1、单元测试853769620

针对的是代码中的每个类，一个类认为是一个代码组件，对每个代码组件都编写一个单元测试类。这个单元测试类中会有多个方法，通常至少对要测试的类中的每个方法都编写一个对应的测试方法。

单元测试，面向的是代码组件的级别，它是最小最细粒度的测试单元

通常而言，通过单元测试检查出来的bug是最多的，所以它是位于测试金字塔模型的最底端

单元测试，需要每个RD对自己编写的代码自己去写单元测试，一般是使用JUnit框架，如果你使用了一些其他的框架，Spring，Spring MVC，都有对应的跟JUnit整合起来进行单元测试的一些框架，逻辑判断的Hamcrest框架，mock对象的Mockito框架。

初步的一些规范：

（1）单元测试类，必须以test来结尾，以要测试的类来打头

（2）针对每个类，一般都要写一个单元测试类来进行测试

（3）单元测试中的每个方法，针对的是被测试类中的每个方法，方法以test打头，跟上要测试的方法名称

下面给出单元测试的一个基本的例子：

public class HelloWorld {

   public String sayHello(String name) {

       return "hello, " + name;

  }

}

public class HelloWorldTest {

   private HelloWorld helloWorld = new HelloWorld();

   @Test

   public void testSayHello() {

       String name = "leo";

       String result = helloWorld.sayHello(name);

       assertEquals("hello, leo", result);

  }

}

下面是使用测试替身的一个例子

public class HelloWorld {

   private GreetingMessageService greetingMessageService;

   public String sayHello(String name) {

       greetingMessageSerivce.save(name);

       return "hello, " + name;

  }

   public void setGreetingMessageService(GreetingMessageService greetingMessageService) {

       this.greetingMessageService = greetingMessageService;

  }

   public GreetingMessageService getGreetingMessageService() {

       return greetingMessageService;

  }

}

public class GreetingMessageServiceImpl implements GreetingMessageService {

   private GreetingMessageDAO greetingMessageDAO;

   public void save(String name) {

       greetingMessageDAO.save(name);

  }

}

重要的原则：单元测试中，你测试一个类，就只能针对这一个类中的代码来进行测试。如果这个类依赖了其他的类，必须用测试替身将依赖的类和要测试的类隔离开来。为什么呢？记住，单元测试是什么，单元测试其实是最小的测试单位，一个单元测试类就负责测试一个代码组件（一个类），如果这个类依赖了其他的类。那么你就自己模拟一些测试替身，注入到要测试的类中去，将要测试的类和依赖的类隔离开来。

避免说，要测试的类和依赖的类耦合在一起测试，互相如果有bug会互相影响

public class GreetingMessageServiceStub implements GreetingMessageService {

   public void save(String name) {

       System.out.println("received a message: " + name);

  }

}

public class HelloWorldTest {

   private HelloWorld helloWorld;

   @Before

   public void setup() {

       this.helloWorld = new HelloWorld();

       GreetingMessageService greetingMessageService = new GreetingMessageServiceStub();

       this.helloWorld.setGreetingMessageService(greetingMessageService);

  }

   @Test

   public void testSayHello() {

       String name = "leo";

       String result = helloWorld.sayHello(name);

       assertEquals("hello, leo", result);

  }

   @After

   public void teardown() {

       this.helloWorld = null;

  }

}

fake对象的示例

public class UserDAOFake implements UserDAO {

   private Map<Long, User> mockDB = new HashMap<Long, User>();

   public void save(User user) {

       mockDB.put(user.getUserId(), user);

  }

   public void update(User user) {

       mockDB.put(user.getUserId(), user);

  }

   public void remove(Long userId) {

       mockDB.remove(userId);

  }

   public User getUserById(Long userId) {

       return mockDB.get(userId);

  }

}

spy对象的示例

public class GreetingMessageServiceSpy implements GreetingMessageService {

   private String \_name;

   public void save(String name) {

       this.\_name = name;

       System.out.println("received a message: " + name);

  }

   public boolean received(String name) {

       return \_name.equals(name);

  }

}

public class HelloWorldTest {

   private HelloWorld helloWorld;

   @Before

   public void setup() {

       this.helloWorld = new HelloWorld();

       GreetingMessageService greetingMessageService = new GreetingMessageServiceSpy();

       this.helloWorld.setGreetingMessageService(greetingMessageService);

  }

   @Test

   public void testSayHello() {

       String name = "leo";

       String result = helloWorld.sayHello(name);

       assertEquals("hello, leo", result);

       GreetingMessageServiceSpy greetingMessageService =

          (GreetingMessageServiceSpy)helloWorld.getGreetingMessageService();

       assertTrue(greetingMessageService.received(name));

  }

   @After

   public void teardown() {

       this.helloWorld = null;

  }

}

mock对象的伪代码示例

public class HelloWorldTest {

   private HelloWorld helloWorld;

   @Before

   public void setup() {

       this.helloWorld = new HelloWorld();

       GreetingMessageService greetingMessageSerivce = mock(GreetingMessageSerivce.class);

       when(greetingMessageSerivce.save("leo")).thenReturn("success");

       this.helloWorld.setGreetingMessageService(greetingMessageService);

  }

   @Test

   public void testSayHello() {

       String name = "leo";

       String result = helloWorld.sayHello(name);

       assertEquals("hello, leo", result);

  }

   @After

   public void teardown() {

       this.helloWorld = null;

  }

}

重要原则：单元测试绝对不能依赖任何的外部基础设施，比如说mysql、redis、rabbitmq，绝对不能依赖这些东西来写单元测试。如果要依赖的话，直接注入测试替身，用模拟的行为来替代掉。

重要规范：单元测试覆盖率，要保证你写的单元测试覆盖了足够多的代码，保证覆盖的代码至少达到70%，对核心模块的代码要覆盖100%

一个好的单元测试要符合的一个规范和规则

1. 什么是单元测试？
   1. 直接裸奔不写单元测试的弊端
      1. 如果裸奔不写单元测试的话，就直接意味着，这个RD根本就没有对自己编写的代码负责
      2. 最基本的单元测试都没有，凭什么让人相信你的代码是经过测试的呢？
   2. 写更多的单元测试对代码质量提高的稳定曲线
   3. 写更多的单元测试对代码设计质量提高的稳定曲线
2. 优秀的单元测试有哪些特质？
   1. 可读性
   2. 良好的测试代码结构
   3. 精准的测试名称
   4. 测试代码的可重复性
   5. 自动化单元测试的3大工具
      1. 测试框架：JUnit来测试，加上其他一些辅助性质的测试框架
      2. 运行单元测试的自动化构建：基于maven的插件，mvn test，自动把所有的单元测试都自动化跑一遍
      3. 测试替身：主要使用的是Mockito框架，模拟出来各种各样实现某个接口的类，每个方法的模拟的行为是什么
3. 测试替身
   1. 测试替身的几大作用
      1. 隔离要测试的代码组件：将测试替身对象传入要测试的组件
      2. 加快测试执行速度：替换复杂耗时的代码执行
         1. 有的时候会有这样的一种情况，比如你要测试的是类A，类A依赖了类B，类B中的代码非常复杂，执行了大量的数据库读写操作
         2. 导致类B的那个方法跑起来是很慢的
         3. 就是用测试替身代替类B，将类B中的方法实现为非常简单的一些打印日志的模拟的实现就可以了，因为我们重点要测试的是类A，不是类B
         4. 测试替身的一个效果，就是可以加速类A中的方法的执行速度，也就可以加快我们的单元测试运行的速度
      3. 让代码行为变得足够稳定：将随机行为改为固定行为
         1. 类A依赖了类B，类B中会随机生成一个随机数，根据随机数来执行一些行为
         2. 这就会导致类A每次调用类B，执行的结果是随机的，不够稳定
         3. 用测试替身替代掉类B，每次调用就是返回一个固定的值，就可以了，让代码行为变得足够稳定
         4. 就能保证我们的单元测试每次都是稳定运行的
      4. 模拟特殊的异常情况：主动抛出网络调用异常
         1. 类A依赖了类B，现在需要测试的是，如果类B的方法抛出了异常，类A捕捉到异常之后，能否正常执行类A中的catch中的代码逻辑
         2. 此时可以用测试替身替代掉类B，让类B中的方法就固定抛出一个异常
      5. 访问私有数据：测试后获取依赖组件的内部私有数据来断言
         1. 类A依赖了类B，现在测试是要看那一下，类A调用了类B之后，类B中的部分变量是否改变为了我们期望的一个值
         2. 此时就可以用测试替身来替代掉这个类B，在里面增加捕获变量改变历史的一些数据
         3. 最后测试完了以后，我们就可以看一下是否符合期望
   2. 测试替身的几种类型
      1. 测试桩：stub
         1. 最简单的测试替身
         2. 一般就是用一两行代码模拟行为
         3. 比如日志记录组件，可以用测试桩来替代
      2. 伪造对象：fake
         1. 相对复杂一些的情况，需要使用伪造对象来模拟真实场景
         2. 比如DAO组件可以用fake对象
         3. 使用内存数据库来实现数据的增删改查
      3. 测试间谍：spy
         1. 要检查依赖组件的行为，是无法直接断言的，需要使用spy
         2. 将依赖组件做成spy，对方法调用可以保存数据
         3. 对测试组件执行完之后，对spy保存的数据进行断言
      4. 模拟对象：mock
         1. mock对象，指定某个参数传入时返回什么结果
         2. 使用mock对象可以模拟任意复杂度的依赖代码组件
   3. 测试替身的使用规则
      1. 如何选择测试替身
         1. 操作行为用mock对象
         2. 查询行为用stub对象
         3. 复杂的一套业务逻辑用fake对象
         4. 无法断言返回结果用spy对象
      2. 单元测试编写范式：准备、执行、断言
      3. 测试框架：JUnit+Mockito+Hamcrest
         1. 单元测试主要用JUnit
         2. 用Mockito来模拟4种测试替身
         3. 用Hamcrest来配合assertThat断言
4. 单元测试的反面教材
   1. 可读性较差
      1. 基本断言：断言时用了算数运算符，让人看不懂
         1. assertTrue(a > 100)
         2. assertThat(a, greaterThan(100))，Hamcrest框架
      2. 过度断言：对大量的代码逻辑的执行结果进行一次断言，出问题不好排查
         1. 一次性对多个方法进行了测试和断言
         2. 要将对多个方法的测试拆分到不同的测试方法中，尽量降低测试的粒度
      3. 位断言：断言时使用了按位与等特殊操作符号，让人看不懂
         1. 在写单元测试的时候，不要使用位运算符，比如说<<，>>
      4. 人格分裂：将较为复杂的多个不同类别的逻辑放在一个测试类里
         1. 在一个测试类中，对多个不同的类，或者一个类中多种不同的负责逻辑进行测试
         2. 可以尝试将过于复杂的测试逻辑拆分到不同的测试类中去
      5. 逻辑分割：测试代码依赖了外部文件中的内容，很难查看
         1. 测试代码里面，使用了外部的一个文件，比如说a.txt，或者b.csv
         2. 尽量用内嵌在代码中的文本去替代外部的文件
         3. 保证说所有的内容都在单元测试中
      6. 魔法数字：断言时用了magic number，让人看不懂
         1. assertTrue(a > 100)，100，magic number
         2. assertThat(productPrice, greaterThan(price(100)))
      7. 冗长安装：setup方法中代码过于冗长，不拆分一下让人不好看懂
         1. @Before的setup方法中，将过于复杂的逻辑拆分到不同的私有方法中去执行
      8. 过分保护：一个测试方法中有多个断言，有过分保护之嫌
         1. 太过于谨慎了，一个测试方法中，多个断言，对同一个结果，不同角度反复断言
   2. 可维护性较差
      1. 重复：数字、字符串和类似代码逻辑多次重复，需要抽取
         1. 对于测试代码中一些重复性的数字、字符串，抽取出来，抽取成一个独立的变量，不要有太多重复的东西
      2. 条件逻辑：不要在测试代码中加多个if else逻辑判断
      3. 脆弱的测试：随机性的逻辑导致测试经常运行失败，不够稳定
      4. 文件路径：使用绝对路径导致在别人那里测试运行失败
      5. 沉睡的蜗牛：使用Thread.sleep来控制多线程，导致运行缓慢和随机性
      6. 像素完美：对图像类测试代码，使用了过多像素数字
      7. 参数化混乱：过多使用参数化测试技术，导致测试代码不易于调试
      8. 方法见缺乏内聚：一个类中混杂太多的测试方法和测试类实例
   3. 可信赖程度较差
      1. 注释掉的测试：不要注释测试，而是直接删除掉
      2. 歧义注释：有些注释有歧义，误导人理解，需要重写注释
      3. 永不失败的测试：检查抛出异常的测试，却永远接收不到异常
      4. 轻率承诺：测试代码被注释，没有写断言，都是无意义的测试代码
      5. 降低期望：对代码执行的结果要求很低，轻易就通过测试
      6. 平台偏见：不同平台的测试需要划分到不同的类中，对windows平台的测试用一个类，对unix平台的测试用一个类
      7. 有条件的测试：如果依赖条件来进行测试，那么可能会有问题，不要出现if else
5. 可测试的设计
   1. 什么是可测试的设计？
      1. 模块化设计
      2. SOLID原则
      3. 上下文中的模块化设计
      4. 以测试驱动出模块化设计
   2. 无法测试的问题
      1. 无法实例化某个类
      2. 无法调用某个方法
      3. 无法观察到输出
      4. 无法替换某个协作者
      5. 无法覆盖某个方法
   3. 可测试设计的指南
      1. 避免复杂的私有方法
      2. 避免final方法
      3. 避免static方法
      4. 谨慎使用new创建对象：如果你这里用了new来创建对象，会导致我们没法注入测试替身
      5. 避免在构造函数中包含复杂逻辑
      6. 避免使用单例模式
      7. 使用组合优先于使用继承
      8. 将外部组件封装在自己的组件中
      9. 避免远程服务查找和调用

你开发好你所有的功能代码之后，运行完所有的单元测试之后，跑出来一份单元测试覆盖率的报告，这份报告的话呢，需要保留好，因为在后面，会有一个审查的环节

1、90%的代码

2、100%的分支

3、100%的方法

public class A {

public void a1() {

if() {

} else {

}

}

public void a2() {

if() {

a3();

} else {

}

}

private void a3() {

}

public static class InnerA {

}

}

1. 行覆盖率：70%
2. 分支覆盖率：50%
3. 方法覆盖率：70%