Mock 七宗罪

天虹大酒店2层小会议室门前，我看了看表1点59分。敏捷大会讲师论坛，下午我第一个发言，题目是“你用错了Mock”。我走进会议室的一瞬间，清晰地感觉到了各位讲师摩拳擦掌准备把我批个体无完肤。“在我开始发言之前，我想强调一下我不是针对在座的哪一位。”我说道，“我是说我们每个人都用错了。”

我做培训的时候经常在开始玩一个自创的游戏。首先我播放一段视频，里面我用手指有节奏的敲桌子。然后让听众猜我敲击的节奏是来自于那首歌。听众被分成两组，一组完全盲听，第二组知道这首歌是《祝你生日快乐》、《小二郎》、《卖报歌》中的一首。我做过很多次实验，第二组大概有80%以上的几率猜对，第一组则从来没有猜对。在被告知答案之后，第一组的人也觉得很明显。

通过这个游戏，我想告诉我的听众，在我的培训中你不会学到任何新的东西，我只是唤醒你已有知识和思考。同时，每个人从中得到的也是不一样的，这取决于你已有的知识结构和思考方式。

为什么要花那么大篇幅引入今天的主题呢？因为这是一个反直觉的主题。如果我们没在一个频道上，这将是一场互相的伤害。如果你完全没有用过Mock，你几乎什么也不会得到。

背景

在下面的论述中，我们尽可能使用同一个案例^。我们所列举的每一个情形都是来自于我本人参加的开发项目或者咨询项目。为了便于理解和陈述，我们把它映射到这个案例上面。

^该案例取自Martin Fowler先生的Mocks Aren’t Stubs。

我们有一个订单（Order）对象，该对象从仓库（Warehouse）对象中取出产品（Product）。订单对象仅包含一个产品对象和该产品的数量。仓库对象中包含多种不同的产品及其数量。当订单对象试图从仓库中取出产品并填充(fill)自己的时候，有两种结果。如果仓库中有足够的产品，订单填充成功，同时仓库中产品减少相应的数量。否则订单填充失败，并且仓库保持不变。

```
//产品代码（仅Order类）
public class Order {
    private final String product;
    private final int quantity;
    private boolean filled;
```

```
    public Order(String product, int quantity) {
        if (quntity <= 0) throw new InvalidArgumentException("The
quantity shall not equal or be less than 0.");
        this.product = product;
        this.quantity = quantity;
    }

    public void fill(Warehouse warehouse) {
        if(warehouse.hasInventory(quantity)) {
            warehouse.remove(quantity);
            filled = true;
        }
    }

    public boolean isFilled() {
        return filled;
    }
}
```

第一部分：Mock测试的缺陷

在开始谈缺陷之前，我们先来看一下Mock测试主要解决什么问题？

以下内容来自Wiki，括号中的注释略有修改：

如果一个对象具有以下特征，比较适合使用mock对象：

- 该对象提供非确定的结果（比如当前的时间或者当前的温度）
- 对象的某些状态难以创建或者重现（比如网络错误或者文件读写错误）
- 对象方法上的执行太慢（比如在测试开始之前初始化数据库）
- 该对象还不存在或者其行为可能发生变化（比如测试驱动开发中驱动创建新的类）
- 该对象必须包含一些专门为测试准备的数据或者方法（后者不适用于静态类型的语言，流行的Mock框架不能为对象添加新的方法。Stub是可以的。）

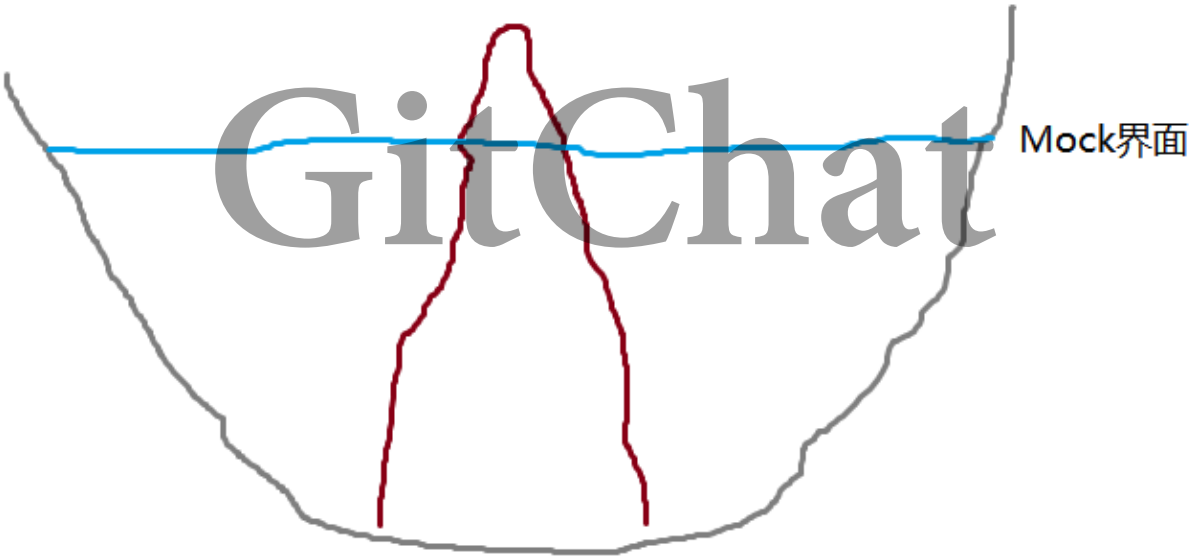
当我们讲到缺陷的时候，一定是说它相对于其他的解决方案。这里我们对比的是直接使用真实对象（Object）的测试，比如不使用Mock的单元测试和集成测试。

一个典型的Mock测试是这样的：

```
//测试代码
public class OrderTest {

    private static String TALISKER = "Talisker";
```

```
    @Test
    public void testFillingRemovesInventoryIfInStock() {
        //设置数据
        Order order = new Order(TALISKER, 50);
        Warehouse warehouse = mock(Warehouse.class);
        when(warehouse.hasInventory(50)).thenReturn(true);

        //执行被测逻辑
        order.fill(warehouse);

        //验证
        verify(warehouse, times(1)).hasInventory(50);
        verify(warehouse, times(1)).remove(TALISKER, 50);
        assertThat(order.isFilled(), is(true));
    }
}
```

被测对象，也经常称为SUT（ System Under Test ），为真实对象，而其依赖则被完全隔离。



借用精益方法中常用的湖水岩石的比喻：岩石代表被测对象及其依赖，水面代表Mock界面。水面以下的部分都被Mock掩盖了。然而，我们的测试是应当确保整个系统正常工作的。由于Mock框架强大的灵活性，这个水面几乎可以任意设置。即便是纪律性非常好的团队，也很难在这个问题上做到一致。比如可以Mock Repositories，也可以Mock Service，甚至Mock一个简单的POJO。这时候确实水面以上的部分被测试覆盖到了，水面以下的部分呢？

1. 阻碍重构

重构是敏捷开发最重要最基本最常用的实践之一。但是Mock测试却经常给重构带来的障碍。

这个测试现在是通过的。考虑到 fill 方法中关于 hasInventory 的检查是属于 Warehouse 的细节，我们尝试把它移动到 Warehouse 类中。

```
// Order class
public void fill(Warehouse warehouse) {
    filled = tryRemove(warehouse);
}

// Warehouse class
public boolean tryRemove(String product, int quantity) {
    //这个判断可以简化，但是为了说明问题，保持原来的调用逻辑
    if(hasInventory(quantity)) {
        remove(product, quantity);
        return true;
    }
    return false;
}
```

重构的基本原则就是对外表现的行为不变，那么相应的，在做重构的时候我们不应当需要修改测试。完成这个重构之后，你会发现测试失败了。即便是我们在 tryRemove 中调用了一样的逻辑，对于被测试它也是不可见的。因为要重构所以要修改测试绝对不是什么好主意，你怎么知道你没有修改行为呢？

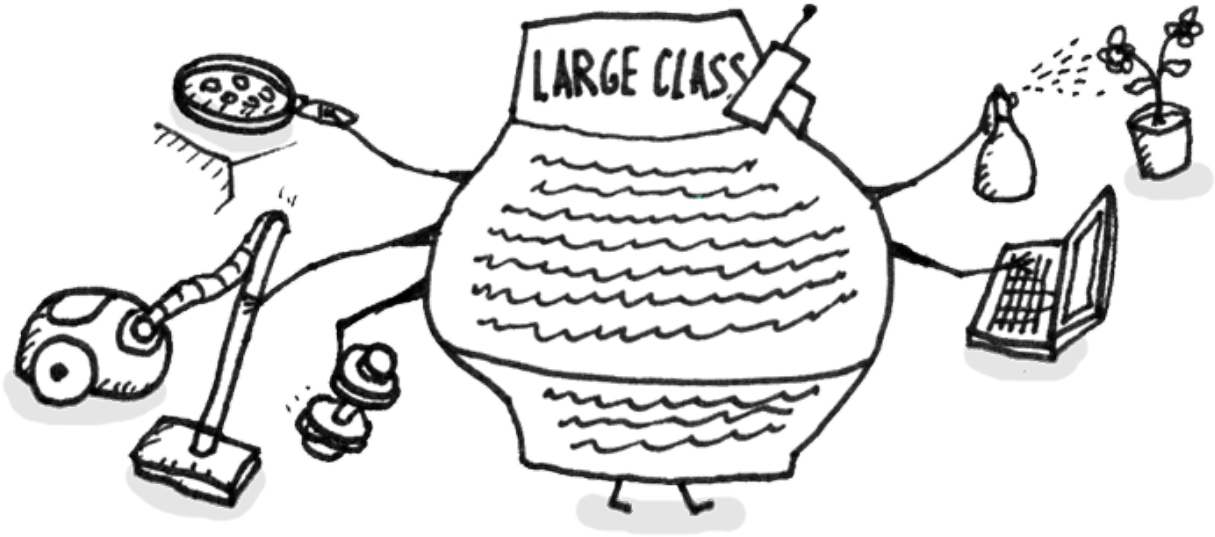
实际上有几种重构手法是天然的跟Mock测试水火不容的。比如 inline，Mock 说“嗨，我来设置这个函数的行为”；inline 说“不用了，我已经把它消除了”。类似的重构还有在类之间迁移方法、上推或者下移方法等等。基本上除了重命名这类简单的重构，你会发现你的实现被绑死在测试上了。

在我看来这一条罪状足够了，只要我还能找到其他替代的方式（第二部分介绍），我就不会用Mock了。

2.掩盖坏味道

Mock不仅会阻碍重构本身，同时也会提高团队对于坏味道的容忍程度。Mock的根本驱动力是构建实际对象的困难。而构建实际对象的困难往往本身就意味着某种坏味道。

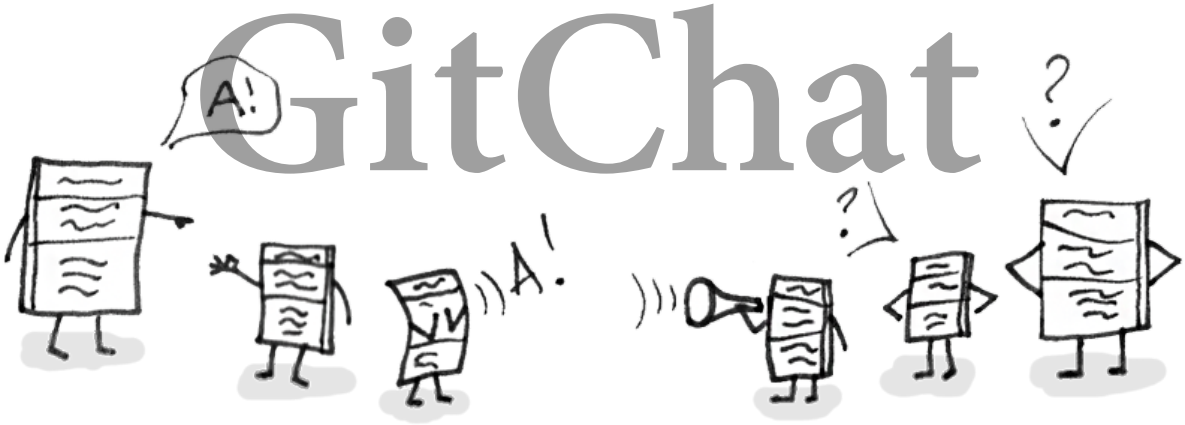
常见的情形是，一个类有过多的依赖。



图片来自于https://sourcemaking.com/

当你要在测试中创建这样的类的时候你会发现仅仅构建这样一个类的实例就是很复杂的。所以，不如用到哪个方法就Mock哪个方法。于是，对于违反了单一职责原则的对象团队往往不能在测试时感受到对象创建的困难。也就缺乏动力进行重构。

另一种情形是过长的调用链。



图片来自于https://sourcemaking.com/

对于过长的调用链，Mock方案是非常具有吸引力的（想象一下在 `remove` 方法中有一长串的 `getA().fetchB().buildC().delete()` ）。因为你可以只Mock第一级调用（`remove`），测试写起来是不是容易多了。

在一个恰当的设计中，每一级调用应当是其实现细节的抽象。当我们使用过长的调用链的时候实际上是把下一级的实现细节暴露了。我的原则是把对框架（或者库）函数的调用定义为边界，任何一个方法到达边界的距离应当尽可能的短（3~4步是我的上限）。过深的的调用意味着对下层函数进行修改的时候涟漪效果会更加明显。

这个时候我们应当做的是改善设计而不是用Mock掩盖错误。

3.测试实现细节

等等，单元测试不就是测试细节的吗？[捂脸哭]GitChat对于表情符号支持不太完善，请自动脑补捂脸哭的符号。

是的，单元测试是测试细节的，但是测试测试的是业务的细节，而不是实现的细节。还是以上文中的代码为例， `fill` 方法到底是调用了 `Warehouse` 的哪个方法就是实现细节。

从测试的目的看，任何测试都应该是行为测试或者业务逻辑测试。即我们测试的是系统或者组件有没有按照期望的方式返回结果。至于这个结果是怎么产生的不应当是测试负责验证的事情。

值得指出的是，业务逻辑是层层下放的，也就是上一层的所有业务细节，一定在下一层有支撑。而一般情况下每一个方法一定是应对于一个业务需求（粒度大小不同）。

从投入产出比来看，为什么不要测试实现细节呢？实现细节变化快于其实现的业务变化频率，测试成本高；同时，因为测试没有到达“边界”，实际上我们获得的信心是有限的。使用Mock测试的时候，我们经常会发现测试基本上是在重复实现的逻辑（比如上文中的测试，你必须知道实现的时候使用了那个方法，才能对那个方法进行Mock）。

4.漏测关键逻辑

案例背景：我们需要计算当天的 `Order` 的总数量。我们为Order类增加一个日期field，并创建 `OrderService` 类来实现计算。

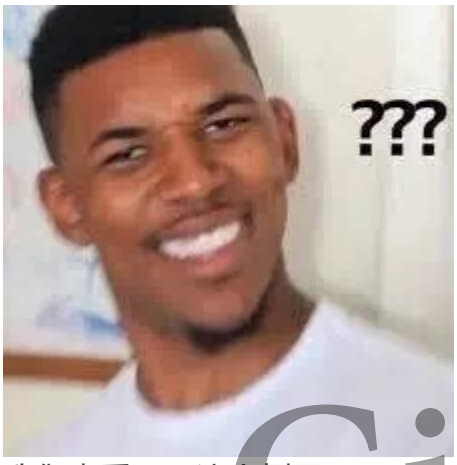

```
@Test
public void sum_total_amount_of_orders_for_today(){
    // today, yesterday, tomorrow
    Order yesterdayOrder = new Order(TALISKER, 100, yesterday);
    Order todayOrder1 = new Order(TALISKER, 200, today);
    Order todayOrder2 = new Order(TALISKER, 600, today);
    Order tomorrowOrder = new Order(TALISKER, 800, tomorrow);


when(orderReporsitory.findByDate(eq(today))).thenReturn(ArrayList
.asList(todayOrder1, todayOrder2));
    // orderReporsitory通过依赖注入的方式注入到OrderService中
    assertThat(orderService.getTodaysAmount(), is(800)));
}

public class OrderService {

    Integer getTodaysAmount() {
        Date today = new Date();
        List orders = orderReporsitory.findByDate(today);
        // 对orders的数量进行求和并返回
        ...
    }
}
```

由于我们在数据库中保存的时间是不带时区的，而在 orderService 中的 today 却是根据操作系统的时区得到的。这就导致搜索得到的结果是跟系统运行的时区设置有关。这是一个非常常见的测试错误，表面上我们测试了 getTodaysTotal 的核心逻辑。但是实际上，操作系统时区这个关键逻辑被屏蔽掉了。

任何时候都要小心，被Mock的对象其行为未必跟我们预期的一致。

5.测试不存在的逻辑

怎么可能测到不存在的逻辑！我隔着屏幕都能感觉到你的疑惑。



我们来看下面这个例子：


```
@Test(expect = IllegalStateException.class)
public void throw_exception_if_an_order_has_no_product_set(){
    Destination mockedOrder = mock(Order.class);
    when(mockedOrder.getQuantity().thenReturn(0);
    ordreService.validate(mockedOrder);
}

public class OrderService {
    void validate(Order order) {
        if (order.getQuantity() <= 0) {
          throw new IllegalStateException("Order has invalid
quantity set.")
        }
    }
}
```

看上去没什么问题，可是事实上 Order 的构造函数可能就已经保证了任何成功构建的实例其 quantity 都不会为0。如果你使用的是真实的对象，是根本不可能犯这个错误的。

6.测试可读性差

应当承认现在的Mock框架的语法已经变得比几年前要好得多了。在jMock时代，被方法名还是要以字符串的形式出现。感受一下这个：

```
warehouseMock.expects(once()).method("hasInventory")
      .with(eq(TALISKER),eq(50))
      .will(returnValue(true));
```

新的语法仍然有很多奇怪的限制。比如，这个是不合法的 `verify(warehouse, times(1)).remove(anyString(), 50);` ，必须写成 `verify(warehouse, times(1)).remove(anyString(), eq(50));`。

另外，你有没有想过下面三种方式有没有差别：

```
//方式1
when(warehouse.hasInventory(50)).thenReturn(true);
//方式2
boolean methodCall = warehouse.hasInventory(50);
when(methodCall).thenReturn(true);
//方式3
boolean methodCall = warehouse.hasInventory(50);
warehouse.hasInventory(30);
when(methodCall).thenReturn(true);
```


我是直到看了Mockito的源代码才明白为啥方式1和方式2等价，方式3会失败。我建议你也去看一下它的实现，确实非常巧妙，但是也增加了很多限制。你可以说这些都是为了提高可读性，但是它增加了很多需要记忆的东西。

7.掩盖性能缺陷

案例：下面这段代码，首先根据 `country` 从缓存中取得相关的订单的ID，然后取出所有订单并计算总金额。

```
public Integer amount getTotal(String country) {
    List<Id> ids = orderCacheService.getAllIds(country);
    Integer total = 0;
    for(Id id : ids) {
        total += orderReporsitory.getOrder(id).getAmount();
    }
    return total;
}
```

虽然我们在测试中包含了对于100K+ ID的测试（当时是为了测试缓存），后来我们发现这个函数在线上平均执行时间为~2000秒（大于半个小时）。而、在测试中由于我们Mock了 `getOrder` 方法，忽视了数据库本身的性能。

第二部分：不用Mock怎么写（单元）测试

我合作过的很多团队，特别是敏捷团队，都对Mock情有独钟。有些团队甚至达到了无Mock不UT的地步。2013开始我在团队中开始推行去Mock化，并取得了不错的效果。下面是我总结的一些经验。

1.消除“单元”情结

要在团队中推行去Mock，就要消除“单元”情结，或者换个角度看待“单元”这个概念。这往往是一个长期的过程，但是第一部分应该给你提供了大量的子弹。

Mockist（Mock主义分子）往往极端重视单元的隔离性。

> A mockist TDD practitioner, however, will always use a mock for any object with interesting behavior.
> ——Martin Fowler

在他们看来单元测试就是测试一个类，甚至是一个方法。所有其他的因素全都应当屏蔽掉。这不仅在原则上是错误的，第一部分也论证了在实践上也是不可能的。正确的测试单元应当是一个业务逻辑单元，比如：

- 用户必须提供正确的邮件地址才能注册
- 用户的默认地址可以被当前地址覆盖
- 用户无法取消正在执行中的工作流

你的测试本身，而不仅是测试的方法名，应该是对于非开发人员也尽量是可读的。

```
@Before
public void setUp() throws Exception {
   warehouse.add(TALISKER, 50);
   warehouse.add(HIGHLAND_PARK, 25);
 }

 @Test
 public void testOrderIsFilledIfEnoughInWarehouse() {
   Order order = new Order(TALISKER, 50);
   order.fill(warehouse);
   assertThat(order.isFilled(), is(true));
   assertThat(warehouse.getInventory(TALISKER), is(0));
 }
```

你会发现有些代码会在不同的测试中被重复执行，比如对邮件地址的格式可能经常被不同的测试覆盖到。这会带来一定的测试效率降低，但是实际开发中因为对象的创建而导致测试效率下降的情况少之又少。

2.文件、网络和数据库

当我们谈起什么时候“应当”使用Mock的时候，IO及其相关操作往往是最先被提出来的。这是因为牵扯到IO的操作往往一方面对象的创建成本高（比如DB connection），另一方面操作执行的速度相对于在内存中创建对象要低得多。

对于数据库我通常采用的策略是使用内存数据库，比如H2。Mock数据库访问层的风险是非常高的。因为现代的数据库有大量的逻辑在里面————SQL、锁、事物等等。

关于网络访问，首先Controller(或者有的框架推荐叫Resource)是可以脱离网络进行测试的。针对对Web Service（比如RESTful API）的测试更多的是测试请求的分发（Dispatch），这个时候把服务器跑起来通常是最经济的手段。好在有内存数据库帮忙，跑起一个服务器并不是很困难的事情。可以参考JHipster的实现。

内存文件系统有Google出品的Jimfs。这个我用的比较少，因为大多数情况基于文件的内容做测试就够了。真正读写文件的时候处理好已知的异常即可。

3.考虑其他的Test Double方式

我在实际工作中也不是完全不使用Mock。如果一个对象比较困难创建，我也会先从Mock开始，测试通过以后逐步把它变成对真实对象的测试。我选择尽量不提交Mock到中心库。在使用Mock的时候我也尽量把它作为Stub使用。比如一开始的例子，我一般都不verify被依赖对象的行为，而只是检查对被测对象的状态（isFilled）。我发现这样可以让测试相对的容易理解，并且不那么脆弱。

```
@Test
public void testFillingRemovesInventoryIfInStock() {
    //setup - data
    Order order = new Order(TALISKER, 50);
    Warehouse warehouse = mock(Warehouse.class);
    when(warehouse.hasInventory(50)).thenReturn(true);

    //exercise
    order.fill(warehouse);

    //verify
    assertThat(order.isFilled(), is(true));
}
```

我自己使用Stub的时候并不多（因为我发现如果遵循SRP的原则实际对象通常都很容易创建和装配）。在我看来通过扩展（extend）原对象的方法来生成Stub也比Mock要好，

因为这样你关注的是状态，而不是两个对象的交互细节。Spies也是一个替代方案它是Stub的一种实现。具体的的定义可以参考Martin Fowler的文章Mocks Aren't Stubs。

4.尝试Reactive Programming

严格的讲这一点并不是解决Mock测试的问题，而是整体上减少有状态的类。我在使用了响应式编程之后发现需要Test Double的机会降低了。因为响应式编程鼓励尽量避免副作用（Side Effects），这使得一个方法几乎不会对外产生依赖。如果有依赖，也是另一个无副作用的依赖。对于一个Function来说，一个输入无条件的对应一个确定的输出。所以，我们设计的方法往往能很快到达"边界"。

在纯粹的Functional Programming中，是不存在面向对象意义上的"对象"的。用户可以定义数据结构，其目的是作为参数或者结果，而不是将操作和被操作的数据放到一起（就像面向对象的做法）。从面向对象编程转向响应式编程在思维方式上是一个非常大的转变。我们也不能针对一个用面向对象思维设计出来的类（比如Order）以响应式编程的方式进行测试。

总结

我们通过一系列案例分析了使用Mock的时候带来的问题。其核心问题是不能给团队提供足够的信心。根本原因则是系统被Mock的部分不一定会按照我们设想的方式工作。最后我们给出了几个避免或者减少使用Mock的几个策略。