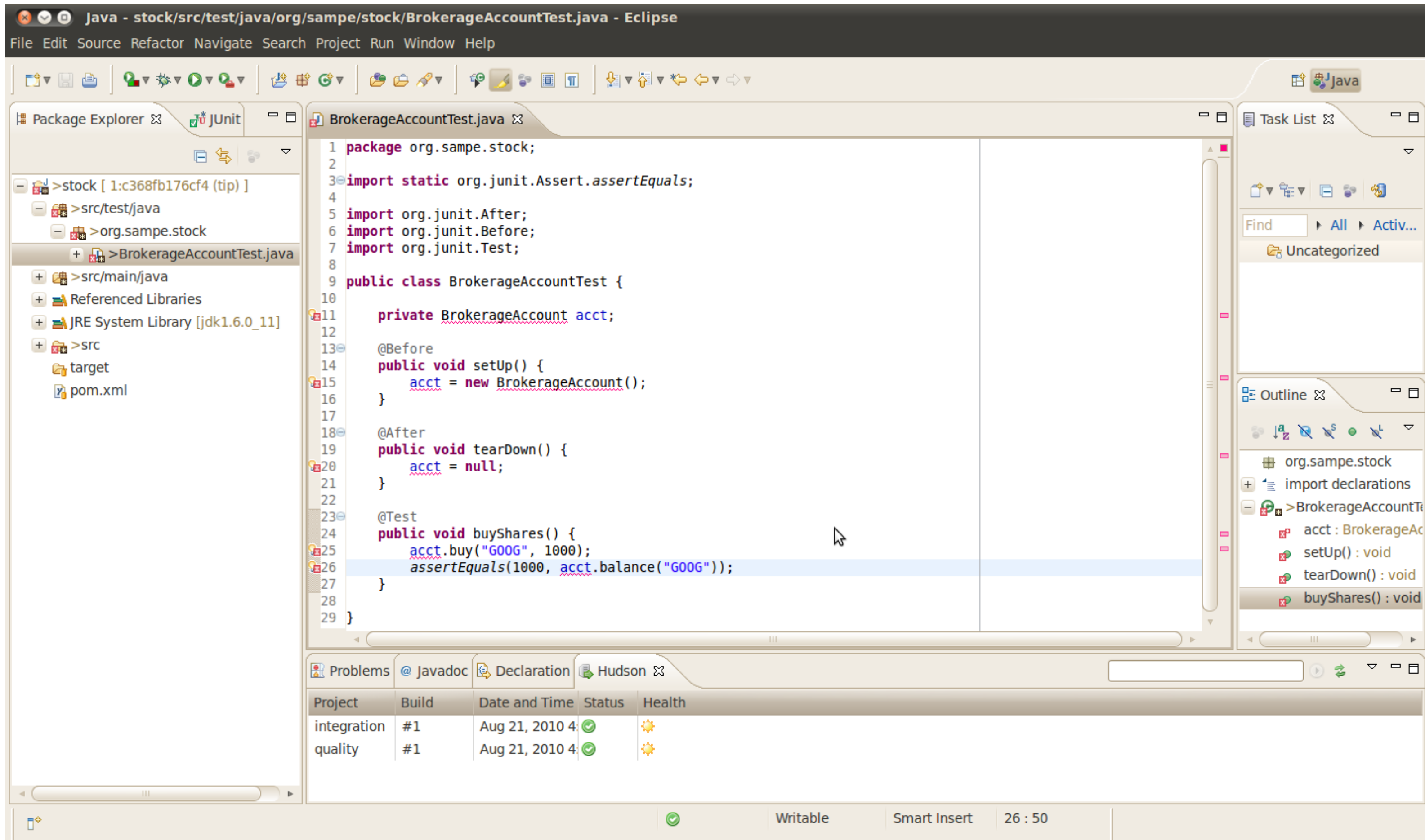
Unrealized Gain or (Loss)

All Investments	$(8.22)

Values may not reflect all of your gains/losses.

Red

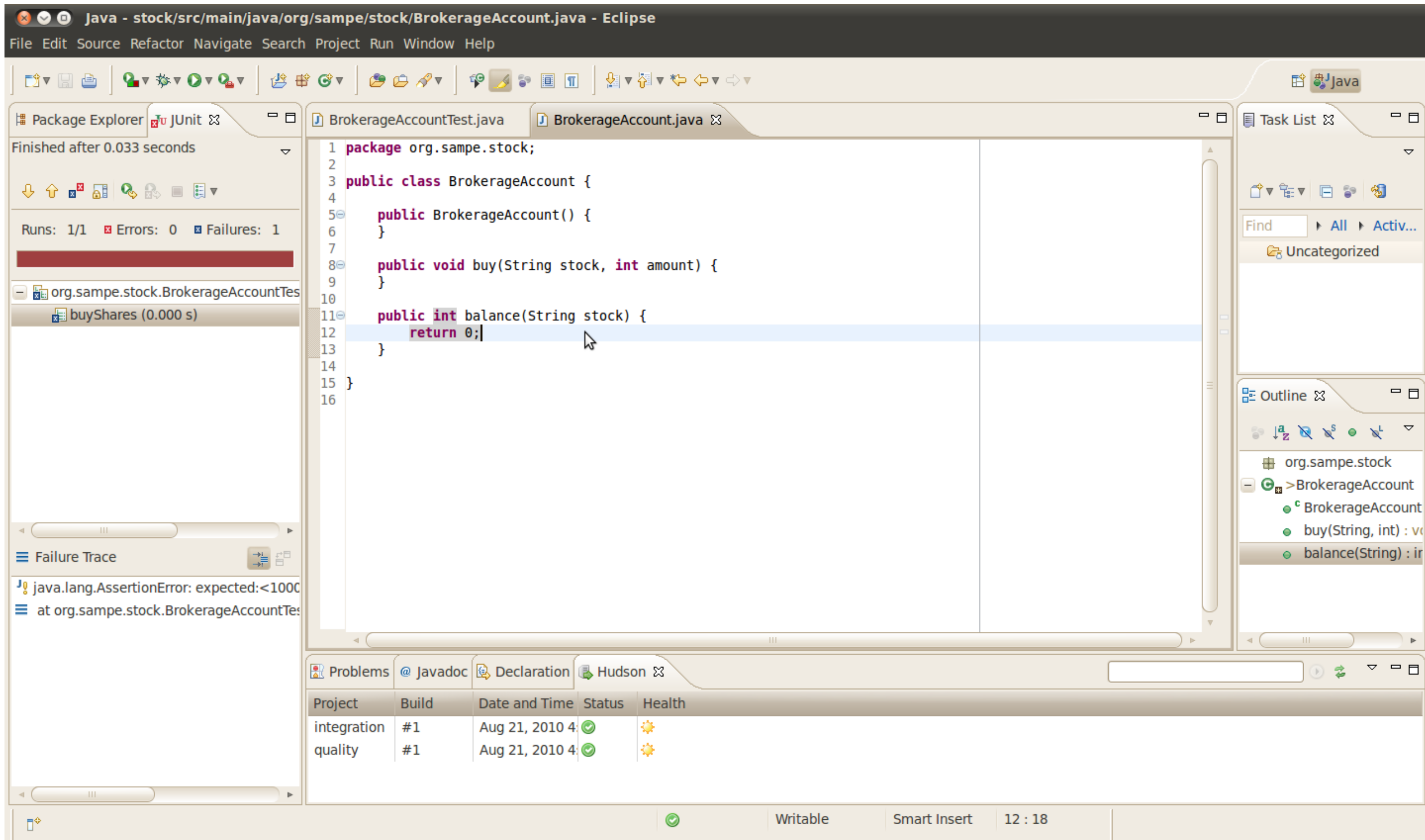
■编译错误



Red

Perficient

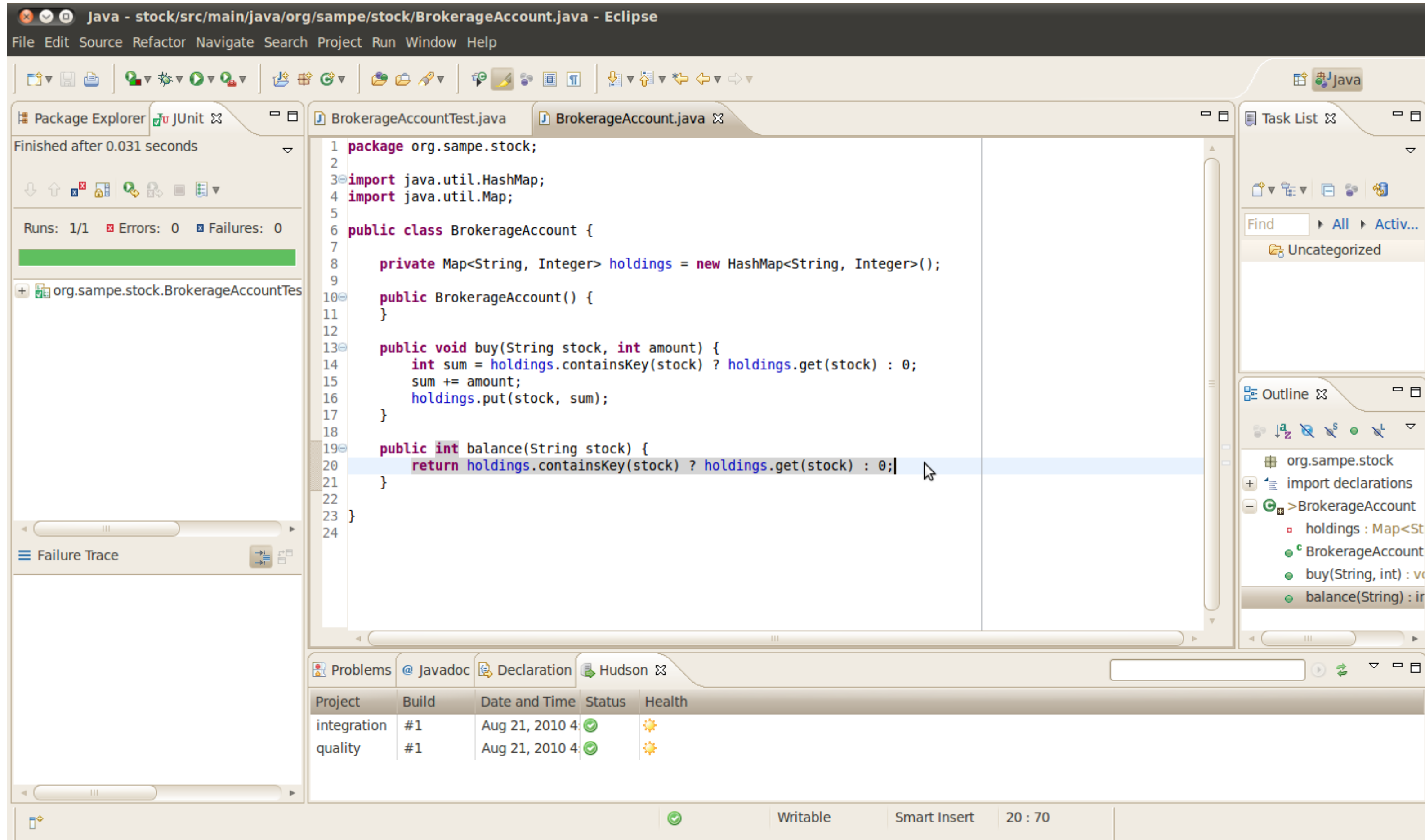
- 修正编译错误后首次运行单元测试



Green

Perficient

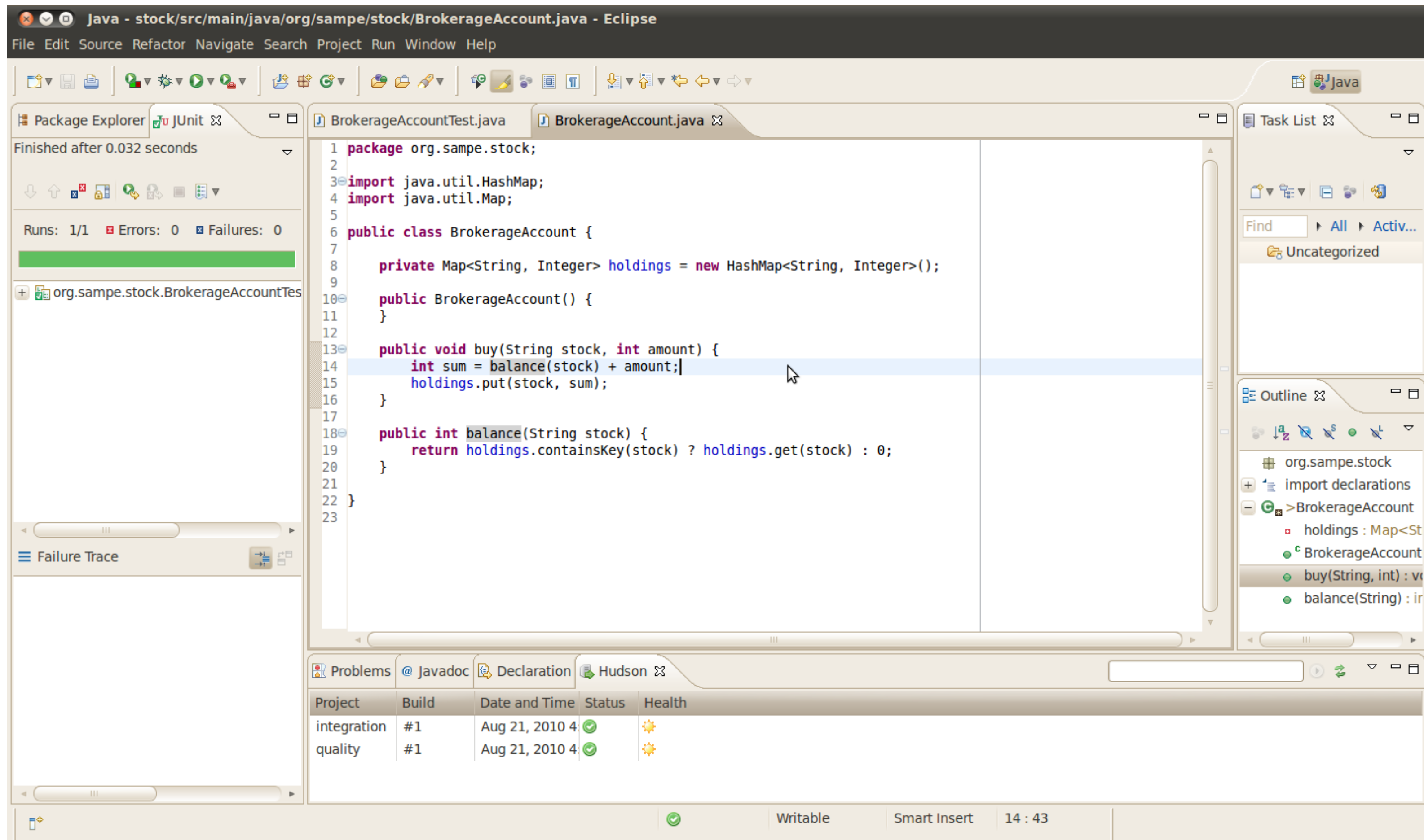
- 快速实现逻辑功能（允许代码重复）



Refactor

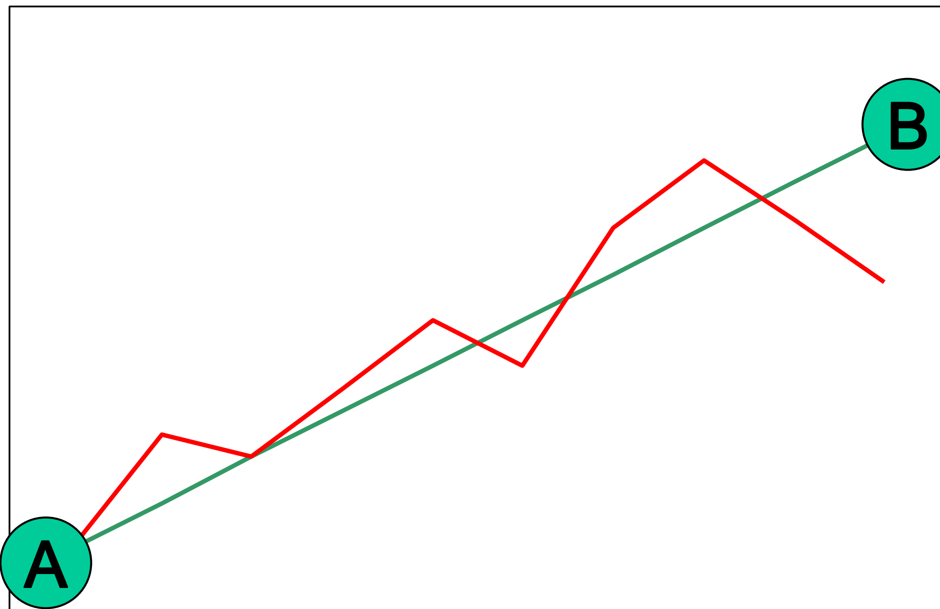
Perficient

- 仅仅通过单元测试是不够的



小步骤(Small Steps)开发更好

- 测试驱动开发提倡采用许多“小步骤”来开发。
  - 你会发现自己离目标越来越近



- 红色路线
  - 会有“废弃和返工”
  - 耗时较长
  - 也许不会很干净利落的直达目标

测试驱动开发鼓励DRY、KISS、YAGNI和SRP

Perficient

- DRY – 不要重复自己(Don’t repeat yourself)
- KISS – 保持简单“愚蠢”(Keep it simple stupid)
- YAGNI – 你不会需要它(You aren’t going to need it)
- SRP – 单一职责原则 (Single Responsibility Principle)

xUnit系列工具简介

xUnit简介

- xUnit是为各种不同编程语言和平台服务的单元测试框架系列。
  - ‘xUnit’里的‘x’是一个通配符，代表的是编程语言和平台。
  - 常用单元测试框架参见：
    - ✓ http://c2.com/cgi/wiki?TestingFramework
  - xUnit家族中的著名成员包括JUnit、CppUnit、NUnit、DBUnit、HttpUnit和JSUnit。
- xUnit可以防止你反复去做那些无聊僵化的测试工作 。

单元测试框架的基本概念

- 单元测试框架的概念起源于Kent Beck的论文《Simple Smalltalk Testing: With Patterns》。
  - http://www.xprogramming.com/testfram.htm
- xUnit框架的主要组成部分应包括 ：
  - 测试夹具(Test Fixture)
  - 测试套件(Test Suites)
  - 测试执行(Test Execution)
  - 断言(Assertion)

单元测试框架的基本概念

- 测试夹具(Test Fixture)
  - 测试夹具是运行测试成功的先决条件或状态的集合，也可称之为测试上下文(test context)。
- 测试套件(Test Suites)
  - 测试套件是指共享同一测试夹具的一组测试。测试运行的先后顺序不应当影响最终的测试结果。
- 测试执行(Test Execution)
  - 设定测试条件和加载测试代码。在测试运行之后，生成测试报告并清理测试环境。
- 断言(Assertion)
  - 断言通常是一个函数或宏，它可用于验证被测单元的行为和状态。

测试中的相关概念

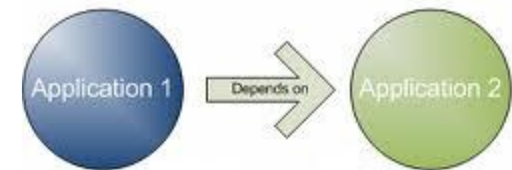
Perficient

被测系统(SUT: System Under Test)

- 被测系统实际上指的是“我们正在测试的东西”，它总是从测试的角度来进行定义的。当我们在编写单元测试时，它实际上指的就是我们正在测试的类或方法。

依赖组件(DOC: Depended-On Component)

- 它是被测系统所依赖的类或大粒度组件。依赖通常表现成某种代理形式，例如方法调用。在测试自动化中，我们经常需要审查和控制它与被测系统之间的互动以达到完全测试覆盖的目标。



一个典型的单元测试

Test Runner
suite
run
create
create
create
Testcase Object
testMethod_1
Test Suite Object
testMethod_1
testMethod_n
Testcase Class
Setup
Exercise
Verify
Teardown
Object
testMethod_n
Exercise
Exercise
Fixture
SUT

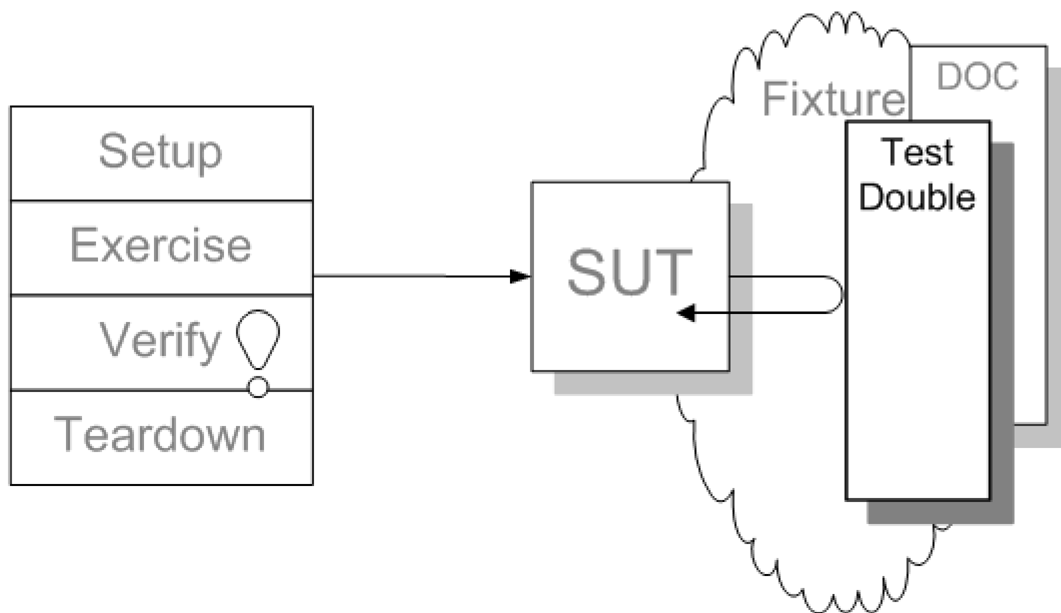
Perficient

运用模拟对象测试驱动开发

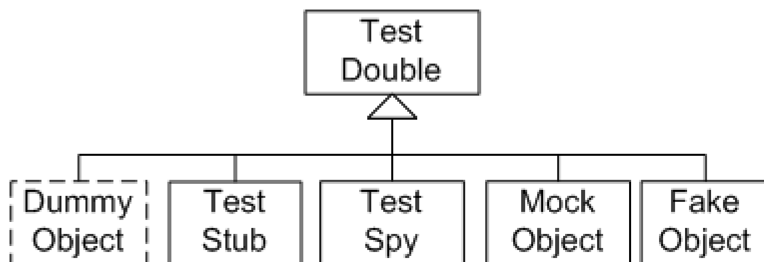
Test Double

Perficient

- Test Double是被用来安装取代真实组件的组件，其目的是让我们的测试能跑起来。



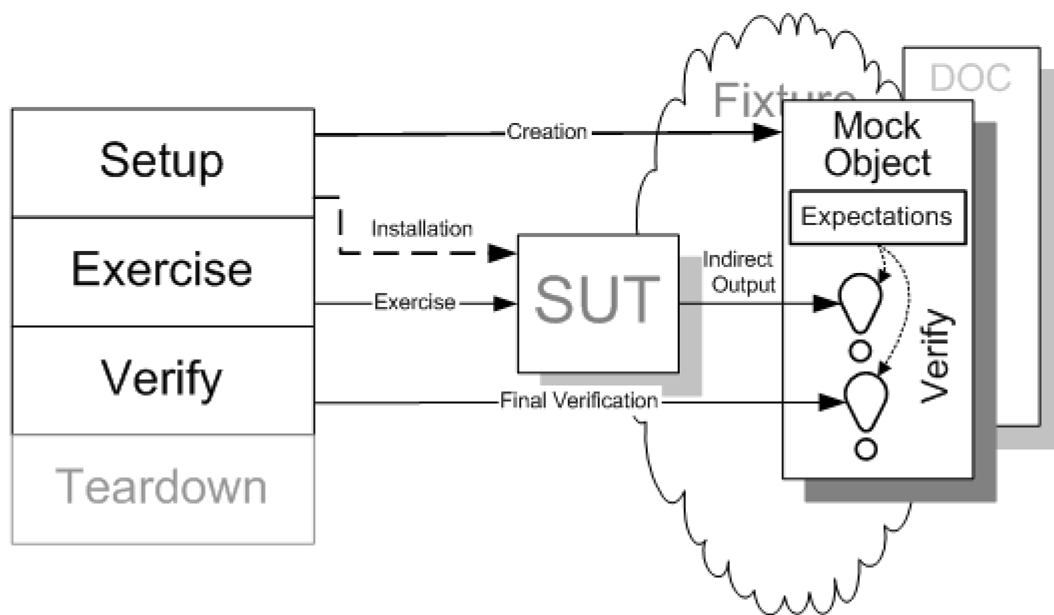
- 根据我们使用它的理由，它可有如下几种变体：



模拟对象(Mock Object)

Perficient

- 模拟对象在测试中被用来取代真正的依赖对象，其目的是在于让测试可以验证它的间接输出。



模拟对象框架的基本概念

- 基于交互的测试(Interaction Based Testing)
  - 指定对象以特定的顺序进行交互。
- 基于状态的测试(State Based Testing)
  - 开始执行某个动作，然后检查预期结果。
- 行为预期(Expectation)
  - 验证特定的方法调用是否与预期一致。
- 记录和重播模型(Record & Replay model)
  - 允许记录对模拟对象的操作，并可在之后重放和验证这些操作。

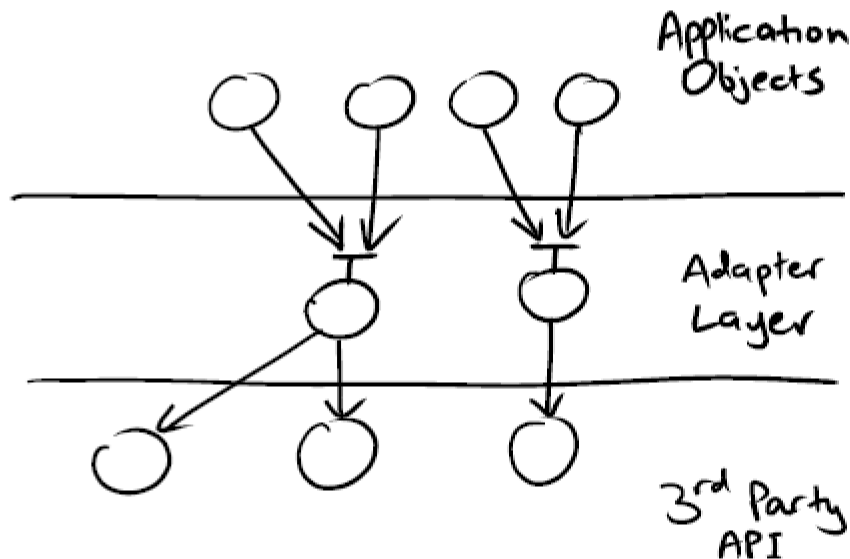
EasyMock示例

Perficient

- 创建模拟对象
  mock = EasyMock.createMock(DbAccess.class);
- 设置模拟对象的预期行为(Expectation)
  EasyMock.expect(mock.get(1, “GOOG”)).andReturn(“1000”);
- 使模拟对象做好迎接测试的准备(Record & Replay)
  EasyMock.replay(mock);
- 验证模拟对象被正确的调用
  EasyMock.verify(mock);

需要遵守的一些原则

- 为被测系统之外的接口和类创建模拟对象。
  - 不要直接为你无法控制的代码（如第三方库）创建模拟对象。



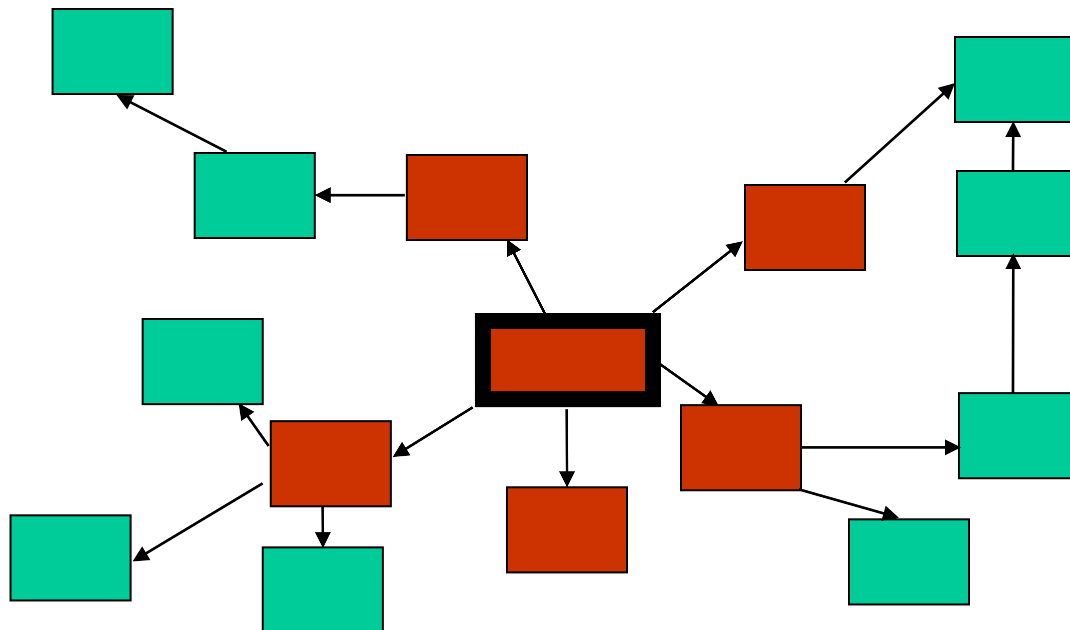

```
public interface SecurityDAO {
    UserInfo findUser(String username)
    UserInfo[ ] findUsersInRole(String rolename);
}
```

Perficient

模拟对象需要多少?

Perficient

- 在给定的测试中需要创建多少个模拟对象？
  - 只需为你的近邻创建模拟对象。



Perficient

单元测试的原则、模式及最佳实践

测试的坏味道(Test Smell)

- 测试的坏味道就是指测试代码中可能有问题的一种征兆。
  - 并不一定总是明确告诉我们什么是错的。
  - 常见的测试坏味道参见：
    - ✓ http://xunitpatterns.com/Test%20Smells.html

测试太慢(Slow Tests)

Perficient

- 测试需要太长时间来运行，以至于开发人员在修改被测系统之后并不愿每次都运行测试来验证。

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContextFactory");
env.put(Context.PROVIDER_URL, "t3://localhost:7001");
Context initCtx = new InitialContext(env);
BrokerageAccount acct = (BrokerageAccount) initCtx.lookup(“ejb/BrokerageAccount ");
acct.buy(“GOOG”, 1000);
assertEquals(1000, acct.balance(“GOOG”));
```

过多断言(Assertion Roulette)

- 在同一个测试方法中，很难准确判断出这几个断言中的哪一个才是测试失败的根本原因。

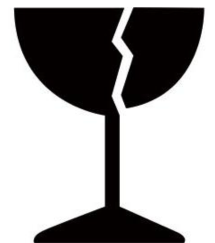
```
acct.buy("GOOG", 1000);
assertEquals(1000, acct.balance("GOOG"));
acct.buy("YHOO", 1500);
assertEquals(1500, acct.balance("YHOO"));
acct.sell("YHOO", 500);
acct.buy("GOOG", 100);
assertEquals(1000, acct.balance("YHOO"));
assertEquals(1100, acct.balance("GOOG"));
acct.buy("YHOO", 2500);
acct.sell("GOOG", 1600);
assertEquals(3500, acct.balance("YHOO"));
assertEquals(1100, acct.balance("GOOG"));
```

脆弱的测试(Fragile Test)

Perficient

- 当某个被测系统发生更改时，会造成对其他被测系统的测试也跟着一起失败。

```
@Test
public void testSellSharesWithAdequateBalance {
    acct.buy(“GOOG”, 1000);
    acct.sell(“GOOG”, 700);
    assertEquals(300, acct.balance(“GOOG”));
}
```



Perficient

不稳定的测试(Erratic Test)

- 一个或多个测试表现失常，有时通过，有时失败。

Test A 通过

```
@Before
public void initializeAccount() {
    acct = new Account(1);
    acct.buy("GOOG", 1000);
}

@Test
public void testSellSharesWithAdequateBalance {
    acct.sell("GOOG", 700);
    assertEquals(300, acct.balance("GOOG"));
}
```

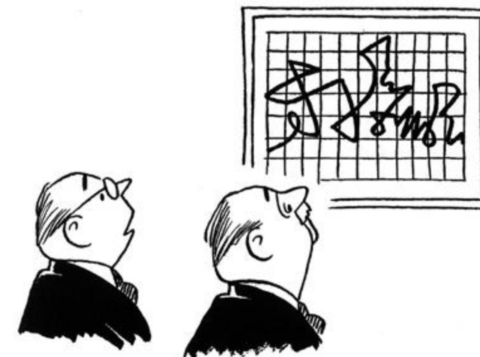
Test B 通过

```
@Before
public void initializeAccount() {
    acct = new Account(1)
    acct.buy("GOOG", 500);
}

@Test
public void testSellSharesWithoutAdequateBalance {
    acct.sell("GOOG", 700);
    assertEquals(500, acct.balance("GOOG"));
}
```

Test Suite 失败

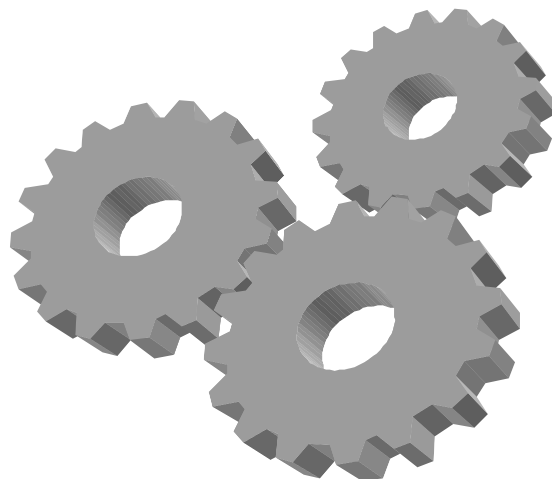
```
@RunWith(Suite.class)
@Suite.SuiteClasses({
    TestA.class,
    TestB.class})
public class AllTests {}
```



测试自动化原则

Perficient

- 这些原则是测试自动化专家们从实践中总结而来的。
- 在绝大多数情况下，它们既适用于单元测试也适用于验收测试。
- 常见的测试自动化原则参见：
  - http://xunitpatterns.com/Principles%20of%20Test%20Automation.html



保持测试独立(Keep Tests Independent)

- 相互作用或次序相关的测试通常会一起失败。
- 一个独立的测试
  - 可以独立运行而无需外力介入
  - 能明确告诉我们缺陷位置

隔离被测系统(Isolate the SUT)

Perficient

- 因为其他软件的行为发生了变化，测试可能会突然失败了。
- 隔离被测系统允许我们
  - 将不同的关注点分开测试
  - 保持测试之间彼此独立
  - 实现健壮的测试

一个测试验证一种情况(Single Condition Test)

Perficient

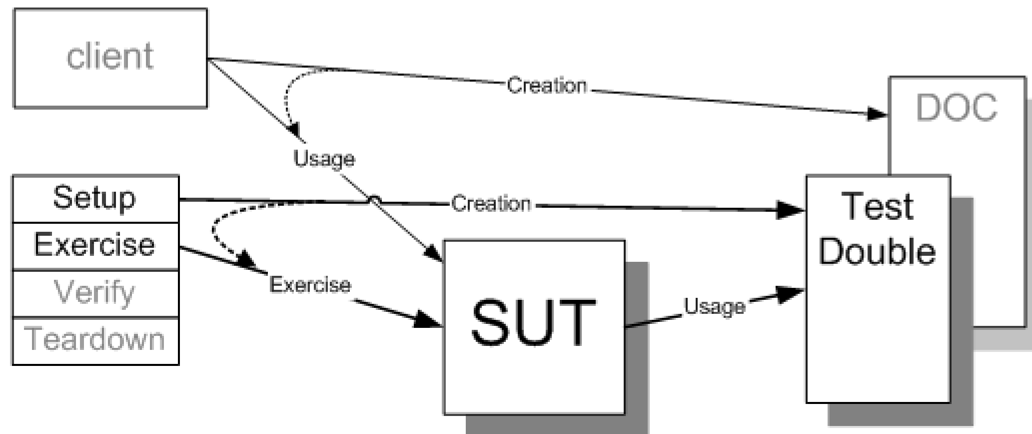
- 在单个测试中验证多种情况会使得缺陷定位变得非常困难。
- 一个测试验证一种情况可以
  - 帮助我们准确定位缺陷
  - 强制测试按照四个标准阶段运行

可测试性设计模式（1/4）

Perficient

控制反转(Inversion of Control)（又名DIP – 依赖倒置原则）

- 高层次模块不应该依赖于低层次模块，它们都应当依赖于抽象。
- 抽象不应该依赖于细节，细节应该依赖于抽象。

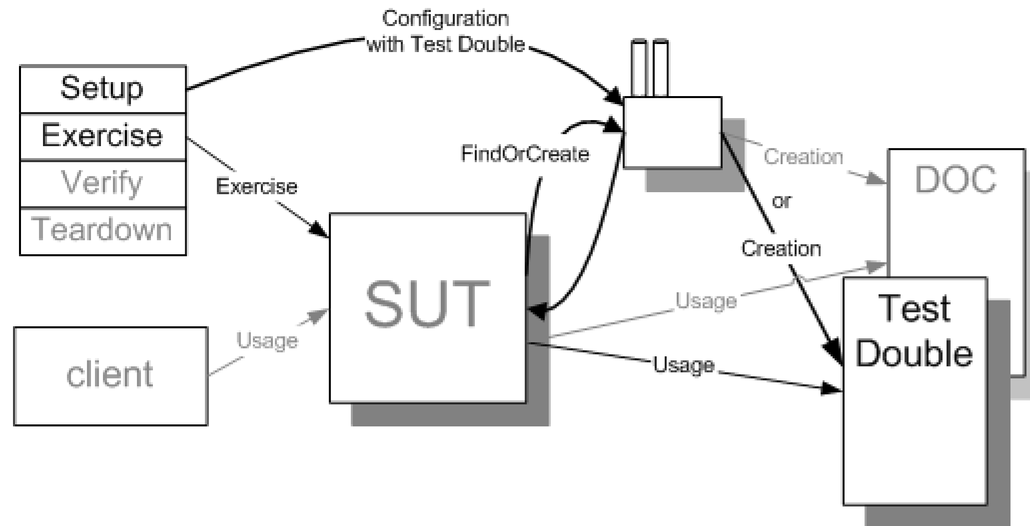


可测试性设计模式（2/4）

Perficient

依赖查找(Dependency Lookup)（又名服务定位器，对象工厂）

- 依赖查找允许被测系统之间的耦合及其依赖关系可在自动测试中被打破。
- 它提供了一些方法来告诉被测系统每次请求中应当使用哪些对象。

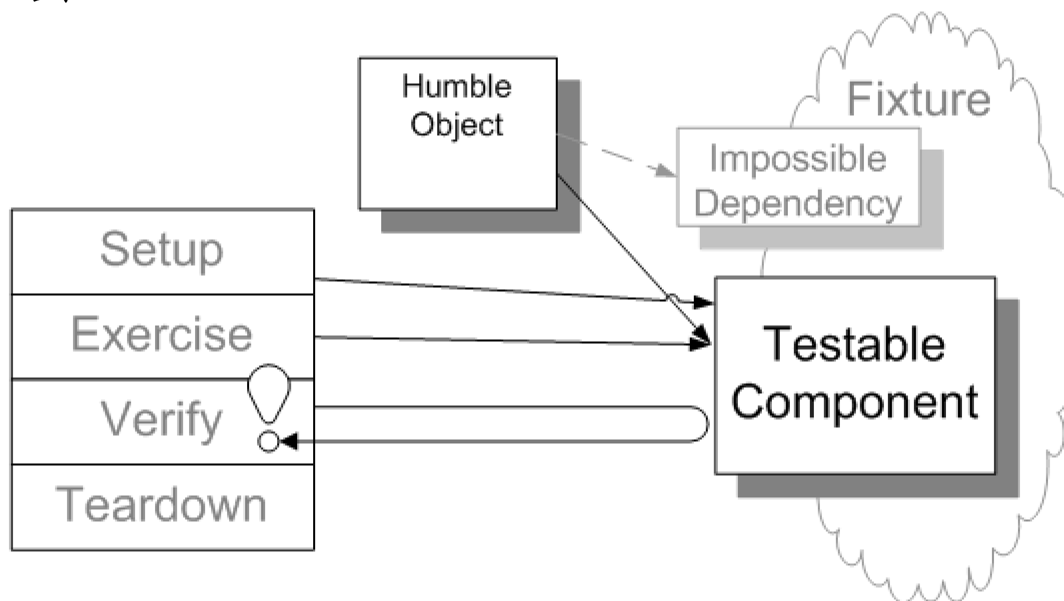


可测试性设计模式（3/4）

Perficient

Humble Object

- 它可用于测试难以实例化的对象中所包含的逻辑，相对于直接测试那些难以实例化的对象而言，其效果更佳。
- 我们从难以测试的组件中提取出所有逻辑并将其移植到可测试性更佳的组件中去。

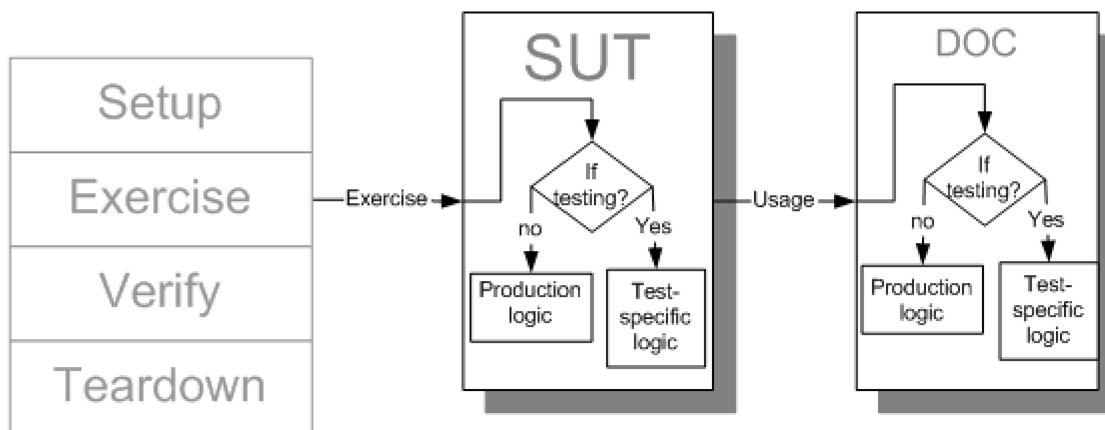


可测试性设计模式（4/4）

Perficient

Test Hook

- 它在自动化测试中向被测系统引入了测试相关的行为，通常是一种万不得已的手段。
- 它更改了被测系统的行为，将测试性代码植入到被测系统或者其依赖对象中去。



最佳实践

Perficient

- 在你的测试用例中所包含的测试方法应当要比你所要测试的类中包含的方法要多。
  - 测试应尽可能细粒度。
- 所有开发人员必须全面了解系统及其测试。
- 如果开发人员修改了系统功能，那么很可能他们也需要更新测试。
  - 警告：你已修改接口，你是否弄坏了别人的代码？
  - 添加新代码之后，所有测试都应当通过并且新的测试被加入系统。
- 所有测试总是一直处在通过状态。
  - 一个测试都不允许总是处在失败状态。
  - 追踪并立刻修复失败的测试。
- 代码提交后运行所有测试来验证系统功能是否正常。
- 在部署之前运行所有测试。

单元测试自动化实践

良好的工程实践

Perficient

- 使用源代码控制系统
- 做单元测试
- 有单元测试自动化工具
- 在提交代码前做代码审查
  - 结对编程
- 频繁提交代码
- 频繁构建系统
- 持续自动化构建
  - 在每次代码提交之后都要重新构建系统并运行所有测试
- 做好验收测试（行为驱动）开发
  - 代码提交后就应开始做验收测试

使用工具

Perficient

- 单元测试自动化框架
  - 例如JUnit
- 集成开发环境
  - 当编写测试时，可以提供代码自动完成和代码生成等辅助功能。
  - 可以直接运行指定的测试
  - 提供重构支持
  - 如Eclipse/NetBeans (包括Emma之类的插件)
- 构建环境
  - 可以在构建过程中自动运行测试
  - 可以计算代码覆盖率
  - 可以产生测试报告
  - 如Maven、Ant 、Hudson 、CruiseControl和Sonar

什么是代码覆盖率(Code Coverage)?

- 代码覆盖率是一种用来查看代码库中（所有的）语句和路径的测试覆盖状况的技术。
- 在《代码大全》第二版中， McConnell将代码覆盖率和逻辑覆盖率(Logic Coverage)作为同一概念引用。
- 如果没有自动化工具，计算代码覆盖率将会是一个工程浩大的工作。
- 对我们而言，重要的是要知道我们能从代码覆盖率数据中得到什么。

集成开发环境里的代码覆盖率报告

- Eclipse下使用Emma插件

Java - stock/src/main/java/org/sampe/stock/BrokerageAccount.java - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help

Package Explorer | JUnit

Finished after 0.125 seconds

Runs: 3/3 Errors: 0 Failures: 0

org.sampe.stock.BrokerageAccountTes

Failure Trace

BrokerageAccountTest.java | BrokerageAccount.java

```java
2
3 import java.util.HashMap;
4 import java.util.Map;
5
6 public class BrokerageAccount {
7
8     private Map<String, Integer> holdings = new HashMap<String, Integer>();
9
10    public BrokerageAccount() {
11    }
12
13    public void buy(String stock, int amount) {
14        int sum = balance(stock) + amount;
15        holdings.put(stock, sum);
16    }
17
18    public int balance(String stock) {
19        return holdings.containsKey(stock) ? holdings.get(stock) : 0;
20    }
21
22    public void sell(String stock, int amount) {
23        int current = balance(stock);
24        if (current > amount) {
25            holdings.put(stock, current - amount);
26        }
27    }
28
29 }
30
```

Task List

Find | All | Activ...

Uncategorized

Outline

- org.sampe.stock
- import declarations
- BrokerageAccount
  - holdings : Map<
  - BrokerageAccou
  - buy(String, int)
  - balance(String)
  - sell(String, int)

Problems | Javadoc | Declaration | Hudson | Coverage

BrokerageAccountTest (Aug 21, 2010 5:28:48 PM)

Element	Coverage	Covered Instructi	Missed Instruc	Total Instructions
stock	100.0 %	123	0	123
src/test/java	100.0 %	70	0	70
src/main/java	100.0 %	53	0	53

Writable | Smart Insert | 23 : 38

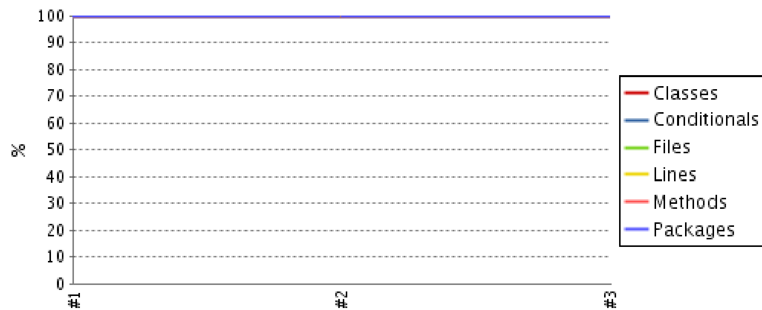
Hudson里的代码覆盖率报告（1/2）

Perficient

Code Coverage

Cobertura Coverage Report

Trend



Project Coverage summary

Name	Classes	Conditionals	Files	Lines	Methods	Packages
Cobertura Coverage Report	100% 1/1	100% 4/4	100% 1/1	100% 11/11	100% 4/4	100% 1/1

Coverage Breakdown by Package

Name	Classes	Conditionals	Files	Lines	Methods
org.sampe.stock	100% 1/1	100% 4/4	100% 1/1	100% 11/11	100% 4/4

Page generated: Aug 21, 2010 5:21:09 PM Hudson ver. 1.372

Hudson里的代码覆盖率报告 (2/2)

Source

org/sampe/stock/BrokerageAccount.java

Line	Hits	Source
1		package org.sampe.stock;
2		
3		import java.util.HashMap;
4		import java.util.Map;
5		
6		public class BrokerageAccount {
7		
8	6	private Map<String, Integer> holdings = new HashMap<String, Integer>();
9		
10	6	public BrokerageAccount() {
11	6	}
12		
13		public void buy(String stock, int amount) {
14	20	int sum = balance(stock) + amount;
15	20	holdings.put(stock, sum);
16	20	}
17		
18		public int balance(String stock) {
19	30	return holdings.containsKey(stock) ? holdings.get(stock) : 0;
20		}
21		
22		public void sell(String stock, int amount) {
23	4	int current = balance(stock);
24	4	if (current > amount) {
25	2	holdings.put(stock, current - amount);
26		}
27	4	}
28		
29		}

Page generated: Aug 21, 2010 5:21:48 PM   Hudson ver. 1.372

Sonar里的代码质量报告

Perficient

Version 1.0-SNAPSHOT - Sat, 21 Aug 2010 17:20 - profile Sonar way

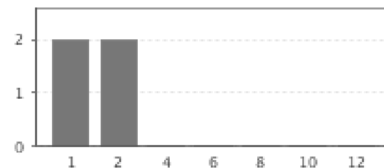
Lines of code
21
29 lines
6 statements
1 files

Classes
1
1 packages
4 methods
+0 accessors

Comments
0.0%
0 lines
0.0% docu. API
4 undocu. API
0 commented LOCs

Duplications
0.0%
0 lines
0 blocks
0 files

Complexity
1.5 / method
6.0 / class
6.0 / file
Total: 6

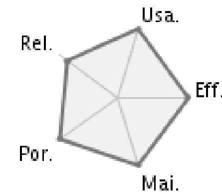


Methods Classes

Code coverage
100.0%
100.0% line coverage
100.0% branch coverage
3 tests
23 ms

Test success
100.0%
0 failures
0 errors

Rules compliance
85.7%



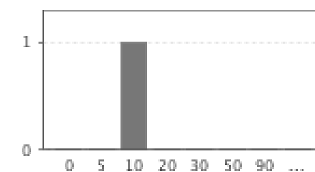
Violations
3

Blocker	0
Critical	0
Major	0
Minor	3
Info	0

Package tangle index
0.0%

LCOM4
1.0 /class
0.0% files having LCOM4>1

RFC
11 /class



Add a measure

回顾

Perficient

- 什么是测试驱动开发?
- 测试驱动开发与传统开发之间有什么区别?
- 测试与设计之间有着什么样的关系?
- 什么是模拟对象(Mock Object)?

参考

- 测试驱动开发
  - http://c2.com/cgi/wiki?TestDrivenProgramming
  - http://c2.com/cgi/wiki?TestFirstDesign
  - http://www.testdriven.com/
  - http://www.xprogramming.com/xpmag/testFirstGuidelines.htm
  - http://www.objectmentor.com/writeUps/TestDrivenDevelopment
  - http://xunitpatterns.com
- 工具
  - http://junit.org
  - http://www.easymock.org
  - https://hudson.dev.java.net
  - http://eclipse.org
  - http://www.vectrace.com/mercurialeclipse
  - http://code.google.com/p/hudson-eclipse

Perficient