第十三讲
测试面向对象程序

OO2018课程组

计算机学院

摘要

- 基本概念
- 如何设计测试数据
- 如何设计测试场景
- 如何基于规格设计测试
- JUnit测试框架
- 作业

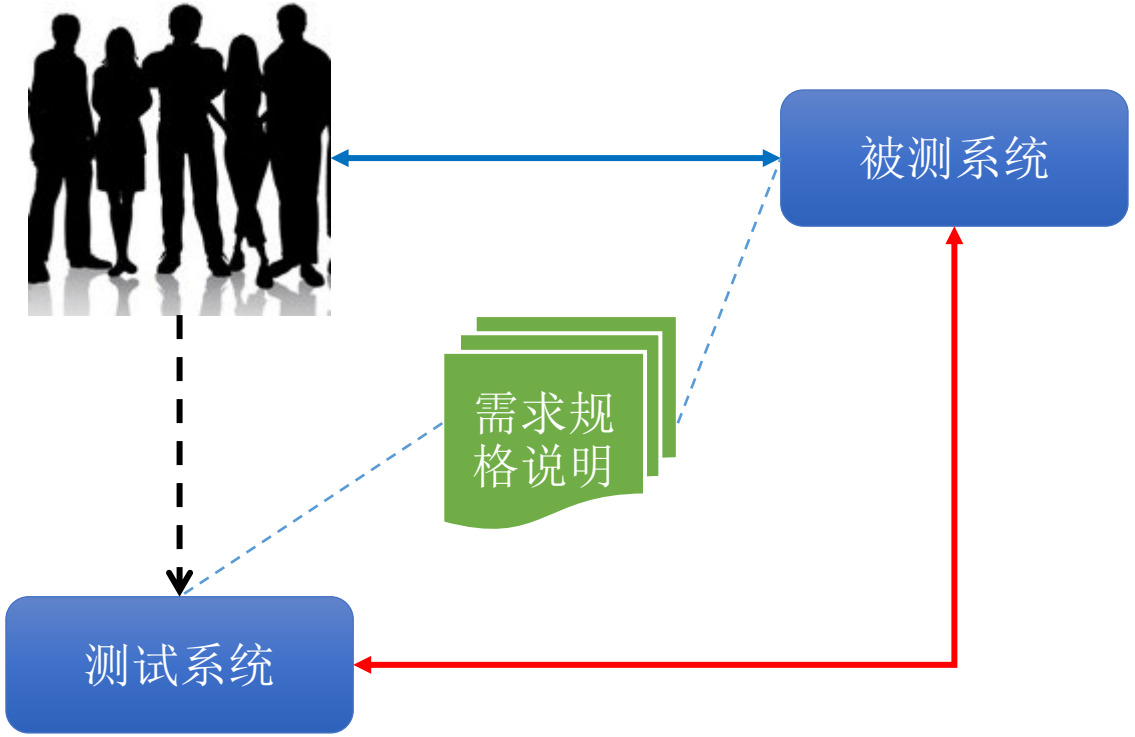
基本概念

- 现实世界中的软件一定存在bug
  - 为什么？
- 测试的目标
  - 发现软件中隐藏的bug
    - 怎么知道隐藏在哪里？
  - 确认项目合同/软件需求的完成情况
- 测试不可以证明软件没有bug
- 被测系统(System Under Test, SUT)
- 测试系统(Test System)

什么是软件bug？
Bug有什么静态和动态特征？

基本概念

- 测试系统的组成
  - 行为
    - 测试准备/初始化行为
    - 测试激励行为
    - 测试判定行为
  - 数据
    - 测试初始化数据
    - 测试激励数据
    - 测试判定数据



如果被测系统与用户在运行时有交互行为，测试系统如何实施测试？

基本概念

- 测试用例(test case)
  - 场景型测试用例
  - 数据型测试用例
- 测试判定结果
  - Pass：测试用例执行通过判断
  - Fail：测试用例执行未通过判断
  - Inconclusive：不确定
- 测试覆盖
  - 代码覆盖：语句覆盖、分支覆盖、路径覆盖
  - 功能覆盖：功能覆盖、场景覆盖
  - 数据覆盖：输入划分覆盖、数据组合覆盖

基本概念

- 测试类型划分
  - 按照SUT信息的详细程度
    - 黑盒测试
    - 白盒测试
    - 灰盒测试
  - 按照测试目标
    - 功能测试
    - 单元测试
    - 性能测试
    - 鲁棒性测试
    - 回归测试
    - …

如何设计测试数据

- 分析被测对象需要提供哪些输入
  - 输入格式
  - 输入内容
- 分析各个输入的范围划分(test partition)
  - 划分等价类
- 分析输入之间的可能依赖关系
- 针对每个输入Xi“抽样”生成相应的数据
  - Mi组数据：xi(1),…xi(mi)
- 进行组合

分析测试输入

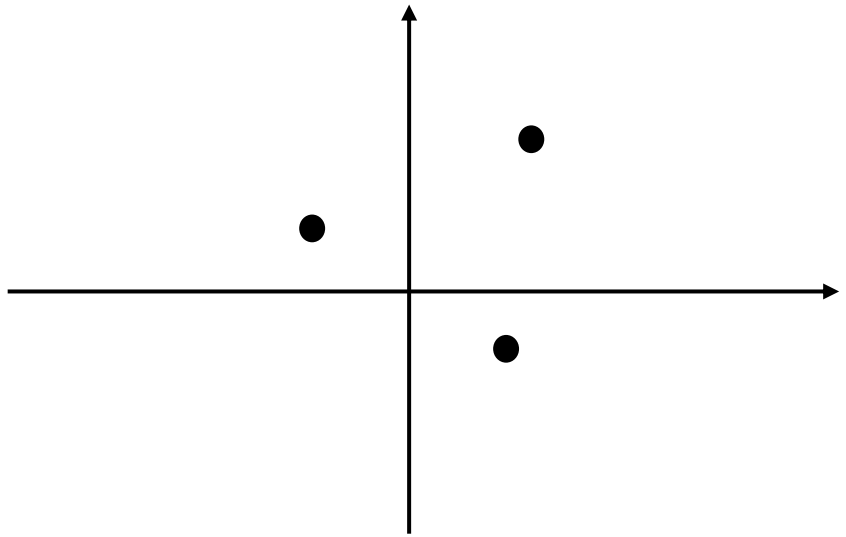
- 显式定义的输入
  - 方法、接口的输入参数
    - 输入的数据类型有语法上的严格定义
    - 输入的内涵或语义有过程规格加以定义
- 隐含定义的输入
  - 运行时从外部获得的输入(System.in、文件、硬件设备读取的数据等)
    - 格式、内容往往缺少类型层次的明确定义
  - 对象状态（属性取值）
    - 类型和内涵通过rep和不变式加以定义
- 一个测试用例可能同时涉及显式输入和隐含输入

分析测试输入

- 参数化显式输入：方法参数
  - 类型、数据范围
  - 测试手段：调用时绑定参数
- 隐含定义输入：对象状态
  - 对象属性类型、目标状态
  - 测试手段：按照一定场景调用相应方法来获得目标状态
- 隐含定义输入：交互式输入数据
  - 组成结构==》类型和范围
  - 测试手段：通过交互设备（控制台/GUI/硬件）提供输入
- 隐含定义输入：批量输入的数据(文件)
  - 文件格式、数据存储结构、类型和范围
  - 测试手段：测试开始前准备好

输入数据的范围划分

- 依据数据的结构pattern来划分
  - 符合结构~不符合结构
- 依据数据的取值范围来划分
  - 合法数据与非法数据
- 依据功能处理的场景来划分
  - 等价类数据



电梯作业中的乘客请求：
(FR,n,DOWN/UP,t)
(ER,n,t)

文件统计作业中的目录输入：
Disk:\\dir

Intersection(IntSet a)
  IntSet a

三角形判定函数输入：
boolean isTriangle(int x1,int y1,
int x2, inty2, int x3, int y3)

分析输入之间的依赖关系

- 有些输入之间具有内在的相关性
  - 一个输入的取值会影响另一个输入的有效取值范围
  - 物流系统GetOrder(String name, Address addr, PCODE pcode, Order order)
  - addr+pcode的有效性
- 在场景型测试中，输入具有先后次序
  - 前面输入引起的对象状态改变会对后面输入的测试效果产生影响
  - 难以复现这种类型的问题

抽样产生输入数据

- 类别覆盖
  - 每个输入的每个类别都应该有“代表性”数据
- 边界关注
  - 不同类别之间“结合部”的数据往往容易发现程序的错误
  - Intersection(IntSet a): a为null； a为empty； a不空但与this无交集； a不空且与this交集只有一个元素； a不空与this交集规模等于this规模。
- 进行组合
  - 全组合
  - 每个输入的每个类别至少出现一次
  - 有依赖关系的输入之间的每个组合类别至少出现一次

如何设计测试场景

- 测试场景关注测试系统与被测系统之间的连续交互
  - 每个交互步骤都是对被测系统的一次激励
  - 每个交互步骤之后都能够检查被测系统的状态和输出
- 与一个程序的交互场景
- 与一个线程的交互场景
- 与一个对象的交互场景

与程序的交互测试场景

- 模拟用户与程序的交互
- 软件为用户提供有用的功能
  - 需要与用户进行交互：获得用户的输入、反馈结果给用户
- 按照交互顺序构造测试场景
  - UML顺序图
- 交互手段多样化
  - 设备相关
- **覆盖**系统与用户的交互场景

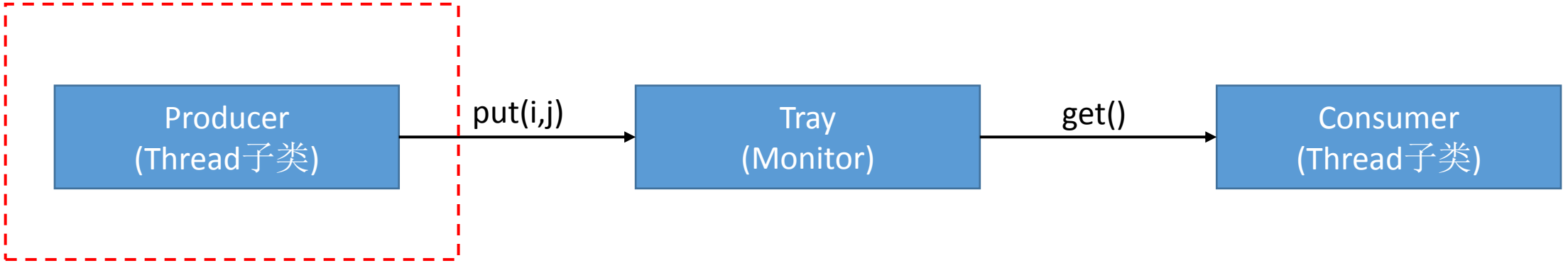
选课系统：学生登录系统后，查看可供选择的课程列表和课的详细信息(学分、学时、类别、开课时间段、授课老师、上课地点)，选课和退选(不迟于开课一周内)

与线程的交互测试场景

- 线程之间通过共享资源进行交互
  - 同步访问控制
  - 每个线程按照自己的逻辑来访问共享对象和进行处理
- 测试必须要解决的四个问题
  - 确定测试目标
    - 线程对共享对象不同状态的处理情况
    - 线程之间的同步控制
  - 构造测试线程
  - 实现测试目标
    - 调用共享对象相关方法来使共享对象达到预期状态
    - 注意使用合适的锁机制来访问共享对象
  - 观察被测线程的行为
    - 线程执行日志
    - 检查共享对象状态的变化

与线程的交互测试场景

- 生产者线程和消费者线程
  - 生产者向一个锁对象(托盘)里存入生产的货物 //synchronized method
  - 消费者从托盘里取走相应的货物 //synchronized method
  - 在货物被取走前，不能放入新的货物 //控制变量表示托盘状态
  - 在货物被取走后，不能再次取货 //控制变量表示托盘状态
  - 三个类：生产者、消费者、托盘



与线程的交互测试场景

```
public class Producer extends Thread {
    private Tray tray;          private int id;
    public Producer(Tray t, int id) {
    tray = t;          this.id = id;          }
    public void run() {
        for (int i = 0; i < 10; i++)
         for(int j =0; j < 10; j++ ) {
            tray.put(i, j);
            System.out.println("Producer #" + this.id   + " put: ("+i +","+j + ").");
            try { sleep((int)(Math.random() * 100));  }
            catch (InterruptedException e) { }
         };
     }
  }
```

与线程的交互测试场景

```
public class ProducerTester {
    private Tray tray;
    private int id;
    Producer producer;
    public ProducerTester() {
        tray = new Tray();        this.id = 1;  producer = new Producer(tray, this.id);
producer.start();  }
    public void test() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = tray.get();
            System.out.println("Consumer #" + this.id + " got: " + value);
        }
    }
}
```

与对象的交互测试场景

- 对象是状态化的存在
- 按照一定的场景来控制和观察对象的状态变化
  - 控制：修改方法，（方法输入参数，this）是控制因素
  - 观察：观察方法
- 控制变量的划分
  - 满足对象方法的Requires **and** 不变式
  - 不满足对象方法的Requires **or** 不变式
- 检查运行结果
  - 是否按照Effects抛出了相关异常
  - 返回值是否满足Effects的要求
  - 对象状态是否满足repOK

与对象的交互测试场景

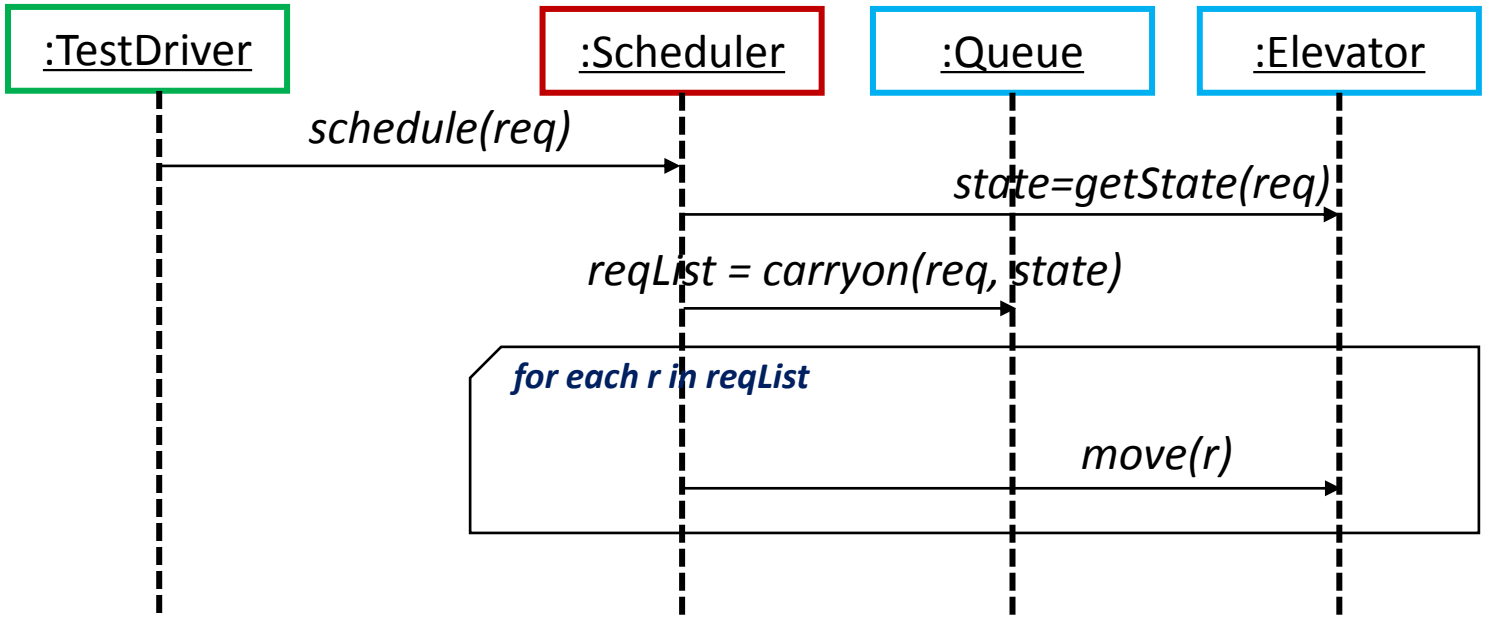
- 根据被测对象的设计规格构造测试场景
  - 测试目标：在[被测对象**状态下]调用[**方法]，获得[**结果]，检查[**约束/条件]是否成立
  - 获得被测对象执行结果的手段
    - 方法返回值
    - 调用观察方法获得对象状态
    - 为被测对象配置其他对象来接收其发送的消息
- 实现测试场景
  - 构造被测对象
  - 配置被测对象
    - 接收被测对象消息的对象
  - 确保被测对象处于期望状态
    - 调用其相关方法
  - 发送测试激励
    - 调用目标方法
  - 获得测试结果
    - 三种手段相结合
  - 检查测试结果

设置目标状态：依据overview、不变式等进行划分
BankAccount类: accountnumber有效， balance >=0, 记账本对象有效

调用被测方法：依据输入参数范围和Requires限制进行划分
BankAccount类的deposit方法：boolean deposit(float amount)
/*@Requires: amount > 0

获得测试结果（依据后置条件）：deposit方法会把交易动作登记到记账本；会返回交易是否成功结果；会更新balance

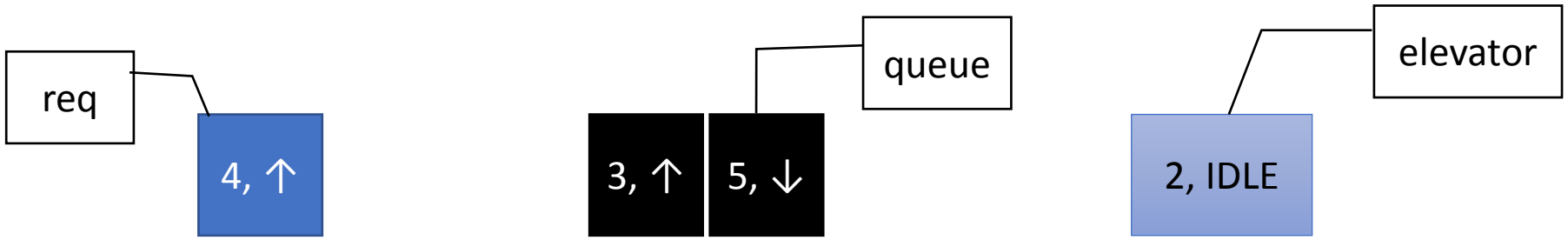
更加一般化的对象交互测试场景



如何配置Queue中的数据？

Elevator中的getState做什么检查？

Elevator中的move做什么检查？



如何基于规格设计测试

```
public Vector<String> scan4subs(String dir)
/**@requires: Files.isDirectory(dir) == true
   *@modifies: none
*@effects: \all String p; \result.contains(p) ==>
p.substring(dir).equals(dir) && Files.isFile(p) */
```

- 过程规格
  - 根据Requires信息来做测试准备
  - 根据Requires + Effects来对方法输入进行数据划分
  - 根据Effects来检查方法执行是否满足规格要求
- 类规格
  - 基于不变式来准备相关对象的属性数据
  - 使用不变式来检查方法执行后的对象状态(调用repOK)
- 迭代规格
  - 使用Effects信息即可
- 类型层次规格
  - 使用子类对象代替父类型对象的使用场合：被测对象方法的参数输入、被测对象

测试是否可以结束了？

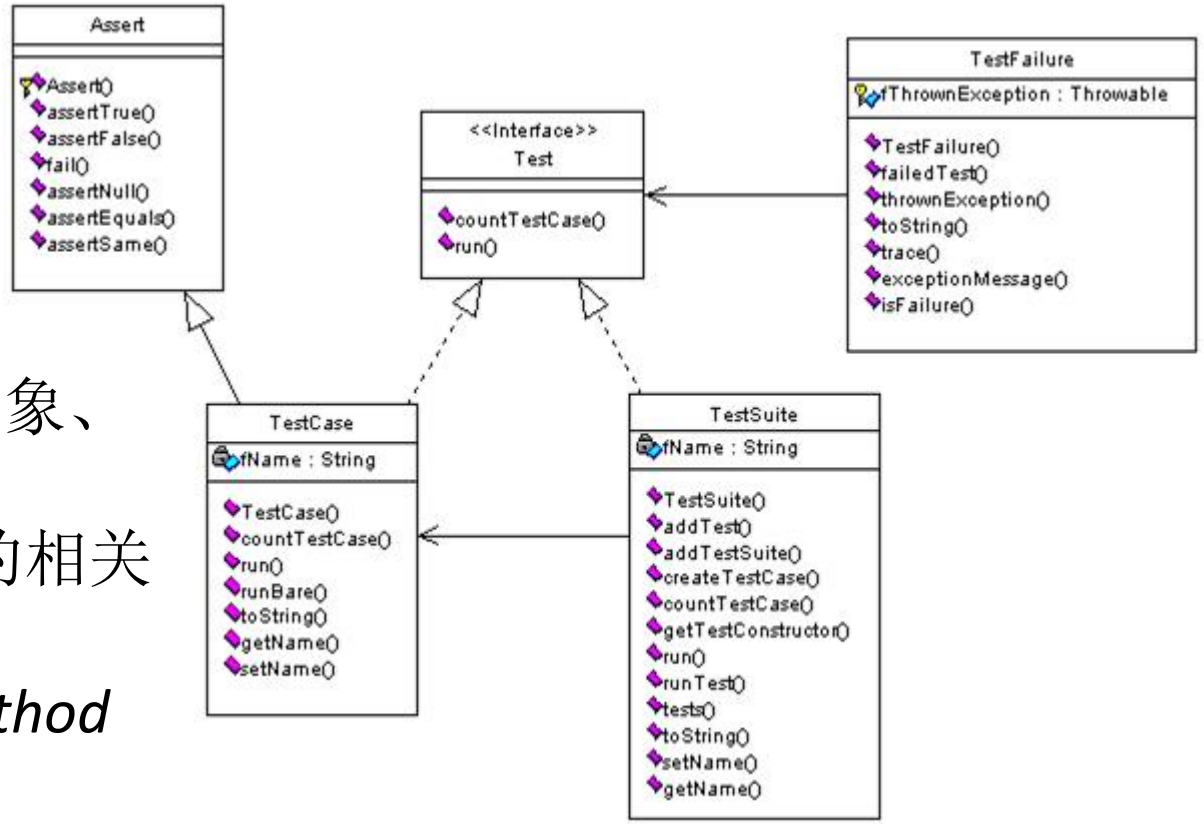
- 你永远不知道代码中还存在哪些bug
- 看看你覆盖了什么
  - 每个方法是否都覆盖了？
  - 方法的输入划分组合是否都覆盖了？
  - 对象交互流程是否都覆盖了？
  - 数据容器对象覆盖了哪些格局？
  - 是否所有代码都被执行了？
  - 是否所有分支都被执行了？

一个高级话题

- 测试系统与被测系统的关系
  - 如何利用测试用例代码和结果，已经成为程序质量研究中的重要内容
- Hot topics
  - Bug分配
  - 自动缺陷定位
  - 自动缺陷修复
  - 测试优化
  - 可靠性分析评估
  - More fancy ideas…

使用Junit高效率的组织和运行你的测试

- 普遍使用的单元测试方法
- 提供了一个框架(junit.framework)
- TestCase
  - setUp：准备测试所需的数据(被测对象、测试桩对象和必要的数据)
  - tearDown：测试完成后清理所建立的相关数据
  - testMethodName：调用被测方法*method*
- 继承TestCase实现自己的测试用例



使用Junit框架编写测试

```
public class IntSetTest extends TestCase {
   private int[] v;
   private IntSet s;

   public IntSetTest(String testName) {
      super(testName);
   }

   protected void setUp() throws Exception {
      super.setUp();
      v = new int[] {5, 3};
      s = new IntSet(v);
   }

   protected void tearDown() throws Exception {
      super.tearDown();
      v = null;
      s = null;
   }

   public void testInclude() {
      int x = 3;
      assertEquals(s.include(x), true);
   }
}
```

使用Junit4来编写测试

```
import org.junit.*;
import static org.junit.Assert.*;
public class <xxxTest> {
    ...
    @BeforeClass
    public static void <name>(){…}
    @AfterClass
    public static void <name>(){…}
    @Before
    public void <name>() {…}
    @After
    public void <name>() {…}
    @Test
    public void <testxxx>() {…}
}
```

@BeforeClass: 一次性setUp, @AfterClass: 一次性tearDown,@Before: setUp, @After: tearDown,@Test: test case

使用断言来检查被测对象的返回值

- public void assertTrue(*String message*, boolean condition)
- public void assertFalse(*String message*, boolean condition)
- public void assertEquals(*String message*, Object expected, Object actual)
- public void assertNotEquals(*String message*, Object expected, Object actual)
- public void assertSame(*String message*, Object expected, Object actual)
- public void assertNotSame(*String message*, Object expected, Object actual)
  - 使用 == 而不是`.equals`来进行比较
- public void assertNull(*String message*, Object obj)
- public void assertNotNull(*String message*, Object obj)
- assert <condition>
- public void fail(*String message*)
  - 强制测试失败

如何处理被测方法抛出异常？

Learn more in http://www.ibm.com/developerworks/java/tutorials/j-junit4/

```
import org.junit.*;
import static org.junit.Assert.*;

public class <xxxTest> {
    ...

    @Test(expected = IndexOutOfBoundsException.class)
    public void <name>() {
        ....
    }
}
```

- 如果被测方法抛出`IndexOutOfBoundsException`，该测试用例执行通过，否则测试用例执行抛出Fail

.class是什么？

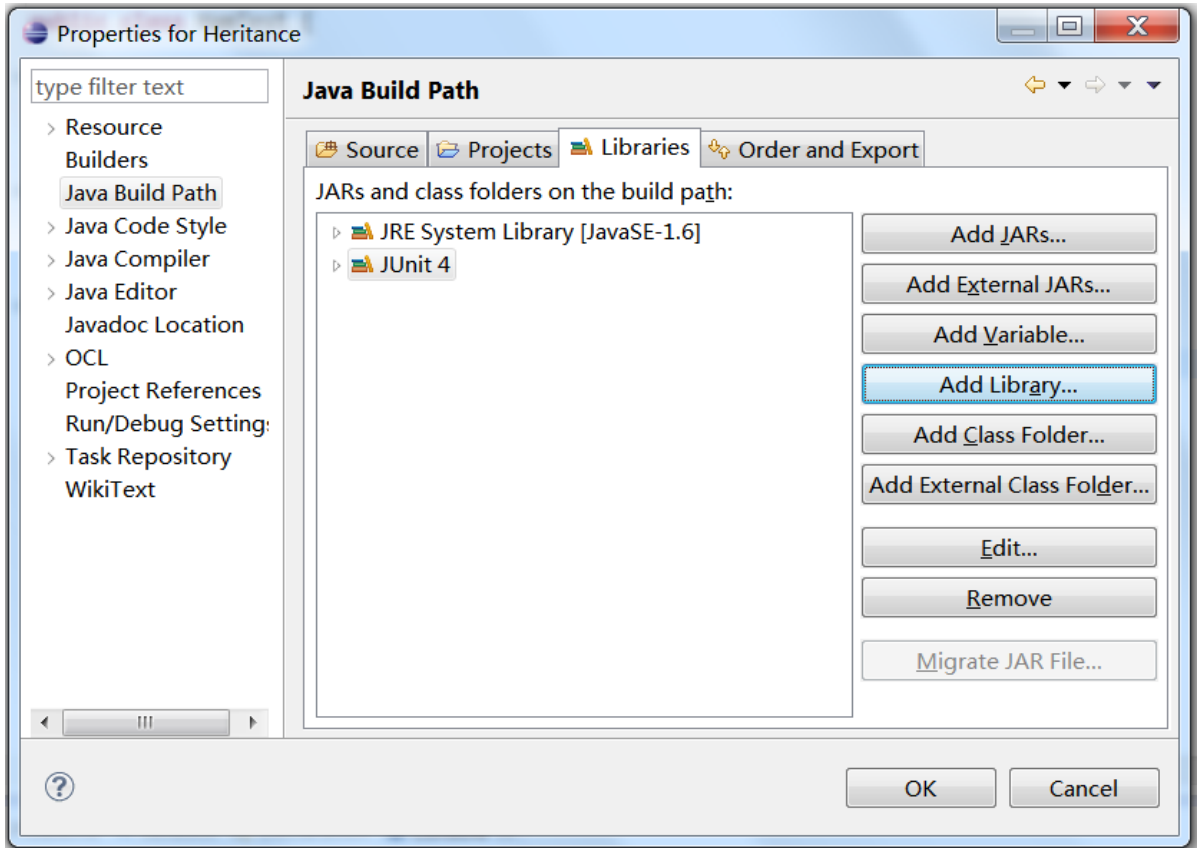
- 任何一个类都可以获得这个.class属性，它的类型是java.lang.Class
- Java语言的反射机制
  - 在不了解类具体规格的情况下来获得类、方法和属性等相关信息，甚至是构造对象和调用对象的机制
- Object与Class的关系
  - Object管理所有对象
  - Class管理所有对象的类型

```
import java.lang.*;
import java.lang.reflect.*;
public class C{
    public void reflectMethods(Class x){
        Method[] methods = x.getDeclaredMethods();
        Method m;
        for(int i = 0; i<methods.length;i++){
          m = methods[i];
          System.out.println(m);
          if(m.getName()=="foo"){
            try{m.invoke(this,i);}catch (Exception e)
            {System.out.println(e);}
          }
        }
    }
    public void foo(int i){
        System.out.println(i);
    }
    public static void main(String[] args){
        C c = new C();
        c.reflectMethods(c.getClass());
    }
}
```

```
public void C.reflectMethods(java.lang.Class)
public void C.foo(int)
1
public static void C.main(java.lang.String[])
```

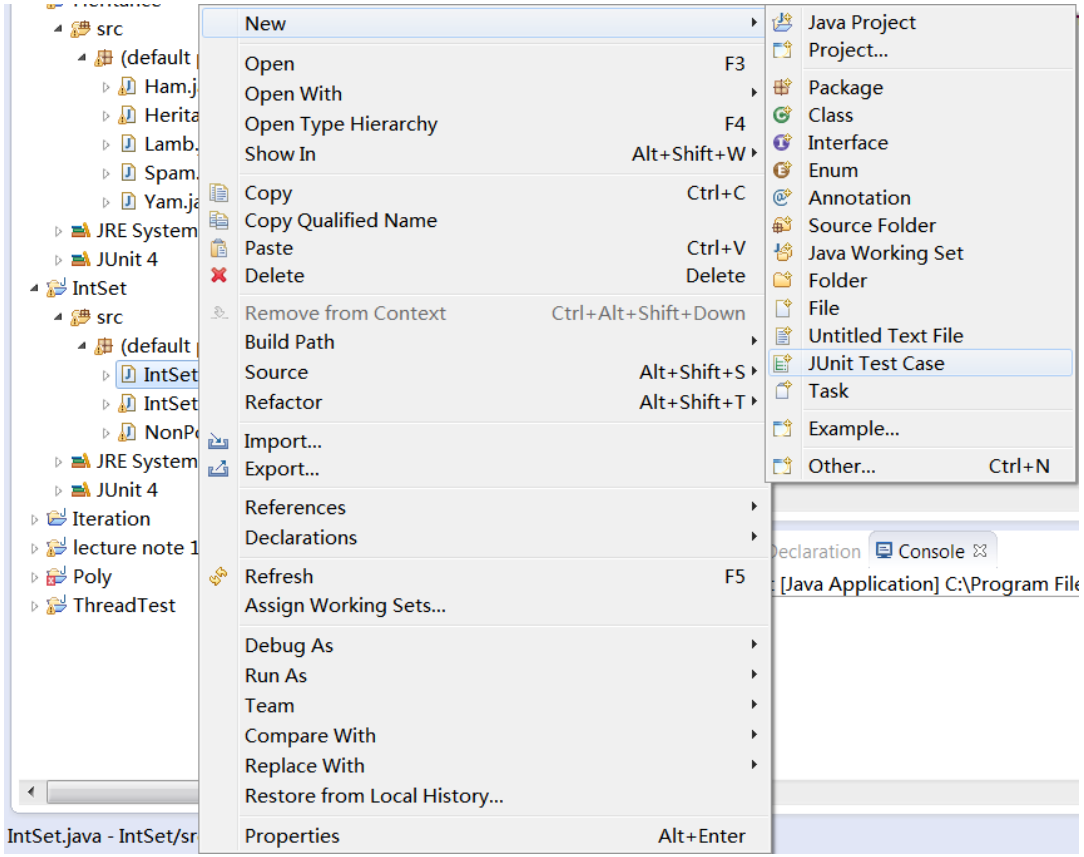
在Eclipse环境下使用Junit

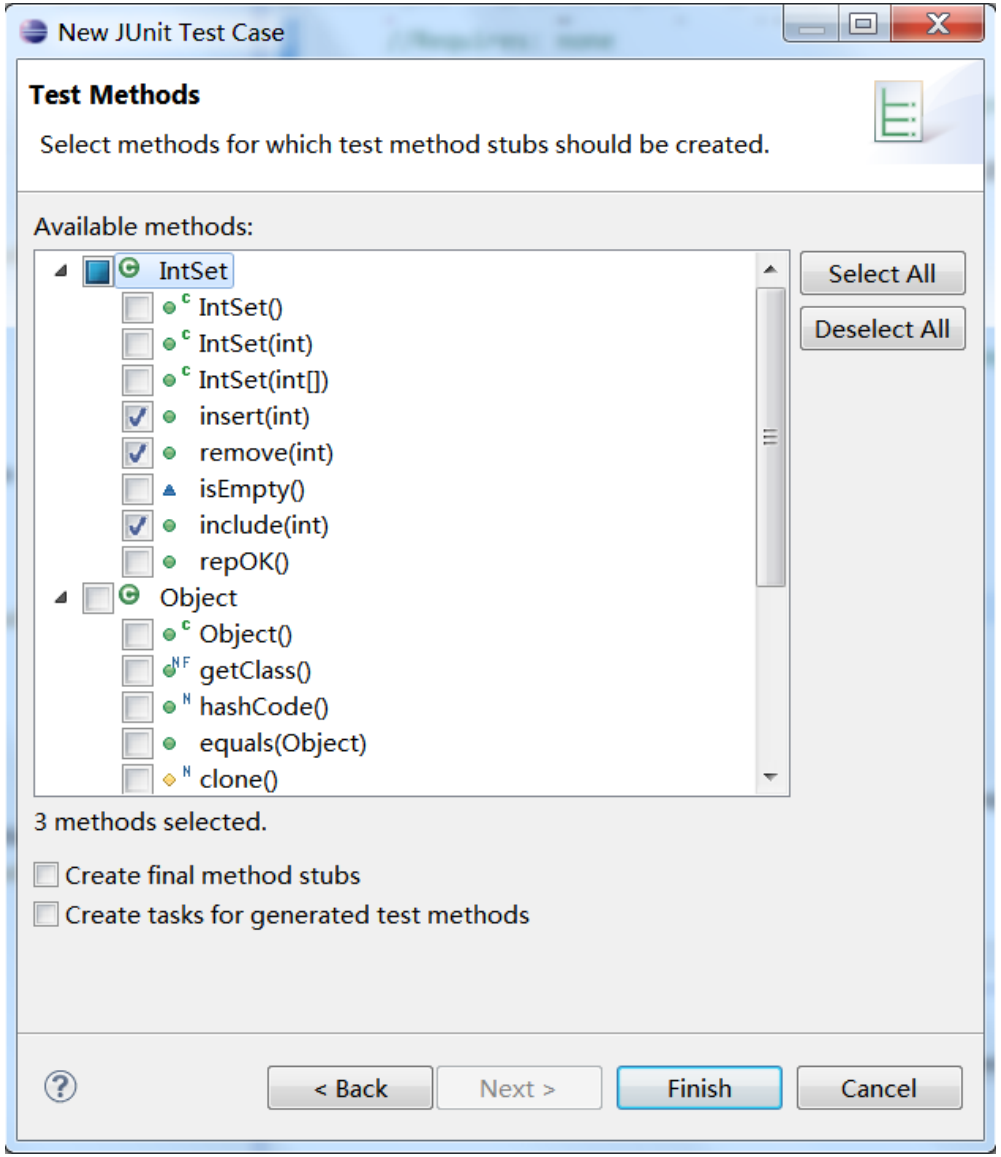
- 在Eclipse项目中加入Junit:
  - **Project** -> **Properties** -> Java Build Path -> Libraries -> **Add Library** -> Junit ->...



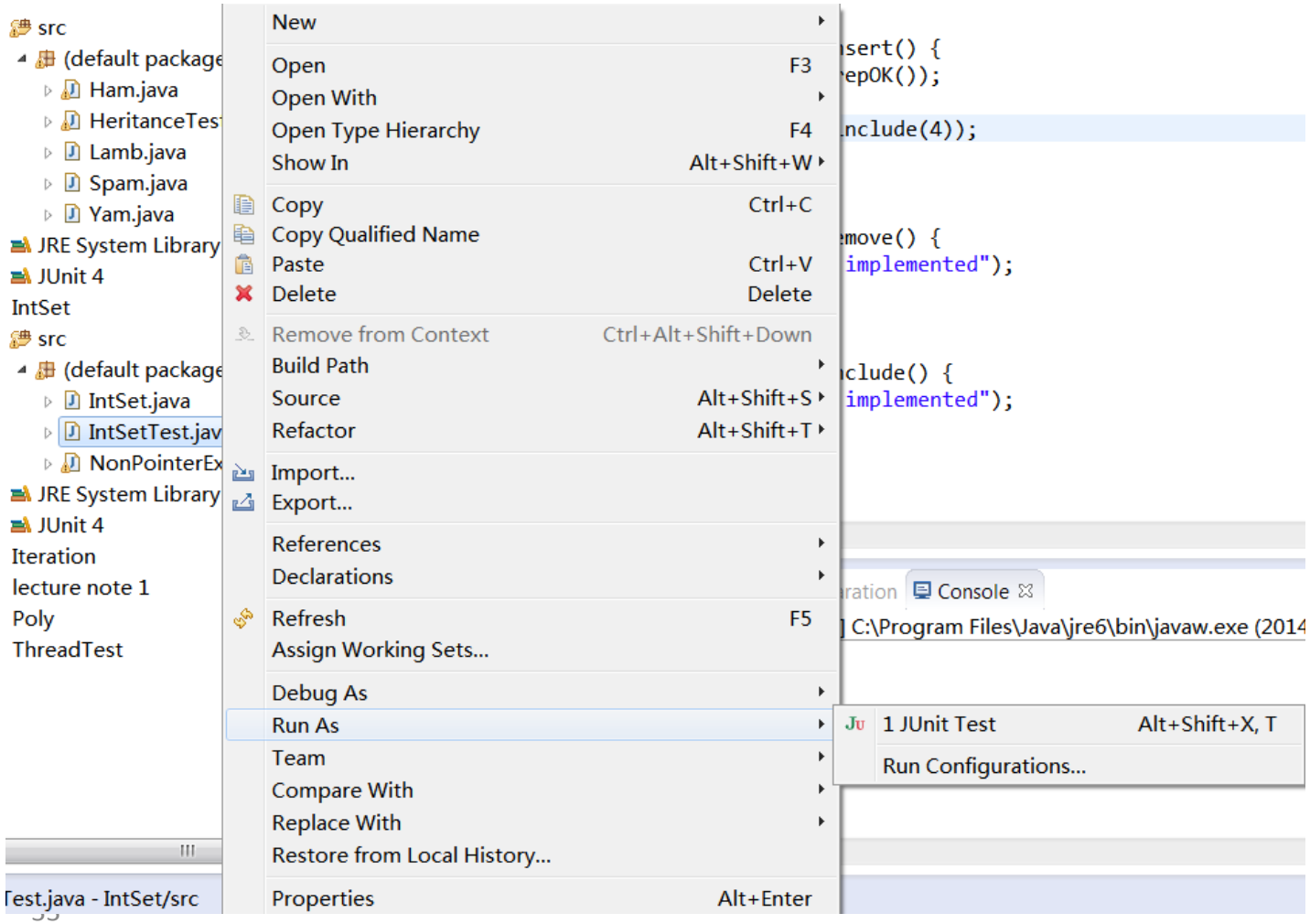
创建测试用例

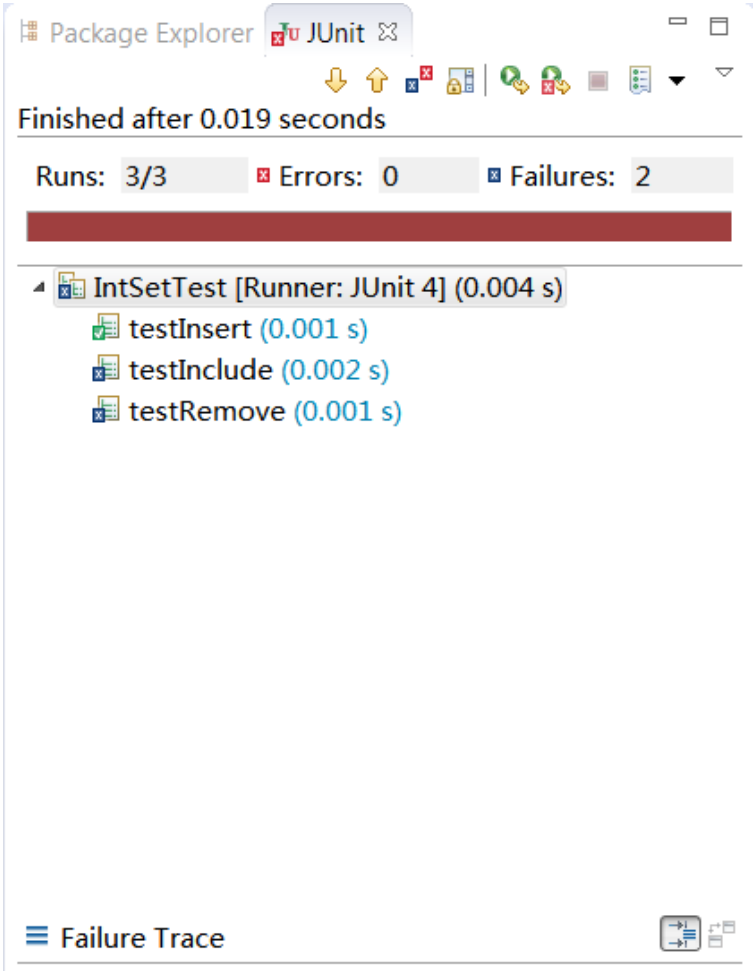
- 可以选择被测试方法
- Junit框架自动生成相应的@Test方法





运行测试


src
(default package
Ham.java
HeritanceTes
Lamb.java
Spam.java
Yam.java
JRE System Library
JUnit 4
IntSet
src
(default package
IntSet.java
IntSetTest.jav
NonPointerEx
JRE System Library
JUnit 4
Iteration
lecture note 1
Poly
ThreadTest
New
Open F3
Open With
Open Type Hierarchy F4
Show In Alt+Shift+W
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Remove from Context Ctrl+Alt+Shift+Down
Build Path
Source Alt+Shift+S
Refactor Alt+Shift+T
Import...
Export...
References
Declarations
Refresh F5
Assign Working Sets...
Debug As
Run As
Team
Compare With
Replace With
Restore from Local History...
Properties Alt+Enter
1 JUnit Test Alt+Shift+X, T
Run Configurations...
Console
C:\Program Files\Java\jre6\bin\javaw.exe (2014
Test.java - IntSet/src


Package Explorer
JUnit
Finished after 0.019 seconds
Runs: 3/3
Errors: 0
Failures: 2
IntSetTest [Runner: JUnit 4] (0.004 s)
testInsert (0.001 s)
testInclude (0.002 s)
testRemove (0.001 s)
Failure Trace

作业

- 针对支持捎带的单电梯作业（第三次作业）
  - 补充电梯类、调度类、请求队列类的规格
  - 使用Junit4来设计和实现测试
    - 按照规格来设计测试场景和数据组合，并根据规格来进行测试判定
    - 电梯类、调度类、请求类每个方法的语句覆盖100%，分支覆盖率>=85%，超出部分获得相应加分
    - 要求能够复现第三次作业被报告的bug
- 提交
  - JSF规格文档、Junit4测试程序、测试覆盖率报告、bug报告（区分哪些是复现的，那些是新发现的）
- 测试检查
  - 根据所提供规格来设计测试用例和进行测试判定，如果相应的bug未在bug报告中，则直接报告出来，测试者获得相应的加分，被测者失去响应的分数
    - 要求提交补充的测试代码和相应的运行结果，按照wrong类型报告bug