Minilab2 Summary

项目目标：

使用JUnit框架对lab1的RESTfulWebService的代码进行单元测试，并使之自动运行。

步骤：

（1）对minilab1的每个java代码文件新建一个JUnit Test Case；
（2）JUnit Test Case尽可能覆盖每个java文件的每个类；
（3）单个文件测试：对所写的每个Junit Test Case执行“Run as –JUnit Test”；
（4）构建自动测试： 使用Eclipse Marketplace安装Infinitest插件，当修改之后并bulid之后，Infinitest便进行自动测试；
（5）根据测试结果修改代码，再测试，直到被测试的代码完全正确。

遇到问题：

（1）对于比较复杂的代码的测试用例，还不是很熟悉如何写测试类，在以后的训练中多积累，增加经验；
（2）自动测试是使用eclipse的Infinitest插件完成的，但是只能每次提交测试代码之后自动测一次，没有达到任务要求的随机测10次。其他的自动测试方法等有时间了继续摸索研究。

Questions:

1.What is unit testing? Which unit testing framework did you use? What are some of the common Java unit testing frameworks?

单元测试：是指对软件中的最小可测试单元进行检查和验证。对于单元测试中单元的含义，一般来说，要根据实际情况去判定其具体含义，如C语言中单元指一个函数，Java里单元指一个类，图形化的软件中可以指一个窗口或一个菜单等。总的来说，单元就是人为规定的最小的被测功能模块。单元测试是在软件开发过程中要进行的最低级别的测试活动，软件的独立单元将在与程序的其他部分相隔离的情况下进行测试。

经常与单元测试联系起来的另外一些开发活动包括代码走读（Code review)，静态分析（Static analysis)和动态分析（Dynamic analysis)。静态分析就是对软件的源代码进行研读，查找错误或收集一些度量数据，并不需要对代码进行编译和执行。动态分析就是通过观察软件运行时的动作，来提供执行跟踪，时间分析，以及测试覆盖度方面的信息。
单元测试（模块测试）是开发者编写的一小段代码，用于检测被测代码的一个很小的，很明确的功能是否正确。通常而言，一个单元测试是用于判断某个特定条件（或者场景）下某个特定函数的行为。

对于程序员而言，如果养成了对自己写的代码进行但愿测试的习惯，不但可以写出高质量的代码，而且还能提高编程水平。要进行充分的单元测试，应专门编写测试代码，并与产品代码隔离，为产品工程建立对应的测试工程，为每个类建立对应的测试类，为每个函数（除很简单的）建立测试函数。

一般认为，在结构化程序时代，单元测试所说的单元是指函数，在当今的面向对象时代，单元测试所说的单元是指类。以我的实践来看，以类作为测试单位，复杂度高，可操作性较差，因此仍然主张以函数作为单元测试的测试单位，但可以用一个测试类来组织某个类的所有测试函数。单元测试不应过分强调面向对象，因为局部代码依然是结构化的。单元测试的工作量较大，简单实用高效才是硬道理。

测试用例的核心是输入数据。预期输出是依据输入数据和程序功能来确定的，也就是说，对于某一程序，输入数据确定了，预期输出也就可以确定了，至于生成/销毁被测试对象和运行测试的语句，是所有测试用例都大同小异的，因此，我们讨论测试用例时，只讨论输入数据。

前面说过，输入数据包括四类：参数、成员变量、全局变量、IO媒体，这四类数据中，只要所测试的程序需要执行读操作的，就要设定其初始值，其中，前两类比较常用，后两类较少用。显然，把输入数据的所有可能取值都进行测试，是不可能也是无意义的，我们应该用一定的规则选择有代表性的数据作为输入数据，主要有三种：正常输入，边界输入，非法输入，每种输入还可以分类，也就是平常说的等价类法，每类取一个数据作为输入数据，如果测试通过，可以肯定同类的其他输入也是可以通过的。下面举例说明：

正常输入
例如字符串的Trim函数，功能是将字符串前后的空格去除，那么正常的输入可以有四类：前面有空格；后面有空格；前后均有空格；前后均无空格。

边界输入
上例中空字符串可以看作是边界输入。
再如一个表示年龄的参数，它的有效范围是0-100，那么边界输入有两个：0和100。

非法输入
非法输入是正常取值范围以外的数据，或使代码不能完成正常功能的输入，如上例中表示年龄的参数，小于0或大于100都是非法输入，再如一个进行文件操作的函数，非法输入有这么几类：文件不存在；目录不存在；文件正在被其他程序打开；权限错误。

如果函数使用了外部数据，则正常输入是肯定会有的，而边界输入和非法输入不是所有函数都有。一般情况下，即使没有设计文档，考虑以上三种输入也可以找出函数的基本功能点。实际上，单元测试与代码编写是“一体两面”的关系，编码时对上述三种输入都是必须考虑的，否则代码的健壮性就会成问题。
常用的Java unit testing frameworks：

2.Out of Manual and Automated testing which one is better and why?

1. JUnit – Java unit testing framework
2. Cactus
3. StrutsTestCase
4. JFCUnit
5. TestNG
6. jMock
7. Grinder
8. Jetif
9. Unitils
10. p-unit
11. XMLUnit
12. Ejb3Unit
13. FEST-Swing
14. Ripplet
15. Feed4JUnit
16. Jubula functional testing tools
17. JsTestDriver
18. Citrus Testframework
19. JBehave
20. JCrawler – load testing framework
    链接地址：http://www.open-open.com/news/view/106d204

3.Do you have unit testing method for testing timeout? How it works?

如下例，求平方根的函数有Bug，是个死循环：

```
public void squareRoot(int n) {
    for (; ;) ;                 //Bug : 死循环

}
```

如果测试的时候遇到死循环，你的脸上绝对不会露出笑容。因此，对于那些逻辑很复杂，循环嵌套比较深的程序，很有可能出现死循环，因此一定要采取一些预防措施。限时测试是一个很好的解决方案。我们给这些测试函数设定一个执行时间，超过了这个时间，他们就会被系统强行终止，并且系统还会向你汇报该函数结束的原因是因为超时，这样你就可以发现这些Bug了。要实现这一功能，只需要给@Test标注加一个参数即可，代码如下：

```
@Test(timeout = 1000)
public void squareRoot() ...{
        calculator.squareRoot(4);
        assertEquals(2, calculator.getResult());
}
```

Timeout参数表明了你要设定的时间，单位为毫秒，因此1000就代表1秒。

4.What are called as test smells in relation with unit testing?

Multiple assertions within one unit test, long-running unit tests etc

5.What is mocking & stubbing? What are key differences between them? Do you know any mocking framework?

Mock测试就是在测试过程中，对于某些不容易构造或者 不容易获取的对象，用一个虚拟的对象来创建以便测试的测试方法。
mock对象
这个虚拟的对象就是mock对象。mock对象就是真实对象在调试期间的代替品。
mock对象使用范畴
真实对象具有不可确定的行为，产生不可预测的效果，（如：股票行情，天气预报） 真实对象很难被创建的 真实对象的某些行为很难被触发 真实对象实际上还不存在的（和其他开发小组或者和新的硬件打交道）等等.
使用mock对象测试的关键步骤
使用一个接口来描述这个对象 在产品代码中实现这个接口 在测试代码中实现这个接口 在被测试代码中只是通过接口来引用对象，所以它不知道这个引用的对象是真实对象还是mock对象。
目前，在Java阵营中主要的Mock测试工具有JMock，MockCreator，Mockrunner，EasyMock，MockMaker等，在微软的.Net阵营中主要是Nmock，.NetMock等。

相同点

mock和stub都可以用来对系统(或者将粒度放小为模块，单元)进行隔离。

在测试，尤其是单元测试中，我们通常关注的是主要测试对象的功能和行为，对于主要测试对象涉及到的次要对象尤其是一些依赖，我们仅仅关注主要测试对象和次要测试对象的交互，比如是否调用，何时调用，调用的参数，调用的次数和顺序等，以及返回的结果或发生的异常。但次要对象是如何执行这次调用的具体细节，我们并不关注，因此常见的技巧就是用mock对象或者stub对象来替代真实的次要对象，模拟真实场景来进行对主要测试对象的测试工作。

因此从实现上看，mock和stub都是通过创建自己的对象来替代次要测试对象，然后按照测试的需要控制这个对象的行为。

不同点

（1）类实现的方式

从类的实现方式上看，stub有一个显式的类实现，按照stub类的复用层次可以实现为普通类(被多个测试案例复用)，内部类(被同一个测试案例的多个测试方法复用)乃至内部匿名类(只用于当前测试方法)。对于stub的方法也会有具体的实现，哪怕简单到只有一个简单的return语句。

而mock则不同，mock的实现类通常是有mock的工具包如easymock, jmock来隐式实现，具体mock的方法的行为则通过record方式来指定。

以mock一个UserService, UserDao为例，最简单的例子，只有一个查询方法：

```
public interface UserService {
enter code hereUser query(String userId);
}

public class UserServiceImpl implements UserService {
private UserDao userDao;
public User query(String userId) {
return userDao.getById(userId);
}
//setter for userDao
}

public interface UserDao {
User getById(String userId);
}
```

stub的标准实现，需要自己实现一个类并实现方法:

```
public class UserDaoStub implements UserDao {
public User getById(String id) {
User user = new User();
user.set.....
return user;
}
}

@Test
public void testGetById() {
UserServiceImpl service = new UserServiceImpl();
UserDao userDao  = new UserDaoStub();
service.setUserDao(userDao);

User user = service.query("1001");
...
}
```

mock的实现，以easymock为例，只要指定mock的类并record期望的行为，并没有显式的构造新类:

```
@Test
public void testGetById() {
UserDao dao = Easymock.createMock(UserDao.class);
User user = new User();
user.set.....
Easymock.expect(dao.getById("1001")).andReturn(user);
Easymock.reply(dao);

UserServiceImpl service = new UserServiceImpl();
service.setUserDao(userDao);
User user = service.query("1001");
...
Easymock.verify(dao);
}
```

对比可以看出，mock编写相对简单，只需要关注被使用的函数，所谓”just enough”。stub要复杂一些，需要实现逻辑，即使是不需要关注的方法也至少要给出空实现。

（2）测试逻辑的可读性

从上面的代码可以看出，在形式上，mock通常是在测试代码中直接mock类和定义mock方法的行为，测试代码和mock的代码通常是放在一起的，因此测试代码的逻辑也容易从测试案例的代码上看出来。Easymock.expect(dao.getById(“1001”)).andReturn(user); 直截了当的指明了当前测试案例对UserDao这个依赖的预期: getById需要被调用，调用的参数应该是”1001”，调用次数为1(不明确指定调用次数时easymock默认为1)。

而stub的测试案例的代码中只有简单的UserDao userDao = new UserDaoStub ();构造语句和service.setUserDao(userDao);设置语句，我们无法直接从测试案例的代码中看出对依赖的预期，只能进入具体的UserServiceImpl类的query()方法，看到具体的实现是调用userDao.getById(userId)，这个时候才能明白完整的测试逻辑。因此当测试逻辑复杂，stub数量多并且某些stub需要传入一些标记比如true，false之类的来制定不同的行为时，测试逻辑的可读性就会下降。

（3）可复用性
Mock通常很少考虑复用，每个mock对象通过都是遵循”just enough”原则，一般只适用于当前测试方法。因此每个测试方法都必须实现自己的mock逻辑，当然在同一个测试类中还是可以有一些简单的初始化逻辑可以复用。

stub则通常比较方便复用，尤其是一些通用的stub，比如jdbc连接之类。spring框架就为此提供了大量的stub来方便测试，不过很遗憾的是，它的名字用错了：spring-mock！

（4）设计和使用

接着我们从mock和stub的设计和使用上来比较两者，这里需要引入两个概念：interaction-based和state-based。

具体关于interaction-based和state-based，不再本文阐述，强烈推荐Martin Fowler 的一篇文章，”Mocks Aren’t Stubs”。地址为http://martinfowler.com/articles/mocksArentStubs.html (PS:当在google中输入mock stub两个关键字做搜索时，出来结果的第一条就是此文，向Martin Fowler致敬，向google致敬).英文不好的同学，可以参考这里的一份中文翻译：http://www.cnblogs.com/anf/archive/2006/03/27/360248.html。

总结来说，
stub是state-based，关注的是输入和输出。
mock是interaction-based，关注的是交互过程。

（5）expectiation/期望

这个才是mock和stub的最重要的区别：expectiation/期望。

对于mock来说，exception是重中之重：我们期待方法有没有被调用，期待适当的参数，期待调用的次数，甚至期待多个mock之间的调用顺序。所有的一切期待都是事先准备好，在测试过程中和测试结束后验证是否和预期的一致。

而对于stub，通常都不会关注exception，就像上面给出的UserDaoStub的例子，没有任何代码来帮助判断这个stub类是否被调用。虽然理论上某些stub实现也可以通过自己编码的方式增加对expectiation的内容，比如增加一个计数器，每次调用+1之类，但是实际上极少这样做。

（6）总结

关于mock和stub的不同，在Martin Fowler的”Mocks Aren’t Stubs”一文中，有以下结束，我将它列出来作为总结：

Dummy
对象被四处传递，但是从不被真正使用。通常他们只是用来填充参数列表。

Fake
有实际可工作的实现，但是通常有一些缺点导致不适合用于产品(基于内存的数据库就是一个好例子)。

Stubs

在测试过程中产生的调用提供预备好的应答，通常不应答计划之外的任何事。stubs可能记录关于调用的信息，比如 邮件网关的stub 会记录它发送的消息，或者可能仅仅是发送了多少信息。

Mocks

如我们在这里说的那样：预先计划好的对象，带有各种期待，他们组成了一个关于他们期待接受的调用的详细说明。

退化和转化

在实际的开发测试过程中，我们会发现其实mock和stub的界限有时候很模糊，并没有严格的划分方式，从而造成我们理解上的含糊和困惑。

主要的原因在于现实使用中，我们经常将mock做不同程度的退化，从而使得mock对象在某些程度上如stub一样工作。以easymock为例，我们可以通过anyObject(), isA(Class)等方式放宽对参数的检测，以atLeatOnce(),anytimes()来放松对调用次数的检测，我们可以使用Easymock.createControl()而不是Easymock.createStrictControl()来放宽对调用顺序的检测(或者调用checkOrder(false))，我们甚至可以通过createNiceControl(), createNiceMock()来创建完全不限制调用方式而且自动返回简单值的mock，这和stub就几乎没有本质区别了。

目前大多数的mock工具都提供mock退化为stub的支持，比如easyock中，除了上面列出的any***,NiceMock之外，还提供诸如andStubAnswer(),andStubDelegateTo(),andStubReturn(),andStubThrow()和asStub()。

上面也谈到过stub也是可以通过增加代码来实现一些expectiation的特性，stub理论上也是可以向mock的方向做转化，而从使得两者的界限更加的模糊。

6.What is test-driven development (TDD)?

测试驱动开发，英文全称Test-Driven Development，简称TDD，是一种不同于传统软件开发流程的新型的开发方法。它要求在编写某个功能的代码之前先编写测试代码，然后只编写使测试通过的功能代码，通过测试来推动整个开发的进行。这有助于编写简洁可用和高质量的代码，并加速开发过程。

测试驱动开发的基本思想就是在开发功能代码之前，先编写测试代码，然后只编写使测试通过的功能代码，从而以测试来驱动整个开发过程的进行。这有助于编写简洁可用和高质量的代码，有很高的灵活性和健壮性，能快速响应变化，并加速开发过程。

测试驱动开发的基本过程如下：

① 快速新增一个测试
② 运行所有的测试（有时候只需要运行一个或一部分），发现新增的测试不能通过
③ 做一些小小的改动，尽快地让测试程序可运行，为此可以在程序中使用一些不合情理的方法
④ 运行所有的测试，并且全部通过
⑤ 重构代码，以消除重复设计，优化设计结构

简单来说，就是不可运行/可运行/重构——这正是测试驱动开发的口号。

举个比较生动的例子，这个例子你一定已经在很多关于TDD的文献资料上都看到过，但它确实是一个不错的比喻。在此我进行了一些加工和扩展。

盖房子的时候，工人师傅砌墙，会先用桩子拉上线，以使砖能够垒的笔直，因为垒砖的时候都是以这根线为基准的。TDD就像这样，先写测试代码，就像工人师傅先用桩子拉上线，然后编码的时候以此为基准，只编写符合这个测试的功能代码。
而一个新手或菜鸟级的小师傅，却可能不知道拉线，而是直接把砖往上垒，垒了一些之后再看是否笔直，这时候可能会用一根线，量一下砌好的墙是否笔直，如果不直再进行校正，敲敲打打。使用传统的软件开发过程就像这样，我们先编码，编码完成之后才写测试程序，以此检验已写的代码是否正确，如果有错误再一点点修改。

优势：

1) TDD根据客户需求编写测试用例，对功能的过程和接口都进行了设计，而且这种从使用者角度对代码进行的设计通常更符合后期开发的需求。因为关注用户反馈，可以及时响应需求变更，同时因为从使用者角度出发的简单设计，也可以更快地适应变化。
2) 出于易测试和测试独立性的要求，将促使我们实现松耦合的设计，并更多地依赖于接口而非具体的类，提高系统的可扩展性和抗变性。而且TDD明显地缩短了设计决策的反馈循环，使我们几秒或几分钟之内就能获得反馈。
3) 将测试工作提到编码之前，并频繁地运行所有测试，可以尽量地避免和尽早地发现错误，极大地降低了后续测试及修复的成本，提高了代码的质量。在测试的保护下，不断重构代码，以消除重复设计，优化设计结构，提高了代码的重用性，从而提高了软件产品的质量。
4) TDD提供了持续的回归测试，使我们拥有重构的勇气，因为代码的改动导致系统其他部分产生任何异常，测试都会立刻通知我们。完整的测试会帮助我们持续地跟踪整个系统的状态，因此我们就不需要担心会产生什么不可预知的副作用了。
5) TDD所产生的单元测试代码就是最完美的开发者文档，它们展示了所有的API该如何使用以及是如何运作的，而且它们与工作代码保持同步，永远是最新的。
6) TDD可以减轻压力、降低忧虑、提高我们对代码的信心、使我们拥有重构的勇气，这些都是快乐工作的重要前提。
7)快速的提高了开发效率。