**实验二：EasyMock**

1. **EasyMock的用途**

测试驱动开发是软件开发的重要部分。如果代码不进行测试，就是不可靠的。所有代码都必须测试，而且理想情况下应该在编写代码之前编写测试。但是，有些东西容易测试，有些东西不容易。如果要编写一个代表货币值的简单的类，那么很容易测试，把 $1.23 和 $2.8 相加是否能够得出 $4.03，而不是 $3.03 或 $4.029999998。测试是否不会出现 $7.465 这样的货币值也不太困难。但是，如何测试把 $7.50 转换为 €5.88 的方法呢（尤其是在通过连接数据库查询随时变动的汇率信息的情况下）？在每次运行程序时，amount.toEuros() 的正确结果都可能有变化。

**答案是 *mock 对象***。测试并不通过连接真正的服务器来获取最新的汇率信息，而是连接一个 mock 服务器，它总是返回相同的汇率。这样就可以得到可预测的结果，可以根据它进行测试。毕竟，测试的目标是 toEuros() 方法中的逻辑，而不是服务器是否发送正确的值。（那是构建服务器的开发人员要操心的事）。这种 mock 对象有时候称为 *fake*。

* **mock 对象还有助于测试错误条件**。例如，如果 toEuros() 方法试图获取最新的汇率，但是网络中断了，那么会发生什么？可以把以太网线从计算机上拔出来，然后运行测试，但是编写一个模拟网络故障的 mock 对象省事得多。
* **mock 对象还可以测试类的行为**。通过把断言放在 mock 代码中，可以检查要测试的代码是否在适当的时候把适当的参数传递给它的协作者。可以通过 mock 查看和测试类的私有部分，而不需要通过不必要的公共方法公开它们。
* **mock 对象有助于从测试中消除依赖项**。它们使测试更单元化。涉及 mock 对象的测试中的失败很可能是要测试的方法中的失败，不太可能是依赖项中的问题。这有助于隔离问题和简化调试。

EasyMock 是一个针对 Java 编程语言的开放源码 mock 对象库，可以帮助您快速轻松地创建用于这些用途的 mock 对象。EasyMock 使用动态代理，让您只用一行代码就能够创建任何接口的基本实现。通过添加 EasyMock 类扩展，还可以为类创建 mock。可以针对任何用途配置这些 mock，从方法签名中的简单哑参数到检验一系列方法调用的多调用测试。

1. **EasyMock 简介**

现在通过一个具体示例演示 EasyMock 的工作方式。清单 1 是虚构的 ExchangeRate 接口。与任何接口一样，接口只说明实例要做什么，而不指定应该怎么做。例如，它并没有指定从 Yahoo 金融服务、政府还是其他地方获取汇率数据。

**清单 1.** ExchangeRate

import java.io.IOException;

public interface ExchangeRate {

double getRate(String inputCurrency, String outputCurrency) throws IOException;

}

清单 2 是假定的 Currency 类的骨架。它实际上相当复杂，很可能包含 bug。（您不必猜了：确实有 bug，实际上有不少）。

**清单 2.** Currency **类**

import java.io.IOException;

public class Currency {

private String units;

private long amount;

private int cents;

public Currency(double amount, String code) {

this.units = code;

setAmount(amount);

}

private void setAmount(double amount) {

this.amount = new Double(amount).longValue();

this.cents = (int) ((amount \* 100.0) % 100);

}

public Currency toEuros(ExchangeRate converter) {

if ("EUR".equals(units)) return this;

else {

double input = amount + cents/100.0;

double rate;

try {

rate = converter.getRate(units, "EUR");

double output = input \* rate;

return new Currency(output, "EUR");

} catch (IOException ex) {

return null;

}

}

}

public boolean equals(Object o) {

if (o instanceof Currency) {

Currency other = (Currency) o;

return this.units.equals(other.units)

&& this.amount == other.amount

&& this.cents == other.cents;

}

return false;

}

public String toString() {

return amount + "." + Math.abs(cents) + " " + units;

}

}

Currency 类设计的一些重点可能不容易一下子看出来。汇率是从这个类之外 传递进来的，并不是在类内部构造的。因此，很有必要为汇率创建 mock，这样在运行测试时就不需要与真正的汇率服务器通信。这还使客户机应用程序能够使用不同的汇率数据源。

清单 3 给出一个 JUnit 测试，它检查在汇率为 1.5 的情况下 $2.50 是否会转换为 €3.75。使用 EasyMock 创建一个总是提供值 1.5 的 ExchangeRate 对象。

**清单 3.** CurrencyTest **类**

import junit.framework.TestCase;

import org.easymock.EasyMock;

import java.io.IOException;

public class CurrencyTest extends TestCase {

public void testToEuros() throws IOException {

Currency expected = new Currency(3.75, "EUR");

ExchangeRate mock = EasyMock.createMock(ExchangeRate.class);

EasyMock.expect(mock.getRate("USD", "EUR")).andReturn(1.5);

EasyMock.replay(mock);

Currency actual = testObject.toEuros(mock);

assertEquals(expected, actual);

}

}

老实说，在我第一次运行 清单 3 时失败了，测试中经常出现这种问题。但是，我已经纠正了 bug。这就是我们采用 TDD 的原因。

运行这个测试，它通过了。发生了什么？我们来逐行看看这个测试。首先，构造测试对象和预期的结果：

Currency testObject = new Currency(2.50, "USD");

Currency expected = new Currency(3.75, "EUR");

这不是新东西。

接下来，通过把 ExchangeRate 接口的 Class 对象传递给静态的 EasyMock.createMock() 方法，创建这个接口的 mock 版本：

ExchangeRate mock = EasyMock.createMock(ExchangeRate.class);

这是到目前为止最不可思议的部分。注意，我可没有编写实现 ExchangeRate 接口的类。另外，EasyMock.createMock() 方法绝对无法返回 ExchangeRate 的实例，它根本不知道这个类型，这个类型是我为本文创建的。即使它能够通过某种奇迹返回 ExchangeRate，但是如果需要模拟另一个接口的实例，又会怎么样呢？

我最初看到这个时也非常困惑。我不相信这段代码能够编译，但是它确实可以。这里的 “黑魔法” 来自 Java 1.3 中引入的 Java 5 泛型和动态代理（见 参考资料）。幸运的是，您不需要了解它的工作方式（发明这些诀窍的程序员确实非常聪明）。

下一步同样令人吃惊。为了告诉 mock 期望什么结果，把方法作为参数传递给 EasyMock.expect() 方法。然后调用 andReturn() 指定调用这个方法应该得到什么结果：EasyMock.expect(mock.getRate("USD", "EUR")).andReturn(1.5);

EasyMock 记录这个调用，因此知道以后应该重放什么。

如果在使用 mock 之前忘了调用 EasyMock.replay()，那么会出现 IllegalStateException 异常和一个没有什么帮助的错误消息：missing behavior definition for the preceding method call。

接下来，通过调用 EasyMock.replay() 方法，让 mock 准备重放记录的数据：EasyMock.replay(mock);

这是让我比较困惑的设计之一。EasyMock.replay() 不会实际重放 mock。而是重新设置 mock，在下一次调用它的方法时，它将开始重放。

现在 mock 准备好了，我把它作为参数传递给要测试的方法：

为类创建 mock

从实现的角度来看，很难为类创建 mock。不能为类创建动态代理。标准的 EasyMock 框架不支持类的 mock。但是，EasyMock 类扩展使用字节码操作产生相同的效果。您的代码中采用的模式几乎完全一样。只需导入 org.easymock.classextension.EasyMock 而不是 org.easymock.EasyMock。为类创建 mock 允许把类中的一部分方法替换为 mock，而其他方法保持不变。

Currency actual = testObject.toEuros(mock);

最后，检查结果是否符合预期：assertEquals(expected, actual);

这就完成了。如果有一个需要返回特定值的接口需要测试，就可以快速地创建一个 mock。这确实很容易。ExchangeRate 接口很小很简单，很容易为它手工编写 mock 类。但是，接口越大越复杂，就越难为每个单元测试编写单独的 mock。通过使用 EasyMock，只需一行代码就能够创建 java.sql.ResultSet 或 org.xml.sax.ContentHandler 这样的大型接口的实现，然后向它们提供运行测试所需的行为。

1. **测试异常**

mock 最常见的用途之一是测试异常条件。例如，无法简便地根据需要制造网络故障，但是可以创建模拟网络故障的 mock。

当 getRate() 抛出 IOException 时，Currency 类应该返回 null。清单 4 测试这一点：

**清单 4. 测试方法是否抛出正确的异常**

public void testExchangeRateServerUnavailable() throws IOException {

ExchangeRate mock = EasyMock.createMock(ExchangeRate.class);

EasyMock.expect(mock.getRate("USD", "EUR")).andThrow(new IOException());

EasyMock.replay(mock);

Currency actual = testObject.toEuros(mock);

assertNull(actual);

}

这里的新东西是 andThrow() 方法。顾名思义，它只是让 getRate() 方法在被调用时抛出指定的异常。

可以抛出您需要的任何类型的异常（已检查、运行时或错误），只要方法签名支持它即可。这对于测试极其少见的条件（例如内存耗尽错误或无法找到类定义）或表示虚拟机 bug 的条件（比如 UTF-8 字符编码不可用）尤其有帮助。

1. **设置预期**

EasyMock 不只是能够用固定的结果响应固定的输入。它还可以检查输入是否符合预期。例如，假设 toEuros() 方法有一个 bug（见清单 5），它返回以欧元为单位的结果，但是获取的是加拿大元的汇率。这会让客户发一笔意外之财或遭受重大损失。

**清单 5. 有 bug 的** toEuros() **方法**

public Currency toEuros(ExchangeRate converter) {

if ("EUR".equals(units)) return this;

else {

double input = amount + cents/100.0;

double rate;

try {

rate = converter.getRate(units, "CAD");

double output = input \* rate;

return new Currency(output, "EUR");

} catch (IOException e) {

return null;

}

}

}

但是，不需要为此编写另一个测试。清单 4 中的 testToEuros 能够捕捉到这个 bug。当对这段代码运行清单 4 中的测试时，测试会失败并显示以下错误消息：

"java.lang.AssertionError:

Unexpected method call getRate("USD", "CAD"):

getRate("USD", "EUR"): expected: 1, actual: 0".

注意，这并不是我设置的断言。EasyMock 注意到我传递的参数不符合测试用例。

在默认情况下，EasyMock 只允许测试用例用指定的参数调用指定的方法。但是，有时候这有点儿太严格了，所以有办法放宽这一限制。例如，假设希望允许把任何字符串传递给 getRate() 方法，而不仅限于 USD 和 EUR。那么，可以指定 EasyMock.anyObject() 而不是显式的字符串，如下所示：

EasyMock.expect(mock.getRate(

(String) EasyMock.anyObject(),

(String) EasyMock.anyObject())).andReturn(1.5);

还可以更挑剔一点儿，通过指定 EasyMock.notNull() 只允许非 null 字符串：

EasyMock.expect(mock.getRate(

(String) EasyMock.notNull(),

(String) EasyMock.notNull())).andReturn(1.5);

静态类型检查会防止把非 String 对象传递给这个方法。但是，现在允许传递 USD 和 EUR 之外的其他 String。还可以通过 EasyMock.matches() 使用更显式的正则表达式。下面指定需要一个三字母的大写 ASCII String：

EasyMock.expect(mock.getRate(

(String) EasyMock.matches("[A-Z][A-Z][A-Z]"),

(String) EasyMock.matches("[A-Z][A-Z][A-Z]"))).andReturn(1.5);

使用 EasyMock.find() 而不是 EasyMock.matches()，就可以接受任何包含三字母大写子 String 的 String。

EasyMock 为基本数据类型提供相似的方法：

* EasyMock.anyInt()

EasyMock.anyShort()

* EasyMock.anyByte()
* EasyMock.anyLong()
* EasyMock.anyFloat()
* EasyMock.anyDouble()
* EasyMock.anyBoolean()

对于数字类型，还可以使用 EasyMock.lt(x) 接受小于 x 的任何值，或使用 EasyMock.gt(x) 接受大于 x 的任何值。

在检查一系列预期时，可以捕捉一个方法调用的结果或参数，然后与传递给另一个方法调用的值进行比较。最后，通过定义定制的匹配器，可以检查参数的任何细节，但是这个过程比较复杂。但是，对于大多数测试，EasyMock.anyInt()、EasyMock.matches() 和 EasyMock.eq() 这样的基本匹配器已经足够了。

1. **严格的 mock 和次序检查**

EasyMock 不仅能够检查是否用正确的参数调用预期的方法。它还可以检查是否以正确的次序调用这些方法，而且只调用了这些方法。在默认情况下，不执行这种检查。要想启用它，应该在测试方法末尾调用 EasyMock.verify(mock)。例如，如果 toEuros() 方法不只一次调用 getRate()，清单 6 就会失败。

**清单 6. 检查是否只调用** getRate() **一次**

public void testToEuros() throws IOException {

Currency expected = new Currency(3.75, "EUR");

ExchangeRate mock = EasyMock.createMock(ExchangeRate.class);

EasyMock.expect(mock.getRate("USD", "EUR")).andReturn(1.5);

EasyMock.replay(mock);

Currency actual = testObject.toEuros(mock);

assertEquals(expected, actual);

EasyMock.verify(mock);

}

EasyMock.verify() 究竟做哪些检查取决于它采用的操作模式：

* **Normal —** EasyMock.createMock()：必须用指定的参数调用所有预期的方法。但是，不考虑调用这些方法的次序。调用未预期的方法会导致测试失败。
* **Strict —** EasyMock.createStrictMock()：必须以指定的次序用预期的参数调用所有预期的方法。调用未预期的方法会导致测试失败。
* **Nice —** EasyMock.createNiceMock()：必须以任意次序用指定的参数调用所有预期的方法。调用未预期的方法*不会* 导致测试失败。Nice mock 为没有显式地提供 mock 的方法提供合理的默认值。返回数字的方法返回 0，返回布尔值的方法返回 false。返回对象的方法返回 null。

检查调用方法的次序和次数对于大型接口和大型测试更有意义。例如，请考虑 org.xml.sax.ContentHandler 接口。如果要测试一个 XML 解析器，希望输入文档并检查解析器是否以正确的次序调用 ContentHandler 中正确的方法。例如，请考虑清单 7 中的简单 XML 文档：

**清单 7. 简单的 XML 文档**

<root>

Hello World!

</root>

根据 SAX 规范，在解析器解析文档时，它应该按以下次序调用这些方法：

1. setDocumentLocator()
2. startDocument()
3. startElement()
4. characters()
5. endElement()
6. endDocument()

但是，更有意思的是，对 setDocumentLocator() 的调用是可选的；解析器可以多次调用 characters()。它们不需要在一次调用中传递尽可能多的连续文本，实际上大多数解析器不这么做。即使是对于清单 7 这样的简单文档，也很难用传统的方法测试 XML 解析器，但是 EasyMock 大大简化了这个任务，见清单 8：

**清单 8. 测试 XML 解析器**

import java.io.\*;

import org.easymock.EasyMock;

import org.xml.sax.\*;

import org.xml.sax.helpers.XMLReaderFactory;

import junit.framework.TestCase;

public class XMLParserTest extends TestCase {

private XMLReader parser;

protected void setUp() throws Exception {

parser = XMLReaderFactory.createXMLReader();

}

public void testSimpleDoc() throws IOException, SAXException {

String doc = "<root>\n Hello World!\n</root>";

ContentHandler mock = EasyMock.createStrictMock(ContentHandler.class);

mock.setDocumentLocator((Locator) EasyMock.anyObject());

EasyMock.expectLastCall().times(0, 1);

mock.startDocument();

mock.startElement(EasyMock.eq(""), EasyMock.eq("root"), EasyMock.eq("root"),

(Attributes) EasyMock.anyObject());

mock.characters((char[]) EasyMock.anyObject(),

EasyMock.anyInt(), EasyMock.anyInt());

EasyMock.expectLastCall().atLeastOnce();

mock.endElement(EasyMock.eq(""), EasyMock.eq("root"), EasyMock.eq("root"));

mock.endDocument();

EasyMock.replay(mock);

parser.setContentHandler(mock);

InputStream in = new ByteArrayInputStream(doc.getBytes("UTF-8"));

parser.parse(new InputSource(in));

EasyMock.verify(mock);

}

}

这个测试展示了几种新技巧。首先，它使用一个 strict mock，因此要求符合指定的次序。例如，不希望解析器在调用 startDocument() 之前调用 endDocument()。

第二，要测试的所有方法都返回 void。这意味着不能把它们作为参数传递给 EasyMock.expect()（就像对 getRate() 所做的）。（EasyMock 在许多方面能够 “欺骗” 编译器，但是还不足以让编译器相信 void 是有效的参数类型）。因此，要在 mock 上调用 void 方法，由 EasyMock 捕捉结果。如果需要修改预期的细节，那么在调用 mock 方法之后立即调用 EasyMock.expectLastCall()。另外注意，不能作为预期参数传递任何 String、int 和数组。必须先用 EasyMock.eq() 包装它们，这样才能在预期中捕捉它们的值。

清单 8 使用 EasyMock.expectLastCall() 调整预期的方法调用次数。在默认情况下，预期的方法调用次数是一次。但是，我通过调用 .times(0, 1) 把 setDocumentLocator() 设置为可选的。这指定调用此方法的次数必须是零次或一次。当然，可以根据需要把预期的方法调用次数设置为任何范围，比如 1-10 次、3-30 次。对于 characters()，我实际上不知道将调用它多少次，但是知道必须至少调用一次，所以对它使用 .atLeastOnce()。如果这是非 void 方法，就可以对预期直接应用 times(0, 1) 和 atLeastOnce()。但是，因为这些方法返回 void，所以必须通过 EasyMock.expectLastCall() 设置它们。

最后注意，这里对 characters() 的参数使用了 EasyMock.anyObject() 和 EasyMock.anyInt()。这考虑到了解析器向 ContentHandler 传递文本的各种方式。

1. **mock 和真实性**

有必要使用 EasyMock 吗？其实，手工编写的 mock 类也能够实现 EasyMock 的功能，但是手工编写的类只能适用于某些项目。例如，对于 清单 3，手工编写一个使用匿名内部类的 mock 也很容易，代码很紧凑，对于不熟悉 EasyMock 的开发人员可读性可能更好。但是，它是一个专门为本文构造的简单示例。在为 org.w3c.dom.Node（25 个方法）或 java.sql.ResultSet（139 个方法而且还在增加）这样的大型接口创建 mock 时，EasyMock 能够大大节省时间，以最低的成本创建更短更可读的代码。

最后，**提出一条警告**：使用 mock 对象可能做得太过分。可能把太多的东西替换为 mock，导致即使在代码质量很差的情况下，测试仍然总是能够通过。替换为 mock 的东西越多，接受测试的东西就越少。依赖库以及方法与其调用的方法之间的交互中可能存在许多 bug。把依赖项替换为 mock 会隐藏许多实际上可能发现的 bug。在任何情况下，mock 都不应该是您的第一选择。如果能够使用真实的依赖项，就应该这么做。mock 是真实类的粗糙的替代品。但是，如果由于某种原因无法用真实的类可靠且自动地进行测试，那么用 mock 进行测试肯定比根本不测试强。