

按图索骥地编写代码



■ «|\_讎||鼸鼸軀



现在，设计文档都齐备了，github也配好了，安装了 JDK7和Maveii，空项目已经用 Maven建好了。还安装好了一个免费使用的IntelliJ IDEA <\_1 13.1 Community版，用来编程。 现在就可以按照细化后的类图来编写第一个类TimeSubject®了。

下面就是TimeSubject类的代码：

public abstract class TimeSubject {

protected Map<String, Clock> clocks = new HashMap<String, Clock>();

public void attach(String cityName, Clock clock) { clocks.put(cityName, clock);

}

public void detach(String cityName) { clocks.remove(cityName);

}

public abstract void notifyAUClocks ();

“我有个疑问。这段代码中，TimeSubject类依赖Clock类，而后者还没有创建，您就开 始用它编程了。为什么不先编写Clock类呢？”

㊀以下简称IDEA。

㊁为了最大限度地体会本书所描述的结对编程的过程，建议读者在自己的计算机上，按照本书的描述，来编 写代码。在Windows、OSX和Linux系统上搭建编程操练环境的步骤，请分别参见附录B ~附录D。本 章源代码参见以下链接：[https://github.com/wubin28/book-taming-bad-code-waterfall。](https://github.com/wubin28/book-taming-bad-code-waterfall%e3%80%82)

嗯，好问题！先编写Clock类当然可以。不过先编写TimeSubject类会有额外的好处， 就是能让IDEA帮助咱们创建Clock类。后面会看到。

如果按照类图来实现，抽象的成员方法的名字notiiVO已经被Java语言本身的Object类 给占用了，notifyAll()也被占用了，所以只好把notify()改名为notifyAllClocks() 了。

现在代码中Clock显示为红色，表示这个类还没有定义。不过现在就可以提交代码到git。

“啊？代码编礙未通过就提交？我们公司可是要求我们直到测试运行通过才能提交代码的。”

对，你们公司说得没错，不过我认为这个要求是针对某种特殊情况而言的，即版本管理系统 的代码库是使用客户端-服务器这种集中式管理的情况〇你们公司管理代码版本用的是什么工具？

“SVN。”

嗯，SVN就是用这种集中式管理的方式来管理代码版本的。早先的代码管理工具CVS 也是用这种方式。这种方式最明显的特点就是一旦断网就无法提交代码。

“是呀，用SVN管理代码必须联网。我在家办公的时候，要是连不上公司网络，那就没 法写代码了。”

现在咱们使用的是git,这是一种分布式的代码版本管理工具。用这种分布式的工具提交 代码时，代码仅仅是被提交到使用git的这台计算机的本地代码库中，尚未提交到远程的代码 库中。所以即使提交尚未通过编译的代码到本地，也不会影响在远程的代码库上进行的编译工 作。等咱们一次次提交到本地的代码最后编译运行通过了，再统一 push到远程代码库也不迟。

在提交代码之前，先填写Commit Message提交注解。

“哦，我以前一直都不填Commit Message。”

每次提交代码都需要填Commit Message。因为如果想在写错代码时能回退到写错前的代 码状态，就得依靠它。另外Commit Message还能起到代码注释的作用。

如果能做到当有少量代码改动时就频繁地把代码提交到本地代码库而不管是否通过编 译，且每次提交都能填写有关此次代码改动的意图明确的Commit Message,那么这种每次少 量且意图描述清晰的代码提交，一方面增强了将来阅读代码变动的可读性，另一方面当代码 写错需要回退时也能有助于做到更精细的回退。

这次提交的 Commit Message 不妨写成 Created and wrote class TimeSubject according to the class diagram. r

代码提交完，现在就可以创建那个标红的Clock类了。在IDEA里，可以把光标移到 Clock中，然后按Alt+Enter快捷键，就能让IDEA自动帮咱们写这个类了。

Clock类的3处编译错误在图2-1中用箭头标了出来，图中还显示了在Clock上按Alt+Enter 快捷键后出现的创建Clock类的快捷菜单。

㊀一般情况下，后文每段代码之前都会给出代码提交的Commit Message,并与本书在github上的源代码 一一对应。

public abstract class TimeSubject {

protected Map<String, Clock> clocks

**new HashMap<String, Clock>();**

% public void attach(String cityNawe, Clock clock)

Create Class 'Clock

9 Create Enum \*Clock\*

9 Create Inner Class 'Clock' ^ Create Interface 'Clock\*

^ Add Maven Dependency...

* Bind Method Parameters to Fields ^ Create Field for Parameter 'dock\*
* Define params default value
* Generate delegated method with default parameter value ^ Make 'private'

Make 'protected'

Make package-local

图2-1创建Clock类的快捷菜单

“哦，这么方便！您要是不说，我还要傻乎乎地一点点地写呢。”

IDEA所创建的Clock类的代码如下所示（CM。： Created class Clock.):

public class Clock {

}

按照类图写出的 Clock 类如下所示（CM: Wrote class Clock according to the class diagram.):

-public class Clock {

+public abstract class Clock {

+ private final int UTC—OFFSET = 0;

+ private int localTime = 0;

+

+ public abstract void setLocalTime(int localTime);

}

上面的代码中，带有号的行表示被删除的行，带有“+”号的行表示新添加的行。 上面的代码表示用后面5个带有“ + ”号的行替换前面那个带有号的行。

Clock类写完了，提交代码气现在在IDEA中，已经没有编译失败的错误了。

接下来根据那个类图，从左到右一个一个地编写剩下3个类的代码。首先是UtcTime类。 创建 UtcTime 类的代码如下所示（CM: Created class UtcTime.):

public class UtcTime extends TimeSubject {

©Override

public void notifyAUClocks () {

㊀ CM 即 Commit Message,余同。

㊁为行文简洁起见，下面每段代码都会进行git的代码提交，文中不赘述。

再来实现UtcTime类的notifyAllClocks〇方法。

UtcTime 类的 notifyAllClocks〇 方法如下所示（CM: Implemented method UtcTime.notify- AllClocks().):

public void notifyAUClocks () {

+ for (Clock clock : super.clocks.values()) {

+ clock.setLocalTime(Clock.toLocalTime(this.utcZeroTime));

+ }

}

呃，类图中utcTime这个名字起得真的让人有点纠结。它有两个含义，既可以指 UtcTime这个类的一个对象，也可以指UtcTime这个类中用来保存UTC时间的那个成员 变量。为了区分，把后者改名叫utcZeroTime,表示与UTC时差为0的时间。所以在细 化后的类图中，除了 UtcTime类的类名和PhoneClock类的utcTime成员变量的变量名之 外，其他8处出现UtcTime的地方都要改为utcZeroTime。另外发现这个类图还有一个错 误，上面那个for循环里面的方法clock.setLocalTime()的参数，不应该仅仅从utcTime改为 utcZeroTime,还应该把它转换为时钟所表示的当地时间，因为这是clock.setLocalTime()方 法的接口所要求的。可以用Clock类的一个静态方法toLocalTime()来把utcZeroTime转换 为 local time。

刚根据那个类图写了 3个类，就发现了那个图有3个问题需要修改：一个是n〇tify()方 法名改为notifyAllClocks(), —个是把8处utcTime改为utcZeroTime,还有一个是for循环 里面的那个方法的参数需要转换为local time。我现在就把那个类图打印一份。您一边写代 码，我一边用红笔在类图上改。

目前在细化后的类图中对上述3个问题做出的修改如图2-2所示。

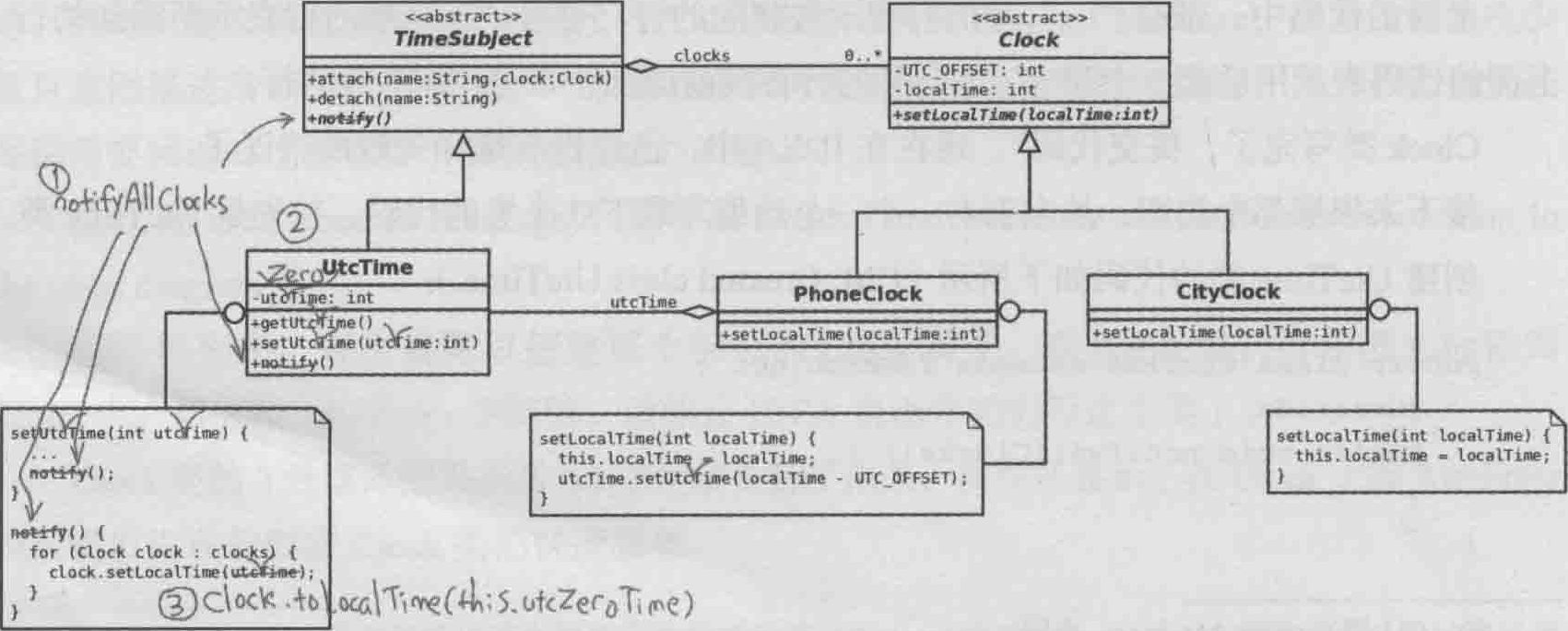


图2-2在细化后的类图上对3个问题进行的修改

在IDEA中，UtcTime类中的notifyAllClocks()方法里的for循环里的那句话，有两处 标出了红色，是因为这里有两个编译错误，一个是Clock.toLocalTimeO这个静态方法没有定 义，另一个是this.utcZeroTime这个成员变量没有定义。

UtcTime类的2处编译错误在图2-3中用箭头标了出来。

public class ptcTime extends TimeSubject {

^Override

ft public void notifyAUClocksO {

for (Clock clock : super.clocks.values()) {

clock.setLocalTime(Clock.t oLoc alTime{this.utcZe roTime});

i i } ①，

图2-3 UtcTime类的2处编译错误

“换我来编会儿吧。咱们先解决后一个问题。把光标移动到utcZeroTime上，还是用 Alt+Enter快捷键来帮咱们创建utcZeroTime这个成员变量。这个快捷键真是太好使了！”

在UtcTime类里面创建出的utcZeroTime成员变量的代码如下所7K ( CM: Added an int field utcZeroTime to class UtcTime.)：

public class UtcTime extends TimeSubject {

+ private int utcZeroTime;

“接下来处理前一个问题，在类Clock中添加静态方法toLocalTime()。”

在类Clock中添加静态方法toLocalTime()的代码如下所示（CM: Added static method Clock.toLocalTime().)：

public abstract class Clock {

- private final int UTC一OFFSET = 0;

+ private static final int UTC一OFFSET = 0; private int localTime = 0;

public abstract void setLocalTime(int localTime);

+

+ public static int toLocalTime(int utcZeroTime) {

+ return utcZeroTime + UTC一OFFSET;

+ }

J

“为了让静态方法toLocalTime()能够访问到成员变量UTC\_OFFSET，把这个成员变量 也转变为静态的了。现在IDEA里面没有编译错误了。接下来按照细化后的类图，来实现 UtcTime 类的成员变量 utcZeroTime 的 getter 和 setter。”

在IDEA里面，可以先把光标定位到UtcTime类的成员变量utcZeroTime下面，然后 按快捷键Alt+Insert调出Generate快捷菜单，来让IDEA帮助生成utcZeroTime的getter和

setter，如图2-4所示。

|  |  |  |
| --- | --- | --- |
| public class UtcTime extends TimeSubject {  private int utcZeroTime; | |  |
|  | Generate |  |
| ^Override  + public void notifyAll  for (Clock clock clock.setLoca | Constructor  Getter  Setter | )){  e(this.utcZeroTine)); |
| }  e } |  |
| equaisO and hashCodeO toStringO  Override Methods... Delegate Methods... Copyright |  |
| 1} |
|  |
|  |
|  |  |  |

图2~4 Generate快捷菜单

“不错，还是快捷键方便。”

生成的UtcTime类的成员变量utcZeroTime的getter和setter的代码如下所不（CM: Generated getter and setter of the field utcZeroTirae of class UtcTime.)：

public class UtcTime extends TimeSubject { private int utcZeroTime;

+ public int getUtcZeroTime() {

+ return utcZeroTime;

+ }

+

+ public void setUtcZeroTime(int utcZeroTime) {

+ this.utcZeroTime = utcZeroTime;

+ }

根据细化后的类图中的注解框里的伪代码，UtcTime类中的setUtcZeroTimeO方法里面 应该有个notifVAllClocks()方法，现在就可以加上它。

在UtcTime类中的setUtcZeroTime()方法里添加notifyAllClocks()方法的代码如下所示 (CM: Added method call notifyAHClocks() in method UtcTime.setUtcZeroTime().)：

public void setUtcZeroTime(int utcZeroTime) { this.utcZeroTime \* utcZeroTime;

+ notifyAUClocks ();

}

接下来，就可以根据类图编写PhoneClock类了。

创建类 PhoneClock 的代码如下所示（CM: Created class PhoneClock.):

+public class PhoneClock {

+}

然后根据类图中注解框中的伪代码来实现PhoneClock类中的setLocalTimeO方法。 PhoneClock 类中的 setLocalTimeO 方法的代码如下所示（CM: Implemented method Phone- Clock.setLocalTime() according to the class diagram.)：

-public class PhoneClock {

+public class PhoneClock extends Clock {

+ @Override

+ public void setLocalTime(int localTime) {

+ this.localTime = localTime;

+ this.utcTime.setUtcZeroTime(localTime - UTC\_OFFSET);

+ }

“哦，按照类图中注解框中的伪代码写完后，在IDEA的PhoneClock类里面，有3个地 方出现了红色的编译错误。”

PhoneClock类的3处编译错误在图2-5中用箭头标了出来。

public class PhoneClock extends Clock {

(^Override i®

public void^setLocalTime(int TocalTim©} { this.localTime = localTime;

this.utcTime.setUtcZeroTimeriocalTime - UTC\_OFFSET);

,1 f©

图2-5 PhoneClock类的3处编译错误

咱们一个一个看这3个编译错误。第1个编译错误是this.localTime,这个localTime实 际上应该来自其父类Clock,所以应该是super.IocalTime,这是细化后的类图中的错误。相 应地，为了让子类能够访问到父类的成员变量，父类Clock中的成员变量localTime也应从 private改为protected。需要改一改类图，把这个问题编为4号。

在细化后的类图中对上述问题做出了对4号问题的修改，如图2-6所示。

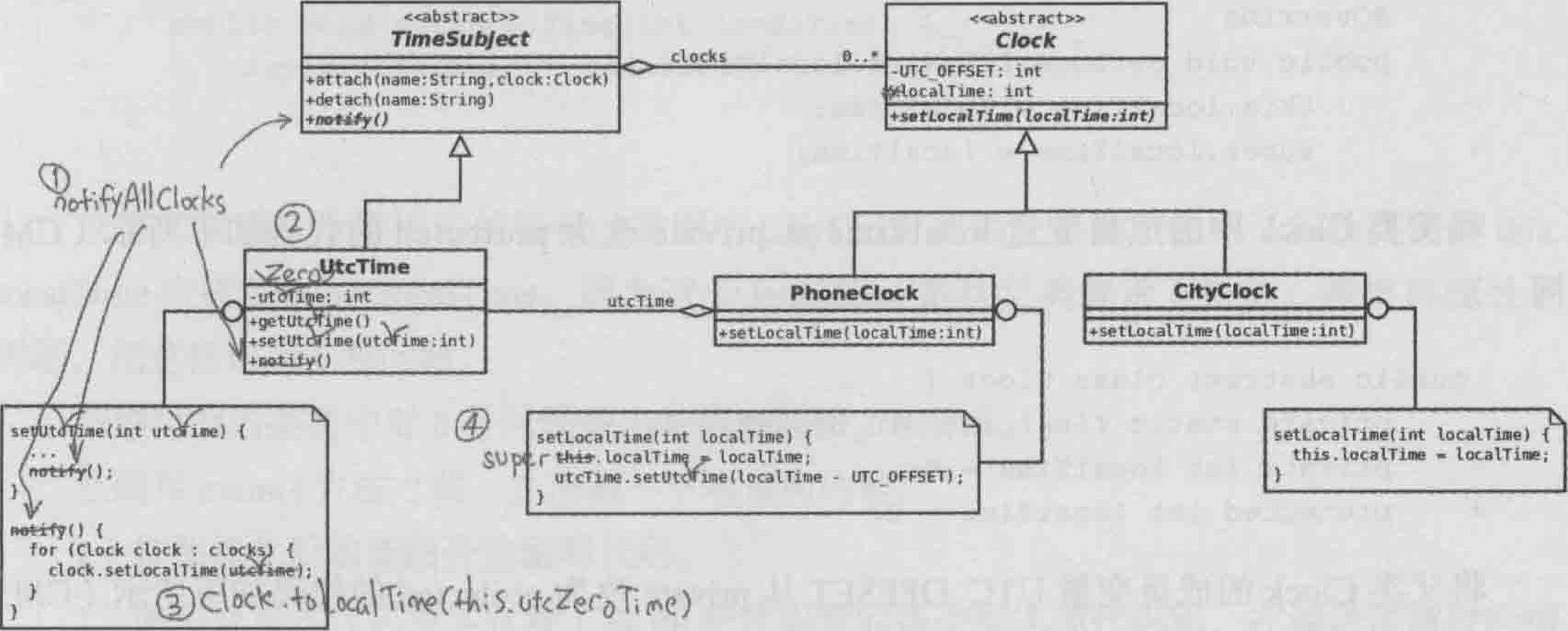


图2-6对4号问题的修改

第2个编译错误是this.utcTime。这是由于在类图中PhoneClock类左侧菱形符号所表示 的它所持有的成员变量utcTime还未创建，一会再创建。第3个编译错误是UTC\_OFFSET, 这个错误的原因与第1个编译错误类似，即UTC\_OFFSET实际上也应来自父类，所以父类 的成员变量UTC\_OFFSET应该改为protected,以便于子类访问，而不应该是private。类图 应该再改一下，在图中把这个问题编为5号。

在细化后的类图中对5号问题做出的修改如图2-7所示。

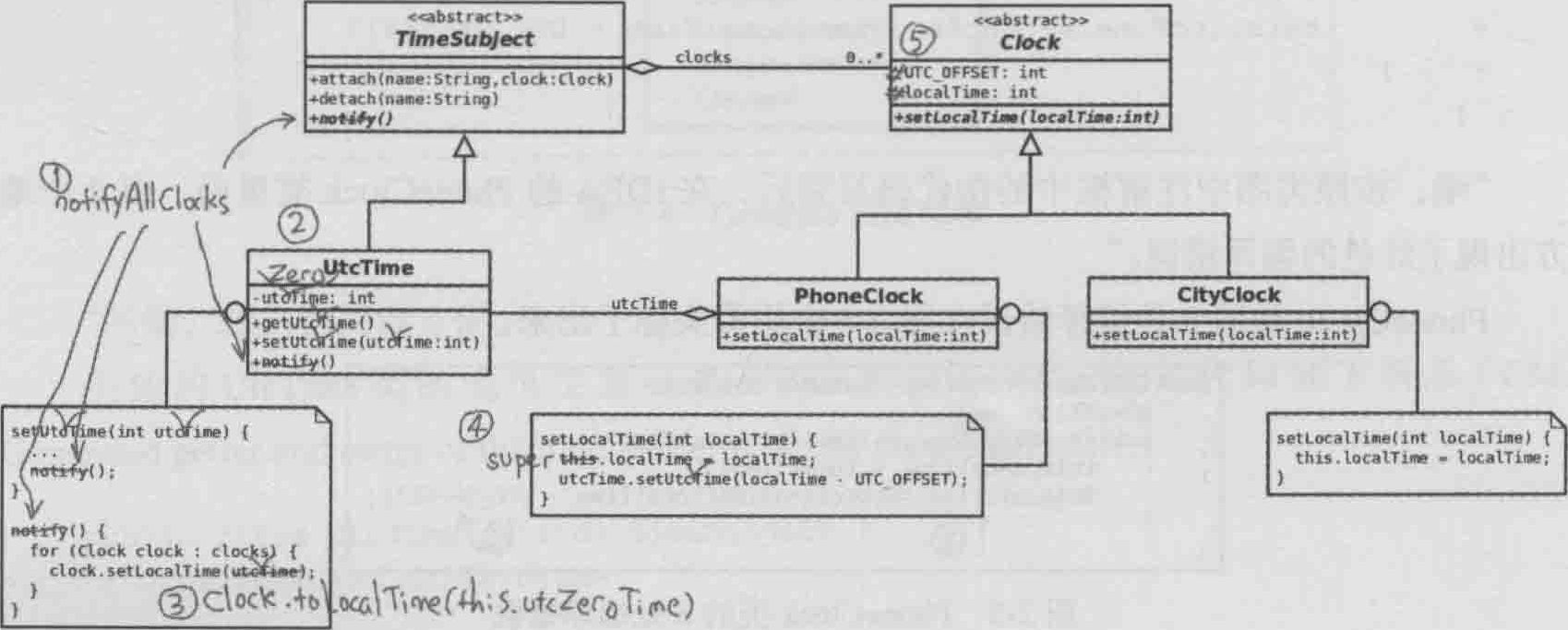


图2-7对5号问题的修改

类图改好后，相应地来改代码。

将 PhoneClock 类中的 setLocalTime()方法中的 this.localTime 改为 super.LocalTime 的代 码如下所不（CM: Made field Clock.localTime protected.):

public class PhoneClock extends Clock {

^Override

public void setLocalTime(int localTime) {

- this.localTime = localTime;

+ super.localTime = localTime;

将父类Clock中的成员变量localTime从private改为protected的代码如下所示（CM 同上)：

public abstract class Clock {

private static final int UTC\_OFFSET = 0;

- private int localTime = 0;

+ protected int localTime = 0;

将父类Clock的成员变量UTC\_OFFSET从private改为protected的代码如下所示（CM: Made field CIock.UTC\_OFFSET protected.)：

public abstract class Clock {

- private static final int UTC\_OFFSET = 0;

+ protected static final int UTC—OFFSET = 0;

“好了，现在该修复前面说的第2个编译错误this.utcTime 了。还是把光标定位到 PhoneClock类中红色的utcTime上，用Alt+Enter快捷键来帮咱们创建这个成员变量。”

在PhoneClock类中创建utcTime成员变量的代码如下所示（CM: Added field PhoneClock. utcTime.)：

public class PhoneClock extends Clock {

+ private UtcTime utcTime;

+

^Override

public void setLocalTime(int localTime) {

现在IDEA里面没有编译失败的代码了。根据类图现在只剩下最后一个类CityClock 了。 先用IDEA创建一个新类CityClock。

创建新类 CityClock 的代码如下所不（CM: Created class CityClock.):

+public class CityClock {

+ }

根据CityClock的类图和其注解框中的伪代码，该实现它所继承的setLocalTimeO方 法了。

CityClock 类的 setLocalTime()方法的代码如下所示（CM: Implemented method CityClock. setLocalTimeQ.)：

-public class CityClock {

+public class CityClock extends Clock {

+ @Override

+ public void setLocalTime(int localTime) {

+ super.localTime ® localTime;

+ }

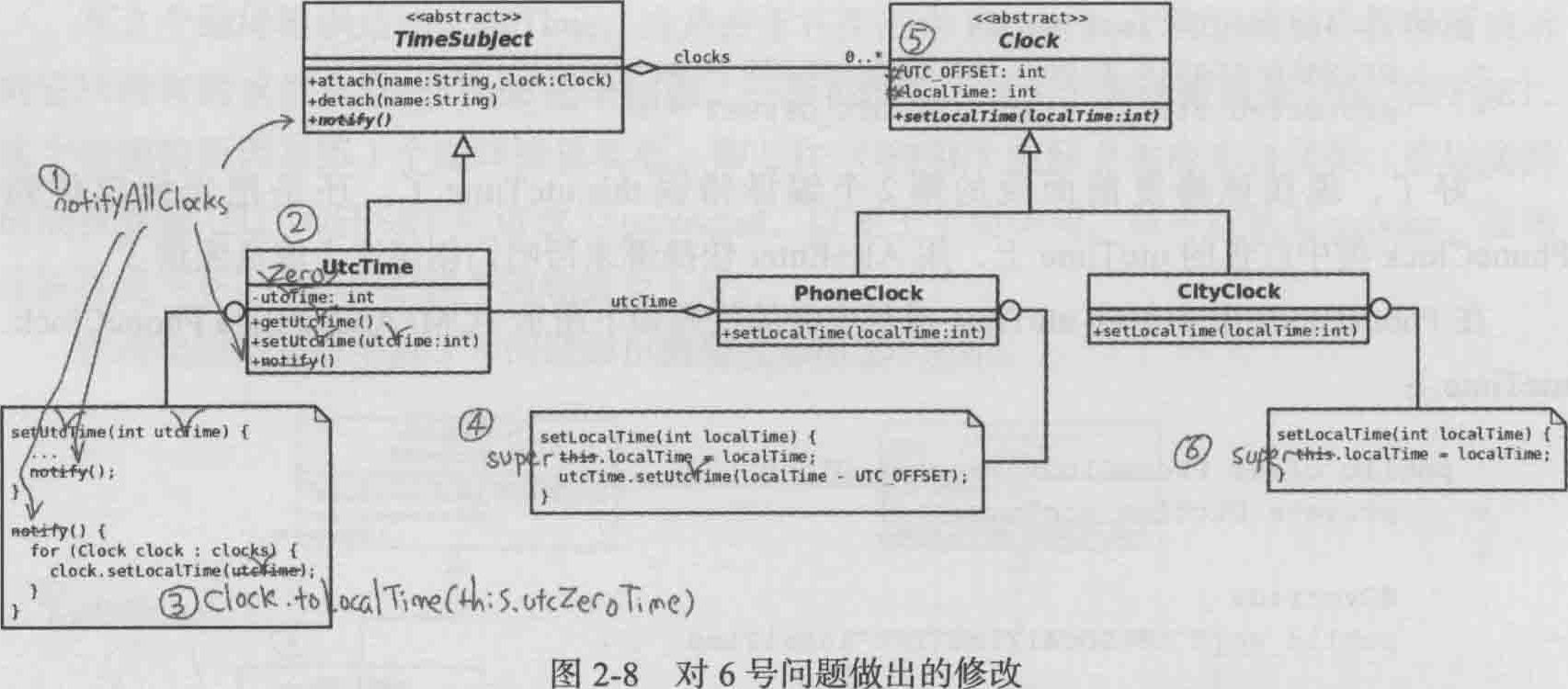
}

这里又发现一个类图中的错误。类图中CityClock类的注解框中的第2行伪代码的this. localTime应该是super.localTime，因为这个localTime是从父类继承下来的。需要再改一下 类图，把这标记为6号问题。

在细化后的类图中对6号问题做出的修改如图2-8所示。

在编写main〇方法之前，先回顾一下本章的内容。

1. )按照细化后的类图开始编写代码。
2. )使用分布式代码版本管理工具git把代码暂时提交到本地代码库，在编译未通过的情 况下，一步一步多次地提交代码。



1. )每次提交代码都写明Commit Message提交注解。
2. 随着编程的进行，修改了细化后的类图中的多处错误。
3. 通过操练我们学到了以下技能：
4. 在IDEA中把光标定位到那些红色的有编译错误的代码上，然后按快捷键Alt+Enter, 能快速帮助我们生成所需要的代码。
5. 如果能做到当有少量代码改动时就频繁地把代码提交到本地代码库而不管其是否通过 编译，且每次提交都能填写有关此次代码改动的、意图明确的Commit Message,那么这种每 次少量且意图描述清晰的代码提交，一方面增强了将来阅读代码变动的可读性，另一方面当 代码写错需要回退时也能有助于做到更精细的回退。

■■麵繼 ■鼸繼 \_肅 匿靄 ■

第3章

写main〇方法测试一下

"类图上的所有类都实现完了。咱们现在可以写个mainO方法来测试一下了。”

先创建一个包含main()方法的类HotelWorldClocksRunner。

HotelWorldClocksRunner类的代码如下所示（CM: Added class HotelWorldClocksRunner with a main() method to have a try.)：

+public class HotelWorldClocksRunner {

+ }

然后在这个类里面写mainO方法。

main()方法的代码如下所示（CM: Added the main() method to class HotelWorldClocksRunner and wrote the expected code there.)：

public class HotelWorldClocksRunner {

+ public static void main(String[] args) {

+ TimeSubject utcTime = new UtcTime();

+ utcTime•attach("beijing", new CityClock(8));

+ utcTime.attach("london", new CityClock(0));

+ utcTime•attach("moscow", new CityClock(4));

+ utcTime.attach("Sydney", new CityClock(10));

+ utcTime.attach("newYork", new CityClock(-5”；

+ Clock phoneClock = new PhoneClock(utcTime);

+

+ phoneClock.setLocalTime(9);

+

+ utcTime .printTimeOfAUClocks ();

这段代码分3个部分，第1部分是做准备工作；第2部分是调用手机时钟的setLocalTimeO 方法来设定时间为北京时间上午9点，以触发所有城市时钟的自动调整；第3部分是打印所 有时钟的本地时间。

在第1部分中，我们先创建一个具有TimeSubject类型的UtcTime实例；再把5个城市 时钟的实例都分别attach到这个UtcTime实例上，每创建一个城市时钟实例，都把该城市 与UTC时间的时差作为构造器的参数传进这个新创建的实例中。比如北京比UTC时间早8 小时，所以在attach北京时钟时，用new CityClock(8)来创建北京时钟实例。最后创建手机 时钟实例phoneClock,并把上面准备好的UtcTime实例作为构造器的参数传进去，以便在 PhoneClock 类的 setLocalTime()方法中，调用 UtcTime 类的 setUtcZeroTime()方法，来自动 调整所有城市时钟的时间。

现在咱们先创建CityClock类的带有时差参数的构造器。

“等等！我觉得带有UtcTime参数的创建PhoneClock的实例那句话写得有问题。因为

PhoneClock和CityClock都继承同一个父类Clock,为何创建CityClock实例时要提供时差参 数，而创建PhoneClock实例时却没有提供时差参数？难道PhoneClock的实例都不需要时差 参数吗？”

问得好！ PhoneClock的实例在创建时，确实也和创建CityClock实例时一样，需要传入 当地时间与UTC时间之间的时差。需要改一改这个main()方法。

修改main()方法中创建PhoneClock实例的代码如下所示（CM: Updated the main〇 method to make the constructor of PhoneClock is the same with CityClock and add the UtcTime object to the phoneClock using method PhoneClock.setUtcTime().)：

UtcTime•attach("moscow", new CityClock(4));

UtcTime.attach("Sydney", new CityClock(10));

UtcTime.attach("newYork", new CityClock(-5));

- Clock phoneClock = new PhoneClock(utcTime);

+ Clock phoneClock = new PhoneClock(8)/

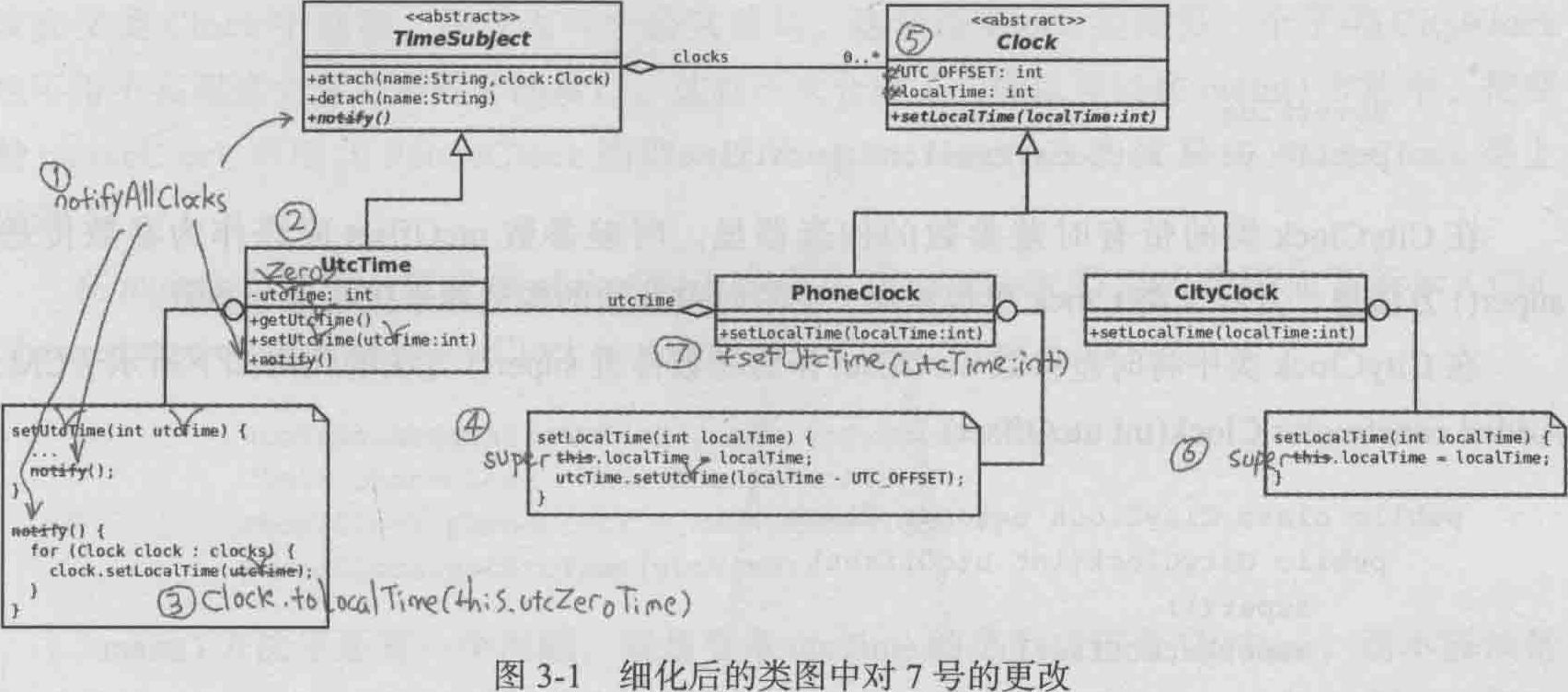
+ phoneClock.setUtcTime(utcTime);

phoneClock.setLocalTime(9);

在创建PhoneClock实例时，把北京时间距离UTC时间的时差8作为构造器的参数传进 去，然后在PhoneClock类上增加一个接口 setUtcTime()方法，来把utcTime传给phoneClock 实例。

“这样应该没问题了。再更改一下类图，在图中把这个更改标记为7号。”

,在细化后的类图中对7号的更改如图3-1所示。



这个main〇方法有不少编译错误，大概有4处。

HotelWorldClocksRunner类的main()方法的4处编译错误如图3-2所示。

public class HotelWorldClocksRunner {

public static void maln(Stringt] args) |

TimeSubjact utcTime = new UtcTimef); \_\_ utcTime.attachr'beijing", new CltyClock(B)); utcTime.attachC'london", new CityQockJ^JJ utcTime.attach("w>scowM, new CityQockXAJJ utcTime.attach("Sydney", new CitvClock(別.）n utcTime.attach(MnewYork,\*, new CitvClock(-5)) Clock phoneClock = new PhoneClock(8^; phoneClock.setUtcTime(utcTime); <—■-\2J

^ ■ (i)

phoneClock.setLocalTime(9); utc Time.p rintTimeO fAllClocks();

白a ^ ④

图3-2 HotelWorldClocksRunner类的main()方法的4处编译错误

现在从上往下看看这4个编译错误。第1个编译错误是CityClock类还没有一个接受该 城市与UTC时间的时差的构造器；第2个编译错误是PhoneClock类也没有接受一个其所在 城市与UTC时间的时差的构造器；第3个编译错误是PhoneClock类还没有创建setUtcTime() 方法；第4个编译错误是UtcTime类还没有创建printTimeOfAllClocks()方法。

接着用Alt+Enter快捷键来让丨DEA帮助咱们创建这些默认的构造器和方法。

现在先消除第1个编译错误，创建CityClock类的带有时差参数的构造器。

创建CityClock类的带有时差参数的构造器的代码如下所示（CM: Added constructor CityClock(int utcOfFset).)：

public class CityClock extends Clock { + public CityClock(int utcOffset) {

+ super();

+ }

+

©Override

public void setLocalTime(int localTime) {

在CityClock类的带有时差参数的构造器里，时差参数utcOffset应该作为参数传进 super〇方法里，并在父类Clock里也添加一个带时差参数的构造器来接收这个参数。

在CityClock类中将时差参数utcOffset作为参数传进super()方法的代码如下所示（CM: Added constructor Clock(int utcOffset).)：

public class CityClock extends Clock { public CityClock(int utcOffset) {

* super ();

+ super(utcOffset);

}

在父类Clock里添加一个带时差参数的构造器的代码如下所示（CM同上)：

public abstract class Clock {

* protected static final int UTC\_OFFSET = 0;

+ protected static int UTC一OFFSET;

protected int localTime =\* 0;

+ public Clock(int utcOffset) {

+ UTC一OFFSET = utcOffset;

+ }

+• • j

public abstract void setLocalTime(int localTime);

因为Clock类的成员变量UTC\_OFFSET需要在其构造器里赋值，所以就不能是final 的了。

再消除第2个编译错误，创建PhoneClock类的带有时差参数的构造器。

创建PhoneClock类的带有时差参数的构造器的代码如下所示（CM: Created constructor PhoneClock(int utcOffset).)：

public class PhoneClock extends Clock { private UtcTime utcTime;

+ public PhoneClock(int utcOffset) {

+ super(utcOffset);

+ }

+

现在消除第3个编译错误，创建PhoneClock类的setUtcTime()方法。不过在main() 方法中，变量phoneClock被声明为Clock类型了。如果是这样的话，setUtcTime()方法应

该在父类Clock中创建，而成为一个公共接口，这使得Clock类的另一个子类CityClock 也不得不实现这个它并不需要的接口。这就不大合理了。所以可以在mainO方法中，把变 量phoneClock声明为PhoneClock类型，这样setUtcTime()方法就只在PhoneClock类上 创建了。

在main()方法中，把变量phoneClock声明为PhoneClock类型的代码如下所示（CM: Changed type of varialbe phoneClock in main() to be PhoneClock.)：

utcTime.attach("newYork", new CityClock(-5));

* Clock phoneClock = new PhoneClock(8);

+ PhoneClock phoneClock = new PhoneClock(8);

phoneClock.setUtcTime(utcTime);

“main()方法里还有一个问题，就是变量utcTime的类型应该是UtcTime,而不应该是 其父类TimeSubject。因为根据类图来看，PhoneClock类持有一个UtcTime的实例，以便调 用后者的setUtcZeroTime()方法。而这个方法只在UtcTime类中定义了，其父类TimeSubject 并没有定义。所以为了调用UtcTime类中的setUtcZeroTime()方法，main()方法里的变量 utcTime的类型应该是UtcTime。”

好的，这就改过来。

在main()方法中，把变量utcTime声明为UtcTime类型的代码如下所示（CM: Changed type of varialbe utcTime in main() to be UtcTime.)：

public class HotelWorldClocksRunner {

public static void main(String[] args) {

* TimeSubject utcTime = new UtcTime();

+ UtcTime utcTime = new UtcTime();

utcTime•attach("beijing”， new CityClock(8));

现在可以创建PhoneClock类的setUtcTime()方法来消除第3个编译错误了。

在 PhoneClock 类中创建 setUtcTime()方法的代码如下所示（CM: Created method Phone- Clock.setUtcTime(UtcTime).)：

super.localTime - localTime;

this.utcTime.setUtcZeroTime(localTime - UTC\_OFFSET);

}

+

+ public void setUtcTime(UtcTime utcTime) {

+ this.utcTime ~ utcTime;

现在可以创建UtcTime类的printTimeOfAUCbcksO方法来消除第4个编译错误了。这 个方法专门是为测试用的，所以就不在类图中画出来了。

创建 UtcTime 类的 printTimeOfAllClocks()方法的代码如下所示（CM: Created method UtcTime.printTimeOfAllClocksQ.)：

clock.setLocalTime(Clock.toLocalTime(this.utcZeroTime));

}

}

+

+ public void printTimeOfAUClocks () {

+ for (String clockName : super.clocks.keyset()) {

+ System.out.println(clockName + " + super.clocks.get(clockName).getTime());

+ }

+ }

}

因为utcTime能从其父类TimeSubject里继承Map类型的成员变量clocks,所以就能在 printTimeOfAllClocks()方法里写一个循环语句，打印所有的时钟名称和时间。

现在就差创建Clock类的getTime〇方法了。

创建 Clock 类的 getTime()方法的代码如下所示（CM: Created method Clock.getTime().):

public static int toLocalTime(int utcZeroTime) { return utcZeroTime + UTC\_OFFSET;

}

+

+ public String getTime() {

+ return String.valueOf(this.localTime);

+ }

}

看起来现在终于可以运行一下这个mainO方法了。在IDEA里打开那个main()方法，把 光标定位到类名上，然后按Ctrl+Shifi+FlO组合键运行一下。结果出来了，奇怪，所有的城 市的时间都是9点。

第一次运行main〇方法的结果如图3-3所示。

“现在的问题是，除了北京以外，其他所有城市的当地时间都是错的，而且都是9。加个 断点调试一下吧。“

在调试前，咱们先看看这章做了什么工作：

1. 开始编写main〇方法，并通过打印语句，测试了一下先前按照细化后的类图所编写 的代码。
2. )在创建PhcmeClock的实例时发现并修复了构造器的参数问题。
3. 先在main()方法中编写调用了暂时不存在但期望存在的接口的代码（即意图代码)， 然后循着这些意图代码的红色编译错误，来编写生产代码以消除这些错误。
4. 修改细化后的类图以反映设计的修改。

D 0 乂 fll d\* :令♦ ^ j HoteiWortdC丨ocksRwmgr ▼ j ► # f 掌函

C} book-taming-bad-code-waterfall (~/buf

C] .idea D src ▼ D main ▼ C3 |ava

▼ ILJ tbc.waterfall CSyClock

* d Clock

‘ Un^oIU/Arl«lrtn>VcDi inirikt- n, otdwoncii. GCivSrtunncf

#\*b PhoneClock

* TlmeSubject
* h UtcTime

@ .gitignore

91 book-taming-bad-code-waterfall. iml m pom.xml § README, md ll External Libraries

曰/\*\*

^ Created by ben on 14-2-25.

*G \*/*

**public class HotelWorldClocksRunner {**

**E=j public static void main(String[] args) f UtcTime utcTime = new UtcTimeO; utcTime.attachC'beijing'\*, new CityClock(8) J; utcTime.attachi^london'\*, new CityClock(0)); utcTime.attach("«oscow\*\*r new CityClock(4)); utcTime.attach("Sydney", new CityClock(lG)}; utcTimG.attach("newYork\*\*/ new CityClock(-5)); PhoneQock phoneClock = new PhoneClock(8); phoneClock.setUtcTime(utcTime);**

**phoneClock.S0tLocalTimG(9);**

File £dlt View CJavtgate £ode Anal^e Refactor guild Ran lools VCS Window Help

rnb°〇k-taming4)ad^:ode-waterfall D src D main i!t! java & tbc [■ th waterfair

teoJd TSIzv

1：0n Project

HotetWorldC丨ocksRunner.java x !

**package tbc.waterfall;**

n'ot-aiWondCl- «cksRuuner

**utcTime.printTimG**〇**fAHClocks();**

Run p-] HotelWoddClocks Runner

\* '/opt/jdkl.7.G\_51/bin7]ava I london： 9 ^ BOSCOW：

beijing:

5I-S neWYork Sydney：

@ Process finished with exit code 0

-Didea. launcher. port=7534 -Didea .launcher. bin. path-=/opt/idea- IC^lH7696/bin -DfUe e

图3-3第一次运行main()方法的结果

调试一下

先看看需要在哪里加断点。从main〇方法看起。最后一句utcTime.printrimeOfAllClocksO仅 仅是打印dock中保存的当地时间的结果，而结果在打印前已经是错的了。所以需要检查给每 个clock的当地时间赋值的地方。这件事在main〇方法中，是由“phoneClock.setLocalTime(9);” 这句话来做的；而在PhoneClock类的setLocalTime〇方法中，这件事又委托给了 UtcTime类 的setUtcZeroTime〇方法；而在UtcTime类的setUtcZeroTime()方法中，这事又委托给该类的 notifyAllClocks〇方法了。notifyAllClocks〇方法里有一个for循环，用来给Map中保存的每一个 城市时钟的当地时间进行赋值。我们可以在这个for循环里的那个语句上加一个断点。

在UtcTime类的notifyAllClocks〇方法里的for循环里加断点，如图4-1所示。

|  |  |  |
| --- | --- | --- |
| public class UtcTime extends TimeSubject { | |  |
|  | private int utcZcroTi«e; |  |
| 田 | public Int getUtcZeix>Tl»©() { return utcZeroTlae; } |  |
|  | pubLlc woid setUtcZeroTime(int utcZeroTime) { |  |
|  | this.utcZeroTiae = utcZeroTime; |  |
| B | notifyAUnocksO ;  ) |  |
|  | ^Override |  |
| •，申  • | public wold h〇ti^yAUa〇cks{) {  for (Clock ciock : super.clocks.values()) {  satUcalTl^ | 1^)J； |
| 0 | }  } |  |

图4\*1在for循环里加断点

“在加断点的这个语句中，真正计算城市时钟的当地时间的是Clock类的静态方法 toLocalTime〇。在这个方法里的唯 条return语句上也加上断点。”

在Clock类的toLocalTime()方法里的return语句上加断点，如图4-2所示。

•1 public abstract class Clock {

protected static int UTC\_OFFSET; protected int localTime =0;

：4i public Clock(int utcOffset) {

UrC\_OFfS£T = utcOffset;

^ }

•i public abstract void sotLocalTime(int localTime);

白 public static int ItoLocalTimeUnt utcZeroTime) {

• HtfiE return utcZeroTime 國抓

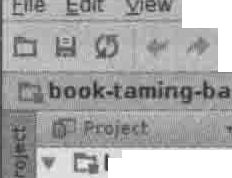
$ >

图4-2在Clock类的toLocalTime()方法里的return语句上加断点

“在IDEA中以debug方式运行main()方法，程序停到了第1个断点上，即首次进入 UtcTime类的noitfyAllClocks()方法中的for循环中的那条语句上。单击下面的变量值可以观 察一下。”

第1个断点：首次进人UtcTime类的noitfyAllClocks()方法中的for循环中的那条语句 的变量值如图4-3所示。

Navigate £ode Analyze Beractor BuBd Ryn I〇〇!s VCS Window hleip



批® d•改汉令办1 杉 1 . Hot«»WprldCl〇clwRMnfi\*f \* i ► • V ^ iSl Of 於醱？

丨d\*code，aterfdll v.: 丨■丨 •丨 c UtcTime

〇 十 ft•卜 cfc HoteiWorldCIccksRunner.java x ,c utcTime.|ava x ,Clock.java x , c CityClocIcjava x «

book-taming4)ad-code-waterfall v-rtii ■

D .idea D src ▼ C] main ▼ D java

▼ Cltbc.waterfail

* ‘ CityClock
* b Clock

c\* . H〇teJWorldiS3®\*:JaRunn.

#r6 PhoneClock 〇 'h TtmeSubject 9 丨 UtcTlme

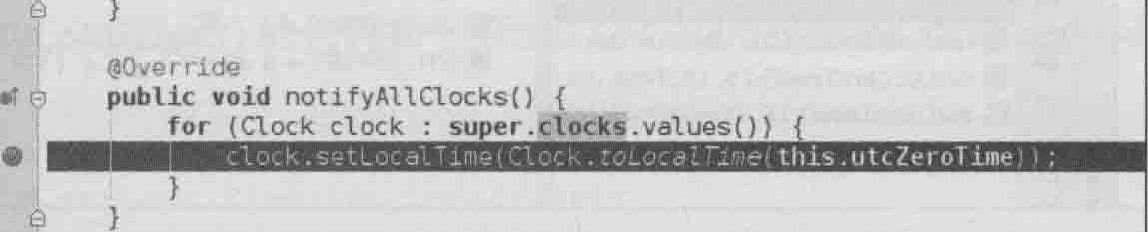
13 .gitignore

31 book-tammg-bad-code-waterfatl.iml m pom.xml R README, md

public class UtcTime extends TimeSubject { private int utcZeroTine;

B public int getUtcZoroTime() { return utcZeroTine;

=j public void setUtcZeroTime(ljit utcZeroTime) { this.utcZeroTiae » utcZeroTime; notifyAHCLocks();



-,-rj ■■ T-jEV-— ~^~~ -~r^!~—

Ca Debugger 回 Console E> Hi Frames \*\* "fnain\*®! in gr... K

S this - {tbc.waterfall.(JtcHme@415}

5 = {java.utii.HashMap$Valueiterator@421}

S clock - {tbc.waterf3ii.CltyClock@422>

!8I locatTime«0

super.docks - {|ava.utii.Ha^hMGp@423> size = 5 ▼ S |〇) = {j»«.util.HashMap$Entry@426>"london\* -> {tbc.waterfall.CityClocic@422> ► 3 java.lang.String - lane ^tnn-5^42' —london\*

▼ S value： tbc.waterfall.CityCtock => {tbc.waterfal(.CityClock@422>

[ffl localTime \* 0

* S |1| = {java.utii.HashMap$Entry@428>"moscoW -> {tbc.waterfaIl.atyCloclc@430}
* B [21 - {{ava.util.HashMap$Entfy@431>"bei{ingr •> {tbcwaterfall.CityCtock®433}

Debug ^ HotetWoridCloctisRunneT-

©

S Variables

控set

.etUtcZeroTl

ietLocamme

ie(hl5^JtcTTm^Sc|

i()：16, FhoneClock (tba

^ © maln()：17. HotefWorldClocIcsRunneij

Q

m

圏1

,§:TODO H Termin«l % g; Changes

图4-3首次进入for循环中的变量值

“刚刚进人这个for循环时，clock的值用@422来标记。在下面的clocks这个Map里面， 对应@422的是伦敦的时钟，它的localTime被初始化为0, 一切正常。按F8键让程序继续 执行，结果停到了第2个断点上，即Clock类的静态方法toLoca]Time()里面那条return语 句上。”

第2个断点：Clock类的静态方法toLocalTime()里面那条return语句的变量值如图4-4

所示。

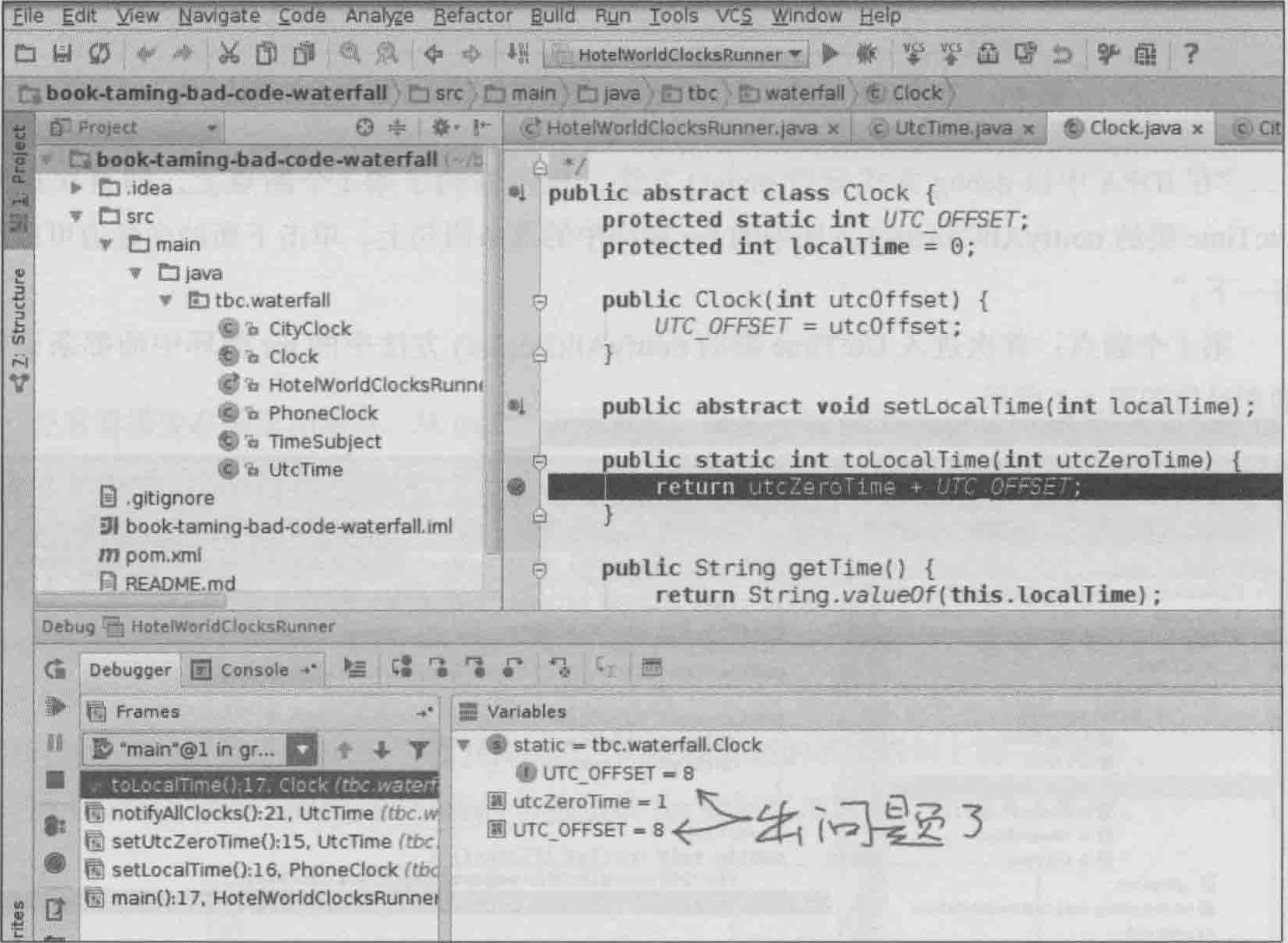


图4-4 Clock类中那条return语句的变量值

“这里就有问题了。Clock类的静态方法toLocalTime()所做的事情，应该是将传人 的UTC时间，加上伦敦与UTC时间的时差UTC\_OFFSET,来生成伦敦的当地时间并 返回。从下面的变量值来看，传入的UTC时间是1，这没有错，因为在那个main()方 法中，phoneClock创建时传人的时差是北京与UTC时间的时差8,后面phoneClock. setLocalTirae(9)中设置的本地时间是9点，所以UTC时间应该是1。而伦敦与UTC时间的 时差UTC\_OFFSET,在main()方法创建伦敦时钟时已经通过构造器的参数设置为0 了，这 .里却是8。”

分析得没错。问题出在Clock类中的成员变量UTC\_OFFSET不应该是静态的。在Clock 类中设计一个成员变量UTC\_OFFSET的初衷是让该类的每一个实例，即城市时钟或手机时 钟，都拥有自己的UTC\_OFFSET,以表示每个时钟所在城市与UTC时间的各自的时差。而 UTC\_OFFSET目前是Clock类的静态的成员变量，静态成员变量意味着该变量属于Clock 类，被每一个该类的实例所共享，所以把UTC\_OFFSET定义为静态的就实现不了上述初衷。

“对。静态的UTC\_OFFSET现在被各个Clock类的实例所共享，每一个Clock类的实例 创建时，构造器都会重写这个静态成员变量。在tmin()方法中最后一个创建Clock类的实例 是new PhoneClock⑻语句，所以UTC\_OFFSET最终被设置为8 了，并且令所有的Clock类 及其子类的实例的UTC\_OFFSET都是8,而UTC时间是1,两者相加就是本地时间9。难怪 所有城市时钟的本地时间最后全是9呢。”

现在首先要把Clock类的UTC\_OFFSET成员变量的static关键字给去掉。

去掉Clock类的UTC\_OFFSET成员变量的static关键字的代码如下所示（CM: The field UTC OFFSET of class Clock must not be static so that all subclasses could have its own UTC\_ OFFSET. That's the root cause of the bug that the local time of all city clocks are set to the same time.)：

public abstract class Clock {

- protected static int UTC一OFFSET;

+ protected int UTC一OFFSET;

protected int localTime = 0;

既然成员变量UTC\_OFFSET不再是静态的了，那么Clock类的静态方法toLocalTime() 也没有存在的必要了。

删除Clock类的静态方法toLocalTime()的代码如下所示（CM: Removed the static method Clock.toLocalTime().)：

public abstract void setLocalTime(int localTime);

public static int toLocalTime(int utcZeroTime) { return utcZeroTime + UTC\_OFFSET;

public String getTimeO {

return String.valueOf(this.localTime);

Clock类的静态方法toLocalTime()被删除了，但将UTC时间转换为时钟所表示的当地 时间并赋值给Clock的实例的成员变量localTime这件事还是需要做的，所以可以在Clock类 上增加一个接口，用来做这件事。这个接口可以叫setLocalTimeFromUtcZeroTime(),类图也 需要添上这个接口，这应该是8号改动。

在细化后的类图中8号更改如图4-5所示。

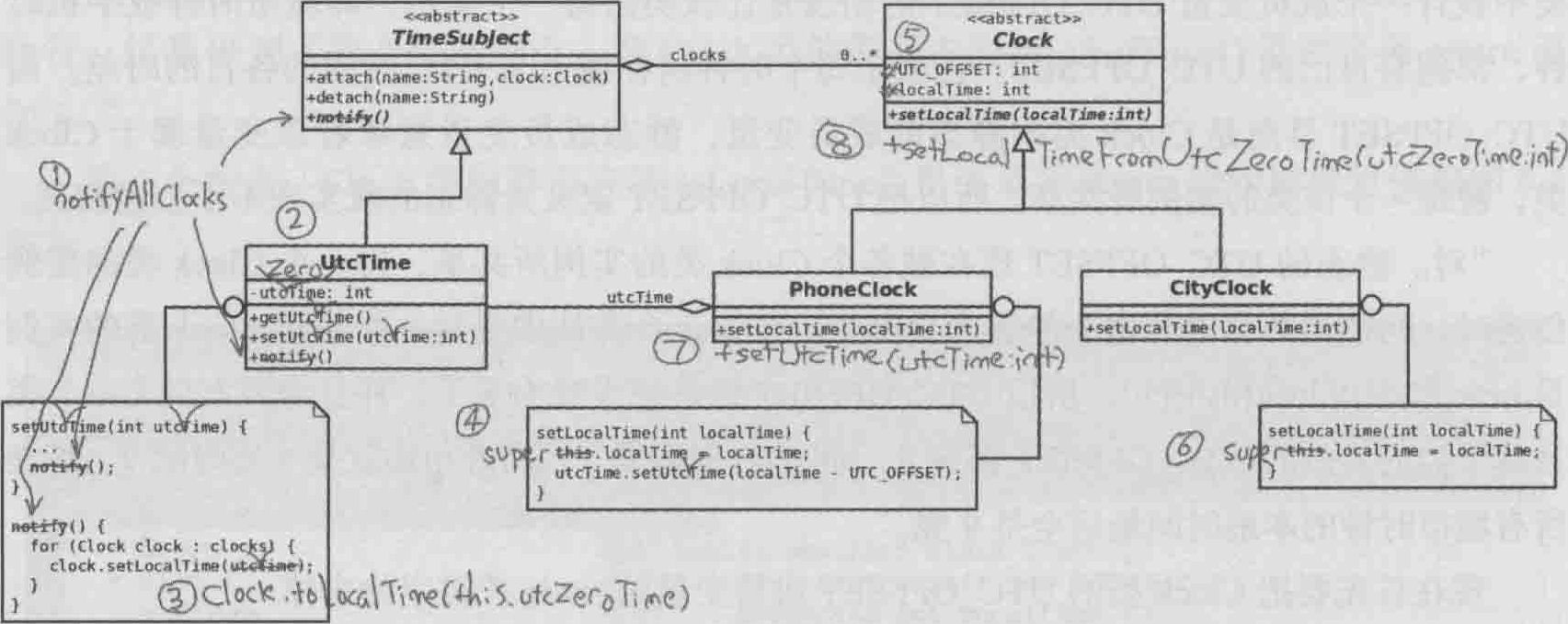


图4-5在细化后的类图中8号的更改

在 UtcTime 类中的 notifyAllClocks()方法中，将对 Clock 类的接口 setLocalTime()的 调用改为对新接口 setLocalTimeFromUtcZeroTime()的代码如下所示（CM: Expected class Clock to have an interface of setting local time from UTC zero time in method UtcTime.notify- AllClocksQ.)：

public void notifyAllClocks() {

for (Clock clock : super.clocks.values()) {

- clock.setLocalTime(Clock.toLocalTime(this.utcZeroTime));

+ clock.setLocalTimeFromUtcZeroTime(this.utcZeroTime);

}

}

“这个新接口还未实现，还是用Alt+Enter快捷键来让IDEA帮助实现它。”

实现 Clock类的新接口 setLocalTimeFromUtcZeroTime()的代码如下所示（CM: Created method Clock.setLocalTimeFromUtcZeroTime〇.)：

public String getTimeO {

return String.valueOf(this.localTime);

}

+

+ public void setLocalTimeFromUtcZeroTime(int utcZeroTime) {

+ this.localTime = utcZeroTime + this.UTC\_OFFSET;

+ }

}

再运行一下main()方法。

第2次运行main()方法后的结果如图4-6所示。

Run rh HotelWoridClocksRunner

/opt/jdkl.7.0\_51/bm/]ava -Didea.launch london： 1 ■oscov： 5

t X- ^ a © ▲黼 MesHQ

filing： 9 ^一

newYork: -4 — r\ \* J\*一

Sydney： 11

Process finished with exit code 0

图4-6第2次运行mainO方法后的结果

城市时钟的当地时间基本上都正确了，除了纽约的时间是-4,基本上都正确了，还有 些问题。这个问题的原因是咱们目前用整数来代表时间，当时间运算超出0到23这个范围 时，程序需要处理，否则就会出现-4这种情况。咱们可以找到进行时间运算的地方，就 在 Clock类中的 setLocalTimeFromUtcZeroTime〇 方法中，可以将运算式 utcZeroTime+this. UTC\_OFFSET作为参数传人Clock类的新的makeHourWithinOTo23()静态方法里，这个静态 方法就是处理上述时间范围的。

在 Clock 类中的 setLocalTimeFromUtcZeroTime()方法中，将运算式 utcZeroTime+this. UTC\_OFFSET作为参数传人Clock类的makeHourWithinOTo23()静态方法的代码如下所示 (CM: Wrote client code of the expected new interface Clock.raakeHonrWithinOTo23() in Clock.set LocalTimeFromUtcZeroTime().)：

public void setLocalTimeFromUtcZeroTime(int utcZeroTime) {

- this.localTime = utcZeroTime + this,UTC\_OFFSET;

+ this.localTime = Clock.makeHourWithinOTo23(utcZeroTime + this.UTC OFFSET);

“Clock类的这个makeHourWithinOTo23〇静态方法还未创建，下面就创建它。”

Clock类的静态方法makeHourWithinOTo23()的创建代码如下所示（CM: Created and implemented method Clock.makeHourWithinOTo23().)：

public void setLocalTimeFromUtcZeroTime(int utcZeroTime) {

this.localTime = Clock.makeHourWithinOTo23(utcZeroTime + this.UTC一OFFSET);

}

+

+ private static int makeHourWithin〇To23(int hour) {

+ return (hour + 24) % 24;

十 }

}

“现在再运行一下mainO方法。”

第3次运行main〇方法的结果如图4-7所示。

|  |  |  |
| --- | --- | --- |
| ► | t | /opt/jdkl.7.0\_51/bin/java -Didea.laur "London: 1 |
| 騮 | 善 | moscov： 5 |
| 11 |  | beijmg： 9 nevYork： 20 |
| \_ | a | Sydney： 11 |
|  | © | Process finished with exit code G |

Run HotelWorldClocksRunncr

图4-7第3次运行main()方法的结果

siJIJOAJ?

“这次终于看起来没有问题了！”

虽然运行结果符合期望，但是这种测试后行的开发方法还是暴露出下面这些问题：

1. )文档经常与代码缺乏同步造成理解偏差。即使这个编程操练很小，然而咱们在纸上对 细化后的类图做了多达8处的修改。而且以后若继续开发或修复bug,还得继续修改这个文档 来同步。实际上，我们不仅要同步这个细化后的类图，还要在变化发生时同步前面所生成的诸 如需求列表、领域词汇表、领域模型类图和Use Case用例图这些文档。这个同步的工作如果 靠人来做，不仅费时费力，还不可靠。因为代码是程序员写的，所以维护像类图这样的文档与 代码一致这项工作，只能由程序员自己来做。但在项目进度的压力下，有几个程序员能坚持做 这样繁琐的维护工作呢？如果无法坚持同步文档，那么这就好比“刻舟求剑”，只要您不再刻 “记号”来同步文档，不断变化的代码会让您无法根据刚刚刻下的“记号”找到您需要的“剑”。 测试后行的开发方法是把文档作为开发团队成员之间沟通的基础工具的。而这些过时的文档， 随着时间的推移，会越来越离谱，最终会变成一个“撒谎的路标”，其效果还不如没有路标。 虽然能够通过一些工具来让代码与类图相互转换，但是转换过来的类图过一段时间还会过时。
2. )写mainO方法进行的测试无法让计算机自动判断软件行为是否符合预期而导致效 率低下。咱们编写的那个测试用的main〇方法，虽然帮助我们发现了城市时钟的时间全是 9点和出现-4点这些问题，但预期的执行结果是保存在人脑中的，每次测试都要手工运行 main〇方法，并靠眼睛和大脑去判断main()方法的输出结果是否符合人脑中的预期值，不但 劳神费力，而且效率低下。
3. 问题产生后没能立即发现而耽误修复时间。那个城市时钟的时间全是9点和出现-4 点这些问题，直到写mainO方法时才发现。能否早一些发现呢？每个公司的主管都知道，软 件缺陷发现得越早，所花费的时间和金钱就越低。如果程序员觉得不能让测试工程师太清 闲，就不写main()方法来测试，而直接把写完的生产代码丟给测试工程师去测试，那么经 常会出现这样的情况：上述问题会等到测试工程师读文档、写测试用例、搭建测试环境、运 行测试这一系列工作完成后才被测试出来；然后测试工程师会兴奋地在bug跟踪系统里填写 bug报告，并且贴各种截屏、抓各种log,然后报告给开发经理；开发经理会皱着眉头地按优

先级对这些bug排序，并分配给当初开发这个代码的程序员；于是这个程序员不耐烦地停下 手中的工作，打开bug跟踪系统，读冗长的bug报告，看各种截屏和丨Og,然后搭建环境重 现问题，并试图回想当初是如何编程的，如果幸运地找到了问题所在，这才能最后修复这个 问题。上面整个过程或许会长达好几天，像黑洞一样无情地耗费了测试工程师、开发经理和 程序员的大量工作时间，让他们没有足够的时间来做更重要的事情，导致程序员更没有时间 自己做测试，进而开启了恶性循环的黑洞。最终软件缺陷发现得越来越晚，浪费的时间和金 钱越来越多。如果程序员能在编写完有问题的代码后立即就知道这个问题，那么这个黑洞是 不是完全可以消除呢？

1. 照搬设计模式导致设计出不必要的抽象，编写出从未被调用的方法造成时间浪费。 TimeSubject这个父类和其子类Utc.Time是否可以合并呢？假如TimeSubject有多个子类，那 么把这些子类的共性提取出来而形成TimeSubject这个父类，还是可以理解的。但是目前父 类TimeSubject只有一个子类UtcTime。完全可以把子类UtcTime并入父类TimeSubject中， 当将来有TimeSubject类的新的子类时，比如出现了另一种新的世界计时系统时，再提取 TimeSubject这个父类也不迟。不必要的抽象增加了程序员理解代码的复杂性，无情地消耗着 程序员的宝贵时间。当初为什么要有TimeSubject类呢？都是照搬“四巨头”的类图惹的祸。 TimeSubject类的detach()方法和UtcTime类的getUtcZeroTime()方法是根据“四巨头”所画 的Observer设计模式的类图编写出来的，但是却没有被任何代码调用。程序员每次在维护代 码时，都会阅读这些从未被调用的代码，这会浪费程序员的宝贵时间。
2. 程序调试过程无法让计算机来替代并自动化地反复使用，导致代码维护时间剧增。 虽然我们的程序出问题时，用设置断点调试程序的办法，能仔细观察相关变量的取值变化， 并解决问题，但这个过程很繁琐：首先需要在IDE里把程序运行起来（有些系统甚至无法在 IDEe里运行，而只能看log这样的日志)，然后重现错误，观察错误表现，定位错误，猜测 出错点，并相应地设置断点，再debug运行程序，观察断点处的变量值，分析并查找错误。 这个过程是不是很麻烦？但最要命的不是麻烦，而是这个人工的过程无法让计算机来替代并 自动化地反复使用。如果每次发现bug都这样繁琐地做一遍这样的debug过程，那么debug 也将会像黑洞一样无情地耗费程序员的大量时间和精力。

如果把上面列出的5个问题归纳一下，可以看出测试后行的开发方法如下：

1. “文档经常与代码缺乏同步”，会让这些文档的读者一包括程序员、测试工程师、产 品专家等——在理解代码行为方面反馈迟缓。

2 ) “写tnainG方法进行的测试无法让计算机自动判断软件行为是否符合预期”，会让程 序员在感知代码问题方面反馈迟缓。

3) “问题产生后没能立即发现”，会让程序员、测试工程师和开发经理等所有与代码相关

㊀ IDE: Integrated Development Environment，集成开发环境。

的人在感知代码问题方面反馈迟缓。

1. “照搬设计模式导致设计出不必要的抽象和编写出从未被调用的方法”所增加的无谓 的复杂性，和“程序调试过程无法让计算机来替代并自动化地反复使用”，都会让程序员在 维护代码质量方面反馈迟缓。

从这个编程操练可以看出，测试后行的开发方法，会使我们在理解代码行为、感知代码 问题和维护代码质量方面都反馈迟缓，其后果就是浪费所有这些人的时间、精力和金钱。

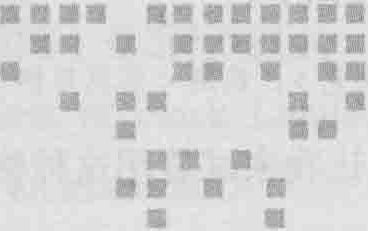
讨论了这些问题之后，咱们接下来用测试驱动开发（Test-Driven Development, TDD 这样测试先行的方法再做一遍这个编程操练，然后回过头来看看这些问题是否能够得到 解决。

在用TDD重做这个编程操练之前，咱们看看本章都做了哪些事：

1. 分析代码行为并加断点调试程序，找到了设计时出现的错误。
2. 在细化后的类图上更改这些错误。
3. 发现并修复了小时数为负数的问题。
4. 通过操练我们学到了以下知识：
5. 测试后行的开发方法所表现出来的问题包括：文档经常与代码缺乏同步造成理解偏 差，写main()方法进行的测试无法让计算机自动判断软件行为是否符合预期导致效率低下， 问题产生后没能立即发现耽误修复时间，照搬设计模式导致设计出不必要的抽象和编写出从 未被调用的方法造成时间浪费，程序调试过程无法让计算机来替代并自动化地反复使用导致 代码维护时间剧增。
6. 用测试后行的开发方法所开发出来的代码，会使与代码相关的所有人，在对代码的行 为理解、问题感知和质量维护方面都反馈迟缓，其后果就是浪费这些人的时间、精力和金钱。

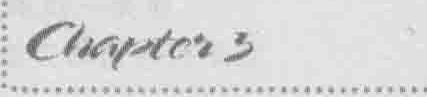
© #JaL： <http://en.wikipedia.org/wiki/Test> ^ Drive Development

用TDD重做编程操练题目



0

**第5章**



现在让我们用TDD 0开发方法，再重新做一遍0关于酒店世界时钟的编程操练。完成后 再和前面的测试后行的开发方法做一个对比。

“需求没变吧？”

没变。不过从现在开始，请不要再说“需求”了，要说”产品特性"（Feature)。因为“需 求”这个字眼带有1虽制、专制和持久”的内涵®，这和软件开发要“适应变化”的特点不符。 想想我们在第一个编程操练中曾把需求都编上号的一个暗示是什么？这暗示着这些编上号的 需求就不能再改了。如果要改，就得走需求变更流程。如果这是一个固定总价的软件开发合 同中的需求，要想实现这些变更，或许就意味着客户要另外支付费用。这样一来，一个不愿 支付额外费用的客户又如何能适应软件开发的变化呢？

相比前面把需求编号的做法，咱们现在不妨换一种方式来描述产品特性。即在卡片或报 事贴上写User Story® (用户故事)，来重新描述第一个编程操练中的那个题目。

User Story可用多种常用的格式®来编写，比如，使用下面这种最经典的格式：

As a <role>, I want <goal/desire> so that <benefit>.

㊀由于篇幅所限，本书所列的代码只能是一些片段，无法随时看到代码全貌。为能够最有效地体会本书编程 过程中的代码的变化，请读者打开自己计算机上的IDEA,跟着本书的描述一步步地输人代码并运行，仔 细体会，必有收获。

㊁ Kent Beck, Cynthia Andres，《Extreme Programming Explained: Embrace Change》，Addison-Wesley, 2nd Edition, November 26, 2004。

㊂User Story，是在软件开发中所写出的几句文字，来用软件最终用户所习惯的语言描述该用户的产品特性。

参见：<http://en.wikipedia.org/wiki/User_storyo> ㊃参见：[http://en.wikipedia.org/wiki八Jser\_story。](http://en.wikipedia.org/wiki%e5%85%abJser_story%e3%80%82)

翻译成中文如下：

作为一位< 角色 >,我想<实现目标/拥有愿望 >来<获得收益>。

“有意思！让我来试着用这种格式写一下。”

作为一位酒店大堂服务员，我想在大堂里北京、伦敦、莫斯科、悉尼和纽约这些城市时钟不准时，用设置自 己手机时间的方法，自动统一调整这些城市时钟的时间，来避免逐一根据时差调整这些时钟的繁琐工作。

看起来不错，格式是对了。不过这个User Story依赖酒店大堂所挂的时钟所表示的城 市。要是酒店新增或减少了城市时钟时，是不是还要修改这个User Story呢？编写出来的 User Story最好能像一个独立0的成年人，而不要像一个依赖家长照顾的孩子。不妨改成下面 这样：

作为一位酒店大堂服务员，我想在大堂的城市时钟不准时，用设置自己手机时间的方法，自动统一调整这些 城市时钟的时间，来避免逐一根据时差调整这些时钟的繁琐工作。

“那像有多少城市时钟和每个城市的时差这样的细节，也是需求，啊，不，是产品特性 的一部分呀。该在哪里表达呢？”

这些细节留到后面与客户或代表客户的产品专家沟通时再确定，然后把要点写在卡片的 背面，以做进一步沟通之用。不过有一点需要注意，User Story仅是一个用来进一步沟通的 指路牌，没有必要在上面写上所有细节。

“现在有了 User Story,那么接下来该做什么呢？既然咱们现在换成了 TDD开发，是不 是要先写测试呢？”

在TDD开发中，测试确实要在生产代码之前写。刚刚写好的User Story就可以指导咱 们进行测试。不过在写测试之前，咱们需要想一想，第一个测试要测什么？或者说这个测试 要测试生产代码所要完成的哪个功能？

“把前面那个nrninO方法里所测试的功能拿来写测试怎么样？”

可以是可以，不过main()方法里一口气测试了 5个城市时钟，是不是太多了？将来一个 城市时钟测试失败时，还得打开这个包含5个城市时钟的测试来看究竟是哪个失败，太麻烦 了。咱们不妨每个测试只测试一个城市时钟的时间，以便于定位错误。

目前已知有北京、伦敦、莫斯科、悉尼和纽约这5个城市的时钟，这5个城市与UTC 时间的时差分别是+8、0、+4、+10和-5,与UTC时差为0的伦敦时钟最简单，所以不妨 先测试伦敦时钟，即要测试在手机设置为北京时间9点后，伦敦时钟会不会被设置为凌晨 1点。

㊀出自编写User Story的INVEST 口诀，即要把User Story编写成：独立的（Independent)、可商量的 (Negotiable)、有价值的（Valuable)、可估算的（Estimable)、合理规模的（Sized appropriately)和可测试的 (Testable)。参见：<http://en.wikipedia.org/wiki/INVEST_(mnemonic)0>

要测试这一点，首先要有一个伦敦时钟的对象，比如叫londonClock。这个对象应该有 一个getTimeO方法，用来取出该时钟的时间，然后和期望的1点进行对比。

下面就是这第一个测试的代码（CM: Added test the\_time\_of\_clock\_London\_should\_be\_l\_ after\_thej3hone\_clock\_is\_set\_to\_9\_Beijing\_time() and wrote an assertion.)：—

public class HotelWorldClocksTest {

@Test

public void the\_time\_of\_clock\_London\_should\_be\_l\_after\_the\_phone\_clock\_is\_ set\_to\_9\_Beijing一time() {

// Arrange //Act // Assert

assertEquals(1, londonClock.getTime());

“这个测试的方法名可真够长的。”

对。虽然长一些，但这样可以写得像一句通顺的英语® (中文意思：当手机时钟被设置为 北京时间9点后，伦敦时钟的时间应该是1点。)，并且用下划线连接每个单词，以便于 阅读。

“这里面的Arrange、Act和Assert这3行注释是什么意思？”

这表示了一个测试通常要做的3件事，即Arrange是做一些测试前的准备工作，Act是 调用要测试的方法来运行，Assert是判断上面运行的结果是否符合预期。

“嗯，测试一般都按这个顺序来做这3件事。不过为什么要空着前两件事不做，反而先 做第3件事，即写assert语句呢？”

您有没有听说过“分形”这个概念？英文叫Fractal。

“没有。”

“分形”指的是这样一种曲线或图案，它包含了和自己的形状完全相同的一些小的曲线 和图案。就好比人类的父母生出的孩子都和自己很像一样。而这种相似性是不是会产生一种 和谐的结果呢？在大的测试驱动开发方面，我们是用测试来反向驱动出生产代码的。而在其 中小的一个测试里面，我们又用先写Assert来反向驱动出Act和Arrange这两步。这是不是 体现出分形的和谐？另外，代码关键字和语句都是英文，如果由此衍生出来的代码中的命名

㊀本章源代码参见以下链接：[https://github.com/wubiii28/tbc-liotel-world-docks-test-first。](https://github.com/wubiii28/tbc-liotel-world-docks-test-first%e3%80%82)

©如果团队成员大多不说英语，可以把测试方法的命名写成中文。像Java这样现代的编程语言都支持中文 的代码命名。测试一方面是固化生产代码的行为的固化剂，另一方面也是描述生产代码行为的文档。只要 测试命名这样的文档和代码的行为能够做到“知行合一”，那就达到目的了。

和注释也是英文的话，那么是不是也很和谐？

“嗯，明白了。”

咱们先看看这条assert语句。assertEquals()方法是org.junit.Assert类的一个静态方法， 用来判断两个值是否相等。这个方法有两个参数。第一个参数是期望值，这里是数值1，表 示这个assert语句期望伦敦时钟的时间是1点。第二个参数是londonClock变量的getTime() 的返回值。变量londonClock还未定义，不过能否通过它来看出咱们的生产代码的接口设 计呢？

“londonClock变量的类型应该是一个类，这个类有一个成员方法getTime〇。”

没错。这样就能从测试代码“驱动”出生产代码了。这种在变量定义前直接写出使用该 变量的代码的编程方式，咱们已经在前面操练时用到过了。它还有一个名字——“意图式编 程” (Programming by Intention) e〇

写完了 Assert，咱们现在可以接着写这个测试的Act。如何才能让londonClock的时间 变成1点呢？需要把手机时钟设置为9点，另外手机时钟还需要持有londonClock，以便于 把手机时钟设置为9点时，顺便把londonClock设置为1点。另外还需要在assertEquals上按 Alt+Enter快捷键生成import语句。

在 HotelWorldClocksTest 类中设置 9 点和 londonClock 的代码如下所示（CM: In the test, asked the phoneClock to hold the londonClock and set the time of the phoneClock to be 9 so that the phoneClock could update the time of the londonClock.)：

+import static org.junit.Assert.assertEquals;

+ phoneClock.setCityClock(londonClock);

+ phoneClock.setTime(9);

接下来该写Arrange 了。在前面Assert和Act里面的londonClock和phoneClock变量, 在这里都应该被定义，并被初始化。另外，伦敦与UTC时间的时差是0,手机所在的北京时 间与UTC时间的时差是8,这两个时差值也应该在实例化CityClock和PhoneClock类时，作 为构造器的参数，分别传到这两个实例中。

在HotelWorldClocksTest类中定义并初始化londonClock和phoneClock变量的代码如下 所示（CM: Instantiated a CityClock and a PhoneClock with UTC offset in the test.):

// Arrange

+ CityClock londonClock = new CityClock(0);

+ PhoneClock phoneClock = new PhoneClock(8);

㊀参见：<http://www.netobjectives.com/resources/programming-intention>

现在，在这第一个测试方法中，有7处代码显示了编译错误的红色。不要畏惧这些 错误，反而应该感谢它们。因为这些错误就像茫茫大海中的灯塔，为咱们继续前进指明 了方向。

“这块我来做吧。从上往下一个一个地解决这些错误。首先是CityClock是红色的，表示 这个类还未创建。按Alt+Enter快捷键来创建它。”

创建 CityClock;类的代码如下所示（CM: Created class CityClock.):

♦public class CityClock {

+ }

“这个类里该写什么内容呢？”

先空着它吧。俗话说，不见兔子不撒鹰。等后面需要它的时候再写不迟。

“接下来再用Alt+Enter快捷键解决CityClock类缺少一个带有参数的构造器的错误。”

创建带有参数的CityClock类的构造器代码如下所示（CM: Created constructor of CityClock with a UTC offset time as the parameter.):

public class CityClock {

+ public CityClock(int utcOffset) {

“下一个要解决的错误是PhoneClock类未创建。这里也用Alt+Enter快捷键创建。”

创建 PhoneClock 类的代码如下所示（CM: Created class PhoneClock.):

^public class PhoneClock {

+ }

“接下来还是要解决PhoneClock类缺少一个带有参数的构造器的错误。用Alt+Enter快 捷键创建这个构造器。”

创建带有参数的PhoneClock类的构造器代码如下所示（CM: Created constructor of PhoneClock with a UTC offset time as the parameter.)：

public class PhoneClock {

+ public PhoneClock(int utcOffset) {

+

+ }

}

“接下来的错误是PhoneClock类少了一个setCityClock〇方法。还是用Alt+Enter快捷键 创建。”

在 PhoneClock 类中创建 setCityClock()方法的代码如下所示（CM: Created method Phone-

Clock.setCityClock().)：

+ public void setCityClock(CityClock cityClock) {

+

+ }

“接下来的错误是PhoneClock类还缺少一个setTime()方法。”

在 PhoneClock 类中创建 setTime()方法的代码如下所示（CM: Created method PhoneClock. setTime().)：

+

+ public void setTime(int time) {

+

“最后还有一个CityClock类缺少getTime()方法的错误。用Alt+Enter快捷键补上这个 方法，

能不能让这个方法返回一个假数据，好让这个测试尽早通过呢？

“应该可以的。只要返回1,测试就肯定通过了。不过这么写代码不是太烂了吗？”

烂有烂的好处。头一个好处就是测试运行通过了，这样好让咱们结束上一环节的工作， 进入下一环节的重构。另一个好处就是能让咱们看到重构的方向。先写出这段烂代码，然后 我告诉您它怎么就指出了方向。

在类CityClock里创建getTime()方法并返回1的代码如下所不（CM: Created method CityClock.getTime() and made the test green with a fake value.)：

+ public int getTime() {

+ return 1;

+ }

按Alt+l快捷键进人左侧的Project工具窗口，将光标定位到HotelWorldClocksTest类 上，然后按Ctrl+Shift+FlO组合键来运行这个测试。测试变绿，运行通过。

在进一步发现重构方向之前，咱们先看一看本章所做的事情。

1. )用User Story的格式，重新描述了产品特性。
2. )根据User Story的描述，从所有待测的城市时钟中挑选了一个情况最简单的城市时钟 来编写第一个测试。
3. 在编写生产代码之前，就在测试代码中编写了符合自己意图的代码，并通过修复意 图代码的编译错误来驱动生成了生产代码。
4. 通过操练我们学到了以下技能：
5. 把以前所提到的带有强制、专制和持久色彩的“需求”一词，换成了“产品特性”这 样基于沟通的字眼，能够让产品特性在频繁沟通的保证下更加具有价值。
6. 把测试方法的名字写得很长，并在单词之间加上空格，使得测试方法名成为易于阅读 的文档。
7. 一个测试通常包括Arrange、Act和Assert这3部分，分别表示测试准备、调用待测 方法和验证结果是否符合预期。
8. 在一个测试中先写最后的Assert部分，然后推导出Act和Arrange这两部分代码，会 更加符合TDD的测试驱动风格，营造出分形的和谐气氛。
9. 没有适当的面向对象的设计，就无法写出合理的意图代码。所以在测试中编写意图代 码就能促进在写测试代码时做出适当的设计。
10. 在测试中写好了所有的意图代码后，由于生产代码还未编写，势必有许多编译错误， 而这种用修复意图代码编译错误的方法来驱动出生产代码的做法，能够让生产代码的开发过 程更加有方向性。
11. 在逐个解决上述编译错误的过程中，只写能让编译通过的、尽量少的代码，比如空类 或空方法，而把编写其具体实现的内容留到后面运行测试出错时再写，同样能够让生产代码 的开发过程更加有方向性，且有助于编写出不多不少并能让测试运行通过的代码，避免了时 间的浪费。
12. 用直接返回假数据的方法，让测试尽快运行通过，这样能够快速结束当前编写测试的 环节，进人下面的重构环节，并能进一步指出重构的方向。

消除假数据所带来的重复代码

“咱们用直接返回假数据1这个烂代码让测试运行通过了。接下来要做什么呢？”

接下来就要对付这个烂代码。您读过Martin Fowler的《重构》e—书吗？知道最难闻的 代码“腐臭”是什么吗？

“读过。不过熊节翻译的中译本把Code Smell翻译成代码的坏味道。最难闻的当属重复 代码。”

很好。我觉得由于中西方观念的差异，外国人很在意的坏味道，在中国这样人口稠密的 国家，就会不那么引人注意。比如在北京南城每天步行接送孩子上下学的人们，对于街上空 气中经常会有的坏味道早已司空见惯。所以我觉得用“腐臭”这个字眼更能引起我们的重视。 大家都希望自己能流芳千古，而不要“腐烂发臭”。

好了，言归正传。在这个返回假数据1的代码中，您能闻到重复代码的“腐臭”吗？ “没看到重复呀。”

再仔细看看，这个在生产代码里直接返回的假数据1,还在哪里出现了？

“那只能是在测试代码中那个assert语句中出现了。”

生产代码中直接返回假数据1的代码如下所示：

public class CityClock {

public int getTimeO {  
return 1;

© MartinFowler著，熊节译。《重构：改善既有代码的设计》，人民邮电出版社，2010年4月第1版

测试代码中进行判断的assert语句中出现1的代码如下所示：

public class HotelWorldClocksTest {

@Test

public void the\_time\_of\_clock\_London\_should\_be\_l\_after\_the\_phone\_clock\_is\_ set一to\_9\_Beijing一time() {

// Assert

assertEquals(1, londonClock.getTime());

}

}

对。在生产代码与测试代码之中所出现的重复代码也必须去除。刚才说过，烂代码能给 咱们指出重构的方向。那么这种重复，就是一个要重构的方向。

“明白了。另外我还观察到，CityClock和PhoneClock这两个类，都有一个以UTC时差 utcOffset作为参数的构造器。这个参数传进构造器后，都会保存在各自的类的成员变量里。 这样一来，这两个相同类型和用途的成员变量之间也出现了重复。”

没错。将来咱们可以在这两个类之上，再提取一个父类，然后把这两个类中的utcOffset 成员变量都移动到父类中，这样就能消除重复。除此之外，对现在的代码咱们还有什么可以 进一步做的任务吗？

“目前的测试仅仅测试了设置手机时钟后，伦敦这一个城市的时钟的调整情况，还没有 测试同时调整多个城市时钟的情况。”

对。另外，咱们的测试目前只考虑了比UTC时间早的城市，还没有考虑比UTC时间晚 的城市，比如纽约，后者可能会出现时间是负数的情况。

饭要一口一口地吃，目前咱们发现的这4个任务，也要一个一个地做。俗话说：好 脑子不如烂笔头。为了防止咱们在做的过程中忘记哪个任务，先把这4个任务，在代码 中相关的位置上，都一一写上TODO注释。这样一来，咱们只要按Alt+6快捷键，再按 Ctrl++e快捷键展开所有的TODO,就能方便地调出来查看。这4个任务中，消除假数据 1那个重复代码可以先做，所以把它写成TODO-working-on,表示咱们正在解决这个 任务。

按Alt+6快捷键后，这4个TODO任务显示在IDEA界面底部，如图6-1所示。

“该如何消除假数据1这个重复代码的‘腐臭’呢？”

这个假数据1在测试和生产代码中各出现一次。在测试代码中咱们没法把它变成变量的 形式，但在生产代码中，咱们是可以把它转变成变量的形式的。想想看，在CityClock类中 的getTime()方法里，哪些变量运算后能得到这个数值1呢？

0 Ctrl++表示先按Ctrl键并保持住，然后按键盘上的加号键。

'*Crt#M b/ tnt\_ 〇fi* 14-5\*13.

i CityClock {

*The const rumors of (*

\* t'lava \* ^]tl

TOOO Cufr\*nt Flle

图6-1 IDEA界面底部显示4个TODO任务

£ile Edit yiew ^avtgate Code

Befactor fiuild Ryn loots vCi Window Help

Q hotelwortddocks . C3 src，D main D lava，B tbc .. 53 tdd . E3 hotelworlddodcs < CityClock ；

Bj jP ffr»»art . • 0 •； ❖ . IWorldClotkBTest.ji

f hotelworlddodw ' y»：：:.ta(i/-ur；〇f e；1 ' patkaqc

Ar' ► # S Q

：PhaneClockjavajt ft CityClock.|ava x [

3. Idea D .settings Dwc ，D ma，n \*» Djava

\* £3 tbc.tdd.hotelworidclocks

| public class Qtj

1000： The constructors ol CityCl» !穸 public CityClock(lnt utcOffset) {

■ CttyClod » Phoned

：lock

C] tbc.tdd.hotdworklc)ocks n HotefworWCIocksTest

public int getTi»e() {

r〇〇〇々oricing\_(M: n» fate t^Iue in tlae prodbctJaa < return 1：

*I the expected v^lue in the*

*)*

■e f Found 4 TODO items in 2 files

▼ J£j tbc.tdd.hotelw〇r1dck)dts (4 it^rm in 2 Kes) t 〇 CityClock.Java

m

画（7. 8>TODO: Fhe consfoirfors of CifyCtodr PhoneCtock are dup"caferf

J jri-i A<'y'^T#~iryfT7T7/??T3^yTjm

：-s；

B (25. 8} *H TODO: Negative hour* H (26. 8) *)t TODO: Set time to multiple <Ay docks*

■

I3JJ If s UIF-S :

“这个数值1其实表示的就是伦敦的当地时间凌晨1点。这个当地时间可以用UTC时间 与该城市与UTC时间的时差相加得到。也就是说：当地时间=UTC时间+当地时间与UTC 时间的时差。”

没错。在CityClock类中的getTimeO方法里，这个假数据1用两个变量之和来替代的代 码如下所示（CM: Replaced the fake value with the intention code in method CityClock.getTime().):

public int getTime() {

// TODO-working-on: The fake value in the production code and the expected value in the test code are duplicated

- return 1;

+ return this.utcOffset + this.utcZeroTime;

}

这里，上面两个变量之和，就是咱们的意图代码。其中，this.utcOffset表示当地时间与 UTC时间的时差，this.utcZeroTime表示UTC时间。

“现在这行意图代码中又出现了两个红色的编译错误，表示这两个成员变量尚未定义。 让我来一个一个地用Alt+Enter快捷键来对付它们吧。”

先对付utcOffset。在类CityClock中创建utcOffset成员变量并在该类构造器中将其赋值 的代码如下所示（CM: Created field utcOffset in class CityClock and assigned a value to it in the constructor.)：

public class CityClock {

+ private int utcOffset;

// TODO: The constructors of CityClock and PhoneClock are duplicated public CityClock(int utcOffset) {

+ this.utcOffset « utcOffset;

}

再对付utcZeroTime。在类CityClock中创建utcZeroTime成员变量的代码如下所示 (CM: Created field utcZeroTime in class CityClock.)：

public class CityClock {

private int utcOffset;

+ private int utcZeroTime;

“这个CityClock类的utcZeroTime成员变量，该由谁给它赋值呢？”

应该可以由PhoneClock这个手机时钟类，在它调用setTime()方法时，根据传入的 当地时间和手机时钟的UTC时差，来计算出UTC时间，并给它所持有的CityClock对 象的utcZeroTime成员变量赋值。为了能做到这一点，我希望在PhoneClock类里面，能 有一个类型为CityClock类的成员变量cityClock。而且我希望能够调用CityClock类的 setUtcZeroTime〇 方法，来为 cityClock 的 utcZeroTime 成员变量赋值。

在PhoneClock类中的setTime()方法中为该类所持有的CityClock实例设置UTC时间的 意图代码如下所75 (CM: Wrote intention code in method PhoneClock.setTime〇.):

public void setTime(int time) {

+ this.cityClock.setUtcZeroTime(time - this.utcOffset);

}

“兔子来了，严阵以待的setTimeO方法终于该撒出去了。”

哈哈，对。

“现在又可以沿着意图代码中的红色编译错误的指引来编程了。”

没错。上面这条语句中的this.cityClock有编译错误，表示PhoneClock类缺少一个名为 cityClock的成员变量。现在就创建它。另外，它应该在该类的那个空的setCityClock()方法 里被赋值。又盯上一只“兔子”。

在PhoneClock类中创建cityClock成员变量并赋值的代码如下所示（CM: Created field cityClock and assigned a value to it in class PhoneClock.)：

public class PhoneClock {

+ private CityClock cityClock;

public void setCityClock(CityClock cityClock) {

+ this.cityClock = cityClock;

接下来再处理前面那行意图代码中最后的部分this.utcOffset的编译错误。这个错误也是 由于PhoneClock类还未创建utcOffset这个成员变量而造成的。现在就创建它。另外，这个 成员变量应该在PhoneClock类的构造器中被赋值。

在PhoneClock类中创建utcOffset成员变量并赋值的代码如下所示（CM: Created field utcOffset and assigned a value to it in class PhoneClock.)：

public class PhoneClock {

private CityClock cityClock;

+ private int utcOffset;

public PhoneClock(int utcOffset) {

+ this.utcOffset = utcOffset;

最后处理前面那条意图代码中最后一个编译错误：CityClock类还未创建 setUtcZeroTime〇方法。先把光标定位到这个错误上，然后用Alt+Enter快捷键创建该方法， 并在这个方法里将参数utcZeroTime赋值给CityClock类的成员变量。

在CityClock类中创建setUtcZeroTime()方法并将其参数utcZeroTime赋值给该类的成员 变量的代码如下所示（CM: Created method CityClock.setUtcZeroTime() and assigned the UTC zero time to the field of class CityClock.)：

public class CityClock {

+ public void setUtcZeroTime(int utcZeroTime) { + this.utcZeroTime = utcZeroTime;

好了，所有的错误都修复了。看起来应该可以运行测试了。因为前面已经运行过测试 了，所以咱们只要按Ctrl+F5快捷键，就能重新运行测试。

按一下试试，测试运行通过。

现在可以把那个解决假数据1的TODO删掉。按Alt+6和Ctrl++快捷键看一眼剩下的 TODO,并从中挑选下一个要处理的任务。

在挑选下一个TODO之前，咱们先看看本章都做了什么工作：

第6幸消除假教据所带来的重复代码 U

1. 在用假数据让测试快速通过的情况下，找到了在测试代码和生产代码之间存在重复 代码的“腐臭”，并在生产代码中用变量来替换假数据的方法来消除重复代码。
2. 把当时所有发现的代码“腐臭”和要继续做的任务都以添加TODO注释的方式，写 在代码相应位置，并方便地用快捷键调出查看，用来备忘并从中寻找下一个要处理的任务。
3. )把当前正在处理的TODO标记成TODO-working-on,以该TODO为导向，继续使用 意图式编程的方法，循着意图代码中的红色编译错误的指引，驱动出生产代码。

4)通过操练我们学到了：把在编程中随时发现的要处理的问题，在代码中相应的位置 写成TODO,并在将来使用快捷键Alt+6来查看，能让随时发现的问题不致中断现有的工作， 并且不会遗忘任何发现要做的事情。

^7於/於w •:第7章

让下一个测试足够有意思

“咱们已经测完情况最简单的伦敦时钟的时间调整了。接下来还有北京、莫斯科、悉 尼和纽约这4个城市时钟。反正现在城市时钟也不多，要不要一个接一个地把它们都测一 遍呢？”

要是这剩下的4个城市时钟所属的测试等价类6都各不相同，且与已经测过的伦敦时钟 的测试等价类也不相同，那么可以把它们一个一个地都测一遍。但在此之前，咱们首先要划 分一下测试等价类。

如果把城市当地时间与UTC时间的时差作为划分等价类的标准，那么就可以划分为3 类城市时钟：一类是当地时间比UTC时间早的城市，比如北京、莫斯科和悉尼；一类是当 地时间比UTC时间晚的城市，比如纽约；最后一类是当地时间与UTC时间相同的城市，比 如伦敦。

现在最后一个等价类的代表城市伦敦的时钟已经测过了。下一个测试就可以从另外两个 等价类中选一个城市来测试。

“选北京时钟如何？它的时间应该和手机设置的时间保持一致，足够简单。”

看起来不错。但是我觉得这样做没有意思。因为现有处理伦敦时钟这种当地时间与UTC 时差为0的代码应该也能够适用于北京时钟这样UTC时差为正整数的情况。换句话说，如 果咱们现在增加一个针对北京时钟的测试，那么这个测试在不修改任何生产代码的情况下就 能运行通过。这不够有意思。

㊀等价类划分就是解决如何选择适当的数据子集来代表整个数据集的问题，通过降低测试的数目去实现“合 理的”覆盖，覆盖了更多的可能数据，以发现更多的软件缺陷。一引自百度百科

而如果选第二个等价类的代表纽约时钟来作为下一个测试，那么根据目前的代码行为， 这种情况所得到的时间就会是负数，即Negative hour这个TODO任务所描述的情况。这会 驱动咱们重构生产代码，从而使这个测试运行通过。这就有意思得多了。

“嗯，是挺有意思的。”

接下来可以开始做这个Negative hour的TODO任务了。先把它改成TODO-working-on。 现在的TODO列表如图7-1所示。

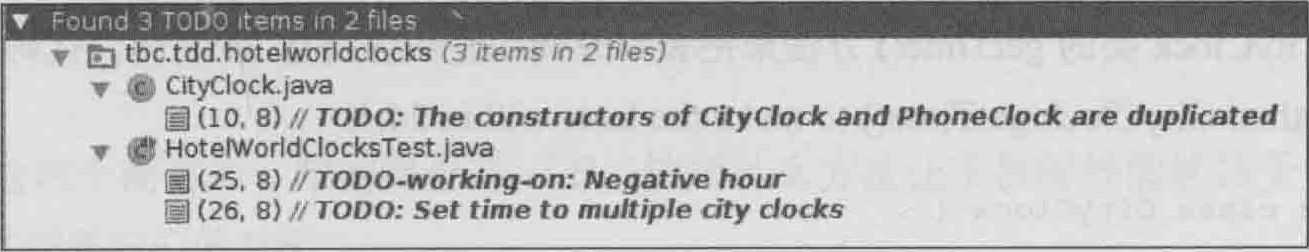


图 7-1 Negative hour TODO 任务

在测试类中为Negative hour这个TODO任务添加一个测试方法的代码如下（CM: Added test the\_time\_of\_clock\_NewYork\_should\_be\_20\_after\_the\_phone\_clock\_is\_set\_to\_9\_Beijing\_ time() and got a negative hour in the actual result.)：

public class HotelWorldClocksTest {

// TODO-working-on: Negative hour + @Test

+ public void the一time一of—clock一NewYork\_should\_be\_20\_after\_the\_phone\_clock is\_set\_to\_9\_Beijing\_time() {

+ // Arrange

+ CityClock newYorkClock = new CityClock(-5);

+ PhoneClock phoneClock = new PhoneClock(8);

+

+ //Act

+ phoneClock.setCityClock(newYorkClock);

+ phoneClock.setTime(9);

+

+ // Assert

+ assertEquals(20, newYorkClock.getTime());

+ }

这个纽约时钟的测试方法和前面那个伦敦时钟的测试方法很类似，只不过所有和伦敦时 钟相关的地方都换成了纽约时钟。写完这个测试后，按Ctrl+F5快捷键运行测试，发现在运 行工具窗口中出现下面的结果：

java.lang.AssertionError:

Expected :20 Actual :-4

“时间是负数的情况出现了。这是因为咱们目前是用正整数来表示时间，还没有处理时 间是负数的情况。”

由于最后是调用assert语句中的newYorkClock.getTime()方法来获得那个负数时间的， 所以咱们需要修改CityClock类的getTime()方法。咱们只要把这个方法中计算出来的时间结 果，加上24,再与24取模，就能让表示时间的整数保持在24之内。

修改CityClock类的getTime()方法来把表示时间的整数保持在24之内，代码如下（CM: Updated method CityClock.getTime() to make the hour within 24.)：

public class CityClock {

public int getTime() {

- return this.utcOffset + this.utcZeroTime;

+ return (this.utcOffset + this.utcZeroTime + 24} % 24;

再按Ctrl+F5快捷键运行下测试，运行通过！

咱们可以按Alt+6快捷键调出TODO列表看一下。现在可以把Negative hour这个 TODO删掉了。还剩下两个TODO, 一个是CityClock和PhoneClock这两个类的构造器有重 复，另一个是一次调整多个城市时钟。

“后面那个一次调整多个城市时钟的TODO看起来更有意思一些，咱们可以先处理它。” 好的。把它的TODO改成TODO-working-on。

现在的TODO列表如图7-2所示。

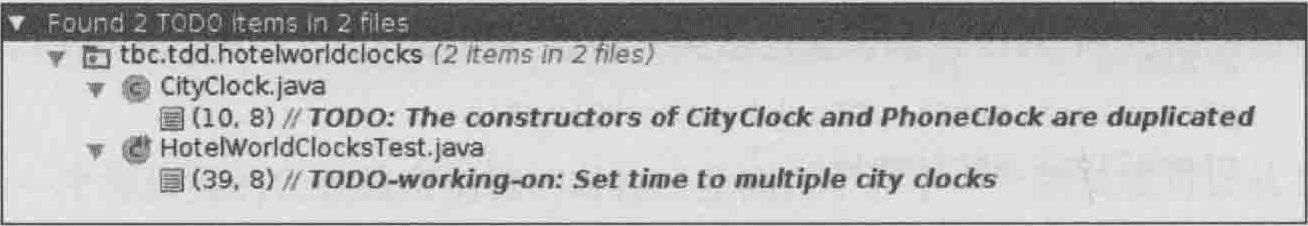


图7-2 —次调整多个城市时钟的TODO “怎样测一次调整多个城市时钟呢？”

咱们同样测试最简单的情况，比如只测试两个城市的情况：当将手机时钟设置成北京时 间9点时，所测试的伦敦和纽约时钟将会分别被调整为1点和20点。

上述测试和其中两个断言的代码如下（CM: Added test the\_time\_of\_clock\_London\_and\_ NewYork\_should\_be\_l\_and\_20\_respectively\_after\_the\_phone\_clock\_is\_set\_to\_9\_Beijing\_tirae() and wrote 2 assertions.)：

public class HotelWorldClocksTest {

// TODO-working-on: Set time to multiple city clocks

+ @Test

+ public void the\_time\_of\_clock\_London\_and\_NewYork\_should\_be\_l\_and\_20\_ respectively\_after\_the\_phone\_clock\_is\_set\_to\_9\_Beijing\_time() {

+

+ // Assert

+ assertEquals(1, londonClock.getTime());

+ assertEquals(20, newYorkClock.getTime());

+ }

}

写好这两个断言后，咱们得思考一下，该用什么方法让手机时钟能够接受多个城市时 钟，来一并调整它们的时间？

“咱们可以把多个城市时钟对象保存到一个ArrayList中，然后传给PhoneClock手机时 钟对象就可以了。”

这确实是一个最简单的做法。不过我有一个问题，这个ArrayList类对应业务领域中 的哪个领域类呢？我们不大可能让一个不懂Java的酒店大堂服务员，用自己的手机去和一 个ArrayList类打交道。咱们不妨创建一个“酒店世界时钟系统”（HotelWorldClockSystem) 领域类来封装这个ArrayList类。这样，咱们就可以把多个城市时钟添加到HotelWorld- ClockSystem类的一个对象中，然后把这个对象传递给上述PhoneClock手机对象，这样会更 加自然。

咱们已经写完了这个测试的Assert的意图代码，接下来就可以从Assert来推导出Act 的意图代码。要调整伦敦和纽约的时钟时间，必须把手机时钟设置为9点，所以可以写出 phoneClock.setTime(9)这个方法调用。

而要让手机时钟去统一更新多个城市时钟的时间，必须把包含多个城市时钟的对象 hotelWorldClockSystem传递给它，所以可以写出这样的方法调用：phoneClock.setHotelWorld ClockSystem(hotelWorldClockSystem)。这样 Act 就写好了。

“让我试试从Act推导出Arrange的意图代码吧。Act里的phoneClock对象需要实例化出 来，hotelWorldClockSystem对象也需要实例化出来。而londonClock和newYorkClock这两个 城市时钟的对象也需要添加到hotelWorldClockSystem对象中，这样HotelWorldClockSystem 类就需要有一个attach()方法。londonClock和newYorkClock这两个城市时钟的对象也需要 分别实例化。”

从Assert推导出Act,进而再推导出Arrange的意图代码如下（CM: Wrote intention code in test the\_time\_of\_clock\_London\_and\_NewYork\_should\_be\_l\_and\_20\_respectively\_after\_the\_ phone\_clock\_is\_set\_to\_9\_Beijing\_tirae().)：

@Test

public void the\_\_time\_of\_clock\_London\_and\_NewYork\_should\_be\_l\_and\_20\_ respectively一after\_the\_phone一clock一is一set\_to\_9\_Beijing\_time(> {

+ // Arrange

+ CityClock londonClock = new CityClock(0);

+ CityClock newYorkClock = new CityClock(-5);

+ HotelWorldClockSystem hotelWorldClockSystem = new HotelWorldClockSystemO;

+■ hotelWorldClockSystem. attach (londonClock);

+■ hotelWorldClockSystem. attach (newYorkClock);

+ PhoneClock phoneClock =\* new PhoneClock(8);

+

+ //Act

+ phoneClock.setHotelWorldClockSystem(hotelWorldClockSystem);

+ phoneClock.setTime(9);

// Assert

assertEquals(1, londonClock.getTime()); assertEquals(20, newYorkClock.getTime());

下面的工作就是循着上面这些意图代码中的红色编译错误，来开始写最少量的程序，以 使编译通过。

“先创建 HotelWorldClockSystem 类〇”

创建 HotelWorldClockSystem 类的代码如下（CM: Created class HotelWorldClockSystem.):

+public class HotelWorldClockSystem {

+ }

“再在这个类中创建attach()方法。”

在 HotelWorldClockSystem 类中创建 attach()方法的代码如下（CM: Created method Hotel- WorldClockSystem.attach().)：

public class HotelWorldClockSystem {

+ public void attach(CityClock cityClock) {

“PhoneClock 类中还少一个 setHotelWorldClockSystem()方法〇”

在 PhoneClock 类中创建 setHotelWorldClockSystem〇 方法的代码如下（CM: Created method PhoneClock.setHotelWorldClockSystem().)：

public class PhoneClock { + public void setHotelWorldClockSystem(HotelWorldClockSystem hotelWorldClockSystem) {

+ }

}

“现在编译错误没有了。咱们不妨按Ctrl+F5快捷键运行一下测试。哦，PhoneClockjava 文件中第19行有一个空指针的错误。”

运行测试后出现的空指针错误信息如下：

j ava.lang.NullPointerException

at tbc.tdd.hotelworldclocks.PhoneClock.setTime(PhoneClock.j ava:19) at tbc.tdd.hotelworldclocks.HotelWorldClocksTest.the\_time\_of\_clock\_London\_ and一NewYork一should一be一1一and—20\_respectively\_after\_the—phone\_cloclc一is\_ set\_to\_9\_Beijing\_time(HotelWorldClocksTest.j ava:52)

PhoneClockjava文件的第19行如下：

this.cityClock.setUtcZeroTime(time - this.utcOffset);

这一行是在PhoneClock类的setTime()方法中调用该类的cityClock成员变量的setUtc- ZeroTimeO方法来设置UTC时间。由于咱们在这个调整多个城市时钟的测试中使用了新创 建的HotelWorMClockSystem的对象来保存多个城市时钟，而没有像以前那样为PhoneClock 类的cityClock成员变量赋值来单独设置一个城市时钟，所以cityClock成员变量就会是空 指针。

“因为现在使用HotelWorldClockSystem的对象来保存多个城市时钟，而对于一个城市 时钟，咱们也可以用这个对象来保存，所以在PhoneClock类的setTimeO方法中就可以用 HotelWorldClockSystem对象的实现来替代原先的cityClock成员变量的实现。可以根据这一 点写出意图代码。”

在PhoneClock类的setTime()方法中，用HotelWorldClockSystem对象的实现替代原 先的cityClock成员变量的实现，把UTC时间设置给所有保存在该对象中的城市时钟的 意图代码如下（CM: Updated method PhoneClock.setTime() to use the clock list stored in the HotelWorldClockSystem.)：

public class PhoneClock {

public void setTime(int time) {

- this.cityClock.setUtcZeroTime(time - this.utcOffset);

+ for (CityClock cityClock : this.hotelWorldClockSystem.getClocks()) {

+ cityClock.setUtcZeroTime(time - this.utcOffset);

+ }

在这个意图代码中，PhoneClock类的hotelWorldClockSystem这个成员变量尚未定义。

这就定义它。另外，它也应该在该类的setHotelWorldClockSystem〇方法中被赋值。

在PhoneClock类中定义hotelWorldClockSystem成员变量并对其赋值的代码如下（CM: The field hotelWorldClockSystem of class PhoneClock was assigned in method PhoneClock. setHotelWorldClockSystem().)：

public class PhoneClock {

+ private HotelWorldClockSystem hotelWorldClockSystem;

public void setHotelWorldClockSystem(HotelWorldClockSystem hotelWorldClockSystem) {

+ this.hotelWorldClockSystem = hotelWorldClockSystem;

}

}

现在，在PhoneClock类的setTime()方法中，getClocks()方法是红色的，这表示还未创 建HotelWorldClockSystem类的getClocks()方法。按Alt+Enter快捷键来创建它。这个方法返 回该类的成员变量this.cityClocks。

创建 HotelWorldClockSystem 类的 getClocks〇 方法并返回成员变量 this.cityClocks 的 意图代码如下（CM: Created method HotelWorldClockSystem.getClocks〇 to return a clock list stored in this class.)：

+import java.util.ArrayList;

public class HotelWorldClockSystem {

+ public ArrayList<CityClock> getClocksO {

+ return this.cityClocks;

+ }

}

在上面这段意图代码中，this.cityClocks是红色的，表示HotelWorldClockSystem类还未 创建cityClocks成员变量。按Alt+Enter快捷键来创建它，并赋初值将其初始化。

在HotelWorldClockSystem类中创建city Clocks成员变量并初始化的代码如下（CM: Created and initialized field cityClocks in class HotelWorldClockSystem.)：

public class HotelWorldClockSystem {

+ private ArrayList<CityClock> cityClocks = new ArrayList<CityClock>();

+

HotelWorldClockSystem类的cityClocks成员变量虽然已经初始化了，但是为它添加城市 时钟的attach〇方法还是空的。咱们现在可以实现这个attach()方法。

“不等到真正需要这个attach()方法时再实现它吗？”

哦，我明白您的意思。咱们是可以把这个方法的实现放到后面运行测试发现与之相关的 错误时再来处理。不过我个人感觉咱们对这个多城市时钟的测试已经接近尾声，前景已经明 朗，信心已经越来越足，所以我想步子可以大一些。TDD虽然讲究用尽量小的步子来开发， 但如果您很有信心，不妨步子大一些。等到处理更复杂的问题时再换成小步子也不迟。宋代 抗金名将岳飞对于用兵曾这样说过：“运用之妙，存乎一心。” TDD开发也是如此。它不是死 板的教条，而是可以灵活调整的方法。

在 HotelWorldClockSystem 类中实现方法 attach()的代码如下（CM: Updated method Hotel- WorldClockSystem.attach() to attach a CityClock.)：

public class HotelWorldClockSystem {

public void attach(CityClock cityClock) {

+ this.cityClocks.add(cityClock);

现在编译没有错误了。运行一下测试看看有什么问题。出现两个空指针错误：

j ava.lang.NullPointerException

at tbc.tdd.hotelworldclocks.PhoneClock.setTime(PhoneClock.java:20) at tbc.tdd.hotelworldclocks.HotelWorldClocksTest.the\_time\_of\_clock\_London\_ should\_be\_l\_after\_the\_phone\_clock\_is\_set\_to\_9\_Beijing\_ time(HotelWorldClocksTest.java:19)

java.lang.NullPointerException

at tbc.tdd.hotelworldclocks.PhoneClock.setTime(PhoneClock.j ava:20) at tbc.tdd.hotelworldclocks.HotelWorldClocksTest.the\_time\_of\_clock\_NewYork\_ should\_be\_20\_after\_the\_phone\_clock\_is\_set\_to\_9\_Beijing\_ time(HotelWorldClocksTest.java:33)

这两个空指针错误都是原有那两个针对单个城市时钟的测试引发的。其中的原因咱 们在前面也提到过：现在是使用HotelWorldClockSystem对象来保存多个城市时钟，而 对于一个城市时钟，咱们也可以用这个对象来保存。但原有这两个测试还没有改用新的 HotelWorldClockSystem对象来保存单个城市时钟，所以会出现空指针。现在就改。

原有两个针对单个城市时钟的测试改用新的HotelWorldClockSystem对象来保存单个 城市时钟，代码如下（CM: Updated the previous 2 tests to use the new interface HotelWorld­ClockSystem.):

public class HotelWorldClocksTest {

public void the\_time一of\_clock一London\_should\_be一1一after—the一phone\_clock\_is—

set—to一9\_Beijing\_time() {

// Arrange

CityClock londonClock = new CityClock(0);

+ HotelWorldClockSystem hotelWorldClockSystem = new HotelWorldClockSystemO;

+ hotelWorldClockSystem.attach(londonClock)/

PhoneClock phoneClock = new PhoneClock(8);

//Act

- phoneClock.setCityClock(londonClock);

+ phoneClock.setHotelWorldClockSystem(hotelWorldClockSystem);

phoneClock.setTime(9);

public void the\_time\_of\_\_clock\_NewYork\_should\_be\_20\_after\_the\_phone\_clock\_ is—set\_to\_9\_Beijing一time() {

// Arrange

CityClock newYorkClock \* new CityClock(-5);

+ HotelWorldClockSystem hotelWorldClockSystem = new HotelWorldClockSystemO;

+ hotelWorldClockSystem.attach(newYorkClock);

PhoneClock phoneClock = new PhoneClock(8);

//Act

- phoneClock.setCityClock(newYorkClock);

+ phoneClock.setHotelWorldClockSystem(hotelWorldClockSystem);

phoneClock.setTime(9);

改完这两个测试，再按Ctrl+F5快捷键运行测试，3个测试都运行通过。

“这个处理多个城市时钟的TODO做完了，可以删除了。再看看还剩下哪些TODO。”

在看剩下的TODO之前，咱们先看看本章都做了什么事情：

1. )用划分测试等价类的方法将测试分类，从每一类测试中选择有代表性的情况进行测试。
2. 在用“尽量简单”的标准选择了第一个测试后，又用能够引发生产代码重构的“尽 量有意思”的标准选择了下一个测试，并将其标记为working-on的TODO来进行处理。

3 )写好一个测试的Assert部分之后，在以此推导出该测试的Act和Arrange部分之前, 用面向对象的设计方法，思考了如何用一个业务领域的类HotelWorldClockSystem来封装一 个Java的ArrayList类，并以这个小设计为基础，来推导出Act和Arrange部分的意图代码。

1. 循着意图代码中红色的编译错误的指引来编写最少的代码，以消除这些编译错误。
2. 用上述写最少代码的方法令编译通过后，运行测试时必然会遇到空指针错误。循着 这些测试运行错误的指引，继续编写期望的意图代码，以解决空指针问题。再循着这些意图 代码所引发的编译错误的指引，编写最少的代码，以消除这些编译错误。如此循环往复，直 至测试运行通过。
3. 当信心充足时，可以把步子迈得大一些。不等出现编译错误和测试运行错误这些指 引时，就可以直接编写那些显而易见的、必须编写的、能让测试运行通过的、最少量的代码。

嗅出代码“腐臭”和新的测试点

“按Alt+6快捷键看看咱们的TODO列表还剩下什么。哦，还剩下最后一个有关构造器 重复内容的TODO。”

目前的TODO列表如图8-1所示。

▼ Found 1 TODO Item in 1 file

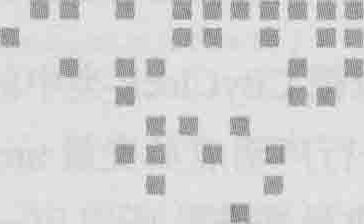
▼ Ej tbc.tdd.hotelworldclocks (1 ttem in 1 file)

图8-1最后一个有关构造器重复内容的TODO

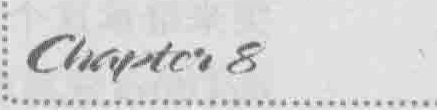
*,*ndPhanea*〇^*



mnmm «■讎■■屬■«



**第8章**



这个TODO要做的事情，是消除CityClock和PhoneClock这两个类的构造器中重复出 现的成员变量utcOffset。解决的方法是把这两个成员变量移动到这两个类的一个共同的父类 的成员变量中。这个父类可以叫做Clock。

还是用意图式编程来做。可以先让CityClock类扩展这个父类Clock。

让 CityC丨ock 类扩展父类 Clock 的意图代码如下（CM: Made class CityClock extend a new class Clock to eliminate the duplicated utcOffset field.)：

-public class CityClock {

+public class CityClock extends Clock{

在上面这段意图代码中，Clock类出现红色编译错误，因为它还未创建。按Alt+Enter快 捷键创建它。

创建 Clock 类的代码如下（CM: Created class Clock.):

+public class Clock {

+ }

接下来可以把CityClock类中的成员变量utcOffset移动到父类Clock中了。可以先在父 类Clock中写一行声明成员变量utcOffset的代码：“ protected int utcOffset;”，因为需要其子 类来继承这个成员变量，所以用protected关键字来修饰。然后在CityClock类中删除它的 utcOffset成员变量，并把所有的this.utcOffset都改为super.utcOffset。其实不这么改也能运 行，但改完后，能清楚地表达这个utcOffset来自父类。

把CityClock类中的成员变量utcOffset移动到父类Clock的代码如下（CM: Made class CityClock use the field utcOffset from the super class Clock.)：

public class Clock {

+ protected int utcOffset;

}

public class CityClock extends Clock{

* private int utcOffset; private int utcZeroTime;

// TODO-working-on: The constructors of CityClock and PhoneClock are duplicated public CityClock(int utcOffset) {

* this.utcOffset = utcOffset;

+ super.utcOffset = utcOffset;

}

public int getTime() {

* return (this.utcOffset + this.utcZeroTime + 24) % 24;

+ return (super.utcOffset + this.utcZeroTime + 24) % 24;

}

“对Clock类的另一个子类PhoneClock也如此办理0”

让PhoneClock类继承Clock类，并把其中的成员变量utcOffset移动到父类Clock，代码 如下所示（CM: Made class PhoneClock use the field utcOffset from the super class Clock.):

-public class PhoneClock {

♦public class PhoneClock extends Clock { private CityClock cityClock;

* private int utcOffset;

private HotelWorldClockSystem hotelWorldClockSystem;

public PhoneClock(int utcOffset) {

- this.utcOffset = utcOffset;

+ super.utcOffset = utcOffset;

public void setTime(int time) {

for (CityClock cityClock : this.hotelWorldClockSystem.getClocks()) { - cityClock.setUtcZeroTime(time - this.utcOffset);

+ cityClock.setUtcZeroTime(time - super.utcOffset);

“运行测试一下。测试全部通过。这个TODO也完成了，可以删掉了。”

别忘了把PhoneClock类中那些因被替换而不再使用的用来保存单个城市时钟的相关代 码也删除掉。

删除PhoneClock类中因被替换而不再使用的用来保存单个城市时钟的相关代码如下 (CM: Removed the finished TODO and unused code from class PhoneClock.)：

public class PhoneClock extends Clock {

- private CityClock cityClock;

private HotelWorldClockSystem hotelWorldClockSystem;

public PhoneClock(int utcOffset) {

super.utcOffset = utcOffset;

public void setCityClock(CityClock cityClock) { this.cityClock = cityClock;

再运行一下测试。测试依然运行通过！

“现在所有TODO都已经完成了，是不是这个用TDD做的操练可以结束了？”

虽然以前所有记录下来的TODO都完成了，但我觉得还差一步工作要做，那就是最后审 阅一下已经完成的测试代码和生产代码，找出还未测到的情况和遗漏的代码“腐臭”。对于 未测的情况，要加新的测试，对于新发现的“腐臭”，要立即重构。只有这一步做完了，才 能告一段落。不过以后要是发现这些测试和生产代码又有了新的未测情况和遗漏掉的代码 “腐臭”，还有重复上述步骤。这样说起来，代码编写没有结束的时候。

“我想起了一个咱们没有测到的地方，就是手机自身的时间，也应该用它设置其他城市 时钟的时候，自动调整好。”

对，咱们需要加上这个测试。再写上一个Assert断言。

“在这个Assert中，我期望phoneClock对象能有一个getTime()方法。这就是我的意图 代码。”

当用手机设置其他城市时钟时其自身的时间也应自动调整好的测试和其中的断言如下 (CM: Added test the\_time\_of\_the\_phone\_clock\_should\_be\_set\_correctly\_after\_its\_setTime()\_ method\_is\_invoked().)：

public class HotelWorldClocksTest {

+ // TODO: the time of the phone clock should be set correctly after its

setTime() method is invoked + @Test

+ public void the\_time\_of\_the\_phone\_clock\_should\_be\_set\_correctly\_after\_its\_

setTime\_method\_is—invoiced (> {

+ // Assert

+ assertEquals(9, phoneClock.getTime());

+ }

}

“从这个Assert可以很容易地推导出这个测试的Act,即把手机时钟设置为9点。然后又 推导出Arrange,即创建一个PhoneClock实例，并且在创建时传人UTC时差。”

从上述Assert推导出这个测试的Act和Arrange的代码如下所示：

public class HotelWorldClocksTest {

@Test

public void the\_time\_of\_the\_phone\_clock\_should\_be\_set\_correctly\_after\_its\_ setTime\_method\_is\_invoked() {

+ // Arrange

+ PhoneClock phoneClock = new PhoneClock(8);

+

+ //Act

+ phoneClock.setTime(9);

+

// Assert

assertEquals(9, phoneClock.getTime());

}

“为了让这个测试快速运行通过，直接让PhoneClock类的getTime()方法返回假数据9。 运行测试。咦，空指针错误？哦，PhoneClock类的setTime〇方法目前需要遍历它的成员变 量hotelWorldClockSystem中保存的各个城市的时钟对象。但咱们这个测试没有为这个成员变 量赋值，所以出现空指针错误。在前面加一个判断就行了。”

让PhoneClock类的getTime〇方法直接返回假数据9并在setTime〇方法里加判断，代 码如下（CM: Made the test to pass by fake it in PhoneClock.getTime().):

public class PhoneClock extends Clock {

public void setTime(int time) {

+ if (this.hotelWorldClockSystem == null) return;

for (CityClock cityClock : this.hotelWorldClockSystem.getClocks{)) { cityClock.setUtcZeroTime(time - super.utcOffset);

+ public int getTimeO {

+ return 9;

+ }

}

在PhoneClock类中加了 getTime()方法后，我发现这个方法和另一个类CityClock中的 getTimeO方法虽然实现不一样，但是方法签名完全一样。是不是可以从这两个方法中提取一 个抽象方法，放到父类Class中，用父类的抽象方法来消除子类方法签名的重复呢？

“应该可以。不过咱们现在手上的有关手机自身时间的测试还没完成呢。”

可以把这个新发现的问题写一个TODO,等完成手上这个TODO之后再处理。

目前TODO列表如图8-2所示。

tbc.tdd.hotelworlddocks (2 rfems in 2 files ▼參 PhoneC丨oclqava

(24, 8) // TODO: PhoneClock.getTimeO and CityClock.getTimeO are duplicated ▼ # HotelWorldClocksTest.java

M (62, 8) /! TODO-working-on: the time of the phone dock should be set correctly after its setTtmeO method fs

图8-2加上一个新发现的TODO待以后处理

“接着处理目前手上的TODO。前面学到了，在PhoneClock类的getTime()方法中直 接返回的那个假数据9,和测试中的9重复了。解决的方法是把生产代码PhoneClock类的 getTime()方法中的9替换成变量。可以在PhoneClock类中创建一个存储手机当前时间的成 员变量time，来替换这个假数据9。”

用PhoneClock类的成员变量time替换假数据9的意图代码如下（CM: Wrote the intention code in PhoneClock.getTime() to return the local time which will be set in the method setTime().)：

public class PhoneClock extends Clock {

//TODO: PhoneClock.getTime() and CityClock.getTime() are duplicated public int getTime() {

- return 9;

+ return this.time;

}

}

“在这个意图代码中，this.time尚未在PhoneClock类中定义。利用Alt+Enter快捷键创建 它，并且在PhoneClock类的setTime()方法中给它赋值。”

创建PhoneClock类的time成员变量并在setTime()方法中对其赋值的代码如下（CM: Created field time in class PhoneClock and assigned value to it in method setTimeQ.)：

public class PhoneClock extends Clock {

private HotelWorldClockSystem hotelWorldClockSystem;

+ private int time;

public void setTime(int time) {

+ this.time = time;

if (this.hotelWorldClockSystem -= null) return;

for (CityClock cityClock : this.hotelWorldClockSystem.getClocks()) { cityClock.setUtcZeroTime(time - super.utcOffset);

“运行测试，通过。现在可以删掉这个处理手机自身时间的TODO。再来处理那个把 getTime〇方法提取到父类作为抽象方法的TODO吧。”

可以直接在Clock类中添加一个getTime〇抽象方法。既然这个方法是抽象的，这个 Clock类也应该改为抽象的。

在 Clock类中添加 getTime()抽象方法的代码如下（CM: Made class Clock abstract and added an abstract method getTime() for its subclasses PhoneClock and CityClock.)：

-public class Clock {

♦public abstract class Clock { protected int utcOffset;

+ public abstract int getTime();

)

“运行测试，通过。”

不过，此时最好在Clock类的那两个子类的getTime()方法前面加上@Override，以便让 人了解它们覆写了父类的getTimeO方法。

在Clock类的两个子类的getTime()方法前面加上@Override，代码如下（CM: Added keyword @Override for the overridden method getTime() in the two subclasses.)：

public class CityClock extends Clock{

+ @Override

public int getTime() {

return (super.utcOffset + this.utcZeroTime + 24) % 24;

}

public class PhoneClock extends Clock {

- // TODO-working-on: PhoneClock.getTime() and CityClock.getTime() are duplicated

+ ©Override

public int getTime() { return this.time;

“运行测试，通过。”

还记得咱们之前划分的测试等价类吗？咱们划分了3个测试等价类，目前只测试了两 个。还有一个没测。

“记得。没测的是当地时间比UTC时间早的城市时钟。当时没测的原因是觉得它能在不 重构生产代码的情况下直接通过，没那么有意思。”

对。此时这个操练的主要工作都做完了，有些富裕时间了。如果咱们对这个没有测试 的等价类信心不足，不妨把它加上。咱们可以测试莫斯科时钟。莫斯科时间比UTC时间早 4小时。只要简单地复制一下伦敦时钟的测试，然后把所有有关伦敦的地方都改成莫斯科就 行了。

测试莫斯科时钟的测试代码如下（CM: Added test the\_time\_of\_clock\_Moscow\_should\_ be\_5\_after\_the\_phone\_clock\_is\_set\_to\_9\_Beijing\_time() to cover the equivalence class of cities with positive UTC offset.)：

public class HotelWorldClocksTest {

+ @Test

+ public void the\_time\_of\_clock\_Moscow\_should\_be\_5\_after\_the\_phone\_clock\_is\_

set\_to\_9\_Beijing一time() {

+ // Arrange

+ CityClock moscowClock = new CityClock(4)/

+ HotelWorldClockSystem hotelWorldClockSystem = new HotelWorldClockSystem();

+ hotelWorldClockSystem.attach(moscowCloclO ;

+ PhoneClock phoneClock = new PhoneClock(8);

+

+ //Act

+ phoneClock.setHotelWorldClockSystem(hotelWorldClockSystem);

+ phoneClock.setTime(9);

+

+ // Assert

+ assertEquals(5, moscowClock.getTime());

运行测试，通过。

“终于要结束了！”

还差一点。在测试代码中，有没有闻到重复代码的“腐臭”？

“没闻到呀。”

哈哈！仔细观察一下目前这5个测试各自的Arrange部分。

“哦。是有两处重复。一处是创建HotelWorldClockSystem对象，另一处是创建Phone-

Clock 对象。”

对。咱们可以先在这个测试类中创建一个用@Bef〇re标注的方法，然后把这两处重复都 提取到这个方法中，最后再在测试类中创建相应的成员变量，供各测试方法使用。首先提取 创建 HotelWorldClockSystem 对象的代码。

在测试类中把创建HotelWorldClockSystem对象的代码提取到@Before标注的方法中， 代码如下（CM: Extracted the duplicated instantiation of class HotelWorldClockSystem into @Before in the test class.)：

public class HotelWorldClocksTest {

+ private HotelWorldClockSystem hotelWorldClockSystem;

+

+ @Before

+ public void initialize() {

+ this.hotelWorldClockSystem = new HotelWorldClockSystem();

+ }

+

new

new

new

new

new

HotelWorldClockSystem

HotelWorldClockSystem

HotelWorldClockSystem

HotelWorldClockSystem

HotelWorldClockSystem

hotelWorldClockSystem = hotelWorldClockSystem = hotelWorldClockSystem = hotelWorldClockSystem = hotelWorldClockSystem =

HotelWorldClockSystem(); HotelWorldClockSystem(); HotelWorldClockSystem(); HotelWorldClockSystem〇; HotelWorldClockSystem();

运行测试，通过。接着提取创建PhoneClock对象的代码。

在测试类中把创建PhoneClock对象的代码提取到@Before标注的方法中，代码如下 (CM: Extracted the duplicated instantiation of class PhoneClock into @Before in the test class.)：

public class HotelWorldClocksTest {

private HotelWorldClockSystem hotelWorldClockSystem;

+ private PhoneClock phoneClock;

^Before

public void initialize() {

this.hotelWorldClockSystem = new HotelWorldClockSystem(); + this.phoneClock = new PhoneClock(8);

PhoneClock phoneClock = new PhoneClock(8);

现在基本上把能发现的测试代码和生产代码的“腐臭”，以及能想到的测试都处理完了。 不过，是否选择将测试代码中的重复代码移动到测试类的以@Bef〇re标注的方法中，以消除 重复，也是因人而异。因为有人认为这些重复的代码在每个测试中都是上下文的一部分，如 果被提取到@Bef〇re中，会造成在阅读测试代码时频繁地跳跃，产生不便，所以这部分人偏 向于不提取测试中的重复代码。

PhoneClock

PhoneClock

PhoneClock

PhoneClock

new

new

new

new

PhoneClock - PhoneClock = phoneClock = phoneClock =

PhoneClock(8); PhoneClock(8); PhoneClock(8); PhoneClock(8);

在把这个TDD开发方法和前面那个测试后运行的方法进行对比前，咱们先看看本章做 了哪些工作。

1. )将子类构造器中出现的重复的成员变量上移到父类中，以消除重复。
2. 将那些因被替换而不再被调用的代码删除掉。
3. 完成了所有列出的TODO,也重新审阅了已经完成的测试代码和生产代码，找出还 未测到的情况和遗漏的代码“腐臭”。对于未测的情况，要增加新的测试；对于新发现的“腐 臭”，要做重构。
4. 在对某个TODO任务进行重构的过程中，随着代码的变化，又浮现出了新的重复 代码，这时把这些新出现的代码“腐臭”写成TODO以做备忘，然后再接着重构当前的 TOD00
5. 遇到了两个子类中有相同签名的方法，但是实现各不相同。此时从这两个方法中提 取一个抽象方法，将其放到父类中，以解决这两个子类的方法签名的重复问题。
6. 虽然有一个以前未测的等价类的测试，能在当前生产代码上直接运行通过，虽然不 那么有意思，但为了增强对这类等价类的信心，还是添加了一个测试来覆盖它。
7. 在测试类中发现了各个测试方法的Arrange部分中出现了重复代码，把这些重复代码 移动到测试类的以@Bef〇re标注的方法中，以消除测试代码中的重复代码。
8. 通过操练我们学到了以下技能：
9. 每次重构代码后都随时运行测试，以检查是否破坏了原有代码行为。
10. 是否选择将测试代码中的重复代码移动到测试类中以@Bef〇re标注的方法中以消除 重复，也因人而异。因为有人认为这些重复的代码在每个测试中都是上下文的一部分，如果 被提取到@8#〇1^中，会造成阅读测试代码时频繁地跳跃，产生不便，所以这部分人偏向于 不提取测试中的重复代码

测试后行vs测试先行

在分别用测试后行的开发方法和TDD开发方法完成了同一个编程操练的题目后，现在 是时候来把这两种方法进行对比了。

为了便于进行对比，需要把这两种方法所开发出来的代码描述出来，而UML类图就是 一种用来描述代码静态结构的方法。我把这两种方法所编写出来的生产代码用类图分别画出 来了，请仔细加以比较，或许能得到一个有意思的发现。

用测试后行的开发方法所编写的生产代码的类图如图9-1所示。

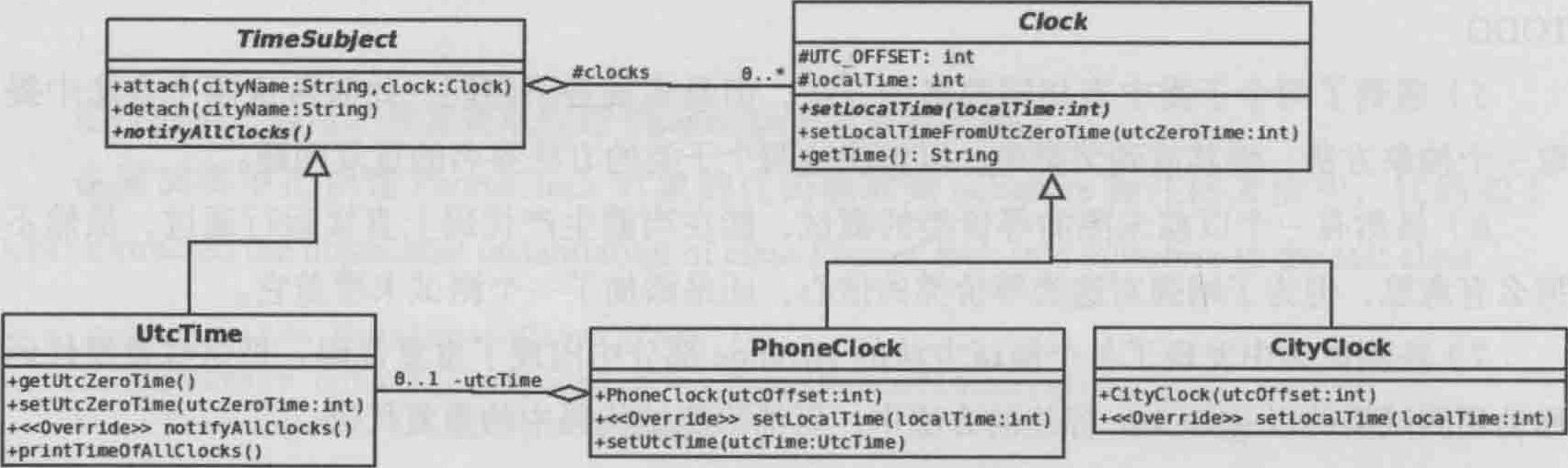


图9-1用测试后行的开发方法所编写的生产代码的类图

用TDD开发方法所编写的生产代码的类图如图9-2所示。

“这两张图很像！而且在用TDD开发出来的生产代码中，把TimeSubject和UtcTime合 并成HotelWorldClockSystem这一个类，竟然解决了前面提到的照搬设计模式导致设计出不 必要的抽象的问题！”

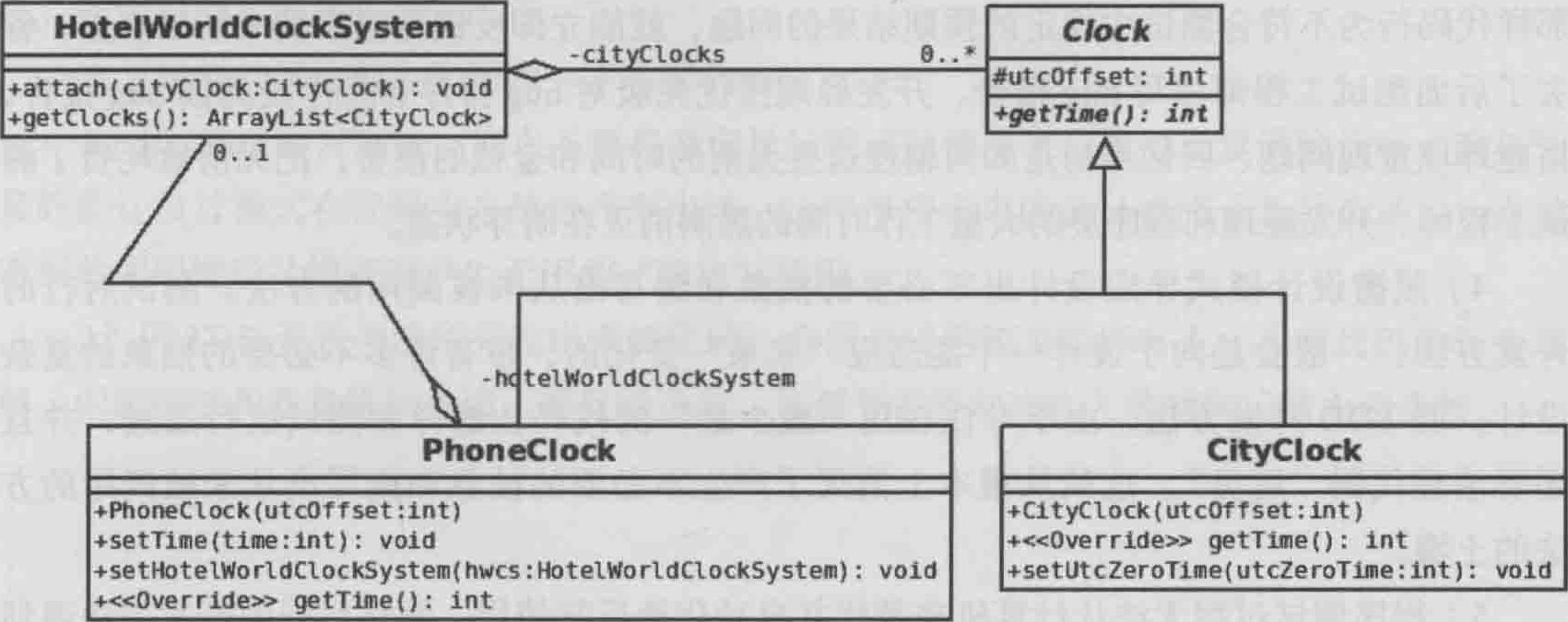


图9-2用TDD开发方法所编写的生产代码的类图

最妙的是，这一切都是咱们在没有提设计模式一个字的情况下，用TDD开发方法，一 点一点地对TODO、测试代码和生产代码进行重构，最后让Observer这个设计模式自己浮现 出来的。

除了在设计模式的运用方面的“照搬”与“浮现”的区别，咱们可以把前面罗列的测试 后行的开发方法所暴露的5个问题再简单回顾一下，并看看用TDD开发方法是否确实解决 了这些问题。

1. 文档经常与代码缺乏同步。测试后行的开发方法是把文档作为代码编写的沟通基础， 然而由于文档与代码的同步工作很繁琐，经常导致文档过时。而TDD开发方法是把代码本 身当成代码编写的沟通基础，这体现在测试代码命名、生产代码命名、TODO注释列表命名、 Commit Message提交注解等各个方面。这些命名既是文档，同时也是能够被运行的代码，这 就保证了文档与代码的天然同步。而作为文档的Commit Message提交注解，会跟随所修改 的代码，作为不可篡改的档案，永久地记录到版本管理系统的代码库里。这也天然地保证了 文档与代码的同步。
2. 写mainO方法进行的测试无法让计算机自动判断软件行为是否符合预期。测试后行 的开发方法，通常需要编写mainO方法来进行测试。但这种测试的预期结果是保存在人脑中 的，需要人工来判断mainO方法的输出是否符合预期，效率低下。而TDD开发方法，能把 测试的期望值也写成Assert语句来告诉程序，使得计算机能够代替人脑来判断结果是否符合 预期，而且这个过程能够通过运行命令来自动化，大大提高了效率。
3. 问题产生后没能立即发现。测试后行的开发方法在编写完相应代码后，没能立即发 现城市时钟的时间全是9点和出现-4点这些问题，直到写main〇方法运行时才发现。而 TDD开发方法，因为是先写测试并能够频繁地运行测试，所以在运行测试时，若发现像上述 那样代码行为不符合测试中规定的预期结果的问题，就能立即反馈给程序员来加以修正。省 去了后面测试工程师填写bug报告、开发经理按优先级对bug排序和程序员阅读bug报告、 搭建环境重现问题、回忆当初是如何编程这些无谓的时间和金钱的浪费，把无情地耗费了测 试工程师、开发经理和程序员的大量工作时间的黑洞消灭在萌芽状态。
4. 照搬设计模式导致设计出不必要的抽象和编写出从未被调用的方法。测试后行的 开发方法，一般会趋向于设计一个能适应“未来”变化的、带有许多不必要的抽象的复杂 设计。而TDD开发方法，由于专注在用“最少量”的代码让编译和测试运行通过，并且 还要治理代码“腐臭”。这就从根本上消灭了产生不必要的抽象和编写出从未被调用的方 法的土壤。
5. 程序调试过程无法让计算机来替代并自动化地反复使用。测试后行的开发方法遇到 bug时一般都会在IDE中设置断点来调试程序，因其过程繁琐、无法让计算机自动复用且 bug出现的范围未被限定，而导致效率低下。而TDD开发方法，bug都会在由计算机自动且 反复运行的测试中被发现，且bug出现的范围被限定在一个个粒度很小的测试之内，便于程 序员定位错误，效率很高。

如果把上面描述的TDD开发方法对这5个问题的解决方法归纳一下，我们能看出TDD 开发方法具有以下优势：

1. 把代码本身当成代码编写的沟通基础，令代码成为可以运行的文档，会让那些代 码即文档的读者——包括程序员、测试工程师、产品专家等——在理解代码行为方面反馈 迅速。
2. 把测试的期望值写成Assert语句告诉计算机，使其能够代替人脑来判断结果是否符 合预期，会让程序员在感知代码问题方面反馈迅速。
3. 先写测试并频繁地运行测试以立即发现错误，会让程序员、测试工程师和开发经理 等所有与代码相关的人在感知代码问题方面反馈迅速。
4. 专注在用“最少量”的代码让编译和测试运行通过，接着治理代码“腐臭”，通过计 算机自动且反复运行的测试，发现那些被限定在一个个粒度很小的测试之内bug，都会让程 序员在维护代码质量方面反馈迅速。

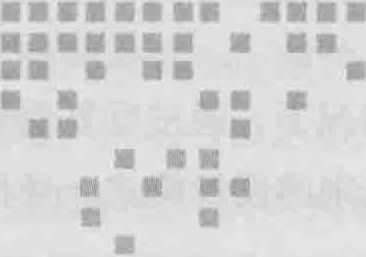
从后一个编程操练可以看出，TDD开发方法，会使我们在理解代码行为、感知代码问题 和维护代码质量方面都反馈迅速，而带来这个优势的根源是用这种方法所编写出的代码。据 此我们可以得出这样的结论：用TDD开发方法所开发出来的代码，会使与代码相关的所有 人，在对代码的行为理解、问题感知和质量维护方面，都反馈迅速；其结果就是节省所有这 些人的时间、精力和金钱。

在用编程操练分别体验了测试后行的开发方法和TDD开发方法之后，我们再反观用这 两种方法所开发出来的代码，哪种代码更像本书所讨论的主题一烂代码呢？要回答这个问

题，我们需要了解什么是烂代码。

在了解烂代码之前，先回顾一下本章的内容：

1. 用TDD开发方法，能在不事先考虑设计模式的情况下，通过反复消除代码“腐臭”， 最终能让设计模式在代码中自然地浮现出来。这样浮现出来的设计模式，身材苗条，一点没 有那些用照搬设计模式而产生无用的“赘肉”代码。
2. 用TDD开发方法所开发出来的代码，会使与代码相关的所有人，在对代码的行为理 解、问题感知和质量维护方面，都反馈迅速，从而节省所有这些人的时间、精力和金钱。



**第10章**

何谓“烂代码”

在正式介绍什么是烂代码之前，先来看一个小故事。

一天，正走在取经路上的唐僧，忽听见山脚下几声震耳欲聋的叫喊：“师父！师父！ ”唐 僧忙走过去，在石缝中见到一个猴子。那猴子问明唐僧是去西天取经的，便说：“我是500 年前大闹天宫的齐天大圣，因为欺骗领导，被如来佛压在这里。不久前观音菩萨依如来佛旨 意去找取经人路过这里，我求他救我。他劝我再莫行凶，皈依佛法，努力保护取经人去西方 拜佛取经，功成后自有好处。我就答应下来，日夜盼望师父前来救我。我愿做你徒弟，保护 您取经。”唐僧满心欢喜，救了悟空出来。

悟空出来后与唐僧一路前行，突然见一猛虎。悟空大叫一声："哪里去！”那猛虎竟然被 惊得动也不敢动，趴在地上任凭悟空一棒打死。吓得唐僧掉下马来，咬着手指说：“天哪！ 昨天我看见那本地猎人打一只斑斓猛虎，还斗了老半天。今日你不费力气，只一棒就把那虎 打个稀烂，强！”

不久，他们又遇到6个歹徒抢劫，悟空又不费吹灰之力将这6人全都打死。气得唐僧 说：“他们虽是强盗，却罪不该死。你一味杀生，如何做得和尚？太可恶了！”悟空受不了师 父的唠叨，一怒之下撂挑子不干，撇下师父去东海龙王那里串门喝茶去了。唐僧只得自己独 自赶路。而知晓这一切的观音菩萨，变做一个老太太去见唐僧，并把为悟空准备的棉衣和嵌 金花帽交给唐僧，还教会了唐僧念紧箍咒。而悟空听从了龙王和观音的劝告，回到了师父身 边。他在行李中瞧见了衣服和帽子，便穿戴起来。唐僧一见，便默默念起紧箍咒，从此那帽 子里的金箍便在悟空的脑袋上生下根来，无法摘下，而且越收越紧，疼得悟空满地打滚。唐 僧问：“你今后还敢无礼吗？”悟空口里虽说：“不敢了。”但仍心怀不善，扯出金箍棒要对唐

僧下手。慌得唐僧又念了3遍紧箍咒，疼得悟空丢了铁棒，趴在地上求饶。当悟空得知这一 切都是观音的安排后，便死心塌地地保护唐僧西天取经。

这里的孙悟空，在我看来，就是一段烂代码。原因在后面会进行说明。

本书所讨论的“烂代码”这个词，在国外一般叫Legacy Code,但是这个英文词翻译成 中文就被译为“遗留代码”。而国内的许多IT从业者，对“遗留代码”又有自己的解释，与 其所对应的英文Legacy Code的本义有所不同。为了讨论方便，在本书中出现的“遗留代码” 的含义就是英文Legacy Code的本义。

在国外讨论软件编写的书籍中，描述与本书主题“烂代码”相近概念的英文词汇有若干 个，包括1^3〇丫〇)(^(遗留代码)、83£1<：(^0(糟糕的代码)、八848311(^\111(13(大泥球)、 Crap (烂东西)、A Mess (—个烂摊子）和Impossible to Work with (不可雕之朽木代码）。

上述描述烂代码的英文词汇中，在国外影响力最大的当属Michael C. Feathers在他所著 Working Effectively with Legacy Code 一书中所描述的“遗留代码” D Feathers在书中对于遗 留代码表达了下面的观点

□遗留代码的严格定义“就是指从其他人那儿得来的代码”。

* “在业内人士的口中，‘遗留代码’一词常常是‘无法理解的、难以修改的代码’的代 名词。”
* “然而，在多年来与形形色色的开发团队共事并帮助他们解决重大的编码问题的过程 中，我总结出了一个不同的定义。对我来说，遗留代码就是那些没有编写相应测试的 代码。明白这一点是很痛苦的。”
* “没有编写测试的代码是糟糕的代码（Bad Code乂不管我们有多细心地去编写它们， 不管它们有多漂亮、面向对象或封装良好，只要没有编写测试，我们实际上就不知道 修改后的代码是变得更好了还是更糟了。反之，有了测试，我们就能够迅速、可验证 地修改代码的行为。”

Michael C. Feathers为“遗留代码”所做的“没写测试”的定义在国外的影响如此之大， 以至于让“没写测试”一举超越“无法理解和难以修改”，在国外成为遗留代码“事实上的 标准定义®”。

㊀ Legacy Code 和 Bad Code 的出处参见：Michael C. Feathers, Working Effectively with Legdcy Code, Prentice Hall PTR^ 2004 年 9 月 22 日，Preface。

㊁ A Big Ball of Mud 的出处参见：Brian Foote，Joseph Yoder, Big Ball of Mud Fourth Conference on Patterns Languages of Programs (PLoP 1 97/EuroPLoP \* 97) Monticello, Illinois, September 1997〇

㊂ Crap、A Mess 和 Impossible to Work with 的出处参见：Daniel Brolund，Ola Ellnestam，Behead Your Legacy Beast，2012 年 2 月 25 日，[www.agical.com/mikmeth/mikadomethod.pdfo](http://www.agical.com/mikmeth/mikadomethod.pdfo)

㊃Michael C. Feathers著，刘未鹏译，《修改代码的艺术》，人民邮电出版社，2007年11月第1版，前言。

㊄ Daniel Brolund，Ola Ellnestam, Behead Your Legacy Beast, 2012 年 2 月 25 日，Preface, [www.agical.com/](http://www.agical.com/) mikmeth/mikadomethod.pdfo

在前面列举的若干与烂代码含义相近的英文词汇中，把烂代码描述得最为生动形象的 当属“大泥球”。该英文词汇的发明人Brian Foote和Joseph Yoder是如下形象地描述"大泥 球”的e:

□大泥球代码就是一个结构混乱、肆意蔓延、轻浮草率、贴满补丁、私搭乱建、一团乱 麻的丛林。

□这类系统明确无误地显现出无序增长、反复修补和权宜修复的迹象。

□信息在系统的互不相干的部分之间杂乱地被共享。

□在这类系统中，经常几乎所有的重要信息都会变为全局的或重复的。

□系统的总体结构可能从未被很好地定义过，就算以前定义过，也被侵蚀得面目全非， 只要有一点点软件架构意^3、的程序员都会躲避这样的“泥潭'

□只有那些不关心软件架构的程序员，和那些或许乐于在一个即将垮掉的堤防上每天做 修修补补差事的程序员，才愿意工作在这样的系统上。

从这里可以看出，那些结构混乱、重复冗余和依赖全局信息的系统，可以被称为“大 泥球”。

从上面Legacy Code和A Big Ball of Mud这两个英文词汇的描述可以看出，本书所讨论 的“烂代码”在国外程序员的观念中，就是指那些没写测试、无法理解、难以修改、结构混 乱、重复冗余和依赖全局信息的代码。

讨论了国外程序对烂代码的看法之后，我们把视点再转到国内程序员。

从目前所掌握的资料来看，国内出版物中，尚无专门论述烂代码，并给出明确定义的书 籍。而烂代码一词经常出现在国内程序员的口头交流、微博和博客中。

我对公益编程操练社区“bjdp.org北京设计模式学习组” &中的一些程序员和新浪微博的 网友（以下简称：微博网友）对烂代码的主要评论进行了归纳，从中可以看出，国内程序员 一般认为，烂代码就是那些难以理解、难以维护和人皆写过的代码。

烂代码难以理解的问题，一般表现在下面3个方面：命名不清、多层嵌套和滥用模式。

□命名不清。“烂代码从烂命名开始。®”“一堆烂代码看着头大，命名能再不靠谱一点 吗？ 命名不清导致的直接后果就是看不懂代码，“看不懂的就是烂代码。®”

□多层嵌套〇 “烂代码一般都具备多层嵌套的特性。我最不喜欢的就是类似for循环下嵌套 着另一个for循环，接着再嵌套一个if语句，然后再嵌套另一个if语句这样的代码。®”

© Wikipedia, Big ball of mud, <http://en.wikipedia.org/wiki/Big_ball_of_mud> ㊁北京设计模式学习组（Beijing Design Patterns Study Group, BJDP)，<http://www.bjdp.org> ㊂微博网友@荣老 ㊃微博网友@哦小印 ㊄北京的Java程序员杜金秀 ㊅怀化的C程序员英界尔

□滥用模式。设计模式是蕴含在代码的内在逻辑里面的，而不是人为给加上的。“好代 码是容易看懂，易于维护，便于扩展，兼顾细节的。烂代码是模式一堆，嵌套层出， 看不清楚，搞不明白。S”设计模式的运用应该是发乎自然，而不是扭捏作态的。“软 件的设计，就像您要放的一个屁。如果您必须要迫使自己把它放出来，那么出来的或 许是一■坨屎。S”这话虽然不雅，但是却指出了一条真理：每个健康的人都会放屁，这 是自然现象，就让它自然地放出来吧（当然要注意场合）〇就如同老子在《道德经》 里所说的：“人法地，地法天，天法道，道法自然。”如果要生生地把它挤出来，那么 很有可能您就需要洗裤子了。

代码难以维护的问题，一般表现在下面3个方面：代码冗余、难以扩展和不如重写。

□代码冗余。“……想到的是公司那些大量冗余烂代码，几万行的一个代码文件，编译 起来是不是要比高质量、重构得当的代码要时间长很多呢？®”“现场分析一些典型的 烂代码，会发现传递信息的粒度太大、代码冗余、方法的定义不精准等问题。®”源 代码拷贝和粘贴是造成代码冗余的最主要的祸根。“拷贝粘贴是万恶之源，为了快速 完成任务，拷贝代码成了权宜之计。®”“烂代码更容易被Copy, Paste后成为更烂的 代码，可变更性更差，导致更高的bug率，导致陷人万劫不复之地。®”

□难以扩展。“烂代码就是耦合度高的代码。®” “烂代码就是代码质量不高、难以扩展、 难以维护、出bug概率高的代码。®”

口不如重写。很多程序员由于缺乏驯服烂代码的心法和手法，在看到难以维护的烂代码 时，往往产生“不如重写”的念头。“我理解的烂代码，就是不想再碰，也不敢碰的 代码。如果实在要改，不如重新来过。®”“重写比驯服也许更经济。®”

烂代码“人皆写过”这一点不难理解。比如在程序员刚刚接触一个崭新的行业领域，或 者刚刚开始学习一门新的编程语言，或者刚从面向过程的开发转向面向对象的开发时，那时 写出的代码就像蹒跚学步的孩子，虽然能走两步，但是跌跌撞撞，经常摔跤。等后来水平提 高了再回头看时，都会有不堪回首的感觉。“有时候刚骂完烂代码，一查版本历史才知道是

©微博网友@放翁\_文初

©程序员Jef Claes在桌子上看到了一个徽章，上面印着下面这段富有哲理的话：Design is like a fart. If you need to force it，it’s probably shit.出自：[https://twitterxom/JefClaes/status/433877873225920512/photo/l#rd。](https://twitterxom/JefClaes/status/433877873225920512/photo/l%23rd%e3%80%82) ㊂微博网友@灵感之源 ㊃微博网友@thinkinlamp ®微博网友李江华 ㊅微博网友@Denny\_Wang\_NJ ㊆北京Java程序员Jeff ®北京软件开发咨询师王洪亮 ㊈北京C#程序员二奎 ㊉微博网友@自律则强

自己写的，然后都不敢相信。e” “我看见自己以前写的那些烂代码，都不想承认那是我写的 代码。s”

总之，如果让国内程序员扪心自问的话，大家都写过一些难以理解和难以维护的烂 代码。

在上面列举的国内外程序员对于烂代码的各种定义和不同描述中，能否归纳出一些有共 性的东西呢？经过对这个问题的长期思考，我认为烂代码应该具有下面3点共性：

1. 烂代码都是能够运行的。不管代码是多么难以理解和难以维护，甚至会有bug,但它 至少应该是可以运行，能看到一些结果的。不能运行的代码就不能被称为代码，而只能被称 为一堆字符串。
2. 烂代码都是需要修改其中的bug或在其中增加新功能的。如果一段代码运行得很好, 而且目前也暂时没有与其相关的bug或开发新功能，那么即使这段代码写得再难以理解和难 以维护，那么它也不会被纳人本书所讨论的要驯服的烂代码的范畴。直到有一天我们需要修 改其中的bug或在其中增加新功能时，我们再称其为烂代码也不迟。这样做的好处是，我们 能够避免因做那些没有任何业务价值的事情而造成浪费®。
3. 烂代码对于其编写者、测试者和维护者来说都是反馈迟缓的。在软件开发中，下面 这种场景#是不是很常见？“添加一点小产品特性，就冒出两个bug!之前是谁写的这代码？ 关联来关联去，也没说明。”添加小产品特性时就会出bug,造成小产品特性完工拖延，这就 是在添加新功能方面反馈迟缓。另外，在使用测试后行的开发方法所进行的本书的第一个编 程操练的过程中，我们也都看到了在理解代码行为、感知代码问题和维护代码质量方面反馈 迟缓的例子。

除了上面的常见场景，咱们可以再逐一看看前面提到的国内外程序员对于烂代码的各种 描述，来看它们是不是在某种程度上都是反馈迟缓的。

难以理解的代码比起容易理解的代码，对于代码的编写者和维护者，在了解这些代码的 行为方面的反馈要迟缓得多。

如果代码没写可以自动化运行的测试，那么我们只能通过测试工程师手工测试来了解 代码的行为是否符合预期。很明显，手工测试比起自动化测试来说要慢很多。所以没写测试 的代码比起有自动化测试的代码，对于代码的测试者，在代码行为的验证方面的反馈要迟缓 得多。

难以维护、难以修改、结构混乱、重复冗余和依赖全局信息的代码，比起那些容易维

©北京Java程序员李小波

©北京C#程序员二奎

㊂本书有关烂代码的第二点共性，是受软件开发咨询师姚若舟（Joseph)的建议的启发。参见：<http://www>. infoq.com/cn/news/2014/03/garbage-code-discuss 〇

⑳微博网友@张克强-敏捷307

护、容易扩展、结构清晰、没有重复、封装良好的代码，对于代码的编写者和维护者，在基 于这些代码之上修改bug或新增功能方面的反馈要迟缓得多。

找出了上述3点烂代码的共性，我们现在就可以给本书所描述的烂代码下个定义了：烂 代码就是那些能够运行的、而且需要修改其中的bug或在其中增加新功能的、但对于代码的 编写者、测试者和维护者来说反馈迟缓的代码。简而言之，烂代码就是反馈迟缓的代码。这 里的反馈包括了解代码行为、验证代码行为和在这些代码之上修改bug或新增功能等方面。

有了烂代码的上述定义，咱们回过头来看看第一个编程操练。

根据烂代码的定义，在编写mainO方法测试之前，所有咱们根据细化后的类图所编写的 那几个类的代码，严格说起来连烂代码都算不上，只能算是一些字符串，因为它们都不能运 行。直到咱们编写了 main()方法，使得那些类可以运行了，这些代码才有了可以被称为烂代 码的资格：能够运行。

当我们第一次和第二次运行mainO方法时，分别发现了两个bug,第一个bug是所有城 市时钟都是9点，第二个bug是出现了 -4点这个不合理的时间，我们需要修改bug。

等我们修复了上述两个bug,第三次运行mainO方法得到期望的结果后，我们能说此时 的代码就不是烂代码了吗？还不行。

因为我们又发现代码中暴露出下面一些反馈迟缓的问题：首先，我们在开发中所时刻依 赖的细化后的类图，随着开发的进行而先后更改了8次，如果这8次修改没有及时同步到类 图中，而以后的开发又严重依赖这份未更新的类图的话，那么这份旧类图就会使我们对于代 码行为的理解发生偏差，令代码在代码编写者和维护者了解其行为方面反馈迟缓。

其次，由于咱们刻板地按照“四巨头”在讨论Observer设计模式时给出的类图而画出 了相应的类图，结果设计出TimeSubject和UtcTime这两个其实可以合并的类，多设计了 TimeSubject.detach()和UtcTime.getUtcZeroTime()这两个从未被调用的方法。这是一种浪费， 而浪费的后果是令程序员在单位时间里，阅读和管理了不必要的代码，从而减少了阅读和管 理必要代码的时间，也使得代码在程序员了解其行为方面反馈迟缓。

最后，咱们这个编程操练是通过手工运行main()方法和设置断点调试程序来获得代码的 行为是否符合预期的反馈的，但是这种反馈，比起可以复用的自动化测试来说，为代码测试 者提供的反馈要迟缓得多。

所以，第一个编程操练最后生成的带有mainO方法的代码，仍然是一段需要继续驯服的 烂代码。

再看看本章开头唐僧收悟空为徒时的悟空，是否也符合烂代码的那3点共性呢？首先， 悟空能轻易地降服取经路上所遇到的狼虫虎豹和妖怪强盗，说明悟空能够起到保护唐僧的作 用，可以运行；其次，悟空有一味杀生行凶这个bug，需要修复，而且还要增加皈依佛法的 “新功能”；最后，悟空接受不了唐僧的教海，撇下师父独自离去，这对于悟空的“维护者”

唐僧来说，悟空对其教化的反馈也是迟缓的。

所以，悟空即使刚刚接受完如来佛把他压在五行山下的驯服，在拜唐僧为师时，也还是 一段需要继续驯服的烂代码。

定义了什么是烂代码后，接下来咱们要做几个驯服烂代码的编程操练。不过接下来的编 程操练题目与之前的那个不同。前面的那个题目是从零开始编写代码，而接下来的题目是要 驯服已有的烂代码。在操练之前，先回顾一下本章的内容：

1. 国外程序员认为烂代码就是那些没写测试、无法理解、难以修改、结构混乱、重复 冗余和依赖全局信息的代码。
2. 国内程序员认为烂代码难以理解和维护，而且自己以前多多少少也写过一些烂代码。
3. 本书对烂代码的定义，包含了烂代码的3点共性：能够运行、需要修改其中的bug 或在其中增加新功能、对于代码的编写者、测试者和维护者来说反馈迟缓。简而言之，烂代 码就是反馈迟缓的代码。

第11章

记录所闻到的“腐臭”

我相信，世界上100%的程序员每天都会面临烂代码的问题。

“那可不一定，有些开发新项目的程序员是从零开始编程的。”

对，我以前在工作中也做过从零开始的开发项目。但是我发现，如果不做代码重构的 话，那些从零开始写出的代码，当天就会变成充满代码“腐臭”的烂代码。过段时间，代码 就会臭不可闻，以至于烂得不可收拾。此时即使推翻重写，若还是不重构的话，最终的代码 还是会沦为烂代码。但即便是像Kent Beck这样的精通重构的TDD开发方法的创立者，每天 也在驯服烂代码。

“此话怎讲？”

Kent Beck提出了 Red-Green-Refacotr (变红-变绿-重构）这样的TDD开发的“六字 真言”。他把TDD开发方法过程演变为这个“六字真言”的一个接着一个的迭代。其中Red 表示先写一个运行失败（在IDE中会变红）的测试，Green表示可以用最省事的办法让该测 试运行通过（在IDE中会变绿)，Refactor表示在测试的保护下把刚才写的那些最省事的代码 中的诸如重复代码这样的“腐臭”给重构好。从这个过程能够看出，Refactor过程其实就是 驯服Green步骤所产生的烂代码。这样说起来，无论是否做代码重构，程序员每天都会面临 烂代码的问题。

“嗯，驯服已有的烂代码，其实就可以看成是从头用TDD写新代码的第3步：Refactor。”

对，不过在做Refactor之前，需要针对要Refactor的行为编写测试。不编写测试，而直 接修改代码不能叫Refactor,只能叫“裸奔” e。

㊀没有测试保护的代码修改就是“裸奔”，这个说法来自软件开发顾问申健的新浪微博@申导。

前面操练的用TDD从零开始写代码的开发方法，为咱们驯服已有烂代码做好了铺垫。 下面咱们就利用前面所学到的技能，来驯服一段已有的烂代码。

这段烂代码所对应的编程操练题目叫Trivia，最初是我读Emily Bache所著的The Coding Dojo Handbook0—书时看到的。该题目专为操练驯服烂代码所设计，包含了 C++、 C#、Java、JavaScript、Objective-C、PHP、Python、Ruby、Scala 等 10 多种语言的版本。这 些语言的版本可以在Emily的github®页面上找到。

Trivia这个题目实现了一个答题闯关的游戏。几个参赛的玩家通过轮流掷色子（又称骰 子）来决定每个人在游戏盘上的位置（以下简称：玩家位置)，然后回答问题。如果回答正确 就会获得金币，否则就被关进禁闭室。被关禁闭的玩家，在下次掷色子时，若掷出的点数是 奇数，则可以走出禁闭室，继续在游戏盘上前进到新的位置，并有机会通过回答问题来贏取 金币；若掷出的点数是偶数，则继续在禁闭室里待着，不能前进和回答问题。一旦产生了第 一个获得6枚金币的玩家，游戏结束。

“看一看源代码3吧。”

源代码只有3个类：Game类实现了这个游戏的所有接口，相当于服务端；GameRunner 类里有个main〇方法，调用了 Game类里面的接口来运行这个游戏，相当于客户端；Game- Test 类是一个测试类，里面只有一个测试 “2+3=5” 的这个必然通过的测试， 用来验证单元 测试框架JUnit是否能正常工作。

先看一下客户端GameRunner类的代码。

GameRunner类的代码如下所示8:

public class GameRunner {

private static boolean notAWinner;

public static void main(String[] args) {

Game aGame = new Game();

aGame.add("Chet"); aGame.add("Pat"); aGame.add("Sue");

Random rand = new Random();

do {

aGame.roll(rand.nextlnt(5) + 1);

参见•• <https://leanpub.com/codingdojohandbook> 参见：<https://github.com/erailybache/trivia>

㊀㊁㊂ ㊃

Trivia 操练题目原有代码参见：<https://github.com/wubin28Abc-trivia-java/tree/exercise> ;本书针对该题目的 操练步骤代码参见：[https://github.com/wubin28/tbc-trivia-java/tree/master。](https://github.com/wubin28/tbc-trivia-java/tree/master%e3%80%82)

为节省篇幅，这里和以后的代码都省略了 package和import语句。

if (rand.nextlnt(9) == 7) {

notAWinner = aGame.wrongAnswe r();

} else {

notAWinner = aGame.wasCorrectlyAnsweredO;

}

} while (notAWinner);

}

}

“这个客户端的代码，首先新建一个Game的对象。然后添加了 3个玩家，并构建了一 个随机数对象。接着在一个do…while循环中，首先掷色子，然后回答问题，若还未产生贏 家，则循环继续。在掷色子时，用上面那个随机数对象来模拟产生色子的点数。在回答问题 时，还用这个随机数对象来模拟问题回答得是否正确这种随机情况。”

可以运行一下这个main〇方法。

运行客户端GameRunner类的main()方法的部分输出结果如下所示：

Chet was added

They are player number 1

Pat was added

They are player number 2

Sue was added

They are player number 3

Chet is the current player

They have rolled a 5

Chet's new location is 5

The category is Science

Science Question 0

Answer was corrent!!!!

Chet now has 1 Gold Coins.

Pat is the current player They have rolled a 4 Pat1s new location is 4 The category is Pop Pop Question 0 Answer was corrent!!!!

Pat now has 1 Gold Coins.

Sue is the current player

Sue now has 5 Gold Coins. Chet is the current player They have rolled a 3 Chet's new location is 8 The category is Pop Pop Question 2 Answer was corrent!!!!

Chet now has 6 Gold Coins.

最后看看Game类，这是咱们要驯服的烂代码。 Game类的代码如下所示e:

public class Game {

ArrayList players = new ArrayListU；

int[] places = new int[6];

int[] purses \* new int[6];

boolean[] inPenaltyBox = new boolean[6];

LinkedList popQuestions = new LinkedList(); LinkedList scienceQuestions = new LinkedList(); LinkedList sportsQuestions = new LinkedList(); LinkedList rockQuestions = new LinkedList();

int currentPlayer = 0;

boolean isGettingOutOfPenaltyBox;

public Game() {

for (int i=0; i < 50; i++) {

popQuestions . addLast (,fPop Question " + i); scienceQuestions.addLast(("Science Question " + i)); sportsQuestions.addLast(("Sports Question " + i)); rockQuestions.addLast(createRockQuestion(i));

}

public String createRockQuestion(int index) { return "Rock Question " + index;

public boolean isPlayable() {

return (howManyPlayers() >= 2);

public boolean add(String playerName) { players.add(playerName); places[howManyPlayers()] = 0; purses[howManyPlayers()] = 0; inPenaltyBox[howManyPlayers()] = false;

System.out.println(playerName + ” was added");

System .out .print In ("They are player number ’’ + players. size ()); return true;

public int howManyPlayers() {

㊀这里列出的代码主要供以后参考，可以粗略看一下。后面将讨论如何阅读这段代码。

return players.size();

public void roll(int roll) {

System. out .println (players. get (currentPlayer) + •’ is the current player"); System.out.println("They have rolled a " + roll);

if (inPenaltyBox[currentPlayer]) {

if (roll % 2 != 0) {

isGettingOutOfPenaltyBox = true;

System.out.println(players.get(currentPlayer) + " is getting out of the penalty box");

places[currentPlayer] = places[currentPlayer] + roll; if (places[currentPlayer] > 11) places[currentPlayer]= places[currentPlayer] - 12;

System.out.println(players.get(currentPlayer)

+ M's new location is "

+ places[currentPlayer]);

System.out.println ("The category is ’’ + currentCategory ()); askQuestion();

} else {

System.out.println(players.get(currentPlayer) + " is not getting out of the penalty box"); isGettingOutOfPenaltyBox = false;

}

} else {

places[currentPlayer] = places[currentPlayer] + roll; if (places[currentPlayer] > 11) places[currentPlayer]= places[currentPlayer] - 12;

System.out.println(players.get(currentPlayer)

+ "'s new location is "

+ places[currentPlayer]};

System.out.println("The category is " + currentCategory()); askQuestion();

private void askQuestion() {

if (currentCategory () =»= ’’Pop”）

System.out.println(popQuestions.removeFirst()); if (currentCategory() == "Science")

System.out.println(scienceQuestions.removeFirst()); if (currentCategory() == "Sports")

System.out.println(sportsQuestions.removeFirst()); if (currentCategory() == "Rock")

private String currentCategory() {

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| if | (places[currentPlayer] | == | 〇) | return "Pop"; |
| if | (places[currentPlayer] | == | 4) | return "Pop"; |
| if | (places[currentPlayer] | == | 8) | return "Pop"; |
| if | (places[currentPlayer] | == | 1) | return "Science"; |
| if | (places[currentPlayer] | == | 5) | return "Science"; |
| if | (places[currentPlayer] | == | 9) | return "ScienceM; |
| if | (places[currentPlayer] |  | 2) | return "Sports"; |
| if | (places[currentPlayer] |  | 6) | return "Sports"; |
| if (places[currentPlayer]  return "Rock"; | | == | 10) | return '\*Sports"; |

}

public boolean wasCorrectlyAnswered() { if (inPenaltyBox[currentPlayer]) {

if (isGettingOutOfPenaltyBox) {

System.out.println("Answer was correct!!!!"); purses[currentPlayer]++;

System.out.println(players.get(currentPlayer)

+ " now has ••

+ purses[currentPlayer]

+ •• Gold Coins.'1)；

boolean winner = didPlayerWin(); currentPlayer++;

if (currentPlayer == players•size()) currentPlayer = 0; return winner;

} else {

currentPlayer++;

if (currentPlayer == players•size()) currentPlayer = 0; return true;

}

} else {

System, out .println (MAnswer was corrent !!!!’’）； purses[currentPlayer]++;

System.out.println(players.get(currentPlayer)

+ " now has n + purses[currentPlayer]

+ ’• Gold Coins.");

boolean winner = didPlayerWin(); currentPlayer++;

if (currentPlayer == players.size()) currentPlayer = 0;

return winner;

public boolean wrongAnswer() {

System.out.println("Question was incorrectly answered");

System.out.println(players.get(currentPlayer) + " was sent to the penalty box"); inPenaltyBox[currentPlayer] = true;

currentPlayer++;

if (currentPlayer == players.size()) currentPlayer = 0; return true;

}

private boolean didPlayerWin() {

return !(purses[currentPlayer] == 6);

}

}

“这么长的代码从头看到尾也得要看一阵子。”

我以前也是从头到尾地看代码，但很快就发现这样读总是不得要领。后来从读书中获得 启发，用读书时找书中章节的方法来读代码就好多了。

“我读书时也是会时不时地看看目录中的章节，以便知道自己看到哪里，还有多少页能 看完。但是代码里面哪里有章节呢？”

我所说的代码里的章节，其实指的是服务端代码里的公共接口，也就是那些声明为 public的方法的定义。这些服务端的接口，一方面被客户端所调用，另一方面也必然会去调 用服务端内部各个私有方法和变量。有这些接口作为指引，就能按图索骥地读遍所有代码。 所以它们的作用就如同书中的章节一样。

现在咱们的任务就是以Game类的公共接口为索引，参照客户端GameRimner类如何调 用这些接口来阅读代码。可以先把这个任务写成TODO,并标上working-on,表示咱们正在 执行这个任务。

在Game类中添加有关阅读代码的TODO如下所示（CM: Working on TODO: Check public interface of the server-side code to see how it is being used by the client-side code.)：

public class Game {

+ // TODO-working-on: Check public interface of the server-side code to see

how it is being used by the client-side code public Game(> {

for (int i = 0; i < 50; i++) {

“可以利用IDEA里面的Structure工具窗口来显示Game类的所有public方法，并将其

设置为能与源代码同步显示，以方便阅读代码。”

利用IDEA里面的Structure工具窗口来显示Game类的所有接口如图11-1所示。

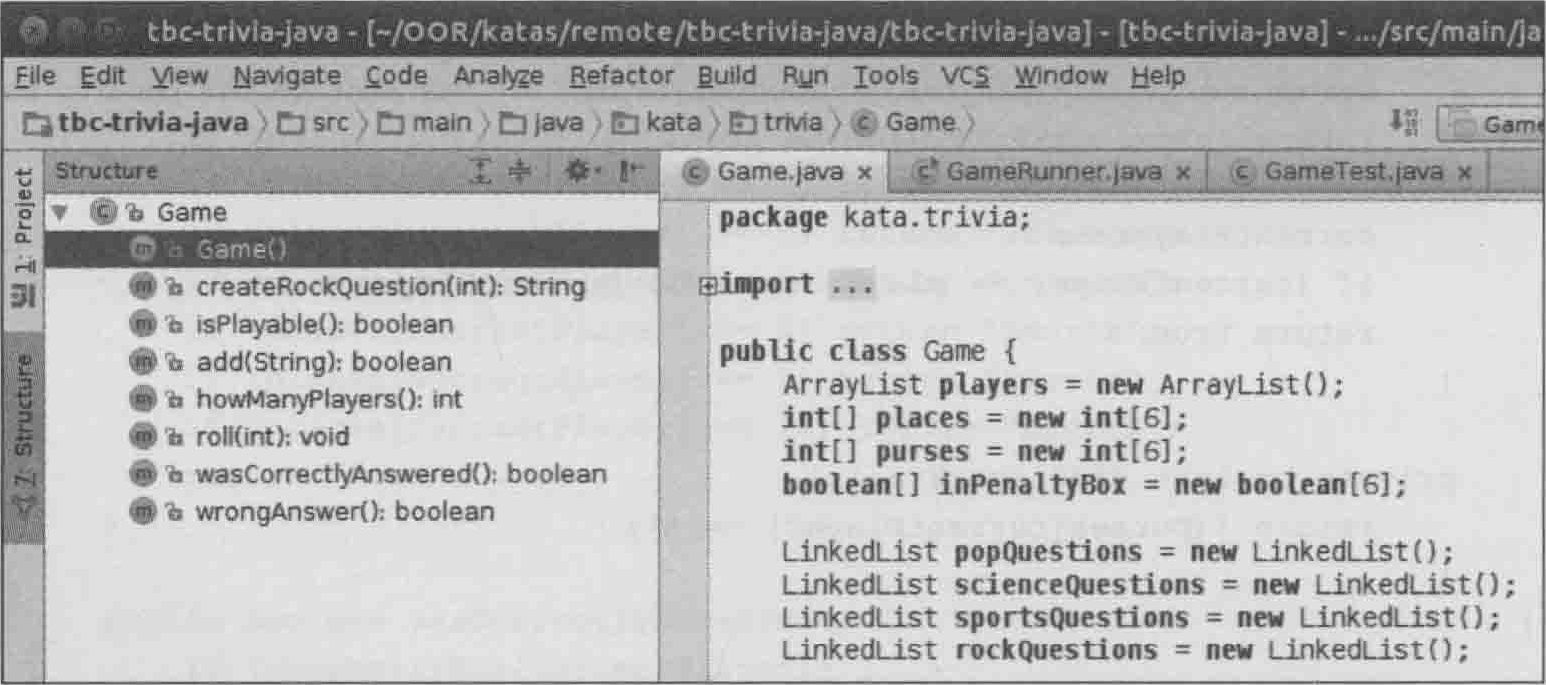


图 11 -1 Structure 工具窗口

Game类有8个public方法，也就是说有8个接口。咱们一个一个地看一下。

Game()这个构造器把游戏中用于提问的流行歌曲（Pop)、科学（Science)、体育（Sports) 和摇滚歌曲（Rock)这4个问题链表通过一个循环赋了初值。

createRockQuestion()方法用于创建有关摇滚歌曲的问题。这个方法有两个问题。首先， 它仅仅被上面那个GameO构造器所使用，客户端没有使用，所以声明为public是没有必要 的。另外，它仅返回有关Rock的问题字符串供rockQuestions链表使用，这与Game()构造 器中其他3个链表的初始化工作没什么不同。所以这个方法其实没有做什么重要的事情，可 以大致对应那22种“腐臭” 6中的那个Lazy Class (冗赘类）“腐臭”，完全可以inline。

“现在就inline它吧。”

肯定要inline它，但现在还不是时候^因为现在working-on的TODO是读代码，而不是 重构代码。另外，现在的代码还没有得到任何测试的保护。在没有测试保护的情况下就修改 代码，前面已经讲过，这不能叫重构，而只能叫“裸奔”，因为不知道这个改动是不是会破 坏原有代码的行为。

“要是现在不管它，接下来继续读代码，就会忘记的。”

可以先写个TODO把在读代码时发现的问题记下来，等将来测试写好了再解决不迟。

在Game类中添加有关将方法createRockQuestion()进行inline的TODO如下所7K ( CM: Added TODO: inline method Game.createRockQuestionQ.)：

©即《重构》一书中所列出的22种代码“腐臭”。

public class Game {

+ // TODO; inline method Game. createRockQuestion ()

rockQuestions.addLast(createRockQuestion(i));

}

}

虽然inline的方法createRockQuestion()能够同时解决该方法不应该是public的问题，但 现在还未到重构代码的时候，所以咱们现在还是写个TODO把这个问题记下来。

在Game类中添加有关将createRockQuestion()方法改为private的TODO，具体如下所 示（CM: Added TODO: Change method Game.createRockQuestion() to be private.):

public class Game {

+ //TODO: Change method Game. createRockQuestion () to be private public String createRockQuestion(int index) { return "Rock Question " + index;

}

看完了两个接口，接着看第3个接口 isPIayableO。这个方法判断当前的玩家数量是不是 大于等于2,如果是的话，就返回true。看起来这个方法的意图是说，游戏只有在有两个或 两个以上的玩家玩时，才可以玩。

“不过这个方法在IDEA中显示为灰色的，表示没有被任何代码调用。”

对。这样说来，这个方法或许在将来有用，但目前没用。这个“腐臭”味道是不是和22 种“腐臭”中的那个Speculative Generality (夸夸其谈未来性）“腐臭”有点像？对付这种“腐 臭”的办法就是删掉它，等将来真的需要时再写不迟。不过现在还没到删除的时候，先写个 TODO。

在Game类中添加有关将isPlayable〇方法进行删除的TODO,具体如下所示（CM: Added TODO: Remove the unused method Game.isPlayable().)：

public class Game {

+ //TODO: Remove the unused method Game. isPlayable () public boolean isPlayable() {

return (howManyPlayers() >= 2);

}

接下来看下一个接口 add()。这个接口将玩家的名字加人players链表，另外还将3个 数组元素的值初始化。这3个数组元素分别记录了下一个玩家位置（placesG)、金币数量 (purses[])和是否在禁闭室（inPenaltyBox[])的情况。接着用System.out.println〇打印一些信 息出来，然后返回true。

“这个方法挺有意思，最后返回true。但这个返回值在客户端没有被使用。”

对，这个true返回得既没有什么道理，又没有代码使用它c■似乎也有些Speculative Generality (夸夸其谈未来性）“腐臭”的味道。

“既然没有代码使用，那就在将来改一下这个接口，让它返回void。”

对，可以先用TODO记录下来。另外，用System.out.println()打印信息，虽然方便开发 时的调试，但使用起来很不灵活，一方面不能长期保存这些信息，另一方面不能在需要时关 闭这些信息输出。不如把这些信息都写到一个log文件里，这样既可以方便地复制或读取log 文件，又可以方便地设置log输出级别来控制是否输出这些信息。针对这个问题可以再写一 个 T0D0o

在Game类中添加有关add()方法返回值未被使用和将System.out.println()替换为log的 两个 TODO 如下所示（CM: Added 2 TODOs: l)The return value of method Game.add〇 is not used; 2)Replace System.out.println() with a log method of a logger.)：

public class Game {

+ //TODO: The return value of method Game. add () is not used, public boolean add(String playerName) {

inPenaltyBox[howManyPlayers()] = false;

+ //TODO: Replace System, out. println () with a log method of a logger

System.out.println(playerName + " was added");

System.out.println("They are player number " + players.size()); return true;

下一个接口是howManyPlayers()。这个接口返回玩家的数量。看起来似乎对客户端有 用，但实际上客户端并没有用到它。反而是Game类内部的其他方法在使用它。

“可以把它从一个public的接口改为private的私有方法。”

对。这个public方法改为private相当于除去了这个接口，使得Game类的整个接口范围 变得更窄。在满足现有客户端的需求的前提下，接口要设计得尽量窄，这样一方面能让这个 类所做的事情更加专注，另一方面能消除维护这些额外接口的成本。记下这个TODO。

在Game类中添加有关将howManyPlayers()方法变为private的TODO如下所7K (CM: Added TODO: The method Game.howManyPlayers() should be private because it is only used by its own calss Game.)：

public class Game {

+ //TODO: The method Game.howManyPlayers() should be private because it is only used by its own calss Game public int howManyPlayers() {

return players.size()/

}

下一个接口是方法roll()。这个方法的参数ro丨丨表示当前玩家所掷色子的点数。如果当前 玩家没有被关禁闭，或者当正在被关禁闭的玩家所掷的点数是奇数，从而从禁闭室中被释放 出来时，可以根据所掷点数来在游戏盘上前进到正确的位置，并接受提问。若该玩家正在被 关禁闭，且所掷的点数是偶数的，那么该玩家继续被关禁闭。

“这个roll〇方法里从1:往下数第2个if语句和第2个else语句中的大部分内容都是重复的。” 对。22种代码“腐臭”中排名第一的Duplicated Code (重复代码）“腐臭”终于出现了。 记下这个TODO。

在Game类的roll()方法中添加有关出现重复代码的TODO如下所示（CM: Added TODO: Duplicate code in method Game.roll().)：

public class Game {

} else {

+ //TODO: Duplicate code in method Game, roll ()

places[currentPlayer] = places[currentPlayer] + roll; if (places[currentPlayer] > 11) places[currentPlayer】= places[currentPlayer] - 12;

下一个接口是wasCorrectlyAnswered〇方法，做玩家答对问题后的事情：若当前玩家没 有被关进禁闭室，或者已经从禁闭室释放出来，就奖励玩家一块金币，然后计算玩家是否已 经获得6块金币而胜出并结束游戏，接着选出下一位玩家，并返回玩家是否获胜的结果；若 当前玩家还未出禁闭室，则仅仅选出下一位玩家，并返回true,表示玩家尚未获胜。

这个方法的返回值我感觉有些别扭：它是一个boolean型，如果返回true表示玩家尚未 获胜，返回false表示玩家已经获胜。这里返回值的true与false,和其所代表的实际语义的 肯定与否定正好相反，让我理解起来得多绕一道弯。为什么不用如下等价的表达呢：如果 返回true表示游戏仍在继续，返回false表示游戏不能继续玩了。还是加一个TODO把方法 didPlayerWin()更名为 isGameStiUInProgress()好一些。

在Game类中添加有关把方法didPlayerWin()更名的TODO如下所示（CM: Added TODO: The name of the method Game.didPlayerWin〇 should be Game.isGameStillInProgress〇.)：

public class Game {

+ "TODO: The name of the method Game.didPlayerWin() should be Game. isGameStilllnProgress() private boolean didPlayerWin() {

return !(purses[currentPlayer] == 6);

既然didPlayerWin()方法要改名为isGameStillInProgress()，那么保存它的返回值的变量 winner也要跟着一起改名为isGameStilllnProgress。再加一个TODO记录这个问题。

在Game类的wasCorrectlyAnswered()中添加有关变量winner改名的TODO如下所不 (CM: Added TODO: Rename variable 'winner' to be 'isGameStilllnProgress'.)：

public class Game {

+ // TODO: Rename variable 'winner' to be 'isGameStilllnProgress'.

boolean winner = didPlayerWin();

“这里又有重复代码了。这个wasCorrectlyAnswered()方法里从上往下数第1个else语句 里的内容，与第2个if语句中最后的部分相重复。另外，第2个else语句里的内容，与第2 个if语句中的内容几乎完全相同，除了 else语句里有一个corrent的拼写错误。有意思的是, 后面那段重复代码，又包括了前面那段重复代码。”

把这两处重复代码分别写到两个TODO里面。后面那处重复代码的TODO后面加个 Outer,表明它包含了前一处的重复代码。

在Game类中wasCorrectlyAnswered()方法里添加有关消除第1处重复代码的TODO如 下所示（CM: Added TODO: Duplicate code in method Game.wasCorrectlyAnswered().):

public class Game {

return winner;

} else {

+ //TODO: Duplicate code in method Game.wasCorrectlyAnswered〇

currentPlayer++;

if (currentPlayer == players.size()) currentPlayer = 0; return true;

在Game类中wasCorrectlyAnswered〇方法里添加有关消除第2处重复代码的TODO如 下所示（CM: Added TODO: Duplicate code in method Game.wasCorrectlyAnswered〇. Outer.):

public class Game {

} else {

+ "TODO: Duplicate code in method Game.wasCorrectlyAnswered(). Outer.

System.out.println("Answer was corrent!!!!"); purses[currentPlayer]++;

System.out.println(players.get(currentPlayer)

Game类最后一个接口是方法wrcmgAnswerO,做玩家答错问题后的事情：将其关进禁闭 室，选出下一位玩家，然后返回true,表示游戏仍在继续。这个方法的返回值虽然被客户端 •GameRunner所使用，但是它永远是true,让人感觉这个返回值似乎没有存在的必要。用一

个TODO把它记录下来。

在Game类中添加有关wrongAnswer()方法的返回值没有必要的TODO如下所示（CM: Added TODO: The return value of method Game.wrongAnswer() is unnecessary and should be eliminated.)：

public class Game {

currentPlayer++；

if (currentPlayer == players.size()) currentPlayer = 0;

+ //TODO: The return value of method Game.wrongAnswer() is unnecessary

and should be eliminated return true;

}

闻到了一些服务端的代码Game类中的代码“腐臭”后，现在可以闻闻客户端的代码 GameRunner类中是否也有“腐臭”。

“既然Game类的wrongAnswer()和wasCorrectlyAnswered()方法都返回游戏是否仍在继 续，那么GameRunner类中的notAWinner成员变量也就可以更名为isGameStilllnProgress。”

对。不过咱们这次操练的重点是重构服务端代码Game类。而且一般客户端的代码都在 用户的计算机上，程序员也无法访问到。所以客户端的代码“腐臭”在本次操练中可以暂时 不管。

现在基本上把代码通读了一遍，同时把所遇见的代码“腐臭”都写成了 TODO。再看看 还有什么“腐臭”遗漏了。

“Game类中的那些成员变量的访问权限最好都改为private,而不要是default。改为 private封装性更好一些。”

在Game类中添加有关把成员变量的访问权限改为private的TODO如下所示（CM: Added TODO: The fields of class Game should be private.)：

public class Game {

+ //TODO: The fields of class Game should be private ArrayList players = new ArrayListO; int[] places = new int [6]; int[] purses = new int[6];

“还有，Game类中的roll()方法的参数roll,表示的是掷色子的点数。最好能改个更确 切的名字，比如叫rollingNumber,而不要和这个方法的名字相同，容易混淆。”

在Game类的roll〇方法中添加有关把参数roll改名的TODO如下所示（CM: Added TODO: Rename the name of the parameter of method Game.rollQ to be 'rollingNumber'.)：

public class Game {

+ // TODO: Rename the name of the parameter of method Game.roll() to be 'rollingNumber'

public void roll(int roll) {

System.out.println(players.get(currentPlayer) + " is the current player"); System.out.println("They have rolled a n + roll);

“好了，现在暂时闻不到其他‘腐臭’了 D”

读完了代码，写好了 TODO,下一步就开始编写测试来固化代码现有行为，以便进行重 构了。不过在写测试前，让我们看看本章都做了哪些事。

1 )阅读答题闯关游戏Trivia的已有烂代码。

1. 在阅读代码的过程中，因尚未编写测试以固化现有代码的行为，所以对随时发现的 代码“腐臭”暂不做修改，而是以TODO注释的形式添加到源代码相应的位置中，待将来写 好测试后再做处理。
2. 阅读代码时发现了以下代码“腐臭”：
3. 1^2^0&38冗赘类（见(^^1611〇(±〇^81丨011()方法）0
4. Speculative Generality 夸夸其谈未来性（见 isPlayable()方法和 add()方法)。
5. 用 System.out.println()打印调试信息。
6. 没有必要的（即无客户端调用的）宽接口（见howManyPlayers〇方法）
7. Duplicated Code 重复代码（见 roll〇 方法和 wasCorrectlyAnswered()方法）〇
8. boolean取值与其所表达语义的真值相反（见wasCorrectlyAnswered()方法）〇
9. 没有必要的default访问权限的成员变量（见Game类中所有成员变量)。
10. 词不达意的变量名（见roll〇方法的参数)。
11. 对于客户端代码的“腐臭”暂不做修改。
12. 通过操练我们学到了以下技能：
13. 所有程序员，不管是高手还是新手，每天都会面对烂代码的“腐臭”。
14. 在对代码做重构之前，必须针对要重构的代码编写测试。没有测试保护的代码修改不 能叫重构，而只能叫“裸奔”。
15. 阅读代码时，可以把代码的公共方法看成图书中的章节，并将其作为指引，来阅读 代码。

**第12章**

用测试描绘用户意图

虽然从操练的角度看，咱们仅仅为了练习而驯服Trivia游戏的烂代码的确无可厚非，但 是从商业价值的角度考虑，如果没有在烂代码上修复bug或增添新功能的需求，而仅仅为 了驯服烂代码而驯服的话，那就是浪费。这一点在前面有关烂代码的第二条特性中讲过。所 以，为了让这个操练离现实情况更近一些，咱们不妨增加一个新特性，并替换掉原先的一个 旧特性。即将“若玩家正在被关禁闭且所掷的点数是偶数的，那么该玩家继续被关禁闭”改 为“若玩家正在被关禁闭且所掷的点数是4时，那么该玩家继续被关禁闭"。换句话说，正 在被关禁闭的玩家若掷出了除4之外的任何点数，都会被从禁闭室里释放出来。

在实现这个新特性前，需要编写测试，来为将来的重构和编写新特性做准备。第一个测 试该测什么呢？

“我看可以把Game类的那些公共接口一个一个都测一遍。比如可以测Game类的构造 器、add()方法、roll()方法、wasCorrectlyAnswered()方法和 wrongAnswer()方法。第一个测 试可以测Game类的构造器。”

这种测试的策略看起来似乎很全面，但有一个比较大的问题。在讨论这个问题之前，咱 们先回忆一下软件设计原则中的依赖倒置原则说的是什么来着？

“原话记不起来了，但大意是要针对抽象编程。让我翻一翻Bob大叔的书査査啊。找到 了，在这里。依赖倒置原则有两点内容，一是高层模块不应该依赖于低层模块，两者都应该 依赖于抽象；二是抽象不应该依赖于细节，细节应该依赖于抽象。e”

既然良好的设计原则讲究写代码需要针对抽象编程，而不是针对具体实现编程，那么同

㊀Robert C. Martin著，邓辉译，《敏捷软件开发：原则、模式与实践》，清华大学出版社，2003年9月第I版

样是代码的测试代码，也应该针对用户意图和公共接口这样的抽象来编写，而不要针对具体 代码实现编写。因为抽象的变动要比具体实现的变动小很多。而针对抽象编写的测试才不会 那么脆弱。前面那个根据User Story来编写TDD测试的酒店世界时钟的编程操练，也是这个 思路，因为User Story—般都描述了用户意图。

构造器的测试一般要验证一个类的对象的构造是否符合预期，但这事应该属于该类自己 的内部实现，用户很少会去关心。如果将测试耦合到构造器的具体实现，那么一方面需要添 加一些诸如反射、Mock这些额外代码来验证这些内部实现，增加了编写测试的成本和代码 复杂性，另一方面一旦这个内部实现发生了变更，这个测试就势必运行失败，同时所有上述 与其相关的额外代码都要进行修改，使得这个测试很脆弱，也增大了维护测试的成本。

“但如果测试不去验证这些接口方法的内部行为，那该测什么呢？”

测试从用户角度所看到的代码意图，即用户意图。好比您是使用Game类的用户，您希 望用了这个类后，它能为您做什么有价值的事情呢？

“用户当然希望Trivia这个游戏能顺利地玩下去。比如能添加玩家、掷色子、在游戏盘 上前进、回答问题、贏金币、关禁闭，还有最后游戏能结束。”

很好呀！这些事情就是代码背后的用户意图。而Game类的每个接口内部的行为，其实 是代码的实现意图。用户意图相比实现意图来说，哪个更不容易变化呢？

“当然是用户意图不容易变化。”

对。这里的用户意图有点接口的意思。编程讲究针对接口来进行，那么相应地，测试就 要讲究针对用户意图来进行。这样才能达到高内聚、低稱合的目标。

“可是有时候我见到的用户，除了谈他要的意图之外，还和我说了一大堆如何实现的具 体技术。这些具体实现的技术算用户意图吗？”

这个问题很好。咱们这里所说的用户意图，仅仅指不关心技术的用户从使用产品的角度 所提出的意图，而不包含实现该意图的技术等其他意图。明智的用户，应该只专注于描述用 户使用层面的意图，而不应染指程序员对编程技术的选择。

“现在用户也不在身边，只有咱们这两个程序员。前面阅读的代码也只是实现层面的内 容，那又如何去确定用户意图呢？”

虽然咱俩不是真正意义上的用户，但不妨戴上用户这顶“思考帽” e，想想要是自己是 用户的话，能有什么用户意图呢？

可以先从最简单的用户意图开始。我现在能想到的最简单的第1个用户意图，是这个游戏 只有一个玩家，他掷了 6 ?欠色子并正确回答了 6次问题，那么游戏在6次掷色子后就能结束。

“要是回答问题出错呢？”

那就可以想出答题出错时最简单的第2个用户意图，即还是这个玩家，他第1次掷色子后

回答问题就答错了，从而被关禁闭。但接下来下一次掷色子的点数是奇数，从而能够去答题, 且答题正确。再接下来的5次掷色子回答问题他都答对了，那么游戏在7次掷色子后就能结束。 “要是被关禁闭后下一次掷色子是偶数呢？”

这就是第3个用户意图，即这个玩家，第1次掷色子回答问题答错被关禁闭，接下来下 一次掷色子是偶数，所以系统将不关心他这次掷色子之后的回答问题的正确与否。再接下来 的6次掷色子回答问题他又都答对了，那么游戏在8次掷色子后结束。

在GameTest测试类中添加有关3个用户意图测试的TODO如下所示（CM: Added 3 user intent tests as TODOs .)：

public class GameTest {

+ // TODO-user-intent: the game should be over if a player rolls the dice and

answers each question correctly for 6 times + // TODO-user-intent: the game should be over if a player rolls the dice for 7

times and answers the question wrongly for 1 time followed by an odd rolling number but then correctly for 6 times +• // TODO-user-intent: the game should be over if a player rolls the dice for 8

times and answers the question wrongly for 1 time followed by an even rolling number but then correctly for 7 times

现在就可以根据上面第1个用户意图来编写测试，即一个用户掷6次色子且6次答题 全对。测试代码如下所示（CM: Added user intent test the\_game\_should\_be\_over\_if\_a\_player\_ rolls\_the\_dice\_and\_answers\_each\_question\_correctly\_for\_6\_times(). Ran it and it passed.)：

@Test

public void the\_game\_should\_be\_over\_if\_a\_player\_rolls\_the\_dice一and\_answers一 each一question一correctly\_for—6一times(} {

// Arrange

Game game = new Game(); game.add("Chet\*');

boolean isGameStilllnProgress = true;

//Act

for (int i = 0; i < 6; i++) { game.roll(1);

isGameStilllnProgress = game.wasCorrectlyAnswered();

}

// Assert

assertFalse(isGameStilllnProgress);

㊀对代码提交的CoramitMessage的改进由于git的特性而无法事后更改，所以本书所列的一些Commit Message和一些代码本身会与github上的略有出人，以本书内容为准。

上面代码中的最后一行assertFalse〇代码，如果写成assertTrue〇也可以。因为咱们 是针对已有的生产代码写测试，所以重点要看这个测试实际运行后的结果，再根据实际结 果调整这条语句和测试。这种根据测试的实际运行结果来编写测试的方法叫做特征测试0 (characterization test) 〇

上面测试代码实际运行的结果是运行通过，而且assertFalseO语句也符合这个测试所对 应的用户意图，所以就不再对上面的代码进行调整了。

第2个用户意图是首次答题错误但接下来掷色子获得了奇数的点数，且以后答题全部正 确，其测试代码如下所示（CM: Added user intent test the\_game\_should\_be\_over\_if\_a\_player\_ rolls\_the\_dice\_for\_7\_times\_and\_answers\_the\_question\_wrongly\_for\_l\_time\_foIlowed\_by\_an\_ odd\_rolling\_number\_but\_then\_correctly\_for\_6\_times(). Ran it and it passed.)：

@Test

public void the\_game\_should\_be\_over\_if\_a\_jplayer\_rolls\_the\_dice\_for\_7\_times\_ and\_answers\_the\_question\_wrongly\_for\_l\_time\_followed\_by\_an\_odd\_rolling\_ number一but\_then\_correctly\_for\_6\_times(> {

// Arrange

Game game = new Game(); game.add("Chet");

boolean isGameStillInProgress = true;

*1/ Act*

game.roll(1); game.wrongAnswer(); game.roll(1);

game.wasCorrectlyAnswered(); for (int i = 0; i < 5; i++) { game.roll(1);

isGameStillInProgress = game.wasCorrectlyAnswered();

// Assert

assertFalse(isGameStilllnProgress);

}

第3个用户意图是首次答题错误但接下来掷色子获得了偶数的点数，但以后答题全部正 确，其测试代码如下所示（CM: Added user intent test the\_game\_should\_be\_over\_if\_a\_player\_ rolls\_the\_dice\_for\_8\_times\_and\_answers\_the\_question\_wrongly\_for\_l\_time\_followed\_by\_an\_ even\_rolling\_number\_but\_then\_correctly\_for\_7\_times\_with\_odd\_rolling\_numbers(). Ran it and it passed.)：

㊀Michael C. Feathers著，刘未鹏译，《修改代码的艺术》，人民邮电出版社，2007年11月第1版，第13章。

@Test

public void the\_game\_should\_be\_over\_if\_a\_player\_rolls\_the\_dice\_for\_8\_times\_ and\_answers\_the\_question\_wrongly\_for\_l\_time\_followed\_by\_an\_even\_rolling\_ number\_but\_then\_correctly\_for\_7\_times\_with\_odd\_rolling\_numbers () {

// Arrange

Game game = new Game(); game.add("Chet");

boolean isGameStilllnProgress = true;

"Act

game, roll (1); game.wrongAnswer(); game.roll(2);

game.wasCorrectlyAnswered(); for (int i = 0; i < 6; i++) { game.roll(1);

isGameStilllnProgress = game.wasCorrectlyAnswered();

// Assert

assertFalse(isGameStilllnProgress);

}

在编写了这3个用户意图测试并运行通过后，生产代码的行为就会被测试所固化。其结 果就是一旦接下来对生产代码所做的改动破坏了上面其所固化的软件行为，测试就会运行失 败。程序员就可以利用频繁运行自动化测试这一点，来确保接下来对生产代码所做的重构， 不会破坏那些已被固化的行为。

“有了这3个测试的保护，咱们就可以放心地对付前面写的那些代码‘腐臭’的TODO 了。” 对。Game类现在已经有14个TODO 了。咱们可以按照这些TODO在代码中出现的先 后顺序一个一个地解决。

在IDEA中的Game类中的丨4个TODO如图12-1所示。

i Current Rb Scope Based

a.

? w

j ▼ Found 16 TODO items in 2 files

g] kata.trMa (26 ttems .n *2* hies)

ill (7. 8) *TODO: The fields of* cfass *Game should be private*

g (26. 16) *TODO: Inline method Game. createRock〇uestfon〇 S* (31. 8) *TODO: Change method Game.createRockQuestionO to be private* B (36. 8) / *TODO: Remove the unused method Game.isPlayableO* ig (41. 8) TODO: *The return value of method Game.addO i<s not used.*



■ (50. 12} *TODO: Replace System, out.printlnO with a log method of s logger*

(56. 8) *TODO: The method Game.howManyPlayersO should be private because it is only used by its own calss Gsme* 圖（61. 8> r〇DO: Rename f fie name of the pardmefer of fnef/iod Game. fo b« TolT/ngMumber *M* (86. 16) *TODO: Duplicate code in method Game.rollO* g (134, 20) v *TODO: Rename variable winner to be isGameStilllnProgress .*

!lj (141. 20) *TODO: Duplicate code in method Game.wasCorrectlyAnsweredO. Inner.*

13 (150. 16) *TODO: DupUuUe code in method Game.wasCorrecttyAnsweredO. Outer.*

p (173, 12) *TODO: The return value of method Game. wrongAnswerO is unnecessary and should be eliminated* ^ (178. 8) *TODO: The name of the method Game.didPlayerWinO should be Game.isGamcStilllnProgressO*

图 12-1 Game 类的 14 个 TODO

“第 1 个 TODO 是 The fields of class Game should be private。这个 TODO 要求把 Game 类的所有成员变量的访问权限都改为private的。”

完成有关Game类的成员变量应该是private的TODO的代码如下所示（CM: Finished TODO: The fields of class Game should be private.)：

public class Game {

一 //TODO: The fields of class Game should be private

* ArrayList players = new ArrayList();
* int [ ] places = new int[6];
* int[] purses = new int[6]/
* boolean[] inPenaltyBox = new boolean[6];
* LinkedList popQuestions = new LinkedList();
* LinkedList scienceQuestions = new LinkedList();
* LinkedList sportsQuestions = new LinkedList();
* LinkedList rockQuestions = new LinkedList()/

int currentPlayer = 0;

boolean isGettingOutOfPenaltyBox'-

private ArrayList players **=new** ArrayList();

private int[] places = new int[6];

private int[] purses = new int[6];

private boolean[] inPenaltyBox = new boolean[6];

+

LinkedList popQuestions = new LinkedList(); LinkedList scienceQuestions - new LinkedList(); LinkedList sportsQuestions = new LinkedList(); LinkedList rockQuestions = new LinkedList();

**private**

**private**

**private**

**private**

+ private int currentPlayer =0;

+ private boolean isGettingOutOfPenaltyBox;

public Game() {

for (int i = 0; i < 50/ i十+) {

改完代码后，别忘了运行测试。测试运行通过。

“把createRockQuestion()方法inline能够解决接下来的两个TODO〇”

完成Game类中有关createRockQuestion()方法的两个TODO的代码如下所7K ( CM: Inlined a method and finished 2 TODOs: l)Change method Game.createRockQuestion() to be private; 2)Inline method Game.createRockQuestion().)：

public class Game {

scienceQuestions . addLast ( ("Science Question ’’ + i)); sportsQuestions.addLast(("Sports Question " + i));

* // TODO: Inline method Game .createRockQuestion ()
* rockQuestions.addLast(createRockQuestion(i))/

+ rockQuestions.addLast(^Rock Question " + i);

* // TODO: Change method Game.createRockQuestion() to be private
* public String createRockQuestion(int index) {
* return "Rock Question **n + index;**
* }

运行测试，通过。

“删除目前没用的isPlayableO方法这个TODO也很简单。”

完成删除Game类的isPlayable〇方法的TODO代码如下所示（CM: Finished TODO: Remove the unused method Game.isPlayable〇.)：

public class Game {

* //TODO: Remove the unused method Game. isPlayable ()
* public boolean isPlayable() {
* return (howManyPlayers() >\* 2);
* )

运行测试，通过。

“接下来解决 The return value of method Game.add() is not used.这个 TODO。”

且慢，解决这个TODO就势必要改变Game类的这个公共接口的返回值。由于这个公共 接口已经被客户端所调用了，所以对其所做的任何改动都要十分谨慎。虽然对于这个题目咱 们可以确信那个有main〇方法的客户端没有调用这个方法，但我还是宁愿把这个TODO放 到最后来做。在实际工作中，这种对服务端的公共接口的改动，也需要尽量往后放一放，以 便有时间来确信这种改动对客户端的影响在可控范围内。

“那就把这个TODO标记为later,表示以后再处理。“

将有关Game类的add()方法的TODO标记为later的代码如下所示（CM: Mark this TODO to handle it later: The return value of method Game.add() is not used.)：

- //TODO: The return value of method Game.add() is not used.

+ // TODO-later: The return value of method Game.add() is not used.

“下一个把System.out.printlnO替换为log方法的TODO由于工作量相对较大，也标记为 later 吧。”

将Game类的有关System.out.printlnO的TODO标记为later的代码如下所示（CM: Mark this TODO to handle it later: Replace System.out.printlnO with a log method of a logger.)：

* // TODO: Replace System.out .println () with a log method of a logger

+ // TODO-later: Replace System, out .println () with a log method of a logger

“下一个TODO是把howManyPlayers()方法改为private的。不像前面那样，这个方法没 有被客户端所调用，所以就可以比较放心地将其由public改为private。”

完成 Game 类有关 howManyPlayers()方法的 TODO 的代码如下（CM: Finished TODO: The method Game.howManyPlayers() should be private because it is only used by its own class Game.)：

public class Game {

* //TODO: The method Game.howManyPlayers() should be private because it is

only used by its own calss Game

* public int howManyPlayers() {

+ private int howManyPlayers() {

return players.size();

}

运行测试，通过。

“下一个TODO是把roll()方法的参数roll改名为rollingNumber,这个改动用IDEA的 rename重构功能就能轻松完成。”

完成Game类有关roll〇方法的参数改名的TODO的代码如下所示（CM: Finished TODO: Rename the name of the parameter of method Game.roll() to be 'rollingNumber'.)：

* //TODO: Rename the name of the parameter of method Game.roll() to be 'rollingNumber'
* public void roll(int roll) {

+ public void roll(int rollingNumber) {

* System.out.println("They have rolled a " + roll);

+ System.out.println(MThey have rolled a " + rollingNumber);

* if (roll % 2 != 0) {

+ if (rollingNumber % 2 != 0) {

* places[currentPlayer】=places[currentPlayer] + roll;

+ places[currentPlayer】=places[currentPlayer] + rollingNumber;

* places[currentPlayer] = places[currentPlayer] + roll;

+ places[currentPlayer] = places[currentPlayer] + rollingNumber;

运行测试，通过。

“下一个TODO是roll()方法里的重复代码，这可以通过提取方法currentPlayerMovesTo NewPlaceAndAnswersAQuestion()来解决〇 ”

完成Game类有关roll()方法中重复代码的TODO的代码如下所示（CM: Extracted

method currentPlayerMovesToNewPlaceAndAnswersAQuestion() and finished TODO: Duplicate code in method Game.roll().)：

public class Game {

* places[currentPlayer] = places[currentPlayer] + rollingNumber;
* if (places[currentPlayer] > 11) places[currentPlayer]=

places[currentPlayer] - 12;

* System.out.println(players.get(currentPlayer)
* + n's new location is "
* + places[currentPlayer]);
* System.out.println(MThe category is " + currentCategory());
* askQuestion();

+ currentPlayerMovesToNewPlaceAndAnswersAQuestion(rollingNumber);

} else {

System.out.println(players.get(currentPlayer) + " is not getting out of the penalty box"); isGettingOutOfPenaltyBox = false;

}

**)else** {

+ currentPlayerMovesToNewPlaceAndAnswersAQuestion(rollingNumber);

+ 1

* // TODO: Duplicate code in method Game.roll ()
* places[currentPlayer] = places[currentPlayer] + rollingNumber;
* if (places[currentPlayer] > 11) places[currentPlayer]=

places[currentPlayer] - 12;

* System.out.println(players.get(currentPlayer)
* + "'s new location is **"**
* + places[currentPlayer]);
* System.out.println("The category is " + currentCategory());
* askQuestion();
* }

+ }

+ private void currentPlayerMovesToNewPlaceAndAnswersAQuestion (int rollingNumber) { + places[currentPlayer] = places[currentPlayer] + rollingNumber;

+ if (places[currentPlayer] > 11) places[currentPlayer]=

places[currentPlayer] - 12;

+ System.out.println(players.get(currentPlayer)

+ + "\* s new location is "

+ + places[currentPlayer]);

+ System.out.println("The category is " + currentCategory());

+ askQuestion();

运行测试，通过。

“下一个 TODO 是将变量 winner 改名为 isGameStilllnProgresSo”

完成Game类有关重命名变量winner的TODO的代码如下（CM: Finished TODO: Rename variable ’winner，to be ’isGameStilllnProgress’.）：

* //TODO: Rename variable 'winner' to be 'isGameStilllnProgress'.
* boolean winner = didPlayerWin《）；

+ boolean isGameStilllnProgress = didPlayerWin();

- return winner;

+ return isGameStilllnProgress;

- boolean winner = didPlayerWin();

十 boolean isGameStilllnProgress = didPlayerWin();

- return winner;

+ return isGameStilllnProgress;

运行测试，通过。

“接下来的TODO是wasCorrectlyAnswered()方法内层的重复代码，可以通过提取方法 nextPlayer()来解决。”

完成Game类有关wasCorrectlyAnswered〇方法中内部重复代码的TODO的代码如下 所示（CM: Extracted method nextPlayer〇 and finished TODO: Duplicate code in method Game. wasCorrectlyAnswered(). Inner.)：

public class Game {

+ " Gold Coins.")；

boolean isGameStilllnProgress = didPlayerWin();

* currentPlayer++;
* if (currentPlayer == players.size()) currentPlayer - 0;

+ nextPlayer();

return isGameStilllnProgress;

} else {

* //TODO: Duplicate code in method Game.wasCorrectlyAnswered(). Inner.
* currentPlayer++ ;
* if (currentPlayer == players.size{)) currentPlayer = 0;

+ nextPlayer();

return true;

Gold Coins.\*');

boolean isGameStilllnProgress = didPlayerWin();

currentPlayer++;

if (currentPlayer == players.size()) currentPlayer = 0; nextPlayer();



return isGameStillInProgress;

+ private void nextPlayer() {

+ currentPlayer++;

+ if (currentPlayer == players.size()) currentPlayer = 0;

+ }

+

public boolean wrongAnswer() {

System.out.println("Question was incorrectly answered");

System.out.println(players.get(currentPlayer) + " was sent to the penalty box")； inPenaltyBox[currentPlayer] = true;

* currentPlayer++;
* if (currentPlayer = players.size⑴ currentPlayer = 0;

+ nextPlayer〇;

运行测试，通过。

“接下来该解决wasCorrectlyAnswered()方法的外层重复代码了。”

这个外层的重复代码有一个笔误：correct被误写为corrent。这个笔误给予我们两点启 示。首先，用复制粘贴的方法生成重复代码后，随着时间的推移，重复的代码会发生一些变 化，与原来的代码不一致，从而增大了消除重复代码的难度。另外，要消除重复代码，必须 先把两段或多段重复代码恢复成相互之间完全相同的状态。

“好的。下面就把这个笔误改过来，使得两段重复代码完全相同。”

将两段重复代码还原为完全一致的代码如下所7K ( CM: Corrected a typo to make sure the duplicate code is exactly the same before extracting a method.)：

* System.out .println ("Answer was corrent !!!!’•）；

+ System.out.println ("Answer was correct !!!!\*')；

“改好了笔误，就可以解决wasCorrectlyAnsweredO方法的外层重复代码了。可以通过提 取 currentPlayerGetsAGoldCoinAndSelectNextPlayer()方法来消除这两段重复代码 〇 ”

完成Game类有关wasCorrectlyAnsweredO方法的外层重复代码的TODO的代码如下 所示（CM: Extracted method cinTentPlayerGetsAGoldCoinAnclSelectNextPlayerO and finished TODO: Duplicate code in method Game.wasCorrectlyAnswered(). Outer.)：

public class Game {

public boolean wasCorrectlyAnsweredO {

if (inPenaltyBox[currentPlayer]) {

if (isGettingOutOfPenaltyBox) {

System.out.println("Answer was correct!!!!"); purses[currentPlayer]++;

System.out.println(players.get(currentPlayer)

+ " now has ’\*

+ purses[currentPlayer]

+**，**• Gold Coins.")；

boolean isGameStillInProgress = didPlayerWin(); nextPlayer();

return isGameStillInProgress;

return currentPlayerGetsAGoldCoinAndSelectNextPlayer(); } else {

nextPlayer(); return true;

} else {

return currentPlayerGetsAGoldCoinAndSelectNextPlayer();

// TODO: Duplicate code in method Game.wasCorrectlyAnswered(). Outer System.out.println("Answer was correct!!!!")； purses[currentPlayer]++;

System.out.println(players.get(currentPlayer)

+ " now has "

+ purses[currentPlayer]

+ " Gold Coins ；

boolean isGameStilllnProgress = didPlayerWin(); nextPlayer();

return isGameStilllnProgress;

}

private boolean currentPlayerGetsAGoldCoinAndSelectNextPlayer() {

System .out .println ("Answer was correct! ! i ! \*'); purses[currentPlayer]++;

System.out.println(players.get(currentPlayer)

+ " now has "

+ purses[currentPlayer]

+ •，Gold Coins.")；

boolean isGameStilllnProgress = didPlayerWin()/ nextPlayer();

return isGameStilllnProgress;

private void nextPlayerO {

运行测试，通过。

“下一个TODO是wrongAnswer()的那个没有存在必要的永远返回true的返回值。由于 客户端正在使用这个返回值，不便于修改，所以把这个TODO标记为later。”

将Game类有关wrongAnswer()方法没有必要存在的返回值的TODO标记为later的代 码如下所示：

* //TODO: The return value of method Game.wrongAnswer() is unnecessary

and should be eliminated

+ // TODO-later: The return value of method Game.wrongAnswer() is

unnecessary and should be eliminated

“最后一个 TODO 是把方法 didPlayerWin()改名为 isGameStillInProgress()。”

完成Game类有关didPlayerWin()方法重命名的TODO的代码如下所示（CM: Finished TODO: The name of the method Game.didPlayerWin() should be Game.isGameStillInProgress().)：

* boolean isGameSti11InProgress = didPlayerWin〇;

+ boolean isGameStillInProgress = isGameStilllnProgress();

* //TODO: The name of the method Game.didPlayerWin() should be Game.

isGameStilllnProgress()

* private boolean didPlayerWin() {

+ private boolean isGameStilllnProgress<) {

运行测试，通过。现在Game类剩下3个later的TODO 了，如图12-2所示。

|  |  |  |
| --- | --- | --- |
| T000 BS333 Cunrent ft|e Scope Based | |  |
| f I 4.令 | V Found 5 TODO Items in 2 files ▼ kata.trivia T5 aems >n 2 fliesJ | |
| ? W  \_ | €1 (29, 8) ' TODO-laten The return value of method Game.addO is not used.  y (38, 12) TODO-later. Replace System, out. printlnO with a log method of a logger  U (143, 12), TODO-laten The return value of method Game.wrongAnswerO is unnecessary and should be eliminated  ►够 GameRunner.java | |
|  | 图 12-2 | Game类剩下3个later的TODO |

前后两个TODO都是被客户端使用的服务端的接口，还是暂时放一放。现在可以处理中 间那个把System.out.println()替换成log方法的TODO。在实际工作中记录log比较常用的做 法是把log保存到文件中，以便于阅读和分享。所以我打算在这里把SyStem.〇Ut.println()所 打印的日志信息都保存到log文件中。首先需要创建一个能完成这个任务的logger。

完成Game类中有关替换System.out.println()方法的TODO的代码如下所本（CM: Finished TODO: Replace System.out.println() with a log method of a logger.):

java.io.IOException; j ava.util.logging.FileHandler; j ava.util.logging.Logger; java.util.logging.SimpleFomatter;

public class Game {

4-import +import 4-import +import

private ArrayList players = new ArrayListO;

+ private static Logger logger = Logger.getLogger("kata.trivia.Game");

+ private static FileHandler fileHandler = null;

+

public Game〇 {

+ try {

+ fileHandler = new FileHandler (n%h/Game-logging.log'[[1]](#footnote-1), 10000000, 1, true);

+ fileHandler.setFormatter(new SimpleFormatter());

+ ) catch (IOException e) {

+ e.printStackTrace();

+ }

+ logger.addHandler(fileHandler);

运行测试，通过。上面创建 FileHandler 实例的 new FileHandler("%h/Game-logging.log"， 10000000, 1，true)的语句里面有4个参数，其中第1个参数％h/Game-logging.log表示日志文 件名是Game-logging.log,并在用户的home目录下被创建〇第2个参数10000000表7K每个 曰志文件的最大字节数为10兆字节。第3个参数1表示只使用这一个日志文件。第4个参 数true表示新生成的日志会被添加到日志文件的尾部。

得到了一个能够写入日志文件的logger,接下来就可以把System.out.printlnO语句都替 换成logger.info()语句，这样每一句logger.info()语句中的日志都会被记录到日志文件中。

将Game类中System.out.printlnO方法替换为logger.info()方法的代码如下所7Ke:

* // TODO-later: Replace System.out.printlnO with a log method of a logger
* System.out.println(playerName + " was added");
* System.out.println("They are player number " + players.size());

+ logger .info (playerName + ’’ was added");

+ logger.info("They are player number n + players.size());

return true;

public void roll<int rollingNumber) {

* System, out .println (players, get (cur rent Player) + ’• is the current player");
* System, out .println ("They have rolled a •’ + rollingNumber);

+ logger.info(players.get(currentPlayer) + " is the current player");

+ logger • info ("They have rolled a •’ + rollingNumber);

运行测试，通过。“剩下两个later的TODO中，上面那个add()方法的返回值没有被客 户端使用的TODO更容易处理一些。所以现在就可以去掉这个返回值。”

完成Game类中有关add()方法返回值的TODO的代码如下所示（CM: Finished TODO: The return value of method Game.add() is not used.)：

* // TODO-later: The return value of method Game.add() is not used.
* public boolean add(String playerName) {

+ public void add(String playerName) {

public void add(String playerName) {

logger. info (playerName + •’ was added");

logger.info("They are player number " + players•size());

- return true;

}

运行测试，通过。“好了，除了还剩下一个以后处理的later TODO外，所有与代码‘腐 臭’相关的TODO都已经处理完了。”

还有一个味道很重的代码“腐臭”没有被处理。

“是不是《重构》一书中所说的那个Large Class ‘腐臭’ ？”

没错。Trivia这个游戏的所有事情都让Game这一个类来处理，这违反了软件设计的单 一职责原则和开闭原则9。不过在咱们解决Large Class这个“腐臭”之前，先看看本章所做 的工作。

1. 根据用户意图编写了3个测试，运行通过，固化代码已有行为，并以此作为后面重 构的保护网。
2. )按TODO列表的顺序依次解决各个TODO来进行重构。
3. 通过操练我们学到了以下技能：
4. 测试的编写应该面向相对稳定的用户意图这个抽象，而不是面向相对易变的已有代码 的具体实现。
5. 首先编写表现用户意图的测试，并运行通过，来固化软件行为。然后在其保护下按在 代码中出现的先后顺序来处理与代码“腐臭”相关的TODO。
6. 在重构过程中，只要有可能运行测试，就尽量频繁地运行测试，以保证重构不会破坏 已被测试所固化的软件行为。
7. 在重构过程中需要修改那些已被客户端所使用的公共接口时，一定要谨慎，可以尽量

往后放一放，以便有时间来确信这种改动对客户端的影响是否在可控范围内。

1. 对于工作量较大的TODO,可以标记为later以便以后处理。
2. 重复代码可以通过方法提取来解决。
3. 去除重复代码的工作要尽早进行，否则随着时间的推移，重复的代码会发生一些变 化，与原来的代码不一致，从而增大了消除重复代码的难度。
4. 要消除重复代码，必须先把两段或多段重复代码恢复成相互之间完全相同的状态。
5. 把用于日志记录的System.out.println()方法替换为logger.info()方法，并保存到日志 文件中，便于阅读和分享。

前面处理的包含重复代码在内的那十几个代码“腐臭” TODO,为现在咱们处理Large Class “腐臭”打好了基础。

分而治之

釜底抽薪

“对，如果不先处理像重复代码这样的代码“腐臭”，而是一上来就要把一个大类分解为 几个小类，一方面会由于重复代码造成代码过长而难以阅读，从而给分解工作带来困难，另 一方面分解大类的操作可能会破坏重复代码相互之间完全相同的状态，并有可能让重复代码 分散到不同的小类中，增加后面消除重复代码的难度。”

没错。现在就可以观察一下Game类的代码，看看有哪些代码逻辑可以被分离到新的 类中。

“Game类的代码开头有两块定义成员变量的代码，中间被一行空行所分隔。这两块聚在 —起的成员变量，是不是在暗示咱们可以把它们分别放到两个新的类中呢？”

很好。就如同Martin Fowler在《重构》一书中所说的：数据项就像小孩子，喜欢成群 结队地待在一块儿。……这些总是绑在一起出现的数据真应该拥有属于它们自己的对象。

Game类的代码开头两块定义成员变量的代码如下：

public class Game {

private ArrayList players = new ArrayList(); private int[] places = new int[6]; private int[] purses = new int[6]; private boolean[] inPenaltyBox = new boolean[6];

private LinkedList popQuestions = new LinkedList(); private LinkedList scienceQuestions = new LinkedList();

private LinkedList sportsQuestions = new LinkedList{); private LinkedList rockQuestions = new LinkedList();

“前面一块代码定义了4个成员变量，来分别保存玩家的名字、在游戏盘上的位置、钱 包中的金币数量和是否在禁闭室的状态。这些都和玩家相关，所以可以创建一个新的玩家类 Player，并把这4个成员变量移动到Player类中。”

可以添加一个TODO来记录这件事。

在Game类中添加将成员变量playerName、places、purses和inPenaltyBox移动到新 的 Player 类的 TODO 中，代码如下（CM: Added TODO: Move playerName，places，purses and inPenaltyBox to a new class Player.)：

public class Game {

+ //TODO: Move playerName, places, purses and inPenaltyBox to a new class Player private ArrayList players = new ArrayList(>; private int[] places \* new int[6]; private int[] purses - new int[6]; private boolean[] inPenaltyBox = new boolean[6];

“后面那块代码定义了 4个有关问题的成员变量，可以放到一个有关问题的新类中。”

对，这个新类不妨叫QuestionMaker。

在 Game 类中添加将成员变量 popQuestions、scienceQuestions、sportsQuestions 和 rockQuestions 移动到新的 QuestionMaker 类的 TODO 中，代码如下（CM: Added TODO: Move question lists to a new class QuestionMaker.)：

public class Game {

+ //TODO: Move question lists to a new class QuestionMaker private LinkedList popQuestions = new LinkedList(); private LinkedList scienceQuestions = new LinkedList(); private LinkedList sportsQuestions = new LinkedList(); private LinkedList rockQuestions = new LinkedList();

“上面这两个TODO相比起来，后一个看起来更简单一些，所以先处理后一个TODO。”

把有关移动question list的TODO标记为working-on并创建QuestionMaker类，代码 如下（CM: Working on TODO: Move question lists to a new class QuestionMaker. Created class QuestionMaker.)：

public class Game {



+ // TODO-working-on: Move question lists to a new class QuestionMaker

♦public class QuestionMaker {

+ }

“接下来，就可以把这4个question list成员变量移动到QuestionMaker类中了。”

把4个question list成员变量从Game类移动到QuestionMaker类的代码如下（CM: Moved the 4 question lists from class Game to QuestionMaker.)：

public class Game {

// TODO-working-on: Move question lists to a new class QuestionMaker private LinkedList popQuestions = new LinkedList(); private LinkedList scienceQuestions = new LinkedList(); private LinkedList sportsQuestions = new LinkedList(); private LinkedList rockQuestions = new LinkedList ();

public class QuestionMaker {

十 // TODO-working-on: Move question lists to a new class QuestionMaker + private LinkedList popQuestions = new LinkedList();

+ private LinkedList scienceQuestions = new LinkedList();

+ private LinkedList sportsQuestions = new LinkedList();

十 private LinkedList rockQuestions = new LinkedList();

对生产代码所做的重构无非是用新代码替换旧代码。替换的方法可以分为两种：第一 种是不保留旧代码，而直接用新代码替换旧代码，并最终保证原有的测试在新代码上也能运 行通过，这种方法好比“釜底抽薪”，从根本上解决问题；第二种是在保留旧代码的基础上， 在旧代码后面编写新代码，待新代码编写完，将代码行为切换到新代码后仍能保证测试运行 通过，此时再删除旧代码，这种方法好比“抛砖引玉”，照着旧代码这块“砖”编写新代码 这块“玉”。“釜底抽薪”的方法适用于简单的重构。“抛砖引玉”的方法适用于复杂一些的

重构。

“现在咱们使用的就是‘釜底抽薪’的方法。按照咱们的重构意图，在把4个question list成员变量从Game类移动到QuestionMaker类后，在Game类中利用这4个成员变量 对question list进行添加和删除的地方就会出现编译错误。可以利用这些编译错误的指引 来将原先向question list中添加问题的语句，替换为QuestionMaker相关的意图代码，即 QuestionMaker类需要4个添加问题的方法来添加那4类问题

将原先向question list中添加问题的语句替换为QuestionMaker相关的意图代码，如下所 7K (CM: Replaced question-adding code with intention code of using QuestionMaker object.)：

public class Game {

for (int i 二 0; i < 50; i++) {

* popQuestions. addLast (MPop Question •• + i);
* scienceQuestions.addLast(("Science Question " + i));
* sportsQuestions.addLast(("Sports Question " + i));
* rockQuestions. addLast ("Rock Question ’’ + i);

+ questionMaker. addPopQuestion ("Pop Question ’• + i);

+ questionMaker.addScienceQuestion((MScience Question " + i));

+ questionMaker .addSportsQuestion (("Sports Question •’ + i));

+ questionMaker.addRockQuestion("Rock Question " + i);

}

“上面意图代码中的questionMaker是红色的，表示有编译错误。原因是它们都没有创 建。应该在Game类中创建它。”

在 Game 类中创建成员变量 questionMaker 的代码如下（CM: Created field questionMaker in class Game.)：

public class Game {

+ private final QuestionMaker questionMaker = new QuestionMaker();

“创建好了 questionMaker这个成员变量，现在上面意图代码中的4个添加问题的方法又 变成表示编程错误的红色。这需要咱们创建这4个方法。”

在QuestionMaker类中创建4个问题添加方法的代码如下（CM: Created 4 question adding methods in class QuestionMaker.)：

public class QuestionMaker { + public void addPopQuestion(String popQuestion) {

+ popQuestions.add(popQuestion);

+ }

+

+ public void addScienceQuestion(String scienceQuestion) { + scienceQuestions.add(scienceQuestion);

+ }

+

+ public void addSportsQuestion(String sportsQuestion) {

+ sportsQuestions.add(sportsQuestion);

+ }

+

+ public void addRockQuestion(String rockQuestion) {

+ rockQuestions.add(rockQuestion);

+ }

现在4个问题添加方法已经没有编译错误了。接下来该处理Game类的askQuestionO方 法中的4个问题删除方法的编译错误了。不过我忽然注意到，Game类中的ArrayList成员变

量player，和QuestionMaker类中的那4个LinkedList成员变量，都不是类型安全的，可以 添加两个TODO来分别记录下来。

在Game和QuestionMaker类中添加两个有关类型安全的TODO,代码如下（CM: Added 2 TODOs to make question and player lists type-safe.)：

public class Game {

+ //TODO: Make player list type-safe

private ArrayList players = new ArrayListO;

public class QuestionMaker {

//TODO: Make question lists type-safe private LinkedList popQuestions \* new LinkedList(); private LinkedList scienceQuestions = new LinkedList(); private LinkedList sportsQuestions = new LinkedList(); private LinkedList rockQuestions = new LinkedList();

“记下这两个TODO后，下面该处理4个问题删除方法的编译错误了。还是先写使用 questionMaker对象的新的问题删除的意图代码，即QuestionMaker类应该有4个方法来删除 那4类问题。”

在Game类中将问题删除的原有代码替换为使用QuestionMaker类的接口的意图代码， 如下所亦（CM: Wrote question removal intention code.):

public class Game {

if (currentCategory() == "Pop")

logger.info(popQuestions.removeFirst().toString()); logger.info(questionMaker.removeFirstPopQuestion()); if (currentCategory() == "Science")

logger.info(scienceQuestions.removeFirst().toString())/ logger.info(questionMaker.removeFirstScienceQuestion()); if (currentCategory() == "Sports")

logger.info(sportsQuestions.removeFirst().toString()); logger.info(questionMaker.removeFirstSportsQuestion()); if (currentCategory () == nRock,f)

logger.info(rockQuestions.removeFirst().toString()); logger.info(questionMaker.removeFirstRockQuestion());

“写好了问题删除的意图代码后，那4个删除问题的方法显示了有编译错误的红色。应 该来处理这些编译错误，即在QuestionMaker类中创建这4个删除问题的方法。”

在QuestionMaker类中创建4个问题删除方法，代码如下（CM: Created 4 question removal methods in class QuestionMaker.)：

no

public class QuestionMaker {

+ public String removeFirstPopQuestion() {

+ return popQuestions.removeFirst().toString();

+ }

+

+ public String removeFirstScienceQuestion() {

+ return scienceQuestions.removeFirst().toString();

+ }

+

+ public String removeFirstSportsQuestion() {

+ return sportsQuestions.removeFirst().toString();

+ }

+

+ public String removeFirstRockQuestion() {

+ return rockQuestions.removeFirst().toString();

+ }

}

将光标定位到GameTest类名上，并按Ctrl+ShifHFlO组合键运行测试，通过。现在完 成了将question list移动到QuestionMaker类中的TODO。下面可以处理这些question list类 型安全的TODO 了。首先将其标记为working-on。

“可以使用泛型编程来给LinkedList添加一个类型参数<String>，从而实现类型安全。”

完成有关 question list 类型安全的 TODO 的代码如下（CM: Finished TODO: Make question lists type-safe.):

public class QuestionMaker {

* // TODO-working-on: Make question lists type-safe
* private LinkedList popQuestions = new LinkedList();
* private LinkedList scienceQuestions = new LinkedList();
* private LinkedList sportsQuestions = new LinkedList();
* private LinkedList rockQuestions = new LinkedList()/

+ private LinkedList<String> popQuestions = new LinkedList<String>();

+ private LinkedList<String> scienceQuestions = new LinkedList<String>();

+ private LinkedList<String> sportsQuestions = new LinkedList<String>();

+ private LinkedList<String> rockQuestions = new LinkedList<String>();

public String removeFirstPopQuestion() {

return popQuestions.removeFirst().toString(); return popQuestions.removeFirst();

}

public String removeFirstScienceQuestion() {

return scienceQuestions.removeFirst().toString(); return scienceQuestions.removeFirst();

public String removeFirstSportsQuestion() {

* return sportsQuestions.removeFirst().toString();

+ return sportsQuestions.removeFirst();

}

public String removeFirstRockQuestion() {

* return rockQuestions.removeFirst().toString();

+ return rockQuestions.removeFirst();

}

)

按Ctrl+F5快捷键重复运行测试，运行测试仍然通过。

现在咱们已经写好了 QuestionMaker类。既然前面GameTest中的3个测试都没有测试 QuestionMaker类的用户意图，那么现在是不是就可以对QuestionMaker类进行测试呢？

“可以。咱们来测试QuestionMaker类的用户意图：能够添加和删除各类问题。比如咱们 可以针对每类问题，先添加2个问题，然后再测试能否成功地删除其中第1个问题。”

在GameTest类中添加4个有关QuestionMaker类的用户意图的测试TODO，代码如下 (CM: Added 4 TODOs to test the new implemented class QuestionMaker.)：

public class GameTest {

+ // TODO: add two pop questions and could remove the first one

+ // TODO: add two science questions and could remove the first one

+ H TODO: add two sports questions and could remove the first one

+ // TODO: add two rock questions and could remove the first one

}

“现在可以处理这4个用户意图TODO中的第1个：先添加两个流行音乐的问题，然后 测试一下能否删除这两个问题中的第1个问题。先编写这个测试的Assert部分。”

添加两个流行音乐问题并删除第1个问题的测试Assert部分，代码如下（CM: Working on TODO: add two pop questions and could remove the first one. Wrote an assertion.)：

public class GameTest {

* "TODO: add two pop questions and could remove the first one

+ // TODO-working-on: add two pop questions and could remove the first one

+ @Test

+ public void add\_two\_pop\_questions\_and\_could\_remove\_the\_\_first\_one 〇 {

+ // Assert

+ assertEquals (,fPop Question 1", questionMaker. removeFirstPopQuestion ());

+ }

“接下来可以把这个测试的Act和Arrange部分写完。”

添加两个流行音乐问题并删除第1个问题的测试Act和Arrange部分，代码如下（CM: Finished the 4 TODOs for testing the class QuestionMaker.)：

* // TODO-working-on: add two pop questions and could remove the first one @Test

public void add\_two\_pop\_questions\_and\_could\_remove\_the\_\_f irst\_one () {

+ // Arrange

+ QuestionMaker questionMaker = new QuestionMaker();

+

+ //Act

+ questionMaker.addPopQuestion(MPop Question 1");

+ questionMaker. addPopQuestion (\*' Pop Question 2");

+

// Assert

assertEquals("Pop Question 1", questionMaker.removeFirstPopQuestion());

}

“运行测试，通过。有了这个测试就能保证流行音乐问题的添加和删除就没有问题了。 接下来可以类似地处理剩下3个科学、体育和摇滚音乐问题的用户意图TODO。”

完成剩下3个科学、体育和摇滚音乐问题的用户意图TODO的代码如下：

* // TODO: add two science questions and could remove the first one
* //TODO: add two sports questions and could remove the first one
* //TODO: add two rock questions and could remove the first one

+ @Test

+ public void add一two—science一questions\_and\_could一remove一the\_first一one() {

+ // Arrange

+ QuestionMaker questionMaker \*= new QuestionMaker ();

+

+ //Act

+ questionMaker.addScienceQuestion("Science Question 1");

+ questionMaker.addScienceQuestion("Science Question 2");

+

+ // Assert

+ assertEquals ("Science Question l'f, questionMaker.removeFirstScienceQuestion ());

+ }

+

+ @Test

+ public void add\_two一sports\_questions一and一could\_remove\_the一first一one() {

+ // Arrange

+ QuestionMaker questionMaker = new QuestionMaker();

+

+ //Act

+ questionMaker.addSportsQuestion("Sports Question 1");

+ questionMaker.addSportsQuestion("Sports Question 2");

+

// Assert

+ assertEquals("Sports Question 1", questionMaker.removeFirstSportsQuestion());

+ )

+

+ @Test

+ public void add\_two\_rock\_questions\_and\_could\_remove\_the\_first\_one() {

+ // Arrange

+ QuestionMaker questionMaker = new QuestionMaker()?

+

+ //Act

+ questionMaker.addRockQuestion("Rock Question 1"};

+ questionMaker.addRockQuestion(nRock Question 2");

+

+ II Assert

+ assertEquals("Rock Question 1", questionMaker.removeFirstRockQuestion());

+ }

}

“运行测试，通过。现在可以处理将Game类的playerName、places、purses和inPenaltyBox 这4个成员变量移动到新类IMayer的TODO 了。”

前面有关4类问题的成员变量与这4个要被移动到Player类中的成员变量有所不同。前 者那4类问题彼此业务逻辑相似，所以可以一起处理。而后者所包含的玩家的名字、在游戏 盘上的位置、钱包中的金币数量和是否在禁闭室的状态这4个成员变量，它们彼此的业务逻 辑各不相同，所以最好能把这个TODO按照4个成员变量分解为4个TODO。

在Game类中将4个成员变量移动到Player类中的TODO分解为4个TODO,代码如下 (CM: Split the 'TODO: Move playerName, places, purses and inPenaltyBox to a new class Player' into 4 TODOs.)：

public class Game {

//TODO: Move playerName, places, purses and inPenaltyBox to a new class Player

+ //TODO: Move places into class Player private int [] places = new int[6];

+

+ //TODO: Move purses into class Player private int[] purses \* new int[6];

+

+ //TODO: Move inPenaltyBox into class Player

private boolean[] inPenaltyBox = new boolean[6];

// TODO: Move playerName into class Player players.add(playerName);

“先处理把玩家名字移动到Player类的TODO。可以编写意图代码来做这件事，即希 望players这个ArrayList的add()方法可以接受Player类型的参数，来向列表中添加Player 对象。”

用Player的新接口来添加新玩家的意图代码如下（CM: Wrote intention code to add a Player with player name to players.)：

public class Game {

public void add(String playerName) {

* // TODO: Move playerName into class Player
* players.add(playerName);

+ // TODO-working-on: Move playerName into class Player

+ players.add(new Player(playerName));

“因为现在还没有创建Player类，所以上面意图代码中new Player(playerName)语句 中的Player显示有编译错误的红色。为了消除这个错误，现在就可以创建Playei•类和它的构

vtBfl ”

JG.V^O

创建 Player 类和其构造器的代码如下（CM: Created class Player and its constructor.):

-♦-public class Player {

+ public Player(String playerName) {

+ }

+ }

“按Ctrl+F5快捷键运行测试，通过。因为现在向玩家列表中添加玩家名字的不再是字符 串，而变成了 Player对象。为了保证类型安全，现在可以处理那个将玩家列表变为类型安全 的 TODO 了。”

将玩家列表变为类型安全的TODO的代码如下（CM: Finished TODO: Make player list type-safe.)：

public class Game {

* // TODO: Make player list type-safe
* private ArrayList players = new ArrayList();

+ private ArrayList<Player> players = new ArrayList<Player>〇;

“按Ctrl+F5快捷键运行测试，通过。不过在IDEA的Run窗口看到的日志，玩家的名字 都是诸如kata.trivia.Player@lld50c0这样不正常的名字。”

在IDEA的Run窗口中部分玩家名字显示不正常的日志如下：

Jul 22, 2014 11:29:00 AM kata.trivia.Game currentPlayerMovesToNewPlaceAndAnswersAQuestion INFO: kata.trivia.Player@lld50c0's new location is 7

“这是由于Player类中没有覆写t〇String()方法。现在可以处理这件事。”

在Player类中覆写toStringO方法，以使得玩家名字在日志中正确显示，代码如下 (CM: Added overridden method toString() to class Player to make logging printing correct player names.)：

public class Player {

+ private String playerName;

+

public Player(String playerName) {

+ this.playerName = playerName;

+ }

+

+ @Override

+ public String toString() {

+ return this.playerName;

}

}

“按Ctrl+F5快捷键运行测试，通过。这回看到的日志中玩家名字显示正常了。”

在IDEA的Run窗口中部分玩家名字显示正常的日志如下：

Jul 22, 2014 11:39:10 AM kata.trivia.Game currentPlayerMovesToNewPlaceAndAnswersAQuestion

INFO: Sue's new location is 0

我还注意到，在IDEA的Run窗口的前面几行诸如They are player number 3这样记录玩 家总数的日志，有英文语法错误。最好能改成The total amount of players is 3。可以为这个问 题创建一个TODO。

在Game类中添加有关显示玩家总数日志的语法错误的TODO,代码如下（CM: Added TODO: The logging message should be The total amount of players is xx'.)：

public class Game {

+ // TODO: The logging message should be 'The total amount of players is xx'

logger . info ("They are player number ’• + players. size ());

}

现在玩家名字在日志中已经显示正常，所以将玩家名字移动到Player类的TODO就算 是完成了。现在可以来处理玩家总数日志语法错误的TODO,把它标记为working-on。

“处理玩家总数日志中语法错误的TODO很容易，把字符串改一下就好了。”

完成处理玩家总数日志中语法错误的TODO的代码如下（CM: Finished TODO: The logging message should be 'The total amount of players is xx'.)：

public class Game {

- // TODO-working-on: The logging message should be 'The total amount of

players is xx'

- logger.info(MThey are player number " + players.size());

+ logger•info<"The total amount of players is "+ players.size(});

}

“测试运行通过，而且日志中语法问题也改好了。把玩家名字移动到Player类中，接 下来就可以把玩家位置移动到Player类中。我注意到在Game类的add〇方法中，在把一个 Player对象添加到玩家列表后，下面那3行对玩家位置、钱包和在禁闭室的状态进行初始 化的语句，也要相应地移动到Player类中。所以这些初始化语句在此处就没有用了，可以 删除。”

在Game类中把玩家位置的计算逻辑移动到Player类的TODO标记为working-on,删除 玩家位置、钱包和在禁闭室的状态初始化的语句，代码如下（CM: Removed unused code from method Game.addQ.)：

public class Game {

* //TODO: Move places into class Player

+ // TODO-working-on: Move places into class Player

public void add(String playerName) {

players.add(new Player(playerName));

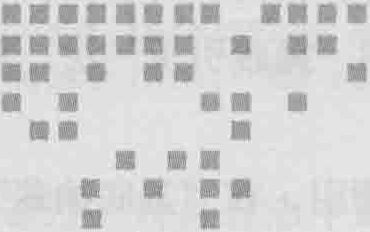
一 places[howManyPlayers()] = 0;

* purses[howManyPlayers()] = 0;
* inPenaltyBox[howManyPlayers()] = false;

在将玩家在游戏盘上的位置移动到Player类之前，先看一看本章都做了哪些工作。

1. 阅读Game这个大类的代码，计划把几个相关的成员变量提取到新的Player类和 QuestionMaker类中，并用TODO记录下来。
2. 使用“釜底抽薪”的方法，把与4个question list成员变量相关的代码从Game类移 动到 QuestionMaker 类。
3. )临时发现在Game类中的ArrayList成员变量player,和在QuestionMaker类中的那4 个LinkedList成员变量，都不是类型安全的，添加了两个TODO分别记录下来。
4. )为新创建的QuestionMaker类编写用户意图测试。
5. 由于彼此的业务逻辑各不相同，把Game类中将4个成员变量移动到Player类中的 TODO分解为4个TODO。
6. 通过操练我们学到了以下技能：
7. 在把一个大类分解为若干个小类前，要先消除重复代码这样的代码“腐臭”。
8. 对于总是聚在一起的那些成员变量，往往可以把它们提取到一个新类中。
9. 重构时替换代码的方法有两种，一种是直接用新代码替换旧代码的“釜底抽薪”法, 另一种是照着旧代码这块“砖”编写新代码这块“玉”的“抛砖引玉”法。前者适用于简单 的重构，后者适用于复杂的重构。
10. 可以利用根据意图修改代码所产生的编译错误的指引，在“釜底抽薪”的重构方法中 进行新旧代码的替换。
11. 对于在重构过程中新识别出来的代码“腐臭”，可以随时添加新的TODO来记录，待 时机成熟时再处理。
12. 对于新分解出来的类，要编写用户意图测试。
13. —个大TODO可以拆解为若干小TODO来完成。
14. 编写代码主要是给人阅读的，需要确保代码在传递信息方面的语法和语义正确，比如 要保证日志信息的语法正确。

“把玩家位置的计算逻辑移动到Player类这件事，看起来有些复杂，可以使用‘抛砖引 玉’的方法来进行。即可以在目前的位置计算逻辑的后面编写意图代码，等新的意图代码编 写完毕，将代码行为切换到它之上，并运行测试通过后，就可以删除原来的位置计算逻辑。 使用‘抛砖引玉’的重构方法的好处是，能够最大限度地让测试频繁得到运行，让复杂的重 构得到测试最大限度的保护。Player类应该有一个moveForwardSteps()方法，接受玩家所掷 色子的点数作为参数，表示该玩家在游戏盘上移动了点数所代表的步数而到达了新的位置。”



■

**第14章**

分而治之-

抛砖引玉

用Player类的新接口让玩家在游戏盘上前进的意图代码如下（CM: Wrote intention code to make the current player move forward rollingNumber steps.)：

public class Game {

private void currentPlayerMovesToNewPlaceAndAnswersAQuestion(int rollingNumber) { places[currentPlayer] = places[currentPlayer] + rollingNumber;

+ players.get(currentPlayer).moveForwardSteps(rollingNumber);

“这个moveForwardStepsQ方法目前显示的是有编译错误的红色，因此需要对 这个错误进行处理。消除这个编译错误最简单的方法是在类Playei■中创建一个空的 moveForwardSteps()方法。”

在类Player中创建一个空的moveForwardSteps()方法的代码如下（CM: Created method Player.moveForwardSteps().)：

public class Player {

+ public void moveForwardSteps(int steps) {

+

+ }

}

“运行测试，依然通过。我注意到，在Game类的currentPlayerMovesToNewPlaceAndAn swersAQuestion()方法中的加法和减法运算，可以使用+=和-=来消除重复代码。”

在Game类中用+=和-=来消除重复代码的代码如下（CM: Removed duplicate code using += and -= in method Game.currentPlayerMovesToNewPlaceAndAnswersAQuestion().):

public class Game {

private void currentPlayerMovesToNewPlaceAndAnswersAQuestion(int rollingNumber) { - places[currentPlayer] = places[currentPlayer] + rollingNumber;

+ places [currentPlayer] +=» rollingNumber;

- if (places[currentPlayer] > 11) places[currentPlayer]=

places[currentPlayer] - 12/

+ if (places[currentPlayer] > 11) places[currentPlayer] -= 12;

“现在Player类的moveForwardSteps()方法还是空的，咱们可以编写意图代码来实现它。 意图代码的实现逻辑与原先的是一样的，只不过在Player类中要有一个place成员变量来保 存玩家位置，还要把表示位置的数值控制在0到11之间。另外，在Game类的currentPlayer MovesToNewPlaceAndAnswersAQuestion()方法中，把表7K位置的数值控制在0到11之间的 逻辑往上移动一行，以使其与上面计算玩家位置的那行代码行挨在一起。”

在Player类中编写意图代码来实现moveForwardSteps〇方法功能代码如下（CM: Wrote intention code in method Player.moveForwardSteps().)：

public class Player {

public void moveForwardSteps(int steps) {

+ this.place += steps;

+ if (this.place > 11) this.place -= 12;

}

public class Game {

private void currentPlayerMovesToNewPlaceAndAnswersAQuestion(int rollingNumber) { places[currentPlayer] += rollingNumber;

- players.get(currentPlayer).moveForwardSteps(rollingNumber);

if (places[currentPlayer] > 11) places[currentPlayer] -= 12;

+ players.get(currentPlayer).moveForwardSteps(rollingNumber);

“在上面Player类的moveForwardSteps()方法的意图代码中，3个place都显了表7K编 译错误的红色，原因是未创建。现在就创建它。”

在Player类中创建成员变量place并将其初始化为0代码如下（CM: Created field place and initialized it in class Player.)：

public class Player {

private String playerName;

+ private int place = 0;

“按Ctrl+F5快捷键运行测试，通过。现在玩家位置已经能在新的Player类中进行计算 了，接下来就可以将这个计算好的结果从Player类中读取出来了。现在编写意图代码来做这 件事，Player类应该有一个getPlaceO方法来获取该玩家在游戏盘上的位置。”

编写意图代码将在Player类中计算好的玩家位置读取出来，代码如下（CM: Wrote intention code in method Game.currentPlayerMovesToNewPlaceAndAnswersAQuestion〇 to get the place from Player.)：

public class Game {

logger.info(players.get(currentPlayer)

+ n's new location is M + places[currentPlayer]);

+ logger.info(players.get(currentPlayer)

+ + "\*s new location is "

+ + players.get(currentPlayer).getPlace());

“在上面的意图代码中，由于Player类还没有创建getPlace()方法，所以getPlace〇显示 的是表示编译错误的红色。现在就处理这个错误。”

在Player类中创建和实现getPlace()方法的代码如下（CM: Created and implemented method Player.getPlaceQ.)：

public class Player {

+ public int getPlace() {

+ return this.place;

+ }

)

“运行测试，通过。观察一下日志，发现出现了两行有关玩家位置的日志。上面一行是 原先的日志，而下面一行是从新的Player类中获取的玩家位置的日志。两行日志完全一样， 说明后者已经成功地实现了前者的功能。”

我忽然发现，Game类中的currentCategoryO方法是根据玩家位置计算问题类别的，而 决定问题类别的逻辑似乎应该归QuestionMakei•类来负责。可以添加一个TODO来记录这 件事。

在Game类中添加将currentCategoryO方法从Game类转移到QuestionMaker类的TODO, 代码如下（CM: Added TODO: Move method Game.currentCategoryO to class QuestionMaker.):

public class Game {

+ // TODO: Move method Game.currentCategory() to class QuestionMaker

private String currentCategory() {

if (places[currentPlayer] == 0) return "Pop"; if (places[currentPlayer] == 4) return "Pop";

“现在有了 Player对象的getPlace()方法，就可以将Game类中获取玩家位置的原有代码 都替换为从Player对象的getPlace()方法中获取。先替换currentCategory()方法中的原有玩 家位置的代码。”

在Game类的currentCategoryO方法中使用Player类的新接口获取当前玩家位置，代码如 下所示（CM: Got the place of the current player from Player in method Game.currentCategory().):

public class Game {

// TODO: Move method Game.currentCategory() to class QuestionMaker

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | private | String currentCategory() | | { |  |
| - | if | (places[currentPlayer] | == | 〇) | return ’’Pop"; |
| \_ | if | (places[currentPlayer] |  | 4) | return ’’Pop"; |
| - | if | (places[currentPlayer] | == | 8) | return "Pop"; |
| - | if | (places[currentPlayer] | == | 1) | return ’'Science”； |
| - | if | (places[currentPlayer] | == | 5) | return "Science"; |
| - | if | (places[currentPlayer] | == | 9) | return MScience,'; |
| - | if | (places[currentPlayer] |  | 2) | return "Sports"; |
| - | if | (places[currentPlayer] |  | 6) | return "Sports"; |
| - | if | (places[currentPlayer] | == | 10) | return "Sports"; |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| + | if | (players.get(currentPlayer).getPlace() | == | 〇) | return | "Pop"; |
| + | if | (players.get(currentPlayer).getPlace() |  | 4) | return | "Pop"; |
| + | if | (players.get(currentPlayer).getPlace() | == | 8) | return | ••Pop"; |
| + | if | (players.get(currentPlayer).getPlace() | == | 1) | return | "Science” |
| + | if | (players.get(currentPlayer).getPlace() | == | 5) | return | "Science" |
| + | if | (players.get(currentPlayer).getPlace() | == | 9) | return | "Science’1 |
|  | if | (players.get(currentPlayer).getPlace() | == | 2) | return | "Sports"; |
| + | if | (players.get(currentPlayer).getPlace() | == | 6) | return | "Sports"; |

+ if (players.get(currentPlayer).getPlace() == 10) return "Sports";

return "Rock";

}

“运行测试，通过。”

我在看刚刚新写的TODO。与其把currentCategory()方法从Game类转移到Question- Maker 类， 还不如转移到 Player 类更合适。因为如果读一读 currentCategoryO 方法的代码就 不难发现，该方法依赖Player类。如果将其转移到QuestionMaker类，会使QuestionMaker 类依赖Player类，增大了类之间的耦合，违背了高内聚、低耦合的设计原则。而将 currentCategoryO方法转移到IMayer类中就能消除这个耦合。另外，该方法的返回值虽然

是表示问题类别的字符串，但在字符串的层面上不会产生新的类之间的耦合，所以转移到 Player类更合适。

将currentCategory()方法从Game类转移到QuestionMaker类改为转移到Player类，代 码如下（CM: Updated ’TODO: Move method Game.currentCategory() to class QuestionMaker’ to be TODO: Move method Game.currentCategory() to class Player, since this method depends on Player.)：

public class Game {

* // TODO: Move method Game.currentCategory() to class QuestionMaker

+ // TODO: Move method Game . currentCategory () to class Player

private String currentCategory() {

if (players.get(currentPlayer).getPlace() == 0) return "Pop"; if (players.get(currentPlayer).getPlace() == 4) return "Pop";

“在Player类有了支持玩家位置的公共方法后，Game类中获取玩家位置的原有代码就 没用了，可以将其删除。不妨将Game类的places成员变量删除，然后让编译器显示编译错 误，以告诉咱们何处还能继续删除。”

将Game类的places成员变量删除的代码如下（CM: Removed field places from class Game.)：

public class Game {

// TODO-working-on: Move places into class Player

* private int [ ] places = new int[6];

“删除了 places成员变量，在Game类中，就能很容易地看到在currentPlayerMovesToN ewPlaceAndAnswersAQuestion()方法中有4处places显示编译错误的红色。现在来删除这些 原有代码。这样，咱们就已经完成了将玩家位置从Game类移动到Player类的TODO。”

完成将玩家位置从Game类移动到Player类的TODO的代码如下（CM: Removed all places-related code from class Game. Finished TODO: Move places into class Player.)：

public class Game {

* // TODO-working-on: Move places into class Player

private void currentPlayerMovesToNewPlaceAndAnswersAQuestion(int rollingNumber) {

* places[currentPlayer] += rollingNumber;
* if (places[currentPlayer] > 11) places[currentPlayer] -= 12; players.get(currentPlayer).moveForwardSteps(rollingNumber); logger.info(players.get(currentPlayer)

+ \*"s new location is ’’

* + places[currentPlayer]);
* logger.info(players.get(currentPlayer)
* + w' s new location is ’’

+ players.get(currentPlayer).getPlace()); logger .info (’’The category is ’’ + currentCategory ()); askQuestion();

“现在可以处理将currentCategory()方法从Game类移动到Player类的TODO 了，把它 标记为 working-on0”

“接下来就可以把currentCategoryO整个方法从Game类移动到Player类中。这个重构比 较简单，可以使用‘釜底抽薪’的重构方法。”

将currentCategoryO整个方法从Game类移动到Player类中的代码如下（CM: Moved method currentCategoryO from class Game to Player.)：

public class Game {

// TODO-working-on: Move method Game.currentCategory() to class Player private String currentCategoryO {

if (players.get(currentPlayer).getPlace() if (players.get(currentPlayer).getPlace() if (players.get(currentPlayer).getPlace() if (players.get(currentPlayer).getPlace() if (players.get(currentPlayer).getPlace() if (players.get(currentPlayer).getPlace() if (players.get(currentPlayer).getPlace() if (players.get(currentPlayer).getPlace() if (players.get(currentPlayer).getPlace()

|  |  |  |
| --- | --- | --- |
| 〇) | return | "Pop"; |
| 4) | return | "Pop"; |
| 8) | return | "Pop"; |
| 1) | return | "Science"; |
| 5) | return | "Science"; |
| 9) | return | ’’Science” ； |
| 2) | return | "Sports"; |
| 6) | return | "Sports"; |

10) return "Sports";

return "Rock\*’；

}

public class Player {

+ // TODO-working-on: Move method Game.currentCategory{) to class Player

+■ private String currentCategoryO {

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| + | if | (this.place | == | 〇) | return "Pop"; |
| + | if | (this.place | == | 4) | return ’’Pop"; |
| + | if | (this.place | == | 8) | return nPopn; |
| + | if | (this.place | == | 1) | return "Science"; |
| + | if | (this.place | == | 5) | return "Science"; |
| + | if | (this.place | == | 9) | return "Science"; |
| + | if | (this.place | == | 2) | return "Sports"; |
| + | if | (this.place | == | 6) | return "Sports"; |
| + | if | (this.place | == | 10) | return "Sports"; |
| + | return "Rock"; | |  |  |  |

+

“这么一移动，原来在Game类中对currentCategory()方法的调用都变成有编译错误的红 色。下面来处理它们，即在Game类中编写意图代码，以从Player类中获取当前玩家的当前 问题类别。也就是说，Player类应该有一个getCmTentCategory()方法。”

在Game类中编写意图代码从Player类的新接口中获取当前玩家的当前问题类别，代码 如下（CM: Wrote intention code to get current category of the current player from Player in class Game.)：

public class Game {

- logger • info (’’The category is •. + currentCategory ());

+ logger . info ("The category is ’’ + players . get (currentPlayer).

getCurrentCategory());

private void askQuestion() {

if (currentCategory() == "PopM>

if (players.get(currentPlayer).getCurrentCategory() == "Pop") logger.info(questionMaker.removeFirstPopQuestion()); if (currentCategory() == "Science")

if (players.get(currentPlayer).getCurrentCategory() == "Science") logger.info(questionMaker.removeFirstScienceQuestion()); if (currentCategory() == "Sports")

if (players.get(currentPlayer).getCurrentCategory() == "Sports”） logger.info(questionMaker.removeFirstSportsQuestion()); if (currentCategory() == "Rock")

if (players.get(currentPlayer).getCurrentCategory() == "Rock")  
logger.info(questionMaker.removeFirstRockQuestion());

“在上面这些意图代码中，有5处getCurrentCategoryO方法显示的是有编译错误的红色， 因为这个方法还未创建。现在就处理这5处编译错误。只要把移动到Player类中的私有的 currentCategoryO方法改名为getCurrentCategory()并定义为公有的就可以了。运行测试，通 过。这样，将currentCategory()方法从Game类移动到Player类的TODO也完成了〇”

完成有关移动Game类的currentCategoryO方法的TODO的代码如下（CM: Updated the method Player.currentCategory() to be Player.getCurrentCategory(). Finished TODO: Move method Game.currentCategory() to class Player.)：

public class Player {

* // TODO-working-on: Move method Game.currentCategory() to class Player
* private String currentCategory() {

+ public String getCurrentCategory() {

if (this.place == 0) return "Pop"; if (this.place == 4) return "Pop"; if (this.place == 8) return "Pop";

“完成了将currentCategory()方法从Game类移动到Player类的TODO，接下来该处理将 玩家的钱包移动到Player类中的TODO 了。把它标记为working-on。”

“这个TODO的重构也相对比较复杂，使用‘拋砖引玉’的方法。先在Game类中从上 到下用关键词purses查找与玩家钱包相关的原有代码。首先在方法currentPlayerGetsAGold CoinAndSelectNextPlayerO中找到了玩家赢取金币的原有代码，然后在这行代码后面编写使 用Player类实现同样贏取金币功能的意图代码，即Player类应该有一个winAGoldCoin()方 法，表示该玩家贏得了一块金币。”

在 Game 类的 currentPlayerGetsAGoldCoinAndSelectNextPlayer()方法中编写意图代码， 以实现使用Player类的新接口让当前用户贏取金币，代码如下（CM: Wrote intention code to make the current player win a gold coin in method Game.currentPlayerGetsAGoldCoinAndSelectN extPlayer().)：

public class Game {

private boolean currentPlayerGetsAGoldCoinAndSelectNextPlayer() {

logger.info(MAnswer was correct!!!!"); purses[currentPlayer]++;

+ players.get(currentPlayer).winAGoldCoin();

“由于Player类的winAGoldCoin〇方法尚未定义，所以上面的意图代码中winAGoldCoin〇 显示的是编译错误的红色’现在来处理这个错误，即在Player类中创建一个空的winAG〇ldCoin〇 方法。”

在 Player 类中创建空的 winAGoldCoin()方法的代码如下（CM: Created method Player. winAGoldCoin().)：

public class Player {

+ public void winAGoldCoin() {

+

+ }

}

“运行测试，通过。编写完使用Player类的贏取金币功能的意图代码后，接下来在Game 类中再用关键词purses继续査找与玩家钱包相关的代码，又在currentPlayerGetsAGoldCoi nAndSelectNextPlayer()方法中查到了获取金币数量的原有代码，然后在其后添加新的使用 Player类获取金币数量的意图代码，即Player类应该有一个名为countGoldCoins()的公共方 法来获取金币数量。”

在 Game 类的 currentPlayerGetsAGoldCoinAndSelectNextPlayer()方法中编写意图代码， 以实现使用Player类的新接口获取当前玩家的金币数量，代码如下（CM: Wrote intention code to count the gold coins of the current player in method Game.currentPlayerGetsAGoldCoinA

ndSelectNextPlayer().)：

public class Game {

logger.info(players.get(currentPlayer)

+ " now has "

+ purses[currentPlayer ]

+ •• Gold Coins.");

+ logger.info(players.get(currentPlayer)

+ + **n** now has **"**

+ + players.get(currentPlayer).countGoldCoins()

+ + " Gold Coins.M);

“在上面的意图代码中，countGoldCoins()方法由于尚未创建，所以是编译错误的 红色。现在就处理这个错误，即创建并实现Player类的countGoldCoins()方法，另外把 winAGoldCoinO 方法也实现了。，，

创建并实现Player类的countGoldCoins()方法，以及实现winAGoldCoin〇方法，代码如 T ( CM: Created and implemented method Player.countGoldCoins〇. Implemented method Player. winAGoldCoinO.):

public class Player {

private String playerName; private int place =0;

+ private int sumOfGoldCoins = 0;

public void winAGoldCoinO {

+ this.sumOfGoldCoins++;

+ }

+ public int countGoldCoins() {

+ return this.sumOfGoldCoins;

}

}

“运行测试，通过。”

忽然发现，Game类的isGameStilUnProgress()方法中有一个魔法数6,其实就是获胜的 玩家贏取的最大金币数，应该用一个名称富有表达力的公共常量来表示，以便于阅读和统一 修改。目前正在把玩家钱包相关的逻辑移动到Player类中，无暇修改，所以添加一个TODO 记下来以后再改。

在Game类中添加有关魔法数6的TODO，代码如下（CM: Added TODO: The magic number 6.)：

public class Game {

//TODO: The magic number 6

private boolean isGameStilllnProgress() { return !(purses[currentPlayer] == 6);

}

“接着在Game类中用关键词purses查找与玩家钱包相关的原有代码。在Game类的最 后一个方法isGameStillInProgress()中查到了获取当前玩家金币数量的原有代码，这个功能可 以用前面创建的Player类的countGoldCoins()方法来实现。”

在Game类的方法isGameStillInProgress()中使用Player类的新接口获取当前玩家金币 数量，代码如下（CM: Got number of gold coins of current player from Player in method Game. isGameStillInProgress().)：

public class Game {

// TODO: The magic number 6

private boolean isGameStilllnProgress() {

* return !(purses[currentPlayer] == 6);

+ return !(players.get(currentPlayer).countGoldCoins() == 6);

)

}

“现在已经把Game类中与玩家钱包相关的代码都用Player的接口替换完了，该到了删 除原有玩家钱包相关代码的时候了。先从删除Game类的purses成员变量开始。”

删除 Game 类的 purses 成员变量的代码如下（CM: Removed field purses from class Game.):

public class Game {

// TODO-working-on: Move purses into class Player

* private int[] purses = new int[6];

“删除了 Game 类的成员变量 purses 后，在 Game 类的 currentPlayerGetsAGoldCoinAndS electNextPlayer()方法里出现了 2处红色的表示有编译错误的purses。现在就删除它们。运行 测试，通过。这样将玩家钱包相关逻辑移动到Player类中的TODO就完成了。”

完成有关将玩家钱包移动到Player类的TODO的代码如下所示（CM: Removed all purses-related code from class Game. Finished TODO: Move purses into class Player.)：

public class Game {

* // TODO-working-on: Move purses into class Player

private boolean currentPlayerGetsAGoldCoinAndSelectNextPlayer() { logger.info("Answer was correct!!!!");

* purses[currentPlayer]++;

players.get(currentPlayer).winAGoldCoin();

logger.info(players.get(currentPlayer)

+ n now has ’\*

* + purses[currentPlayer]
* + w Gold Coins.w);
* logger.info(players.get(currentPlayer)
* + **n** now has ’’

+ players.get(currentPlayer).countGoldCoins()

+ " Gold Coins.w);

“现在终于要处理从Game类向Player类移动玩家在禁闭室的状态inPenaltyBox这最后 一个成员变量了。先把有关这个移动的TODO标记为working-on。”

“对于这个比较复杂的重构也使用‘拋砖引玉’的方法。在Game类中用关键词 inPenaltyBox进行查找，在Game类的roll()方法中找到了玩家是否在禁闭室、走出禁闭室 和呆在禁闭室这3处应用。可以让Player类具有isInPenaltyBox〇、getOutOfPenaltyBox〇和 stayInPenaltyBox()这3个接口，来分别对应这3处应用。”

在Game类的r〇U〇方法中编写意图代码来处理有关玩家在禁闭室状态的代码如下所示 (CM: Wrote intention code to use the field isInPenaltyBox and isGettingOutOfPenaltyBox in class Player in method Game.rollQ.)：

public class Game {

logger. info (players .get (currentPlayer) + •• is the current player’’）； logger.info("They have rolled a M + rollingNumber);

* if (inPenaltyBox[currentPlayer]) {

+ if (inPenaltyBox[currentPlayer] || players.get(currentPlayer).

isInPenaltyBox()) {

if (rollingNumber % 2 !\* 0) {

isGettingOutOfPenaltyBox = true;

+ players.get(currentPlayer).getOutOfPenaltyBox();

logger.info(players.get(currentPlayer) + " is getting out of the penalty boxM)；

currentPlayerMovesToNewPlaceAndAnswersAQuestion(rollingNumber);

} else {

logger.info(players.get(currentPlayer) + " is not getting out of the penalty box"); isGettingOutOfPenaltyBox = false;

+ players.get(currentPlayer).staylnPenaltyBox();

}

} else {

“上面代码中的原有代码出现在if条件中，则其可以与其功能等价的意图代码相‘或’ 而出现在同一个if条件中，这样就不会在代码运行时影响原有代码的功能。”

“在 Game 类的 roll〇 方法中期望 Player 类所具有的 isInPenaltyBox()、getOutOffenalty-

Box()和stayInPenaltyBox()这3个公共方法都尚未创建，所以显示了编译错误的红色。现在 一个一个地创建它们。先创建和实现Player类的isInPenaltyBox()方法。”

创建和实现Player类的isInPenaltyBox()方法的代码如下所示（CM: Created and imple­mented method Player.isInPenaltyBox。.）：

public class Player {

+ private boolean isInPenaltyBox = false;

+ public boolean isInPenaltyBox() {

+ return this.isInPenaltyBox;

+ }

}

“现在Game类的roll()方法中的isInPenaltyBox()不再是红色的了。接下来创建和实现 Player 类的 getOutOfPenaltyBox()和 stayInPenaltyBox()方法 〇”

创建和实现Player类的getOutOfPenaltyBox()方法的代码如下所示（CM: Created and implemented method Player.getOutOfPenaltyBox().)：

public class Player {

+ private boolean isGettingOutOfPenaltyBox = true;

+ public void getOutOfPenaltyBox(> {

+ this.isGettingOutOfPenaltyBox = true;

+ }

}

创建和实现Player类的stayInPenaltyBox()方法的代码如下所示（CM: Created and imple­mented method Player.stayInPenaltyBox〇.):

public class Player {

+ public void staylnPenaltyBox() {

+ this . isGettingOutOf PenaltyBox =» false;

+ }

}

“运行测试，通过。处理完了 Game类的r〇ll()方法中出现的inPenaltyBox，再接着 在Game类中用关键词inPenaltyBox进行查找。下一个出现inPenaltyBox的方法是was- CorrectlyAnsweredO。还是在这个方法里面‘抛砖引玉’，在原有代码旁边写使用Player类的 新的意图代码接口。这里，Player类还应该有一个名为isGettingOutOfPenaltyBox()的公共方 法，与原有的isGettingOutOfPenaltyBox成员变量实现功能等价。”

在Game类的wasCorrectlyAnswered()方法中编写意图代码来使用Player类的新接口 处理有关玩家在禁闭室状态的代码如下所示（CM: Wrote intention code to know if the current player is getting out of penalty box from Player in method Game.wasCorrectlyAnswered().)：

public class Game {

public boolean wasCorrectlyAnswered() {

* if (inPenaltyBox[currentPlayer]) {
* if (isGettingOutOfPenaltyBox) {

+ if (inPenaltyBox[currentPlayer] || players.get(currentPlayer).

isInPenaltyBox()) {

+ if (isGettingOutOfPenaltyBox 丨丨 players.get(currentPlayer).

isGettingOutOfPenaltyBox()) {

return currentPlayerGetsAGoldCoinAndSelectNextPlayer();

} else {

nextPlayer();

“由于上面的意图代码中的isGettingOutOfPenaltyBox〇方法尚未创建，所以是红色的。 现在就创建和实现这个方法。”

创建并实现Player类的isGettingOutOfPenaltyBox()方法的代码如下所7K (CM: Created and implemented method Player.isGettingOutOfPenaltyBox().)：

public class Player {

+ public boolean isGettingOutOfPenaltyBox() {

+ return this.isGettingOutOfPenaltyBox;

+ }

}

“接着在Game类中用关键词inPenaltyBox继续査找。在最后一个方法wrongAnswer()中 又找到了这个关键词。因为此处是当玩家回答问题错误后就被送入禁闭室，所以在它下面添 加Player类的表现这个意图的代码，不妨起名叫sentToPenaltyBox〇方法。”

在Game类的wrongAnswerO方法中编写意图代码来用Player类的新接口将当前玩家送 入禁闭室的代码如下所示（CM: Wrote intention code to send the current player to penalty box in class Player in method Game.wrongAnswer().)：

public class Game {

public boolean wrongAnswer() {

inPenaltyBox[currentPlayer] = true;

+ players.get(currentPlayer).sentToPenaltyBox();

“这个意图代码sentToPenaltyBoxO还未创建，所以是红色。现在就创建并实现它。”

创建并实现Player类的sentToPenaltyBox()方法的代码如下所示（CM: Created and imple­mented method Player.sentToPenaltyBox().):

public class Player {

+ public void sentToPenaltyBox() {

+ this.isInPenaltyBox = true;

+ }

}

“运行测试，通过。现在玩家在禁闭室的状态的原有代码旁边都编写了使用Player类的 接口的新代码，所以到了删除原有代码的时候了。可以从Game类中删除inPenaltyBox和 isGettingOutOfPenaltyBox 这两个成员变量。”

从Game类中删除inPenaltyBox和isGettingOutOfPenaltyBox这两个成员变量的代码如 下所示（CM: Removed field inPenaltyBox and isGettingOutOfPenaltyBox from class Game.):

public class Game {

// TODO-working-on: Move inPenaltyBox into class Player

* private boolean[] inPenaltyBox = new boolean[6];

private int currentPlayer = 0;

* private boolean isGettingOutOfPenaltyBox;

“删除了这两个成员变量后，在Game类中就出现了不少红色的编译错误。把这些有红 色错误的代码删除，再运行测试，通过。这样，将玩家在禁闭室状态的代码移动到Player类 的TODO也完成了。”

在Game类中删除所有与inPenaltyBox和isGettingOutOfPenaltyBox这两个成员变 量相关的红色编译错误的代码如下所示（CM: Removed all inPenaltyBox- and isGetting- OutOfPenaltyBox-related code from class Game. Finished TODO: Move inPenaltyBox into class Player.)：

public class Game {

* // TODO-working-on: Move inPenaltyBox into class Player
* if (inPenaltyBox[currentPlayer]丨丨 players.get(currentPlayer).

isInPenaltyBox()) {

+ if (players.get(currentPlayer).isInPenaltyBox()) {

if (rollingNumber % 2 != 0) {

* isGettingOutOfPenaltyBox - true;

players.get(currentPlayer).getOutOfPenaltyBox();

logger.info(players.get(currentPlayer) + " is getting out of

the penalty box");

currentPlayerMovesToNewPlaceAndAnswersAQuestion(rollingNumber); } else {

logger.info(players.get(currentPlayer) + " is not getting out of the penalty box"); isGettingOutOfPenaltyBox = false; players.get(currentPlayer).staylnPenaltyBox();

public boolean wasCorrectlyAnswered() {

* if (inPenaltyBox[currentPlayer] || players.get(currentPlayer).

isInPenaltyBox()) {

* if (isGettingOutOfPenaltyBox 丨丨 players.get(currentPlayer).

isGettingOutOfPenaltyBox()) {

+ if (players.get(currentPlayer).isInPenaltyBox()) {

+ if (players.get(currentPlayer).isGettingOutOfPenaltyBox()) {

return currentPlayerGetsAGoldCoinAndSelectNextPlayer();

} else {

nextPlayer 〇;

public boolean wrongAnswer〇 {

logger.info("Question was incorrectly answered")；

logger. info (players. get (currentPlayer) + •’ was sent to the penalty box"); - inPenaltyBox[currentPlayer] = true;

players.get(currentPlayer).sentToPenaltyBox();

nextPlayer();

“至此，将Game类中与玩家的名字、在游戏盘上的位置、钱包中的金币数量和是否在 禁闭室的状态相关的代码移动到Player类的工作基本做完了。不过还是发现了一些其他问题 没有解决。”

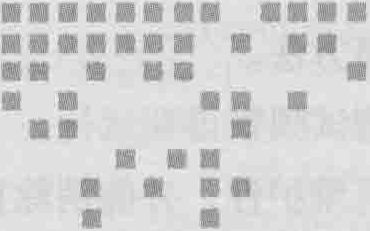
在解决这些问题之前，让我们看看本章又做了什么工作。

1. 使用“抛砖引玉”的方法把相对复杂的玩家位置的计算逻辑移动到Player类中。
2. 临时发现Game类中的currentCategoryO方法，是根据玩家位置计算问题类别的，应 该归QuestionMaker类来负责，添加了一个TODO来记录此问题。

3 )与其把currentCategoryO方法从Game类转移到QuestionMaker类，还不如转移到 Player类更合适，因为这样能更好地实现高内聚、低耦合的设计原则。

4)使用“爸底抽薪”的重构方法把currentCategoryO方法整个从Game类移动到Player 类中。

1. )使用“抛砖引玉”的方法将玩家的钱包移动到Player类中。 •
2. 使用“抛砖引玉”的方法从Game类向Player类移动玩家在禁闭室的状态。
3. 通过操练我们学到了以下技能：
4. 使用+=和-=可以消除重复代码。
5. 应该本着高内聚和低耦合的原则在类之间移动成员方法。
6. 使用“抛砖引玉”的重构方法能够最大限度地让测试频繁得到运行。
7. 将魔法数提取成常量重命名，以增强其可读性、可维护性，并能消除重复代码。
8. 使用“抛砖引玉”的重构方法时，若原有代码出现在if条件中，则其可以与其功能 等价的意图代码做逻辑“或”运算而出现在同一个if条件中，这样就不会在代码运行时影响 原有代码的功能。



第15章

打扫战场

现在咱们这个答题闯关Trivia题目的服务端的类，由原先的1个Game类变为现在的3 个类：Game、QuestionMaker和Player。其中，Game类负责添加玩家、掷色子、处理玩家回 答问题正确和错误的情况这些维护游戏运行的事情；QuestionMaker类负责添加和删除在游 戏中要提出的问题；Player类负责维护玩家的名字、在游戏盘上的位置、钱包中的金币数量 和是否在禁闭室的状态这些与玩家自身相关的事情。现在看起来，这个题目在类的职责划分 上基本没有什么大问题了，大部分代码“腐臭”也已经在前面处理了。在结束这个编程题目 之前，还需要打扫战场，清理一下遗留的代码“腐臭”。

“我刚才想说的没有解决的一个问题，就是在Player类中，我感觉isInPenaltyBox和 isGettingOutOfPenaltyBox这两个成员变量有些冗余，应该可以去掉一个。它们其实就是表本 这个玩家是否在禁闭室这个状态而已，为什么要用两个成员变量来表示这一件事，让人难以 理解呢？”

对，是有些让人难以理解。可以把isGettingOutOfPenaltyBox这个成员变量删除，添加 一个TODO来记录这件事。

在Player类中添加把成员变量isGettingOutOfPenaltyBox删除的TODO的代码如下所ZK (CM: Added TODO: Eliminate field Play.isGettingOutOfPenaltyBox.)：

public class Player {

private int place = 0;

private int sumOfGoldCoins = 0;

private boolean isInPenaltyBox = false;

+ //TODO: Eliminate field Play.isGettingOutOfPenaltyBox

private boolean isGettingOutOfPenaltyBox = true;

咱们可以再回顾一下这几个类，看看还有什么遗漏的代码“腐臭”。

“在Game类的构造器中有两个魔法数，一个是try语句中的10000000,还有一个是for 循环中的50。”

对。另外在roll()方法中，第2个if语句的条件判断(rollingNumber % 2 != 0)不大好读， 不如提取出一个具有良好命名的解释性的变量e来说明这个条件就是“所掷色子点数是奇数” 这件事。

在Game类中添加2个魔法数和1个有关解释性变量的TODO的代码如下所示（CM: Added 3 TODOs for class Game: l)Magic number 10000000; 2)Magic number 50; 3)Introduce explaining variable isRollingNumberOdd.)：

public class Game {

public Game() { try {

+ //TODO: Magic number 10000000

fileHandler = new FileHandler("%h/Game-logging.log", 10000000, 1, true)/

+ //TODO: Magic number 50

for (int i = 0; i < 50; i++) {

questionMaker. addPopQuestion (M Pop Question ’’ + i); questionMaker.addScienceQuestion(("Science Question " + i));

+ //TODO: Introduce explaining variable isRollingNumberOdd

if (rollingNumber % 2 != 0) {

players.get(currentPlayer).getOutOfPenaltyBox();

“在Player类中又有两个魔法数，一个是moveForwardSteps()方法中的12，还有一个是 getCurrentCategory()方法中的 0、4、8、1、5、9、2、6 和 10。”

在Player类中添加2个有关魔法数的TODO的代码如下所示（CM: Added 2 TODOs for class Player: l)Magic number 12; 2)Magic number 0, 4, 8, 1, 5, 9, 2, 6, 10.)：

public class Player {

public void moveForwardSteps(int steps) { this.place += steps;

+ //TODO: Magic number 12

if (this.place > 11) this.place -= 12;

public String getCurrentCategory〇 {

㊀Martin Fowler著，熊节译，《重构：改善既有代码的设计》，人民邮电出版社，2010年4月第1版，第124页

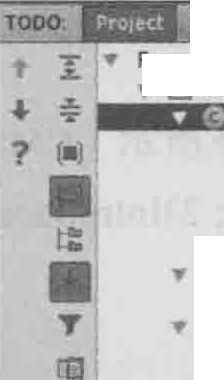
按Alt+6快捷键查看现有的TODO,发现共有9个TODO。

在IDEA中的TODO窗口中显示的9个TODO如图15-1所示。

swerO is unnecessary and should be eliminated

Cuirent Bl« Scop\* S^sed

Found 9 TODO Items In 3 files y Ej kata.trtv»a (9 items tn 3 f--!esl



j] (22. 16) *TODO: Magic number lOOOOOOO*

U (30. 12) *TODO: Magic number 50*

S) (56. 16). *TODO: Introduce explaining variable isRoUingNumberOdd* 圍（133. 12) // rODOJater: Hi© refmn vahie of ❷fhod Game.

6, 10

H TODO: Magic number if (this.place == 0) if (this.place == 4) if (this.place == 8)

0, 4, 8, 1, 5, 9, 2, return "Pop"; return ,fPop"; return "Pop";

Si (137. 8) / TODO: The magic number 6 (jg GameRunner.fava

@(22. 20) ： TOO*〇*4ater: The nsme of the variable notAWinner should be isGsmeStilltnProgress

• Player, java

B(ll. 8) v TODO: Ellmln<ate field Play.isOettlngOutOfPenaltyBox 国（25. 12) // TODO: Magi’c munber 丨 2 m (34.12) if TODO: Magic number 0f 4, 8, 1, % 9, Z 6,10

图15-1在IDEA中的TODO窗口中显示的9个TODO

可以从这9个TODO中挑选简单的先做。那个引入解释性的变量的TODO看起来简单， 可以先做这个。把它标记为working-on。

把if语句的条件提取出来成为一个boolean类型的本地变量就能解决这个不易读的 问题。

完成引入解释性变量isRoUingNumberOdd的TODO的代码如下所示（CM: Finished TODO: Introduce explaining variable isRoUingNumberOdd.)：

public class Game {

if (players.get(currentPlayer).isInPenaltyBox()) {

* // TODO-working-on: Introduce explaining variable isRoUingNumberOdd
* if (rollingNumber % 2 i- 0) {

+ boolean isRoUingNumberOdd = rollingNumber % 2 ! = 0;

+ if (isRoUingNumberOdd) {

players.get(currentPlayer).getCutOfPenaltyBox();

“运行测试，通过。现在有了 iflisRollingNumberOdd)这样的条件语句，可读性就好了很 多。剩下的TODO中，魔法数的也很简单，先处理Game类所有魔法数。”

完成Game类中所有魔法数的TODO的代码如下所示（CM: Finished all magic number TODOs in class Game.)：

public class Game {

+ public static final int NUMBER\_OF\_GOLD\_COINS\_TO\_WON\_AND\_GAME\_OVER = 6;

+ public static final int MAX\_NUMBER\_OF\_BYTES\_WRITING一TO一ONE一FILE = 10000000;

+ public static final int NUMBER一OF\_FILES—TO一USE = 1;

+ public static final int MAX\_NUMBER\_OF\_QUESTIONS = 50;

public Game() { try {

// TODO: Magic number 10000000



fileHandler = new FileHandler("%h/Game-logging.log", 10000000, 1, true); fileHandler = new FileHandler (,,%h/Game-logging. log"

,MAX一NUMBER—OF-BYTES\_WRITING\_TO\_ONE\_FILE ,NUMBER\_OF\_FILES\_TO一USE, true);

// TODO: Magic number 50

for (int i = 0; i < 50; i++) {

for (int i = 0; i < MAX\_NUMBER\_OF\_QUESTIONS; i++) {

- // TODO: The magic number 6

private boolean isGameStilllnProgress() {

一 return !(players.get(currentPlayer).countGoldCoins() == 6);

+ return !(players.get(currentPlayer).countGoldCoins() == NUMBER\_OF\_

GOLD一COINS\_TO\_WON\_AND一GAME—OVER);

}

“运行测试，通过。接下来该处理Player类中的所有魔法数了。”

完成Player类中的所有魔法数的TODO的代码如下所示（CM: Finished all magic number TODOs in class Player.)：

int MAX\_NUMBER\_OF\_PLACE = 12; int CATEGORY一POP\_l = 0; int CATEGORY\_POP\_2 = 4; int CATEGORY一POP\_3 = 8; int CATEGORY一SCIENCE」=1/ int CATEGORY一SCIENCE一2 = 5; int CATEGORY—SCIENCE一3 = 9; int CATEGORY\_SPORTS\_l = 2; int CATEGORY\_SPORTS\_2 = 6; int CATEGORY一SPORTS\_3 = 10;

public class Player {

+ public static final + public static final + public static final + public static final + public static final + public static final + public static final + public static final + public static final + public static final

public void moveForwardSteps(int steps) { this.place += steps;

* //TODO: Magic number 12
* if (this.place > 11) this.place -= 12;

+ if (this.place > MAX\_NUMBER\_OF\_PLACE - 1) this.place -= MAX\_NUMBER\_OF\_PLACE;

public String getCurrentCategory() {

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| - |  | // TODO: Magic number 0, 4, | | | | 8, 1, 5, 9, 2, 6, 10 |
| - |  | if | (this.place | == | 0) return | "Pop"; |
| - |  | if | (this.place | == | 4) return | "Pop"; |
| - |  | if | (this.place | == | 8) return | "Pop"; |
| - |  | if | (this.place | == | 1) return | "Science，，； |
| - |  | if | (this.place |  | 5) return | "Science"; |
| - |  | if | (this.place | == | 9) return | "Science"; |
| - |  | if | (this.place | == | 2) return | "Sports"; |
| - |  | if | (this.place |  | 6) return | "Sports"; |
| - |  | if | (this.place | == | 10) return "Sports"; | |
| + |  | if | (this.place | == | CATEGORY— | P0P\_1) return "Pop"; |
| + |  | if | (this.place | == | CATEGORY一 | P0P\_2) return "Pop"; |
| + |  | if | (this.place | == | CATEGORY一 | POP 3) return "Pop"; |
| + |  | if | (this.place | == | CATEGORY一 | SCIENCE 1) return "Science"; |
| + |  | if | (this.place | == | CATEGORY— | SCIENCE\_2) return "Science"; |
| + |  | if | (this.place |  | CATEGORY一 | SCIENCE\_3) return "Science"; |
| + |  | if | (this.place | == | CATEGORY一 | SPORTS」）return "Sports"; |
| + |  | if | (this.place |  | CATEGORY一 | SPORTS一2) return "Sports"; |
| + | } | if (this.place  return "Rock"; | |  | CATEGORY— | SPORTS一3) return "Sports"; |

“运行测试，通过。接下来该处理Player类的那个冗余的成员变量isGettingOutOf- PenaltyBox 了。把它标记为 working-on。”

要清除成员变量isGettingOutOfPenaltyBox，可以先把使用这个成员变量的地方都替换为 使用另一个成员变量isInPenaltyBox，等没有使用成员变量isGettingOutOfPenaltyBox的地方 后，再清除它。可以先从Player类的getOutOfPenaltyBox()方法开始。

让Player类的getOutOfPenaltyBox()方法使用成员变量isInPenaltyBox的代码如下 所示（CM: Updated the implementation of method Player.getOutOfPenaltyBox() to use field isInPenaltyBox instead.)：

public class Player {

public void getOutOfPenaltyBox() {

- this.isGettingOutOfPenaltyBox = true;

+ this.isInPenaltyBox = false;

}

“运行测试，通过。接下来可以在Game类中把所有与禁闭室相关的代码都换成使用 Player类的isInPenaltyBox成员变量的代码。”

在Game类中把所有与禁闭室相关的代码都换成使用Player类的isInPenaltyBox成员变 量的代码如下所示（CM: Updated all penalty-related code in class Game to use the field Player. isInPenaltyBox.)：

public class Game {

} else {

logger.info(players.get(currentPlayer) + " is not getting out of the penalty box");

- players.get(currentPlayer).staylnPenaltyBox();

+ players.get(currentPlayer).sentToPenaltyBox();

public boolean wasCorrectlyAnswered() {

if (players.get(currentPlayer).isInPenaltyBox()) {

* if (players.get(currentPlayer).isGettingOutOfPenaltyBox()) {
* return currentPlayerGetsAGoldCoinAndSelectNextPlayer();
* } else {
* nextPlayer();
* return true;
* }
* } else {
* return currentPlayerGetsAGoldCoinAndSelectNextPlayer();

+• nextPlayer ();

+ return true;

}

+ return currentPlayerGetsAGoldCoinAndSelectNextPlayer();

}

“运行测试，通过。现在到了删除Player类中与isGettingOutOfPenaltyBox相关的所有代 码的时候了。”

完成删除Player类的成员变量isGettingOutOfPenaltyBox的TODO的代码如下所示 (CM: Removed all isGettingOutOfPenaltyBox-related code from class Player. Finished TODO: Eliminate field Play.isGettingOutOfPenaltyBox.)：

public class Player {

private boolean isInPenaltyBox = false;

// TODO-working-on: Eliminate field Play.isGettingOutOfPenaltyBox private boolean isGettingOutOfPenaltyBox = true;

public void staylnPenaltyBox() {

this.isGettingOutOfPenaltyBox = false;

)

public boolean isGettingOutOfPenaltyBox() { return this.isGettingOutOfPenaltyBox;

public void sentToPenaltyBox() {

this.isInPenaltyBox - true;

}

运行测试，通过。

“Player类还没有进行单元测试呢，添加一个TODO吧。”

在GameTest类中添加有关Player类的单元测试的TODO的代码如下所示（CM: Added TODO: Write tests for class Player.)：

public class GameTest {

+ //TODO: Write tests for class Player }

不过这个有关给Player类添加单元测试的TODO有些大。最好能分解为几个Player类 的用户意图，再针对每个用户意图来编写单元测试。咱们可以先看看Player类的用户意图都 有哪些，然后再写出那些尚未测试的用户意图。

“ Player类主要负责添加玩家和维护玩家在游戏盘上的位置、钱包中的金币数量和是否 在禁闭室的状态。其中添加玩家和维护玩家钱包中的金币数量这两个用户意图已经在前面那 3个有关游戏如何结束的测试中体现出来了。未测试的用户意图就剩下玩家在游戏盘上的位 置和玩家在游戏盘上移动的步数与他要回答的问题类别之间的关系这两个方面。可以针对 这两个方面来添加细化后的TODO。对于玩家在游戏盘上的位置的用户意图，可以写2个 TODO, 一个表示玩家从起点开始前进1步，他所处的位置就是1;另一个表示玩家从起 点开始前进12步，他所处的位置就是0。对于移动步数与问题类别之间的关系，可以写4 个TODO,分别表示玩家移动步数与流行音乐、科学、体育和摇滚音乐这4类问题之间的 关系。”

在GameTest类中将一个大TODO替换为6个小TODO的代码如下所示（CM: Replaced 'TODO: Write tests for class Player' with 6 TODOs for testing place and category of class Player.)：

public class GameTest {

//TODO: Write tests for class Player

//TODO: the place should be 1 if the player moves forward 1 step //TODO: the place should be 0 if the player moves forward 12 steps

//TODO: the category should be Pop if the player moves 12, 4 or 8 steps

//TODO: the category should be Science if the player moves 1, 5 or 9 steps

//TODO: the category should be Sports if the player moves 2, 6 or 10 steps

//TODO: the category should be Rock if the player moves 3, 7 or 11 steps

“可以先处理玩家移动1步的TODO。”

处理玩家移动1步的TODO并编写测试的Assert部分的代码如下所示（CM: Working on TODO: the place should be 1 if the player moves forward 1 step. Wrote an assertion.)：

public class GameTest {

- "TODO: the place should be 1 if the player moves forward 1 step + // TODO-working-on: the place should be 1 if the player moves forward 1 step

+ @Test

+ public void the\_place\_should一be\_l\_if一the\_player\_moves一forward一1一step () {

+ // Assert

+ assertEquals(1, player.getPlace());

+ }

“接着再写这个测试的Act和Arrange部分。运行测试，通过。”

完成玩家移动1步的TODO的代码如下所示（CM: Finished TODO: the place should be 1 if the player moves forward 1 step.)：

public class GameTest {

// TODO-working-on: the place should be 1 if the player moves forward 1 step @Test

public void the一place一should一be\_l\_if\_the\_player\_moves\_forward一1一step〇 {

+ // Arrange

+ Player player - new Player<"Ben’” ；

+

+ //Act

+ player.moveForwardSteps(1);

+

II Assert

assertEquals(1, player.getPlace());

)

“类似地，再处理玩家移动12步的TODO。”

处理玩家移动12步的TODO并编写测试的Assert部分的代码如下所示（CM: Working on TODO: the place should be 0 if the player moves forward 12 steps. Wrote an assertion.)：

public class GameTest {

- //TODO: the place should be 0 if the player moves forward 12 steps

+ // TODO-working-on: the place should be 0 if the player moves forward 12 steps

+ @Test

+ public void the\_place\_should\_be\_0\_if\_the\_player\_moves\_forward\_12\_steps() {

+ // Assert

+ assertEquals(0, player.getPlace());

+ ) 完成玩家移动12步的TODO的代码如下所示（CM: Finished TODO: the place should be 0

if the player moves forward 12 steps.)：

public class GameTest {

- // TODO-working-on: the place should be 0 if the player moves forward 12 steps

@Test

public void the一place\_should一be\_0\_if\_the一player一moves—forward一12\_steps() { + // Arrange

+ Player player = new Player <’’Ben’’};

+

+ //Act

+ player.moveForwardSteps(12);

+

// Assert

assertEquals(0, player.getPlace());

}

“运行测试，通过。接下来该处理玩家移动12、4或8步的时候就会被问到流行音乐的 问题类别的TODO 了。”

处理玩家移动12、4或8步的时候就会被问到流行音乐的问题类别的TODO并编写测试 的 Assert 部分的代码如下所示（CM: Working on TODO: the category should be Pop if the player moves 12, 4 or 8 steps. Wrote an assertion.)：

public class GameTest {

- //TODO: the category should be Pop if the player moves 12, 4 or 8 steps

+ // TODO-working-on: the category should be Pop if the player moves 12, 4 or 8 steps

+ @Test

+ public void the\_category\_should\_be\_Pop\_if\_the\_player\_moves\_12\_or\_4\_or\_8\_steps () {

+ // Assert

+ assertEquals("Pop", player.getCurrentCategory());

+ }

完成玩家移动12、4或8步的时候就会被问到流行音乐的问题类别的TODO的代码如下 所7K (CM: Finished TODO: the category should be Pop if the player moves 12, 4 or 8 steps.):

public class GameTest {

* // TODO-working-on: the category should be Pop if the player moves 12, 4 or 8 steps @Test

public void the\_category\_should\_be\_Pop\_if\_the\_player\_moves\_12\_or\_4\_or\_8\_steps () {

* // Assert

+ // Arrange

+ Player player = new Player("Ben");

+

+ //Act, Assert

+ player.moveForwardSteps(12);

+ assertEquals("Pop", player.getCurrentCategory());

+

+ player.moveForwardSteps(4);

+ assertEquals("Pop", player.getCurrentCategory());

+

+ player.moveForwardSteps(8);

assertEquals("Pop", player.getCurrentCategory());

}

运行测试，通过。

“这个测试有3条assertEquals()语句，每条语句都有Pop这个字符串。换句话说，这个 字符串重复了 3次。可以提取一个变量来消除重复。”

把Pop字符串提取为变量以消除重复的代码如下所示（CM: Extracted variable category in a test to remove duplicate code.)：

public class GameTest {

public void the\_category\_should\_be\_Pop\_if\_the\_player\_moves\_12\_or\_4\_or\_8\_steps () { // Arrange

Player player = new Player ("Ben\*');

+ String category = ’’Pop";

//Act, Assert

player.moveForwardSteps(12);

* assertEquals("Pop", player.getCurrentCategory());

+ assertEquals(category, player.getCurrentCategory());

player.moveForwardSteps(4);

* assertEquals("Pop", player.getCurrentCategory());

+ assertEquals(category, player.getCurrentCategory());

player.moveForwardSteps(8);

* assertEquals("Pop", player.getCurrentCategory());

+ assertEquals(category, player.getCurrentCategory());

}

“运行测试，通过。现在可以类似地处理玩家移动1、5或9步的时候就会被问到科学的 问题类别的TODO 了。但是测试运行却失败了。哦，原来Player类的moveForwardSteps()方 法对于玩家在游戏盘上移动的步数是累积计算的，而不会每次调用时都清0,所以需要修改 有关问题类别的测试。”

处理玩家移动1、5或9步的时候就会被问到科学的问题类别的TODO测试失败的代 码如下所示（CM: Failed a category test since method Player.moveForwardSteps() is cumulative instead of restarted every time. Will change the category tests.)：

public class GameTest {

- // TODO: the category should be Science if the player moves 1, 5 or 9 steps

+ // TODO-working-on: the category should be Science if the player moves 1,

5 or 9 steps + GTest

+ public void the\_category\_should\_be\_Science\_if\_the\_player\_moves\_l\_or\_5\_

or\_9\_steps() {

+ // Arrange

+ Player player = new Player("Ben");

+ String category = "Science";

+

+ //Act, Assert

+ player.moveForwardSteps(1);

+ assertEquals(category, player.getCurrentCategory());

+

+ player.moveForwardSteps(5);

+ assertEquals(category, player.getCurrentCategory());

+

+ player.moveForwardSteps(9);

+ assertEquals(category, player.getCurrentCategory());

+ }

“需要把这4个有关问题类别的用户意图测试中玩家移动的步数换成玩家在游戏盘上所 处的位置。”

对。虽然还是在测试中调用Player类的moveForwardSteps()方法来让玩家在游戏盘上移 动步数，但是需要让步数凑成在游戏盘上的相应的位置。

把4个有关问题类别的用户意图测试中玩家移动的步数换成玩家在游戏盘上所处位置 的代码如下所示（CM: Updated all tests and TODOs about places to use the concept of place. Finished 2 TODOs about Pop and Science category.)：

public class GameTest {

@Test

- public void the\_category\_should\_be\_Pop\_if\_the\_player\_moves\_12\_or\_4\_or\_8\_steps () {

+ public void the\_category\_should\_be\_Pop\_if\_the\_player\_is\_in\_place\_0\_or\_4\_or\_8 () {

H Arrange

Player player = new Player("Ben");

String category - MPopw;

//Act, Assert

player.moveForwardSteps(12);

assertEquals(category, player.getCurrentCategory ());

player.moveForwardSteps(4);

assertEquals(category, player.getCurrentCategory());

* player.moveForwardSteps(8);

+ player.moveForwardSteps(4);

assertEquals(category, player.getCurrentCategory());

>

* // TODO-working-on: the category should be Science if the player moves 1,

5 or 9 steps @Test

* public void the\_category\_should\_be\_Science\_if\_the\_player\_moves\_l\_or\_5\_or\_

9\_steps() {

+ public void the一category\_should\_be一Science一if一the—player一is\_in一place一1一or一 5一or一9 (> {

// Arrange

Player player = new Player("Ben");

String category = "Science";

//Act, Assert

player.moveForwardSteps(1);

assertEquals(category, player.getCurrentCategory());

* player.moveForwardSteps(5);

+ player.moveForwardSteps(4);

assertEquals(category, player.getCurrentCategory());

* player.moveForwardSteps(9);

+ player.moveForwardSteps(4);

assertEquals(category, player.getCurrentCategory());

)

* // TODO: the category should be Sports if the player moves 2, 6 or 10 steps
* // TODO: the category should be Rock if the player moves 3, 7 or 11 steps

+ // TODO: the category should be Sports if the player is in place 2 or 6 or 10

+ // TODO: the category should be Rock if the player is in place 3 or 7 or 11

}

“运行测试，通过。类似地，可以处理有关体育和摇滚音乐的问题类别的TODO。”

完成有关体育和摇滚音乐的问题类别的TODO的代码如下所示（CM: Finished 2 TODOs about Sports and Rock categories.)：

public class GameTest {

* // TODO: the category should be Sports if the player is in place 2 or 6 or 10
* // TODO: the category should be Rock if the player is in place 3 or 7 or 11

+ @Test

+ public void the\_category\_should\_be\_Sports\_if\_the\_\_player\_is\_in\_place\_2\_or\_

6\_or\_10 () {

// Arrange

Player player = new Player("Ben");

String category = ’’Sports’’；

//Act, Assert

player.moveForwardSteps(2);

assertEquals(category, player.getCurrentCategory()); player.moveForwardSteps(4);

assertEquals(category, player.getCurrentCategory()); player.moveForwardSteps(4);

assertEquals(category, player.getCurrentCategory());

+ @Test

+ public void the\_category\_should\_be\_Rock\_if\_the\_j)layer\_is\_in\_place\_3\_or\_7\_or\_ll () {

+ // Arrange

+ Player player = new Player("Ben");

+ String category = "Rock";

+

+ //Act, Assert

+ player.moveForwardSteps(3);

+ assertEquals(category, player.getCurrentCategory(}};

+

+ player.moveForwardSteps(4);

+ assertEquals(category, player.getCurrentCategory());

+

+ player.moveForwardSteps(4);

+ assertEquals(category, player.getCurrentCategory());

+ }

“运行测试，通过。现在除了两个later的TODO之外，所有TODO都处理完了。现在 TODO不像以前那么多了，而且手上也没有正在处理的任务，所以咱们可以发现代码‘腐臭’ 就随时修改，不用再写TODO 了。我注意到有些import语句没有被用到，可以清理一下。”

清理未被用到的import语句的代码如下所示（CM: Cleaned up unused imports.):

import j ava.util.ArrayList;

-import j ava. util. LinkedList; //in file Game, java import java.util.logging.FileHandler;

import static org.junit.Assert.assertFalse;

-import static org.junit.Assert.assertTrue; //in file GameTest.java

运行测试，通过。

“还有什么地方可以重构呢？”

观察一下Game类的构造器，看看还有什么问题。 Game类的构造器代码如下所示：

public Game<) {

try {

fileHandler = new FileHandler(M%h/Game-logging.log"

,MAX\_NUMBER\_OF\_BYTES一WRITING—TO\_ONE\_FILE • NUMBER一OF一FILES\_TO\_USE, true" fileHandler.setFormatter(new SimpleFormatter());

} catch (IOException e) { e.printStackTrace();

}

logger.addHandler(fileHandler);

for (int i = 0; i < MAX一NUMBER一OF一QUESTIONS; i++) {

questionMaker. addPopQuestion ('\*Pop Question ’’ + I); questionMaker. addScienceQuestion ( ("Science Question ’’ + i)); questionMaker.addSportsQuestion(("Sports Question " + i)); questionMaker. addRockQuestion ("Rock Question ’• + i);

“没看出什么问题呀。”

函数中的语句都要在同一抽象层次上。0但是在Game类的构造器中，既有question- Maker.addPopQuestion()这样较高层次的语句，也有logger.addHandler()这样较低层次的语 句。为了便于阅读，可以把这个构造器中有关设置日志文件和准备问题的两段代码提取出来 成为两个方法，使得每个方法中的语句都在同一个抽象层次上。

将Game类的构造器中两段代码提取成两个方法的代码如下所示：

public Game() {

logToAFile(); prepareQuestions();

}

private void logToAFile() {

try {

fileHandler = new FileHandler ("%h/Game-logging. log'\*

,MAX\_NUMBER\_OF\_BYTES\_WRITING\_TO\_ONE\_FILE • NUMBER一0F\_FILES\_T0\_USE, true); fileHandler.setFormatter(new SimpleFormatter());

} catch (IOException e) { e.printStackTrace();

㊀RobertC. Martin著，韩磊译，《代码整洁之道》，人民邮电出版社，2010年1月第1版，第34页

logger.addHandler(fileHandler);

}

private void prepareQuestions() {

for (int i = 0; i < MAX\_NUMBER\_OF\_QUESTIONS; i++) {

questionMaker. addPopQuestion (MPop Question •• + i); questionMaker.addScienceQuestion(("Science Question M + i)); questionMaker. addSportsQuestion ( (\*'Sports Question " + i)); questionMaker.addRockQuestion("Rock Question " + i);

“运行测试，通过。我注意到Game类的私有方法howManyPlayers()没有被使用过，可 以大胆地删除它。写好Commit Message，等以后需要时再从版本管理系统中找出来也不迟。” 在Game类中删除未被使用过的方法howManyPlayers()的代码如下所示（CM: Removed unused method Game.howManyPlayers().)：

public class Game {

* private int howManyPlayers() {
* return players•size(>;

一 )

运行测试，通过。

“Game类的roll()方法有两层嵌套的if-else语句，读起来挺费劲、有没有重构的好 办法？”

Game类的roll〇方法在重构前的代码如下所示：

public void roll(int rollingNumber) {

logger. info (players .get (currentPlayer) + " is the current player\*'); logger.info("They have rolled a " + rollingNumber);

if (players.get(currentPlayer).isInPenaltyBox()) {

boolean isRollingNumberOdd = rollingNumber % 2 != 0; if (isRollingNumberOdd) {

players.get(currentPlayer).getOutOfPenaltyBox();

logger.info(players.get(currentPlayer) + " is getting out of the penalty box");

currentPlayerMovesToNewPlaceAndAnswersAQuestion(rollingNumber);

} else {

logger.info(players.get(currentPlayer) + " is not getting out of the penalty box")；

players.get(currentPlayer).sentToPenaltyBox();

} else {

currentPlayerMovesToNewPlaceAndAnswersAQuestion(rollingNumber);

j

}

“有的。可以使用卫语句（guard clause)来替代嵌套的条件表达式e，即只有一层条件判 断，只要某个条件内部的语句执行完毕，就立即执行return语句返回，退出方法。因为只有 一层条件判断，所以有利于阅读代码。”

Game类的roll〇方法使用卫语句重构后的代码如下所示：

public void roll(int rollingNumber) {

logger.info(players.get(currentPlayer) + " is the current player")； logger.info("They have rolled a " + rollingNumber);

if (!players.get(currentPlayer).isInPenaltyBox()) {

currentPlayerMovesToNewPlaceAndAnswersAQuestion(rollingNumber); return;

}

boolean isRollingNuraberOdd = rollingNumber % 2 != 0; if (isRollingNumberOdd) {

players.get(currentPlayer).getOutOfPenaltyBox();

logger.info(players.get(currentPlayer) + " is getting out of the penalty box’’）；

currentPlayerMovesToNewPlaceAndAnswersAQuestion(rollingNumber); return;

logger.info(players.get(currentPlayer) + " is not getting out of the penalty box");

players.get(currentPlayer).sentToPenaltyBox();

>

“运行测试，通过。Game 类的 wasCorrectlyAnswered()方法中，有一句 ‘return true;’ 语句。这句话在上下文中的含义其实是玩家还被关在禁闭室里，所以这个玩家就不能回答问 题从而贏得金币进而结束游戏，所以返回游戏仍旧可以继续的状态。这么多的内容用一句 ‘return true;’自然表达不清，但可以把true提取成一个解释性的变量，来增强表达性，以利 于阅读。”

在Game类的wasCorrectlyAnswered()方法中提取解释性的变量的代码如下所示（CM: Introduced explaining variable in method Game.wasCorrectlyAnswered().)：

public class Game {

㊀Martin Fowler著，熊节译，《重构：改善既有代码的设计》，人民邮电出版社，2010年4月第1版，第 250页

public boolean wasCorrectlyAnswered() {

if (players.get(currentPlayer).isInPenaltyBox()) {

nextPlayer()/

* return true;

+ boolean theGamelsStillInProgress = true;

+ return theGamelsStilllnProgress;

}

return currentPlayerGetsAGoldCoinAndSelectNextPlayer();

}

“运行测试，通过。在Game类的wasCorrectlyAnswered〇方法中提取了解释性的变量， 那么相应地在wrongAnswer〇方法中返回同样含义的true的地方也需要这样做。”

在Game类的wrongAnswer()方法中提取解释性的变量的代码如下所示（CM: Introduced explaining variable in method Game.wrongAnswer().)：

public class Game {

public boolean wrongAnswer() {

logger.info("Question was incorrectly answered");

logger. info (players. get (currentPlayer) + •’ was sent to the penalty box"); players.get(currentPlayer).sentToPenaltyBox();

nextPlayer();

* return true;

+ boolean theGamelsStilllnProgress = true;

+ return theGamelsStilllnProgress;

}

“运行测试，通过。在Player类的getCurrentCategory()方法中，问题类别的字符串都可 以提取成常量，以消除重复代码。”

在Player类的getCurrentCategoryO方法中提取问题类别字符串作为常量的代码如下所示 (CM: Extracted constants of categories in class Player.)：

public class Player {

+ public static final String POP = "Pop";

+ public static final String SCIENCE = "Science";

+ public static final String SPORTS = "Sports”；

+ public static final String ROCK = "Rock";

public String getCurrentCategory() {

if (this.place == CATEG0RY\_P0P\_1) return "Pop"; if (this.place == CATEG0RY\_P0P\_2) return "Pop"; if (this.place == CATEG0RY\_P0P\_3) return "Pop"; if (this.place == CATEG0RY\_SCIENCE\_1) return "Science"; if (this.place == CATEGORY一SCIENCE一2) return "Science";

if (this.place == CATEGORY一SCIENCE一3) return ’.Science"; if (this.place \*= CATEGORY一SPORTS」> return "Sports"; if (this.place \*= CATEG0RY\_SP0RTS\_2) return "Sports"; if (this.place \*= CATEGORY\_SPORTS\_3) return "Sports"; return "Rock";

if (this.place == CATEG0RY\_P0P\_1) return POP; if (this.place \*= CATEG0RY\_P0P\_2) return POP; if (this.place == CATEGORY—POP一3) return POP; if (this.place == CATEGORY\_SCIENCE一1) return SCIENCE; if (this.place == CATEG0RY\_SCIENCE\_2) return SCIENCE; if (this.place == CATEG0RY\_SCIENCE\_3) return SCIENCE; if (this.place »= CATEG0RY\_SP0RTS\_1) return SPORTS; if (this.place \*= CATEG0RY\_SP0RTS\_2) return SPORTS; if (this.place == CATEGORY一SP0RTS\_3> return SPORTS; return ROCK;

“运行测试，通过。现在在GameTest这个测试类中，共有13个测试，也算一个大类了。 我看可以根据这13个测试相互之间的相关性，提取一些相关的测试到新创建的小一些的测 试类中，以消除过大的类这样的代码‘腐臭’。”

将GameTest类中与问题相关的测试提取到QuestionMakerTest测试类中的代码如下所示 (CM: Created test class QuestionMakerTest and moved all question-related tests into it.)：

+public class QuestionMakerTest {

+ @Test

+ public void add一two\_pop一questions一and\_could一remove一the一first\_one<) {

+ )

+

+ @Test

+ public void add\_two\_science\_questions\_and\_could\_remove\_the\_first\_one() {



+ @Test

+ public void add\_two\_\_sports\_questions\_and\_could\_remove\_the\_first\_one() {

+ }

+

+ @Test

+ public void add\_two\_rock\_questions\_and\_could\_remove\_the\_first\_one() {

“由于现在有了两个测试类，所以可以按Alt+l快捷键把光标定位在Project窗口的test/

java/kata.trivia之上，然后按Ctrl+Shift+FlO组合键来运行测试，通过。类似地，可以把 GameTest类中与玩家在游戏盘上的位置和问题类别相关的测试都提取到新创建的PlayerTest 测试类中。”

把GameTest类中与玩家在游戏盘上的位置和问题类别相关的测试都提取到新创建的 PlayerTest 测试类中的代码如下所示（CM: Created test class PlayerTest and moved all place-and category-related tests into it.)：

+public class PlayerTest {

+ @Test

+ public void the\_place\_should\_be\_l\_if\_the\_player\_moves\_forward\_l\_step() {

+

+ @Test

+ public void the\_place\_should\_be\_0\_if\_the\_player\_moves\_forward\_12\_steps() {

+ }

+

+ @Test

+ public void the一category一should一be\_Pop一if\_the\_player\_is一in\_place\_0\_or一4\_or一8 () {

+ @Test

+ public void the一category一should\_be\_Science\_if\_the\_player—is\_in一place\_l\_or

5—or\_9() {

+ }

+

+ @Test

+ public void the\_category一should\_be\_Sports\_if\_the一player一is\_in—place\_2\_or 6\_or\_10() {

+ }

+

+ @Test

+ public void the\_category\_should\_be\_Rock\_if\_the\_player\_is\_in\_place\_3\_or\_7\_or\_ll () {

+ }

+ }

“运行测试，通过。可以把GameTest、PlayerTest和QuestionMakerTest这3个测试类中 的Arrange部分的代码都提取到@Bef〇re标注的方法中，以消除重复代码。”

在GameTest类中将Arrange部分提取到@Before标注的方法后的代码如下所示（CM: Moved initialization code to @Before in test class GameTest.)：

public class GameTest {

private Game game = null;

private boolean isGameSti11InProgress = true; @Before

public void initialize() {

// Arrange

game = new Game();

game.add("Chet"> ;

isGameStillInProgress = true;

@Test

public void the一game一should\_be一over\_if\_a\_player一rolls一the\_dice\_and\_answers each一question—correctly一for\_6\_times <) {

//Act

for (int i = 0; i < 6; i++) {

game.roll (1);

isGameStillInProgress = game.wasCorrectlyAnswered()/

// Assert

assertFalse(isGameStilllnProgress);

@Test

public void the\_game\_should\_be\_over\_if\_a\_player\_rolls\_the\_dice\_for\_7\_times\_ and一answers\_the\_question\_wrongly一for—l\_time一foll〇wed\_by\_an\_odd 一rolling一 number\_but\_then\_correctly\_for\_6\_times() {

@Test

public void the—game一should一be\_over一if—a\_player\_rolls\_the\_dice\_for\_8\_times\_ and一answers一the一question\_wrongly\_for\_l\_time\_followed\_by—an\_even\_rolling\_ number\_but\_then\_correctly\_for\_7\_times\_with\_odd\_rolling\_numbers() {

在PlayerTest类中将Arrange部分提取到@Before标注的方法后的代码略（CM: Moved initialization code to @Before in test class PlayerTest.) 〇

在QuestionMakerTest类中将Arrange部分提取到@Before标注的方法后的代码略（CM: Moved initialization code to @Before in test class QuestionMakerTest.) 〇

“运行测试，通过。在QuestionMakerTest类中，第1个问题的字符串在添加问题和断言 判断时分别出现了一次，形成了重复代码。可以将其提取成常量来消除重复代码。”

在QuestionMakerTest类中提取常量以消除重复的代码如下所示（CM: Extracted constants in test class QuestionMakerTest to eliminate duplicate code.)：

public class QuestionMakerTest {

+ public static final String POP一QUESTION」="Pop Question 1";

+ public static final String SCIENCE一QUESTION」="Science Question 1";

+ public static final String SPORTS\_QUESTION\_l = "Sports Question 1";

+ public static final String R0CK\_QUESTI0N\_1 = "Rock Question 1";

@Test

public void add\_two\_pop\_questions\_and\_could\_remove\_the\_f irst\_one () {

II Act

* questionMaker.addPopQuestion('\*Pop Question 1’’};

+ questionMaker.addPopQuestion(P0P\_QUESTI0N\_1);

questionMaker.addPopQuestion("Pop Question 2");

// Assert

* assertEquals("Pop Question 1", questionMaker.removeFirstPopQuestion());

+ assertEquals(POP一QUESTION」， questionMaker.removeFirstPopQuestion());

“运行测试，通过。仔细观察一下Game类的构造器的第2条准备提问的语句 ‘ prepareQuestions();’。为什么准备提问的语句会放到Game类的构造器里面，而不是放 在QuestionMaker类的构造器里面呢？可以把准备提问的语句从Game类的构造器中移动到 QuestionMaker类的构造器中〇”

把提问的语句从Game类的构造器中移动到QuestionMaker类的构造器中的代码如下所 ^ (CM: Moved question preparation code from class Game to QuestionMaker.)：

public class Game {

public static final int MAX\_NUMBER\_OF\_QUESTIONS = 50;

public Game() {

logToAFile()； prepareQuestions();

private void prepareQuestions() {

for (int i = 0; i < MAX\_NUMBER一OF一QUESTIONS; i++) {

questionMaker.addPopQuestion("Pop Question " + i); questionMaker.addScienceQuestion(("Science Question " + i)); questionMaker.addSportsQuestion(("Sports Question " + i)); questionMaker.addRockQuestion("Rock Question " + i)/

public class QuestionMaker {

+ public static final int MAX\_NUMBER\_OF\_QUESTIONS = 50;

public QuestionMaker() {

for (int i = 0; i < MAX一NUMBER—OF一QUESTIONS; i++) { addPopQuestion("Pop Question " + i); addScienceQuestion(("Science Question " + i)); addSportsQuestion ( ("Sports Question •• + i)); addRockQuestion ("Rock Question •• + i);

“运行测试，失败！失败的原因是刚才修改了 QuestionMaker类的构造器，使得 QuestionMaker类的对象一旦创建就会在4类问题中的每一类问题中添加50个问题。基于这 种情况，需要修改一下QuestionMakerTest测试。因为问题在QuestionMaker对象创建时都已 经添加好了，所以在测试中就不再添加问题，而只要测试能正确删除第1个问题就可以了。” 修改QuestionMaker类的测试使得不再添加问题的代码如下所示（CM: Updated test class QuestionMakerTest to adapt the fact that the questions had been added by its constructor.)：

public class QuestionMakerTest {

private QuestionMaker questionMaker = null;

^Before

public void initialize() {

// Arrange

questionMaker = new QuestionMaker();

@Test

public void the\_first\_pop\_question\_added\_by\_constructor\_could\_be\_removed() {

//Act, Assert

assertEquals("Pop Question 0", questionMaker.removeFirstPopQuestion());

@Test

public void the\_first\_science一question一added一by\_constructor一could\_be\_removed(> { //Act, Assert

assertEquals("Science Question 0", questionMaker.removeFirstScienceQuestion());

} @Test

public void the一first\_sports\_question\_adcied\_by一construct〇r\_could\_be—removed () {

//Act, Assert

assertEquals("Sports Question 0", questionMaker.removeFirstSportsQuestion());

@Test

public void the\_first\_rock\_question\_added\_by\_constructor\_could\_be\_removed 〇 { //Act, Assert

assertEquals("Rock Question 0", questionMaker.removeFirstRockQuestion());

}

}

“运行测试，通过。在 Game 类中，wasCorrectlyAnswered()和 wrongAnswer()这两个方 法虽然在功能上属于同一层次，但二者的命名风格却不一致。如果将前者命名为answered- Correctly(),后者命名为answeredWrongO似乎更好一些。因为本次编程操练不会修改客户端 代码，而这两个方法又被客户端所调用，所以二者的命名不会被修改。即使是这样，我还是 愿意添加两个later的TODO,来记录此事，待以后条件成熟时再修改。”

在Game类中添加两个有关命名的later的TODO如下所示（CM: Added 2 TODO-laters for suggestions on the names of interface of class Game.)：

public class Game {

+ // TODO-later: The name of method Game.wasCorrectlyAnswered() should be

Game.answeredCorrectly() public boolean wasCorrectlyAnswered() {

+ // TODO-later: The name of method Game.wrongAnswer() should be Game.

answeredWrong()

public boolean wrongAnswer() {

“好了，终于要实现咱们在做这个编程题目最开始时所描述的那个新特性了：正在被关 禁闭的玩家，若掷出了除4之外的任何点数，都会被从禁闭室里释放出来；若掷出了 4,则 还要继续被关禁闭。”

“因为新特性要求被关禁闭的玩家从禁闭室被释放出来的条件是掷出了非4的色子，这 与原来掷出奇数的色子的条件不一致，所以要更新原来的测试，来与新特性保持一致。”

在GameTest类中更新两个含有玩家从禁闭室释放出来的测试来与新特性保持一致的代 码如下所示（CM: Updated two tests according to the new feature TODO.):

public class GameTest {

@Test

- public void the\_game一should一be\_over\_if\_a\_player\_rolls一the\_dice一for\_7\_

times\_and\_answers\_the\_question\_wrongly\_for\_l\_time\_followed\_by\_an\_odd rolling一number\_but\_then\_correctly\_for\_6\_times <) {

+ public void the一game\_should—be\_over—if\_a\_player一rolls\_the—dice一for一7\_ times\_and\_answers\_the\_question\_wrongly\_for\_l\_time\_followed\_by\_a\_

rolling\_number\_which\_is\_not\_4\_but\_then\_correctly\_for\_6\_times() {

//Act

game.roll(1); game.wrongAnswer();

- game.roll(1);

+ game.roll(6);

game.wasCorrectlyAnswered(); for (int i = 0; i < 5; i++) { game.roll(1);

@Test

* public void the\_game\_should\_be\_over\_if\_a\_player\_rolls\_the\_dice\_for\_8\_

times\_and\_answers\_the\_question\_wrongly\_for\_l\_time\_followed\_by\_an\_even\_ rolling\_mimber\_but\_then\_correctly\_for\_7\_times\_with\_odd\_rolling\_nuinbers () { + public void the一game—should一be一over\_if\_a\_player一rolls一the—dice\_for—8\_times\_ and\_answers\_the\_question\_wrongly\_for\_l\_tirae\_followed\_by\_a\_rolling\_ number\_which\_\_is\_4\_but\_then\_correctly\_for\_\_7\_times\_with\_odd\_rolling\_numbers () { //Act

game.roll(1); game.wrongAnswer();

* game.roll(2);

+ game.roll(4);

game.wasCorrectlyAnswered(); for (int i = 0; i < 6; i++) { game.roll(1);

+ // TODO-new-feature-working-on: The player will not be getting out of the

penalty box when the rolling number is 4

}

“运行测试，只有那个掷了 7次色子其中有一次被关禁闭之后又掷出非4色子的测试失 败，其他测试运行通过。其原因是刚刚只根据新特性更新了测试代码，还未更新生产代码。 刚刚更新的两个测试中，其中一个根据新特性编写的测试的数据恰恰也满足原有的逻辑，所 以测试运行碰巧通过。要让那个失败的测试运行通过，只要在Game类的roll()方法中，把 原先计算所掷点数是奇数的地方，换成非4就可以了。”

在Game类的r〇ll()方法中，把原先计算所掷点数是奇数的地方换成非4的代码如下所 示（CM: Finished the new feature TODO.):

public class Game {

public void roll(int rollingNumber) {

* boolean isRollingNumberOdd = rollingNumber % 2 != 0;
* if (isRollingNumberOdd) {

+• boolean isRollingNumberForGettingOutOfPenaltyBox = rollingNumber != 4;

+ if (isRollingNumberForGettingOutOfPenaltyBox) {

players.get(currentPlayer).getOutOfPenaltyBox();

logger.info(players.get(currentPlayer) + " is getting out of the penalty box"};

“运行测试，通过。新特性完成了，咱们这个编程题目也将告一段落了。”

为了体会一下重构的成果，不妨把重构后的测试代码和各个生产代码的类完整地列出 来，与重构前列出的Game类的代码作一对照。

GameTest测试类的代码如下所示：

public class GameTest {

private Game game = null;

private boolean isGameStillInProgress = true;

@Before

public void initialize () {

// Arrange

game = new Game();

game.add("Chet")/

isGaraeStilllnProgress = true;

}

@Test

public void the\_game\_should\_be\_over\_if\_a\_player\_rolls\_the\_dice\_and\_answers\_ each一question一correctly\_for\_6\_times 〇 {

//Act

for (int i = 0; i < 6; i++) { game.roll(1);

isGameStillInProgress = game.wasCorrectlyAnswered();

// Assert

assertFalse(isGameStilllnProgress);

@Test

public void the\_game一should\_be一over\_if一a\_player一rolls一the一dice一for一7一times and\_answers\_the\_question\_wrongly\_for\_l\_time\_followed\_by\_a\_rolling\_ number\_which\_is\_not\_4\_but\_then\_correctly\_for\_6\_times() {

//Act

game.roll (1); game.wrongAnswer(); game.roll(6);

game.wasCorrectlyAnswered();

for (int i » 0; i < 5; i++) { game.roll(1);

isGameStillInProgress = game.wasCorrectlyAnswered ();

// Assert

assertFalse(isGameStilllnProgress);

@Test

public void the\_game\_should\_be\_over\_if\_a\_player\_rolls\_the\_dice\_for\_8\_times\_ and\_answers\_the\_question\_wrongly\_for\_l\_time\_\_followed\_by\_\_a\_rolling\_number which一is\_4\_but一then\_correctly\_for一7一times一with一odd\_rolling\_numbers () {

//Act

game.roll(1); game.wrongAnswer(); game.roll(4)/

game.wasCorrectlyAnswered();

. for (int i = 0; i < 6; i++) {

game.roll(1);

isGameStilllnProgress = game.wasCorrectlyAnswered();

} // Assert

assertFalse(isGameStilllnProgress);

PlayerTest测试类的代码如下所示:

public class PlayerTest {

private Player player = null; private String category = null;

@Before

public void initialize() {

// Arrange

player = new Player("Ben">;

@Test

public void the\_place一should—be一1一if\_the一player\_moves\_forward\_l\_step(> { //Act

player.moveForwardSteps(1);

// Assert

assertEquals(1, player.getPlace());

@Test

public void the\_place\_should\_be\_0\_if\_the\_player\_moves\_forward\_12\_steps() {

"Act

player.moveForwardSteps(12);

// Assert

assertEquals(0, player.getPlace());

}

@Test

public void the\_category\_should\_be\_Pop\_if\_the\_player\_is\_in\_place\_0\_or\_4\_or\_8 () { // Arrange category = "Pop";

//Act, Assert

player.moveForwardSteps(12);

assertEquals(category, player.getCurrentCategory()); player.moveForwardSteps(4);

assertEquals(category, player.getCurrentCategory()); player.moveForwardSteps(4);

assertEquals(category, player.getCurrentCategory());

)

@Test

public void the一category一should\_be\_Science\_if\_the\_player\_is\_in一place\_l\_or一 5-Oir一9〇 {

// Arrange

category = "Science";

//Act, Assert

player.moveForwardSteps(1);

assertEquals(category, player.getCurrentCategory());

player.moveForwardSteps(4);

assertEquals(category, player.getCurrentCategory()); player.moveForwardSteps(4);

assertEquals(category, player.getCurrentCategory());

}

@Test

public void the\_category\_should\_be\_Sports\_if\_the\_player\_is\_injplace\_2\_or\_6\_or\_10 () { // Arrange category = "Sports";

//Act, Assert

player.moveForwardSteps(2);

assertEquals(category, player.getCurrentCategory());

player.moveForwardSteps(4);

assertEquals(category, player.getCurrentCategory()); player.moveForwardSteps(4);

assertEquals(category, player.getCurrentCategory());

}

@Test

public void the\_category\_should\_be\_Rock\_if\_the\_\_player\_is\_injplace\_3\_or\_7\_or\_ll () { // Arrange category = "Rock";

//Act, Assert

player.moveForwardSteps(3);

assertEquals(category, player.getCurrentCategory()); player.moveForwardSteps(4);

assertEquals(category, player.getCurrentCategory()); player.moveForwardSteps(4);

assertEquals(category, player.getCurrentCategory());

QuestionMakerTest测试类的代码如下所7K:

public class QuestionMakerTest {

private QuestionMaker questionMaker = null;

@Before

public void initialize() {

// Arrange

questionMaker = new QuestionMaker();

}

@Test

public void the一 first\_pop\_question\_added一by一constructor一could\_be一removed() {

//Act, Assert

assertEquals ("Pop Question 0•’，questionMaker. removeFirstPopQuestion ());

@Test

public void the\_first一science一question一added\_by一constructor\_could\_be\_removed() { //Act, Assert

assertEquals("Science Question 0", questionMaker.removeFirstScienceQuestion());

@Test

public void the\_first\_sports\_question\_added\_by\_constructor\_could\_be\_removed () { //Act, Assert

assertEquals ("Sports Question 0•’，questionMaker.removeFirstSportsQuestion ());

@Test

public void the\_first\_rock\_question\_added\_by\_constructor\_could\_be\_removed() { //Act, Assert

assertEquals("Rock Question 0", questionMaker.removeFirstRockQuestion());

Game类的代码如下所示：

public class Game {

public static final int NUMBER\_OF\_GOLD\_COINS\_TO\_WON\_AND\_GAME\_OVER = 6; public static final int MAX一NUMBER一OF\_BYTES\_WRITING\_TO\_ONE一FILE = 10000000; public static final int NUMBER—OF一FILES一TO—USE = 1; private final QuestionMaker questionMaker = new QuestionMaker();

private ArrayList<Player> players = new ArrayList<Player>();

private int currentPlayer = 0;

private static Logger logger = Logger.getLogger(Mkata.trivia.Game")； private static FileHandler fileHandler = null;

public Game() { logToAFile();

private void logToAFileO { try {

fileHandler = new FileHandler(M%h/Game-logging.log" ,MAX\_NUMBER一OF\_BYTES\_WRITING\_TO\_ONE一FILE ,NUMBER\_OF-FILES\_TO一USE, true); fileHandler.setFormatter(new SimpleFormatter());

} catch (IOException e) { e.printStackTrace();

logger.addHandler(fileHandler);

public void add(String playerName) {

players.add(new Player(playerName))/ logger.info(playerName + " was added");

logger.info("The total amount of players is " + players.size());

}

public void roll<int rollingNumber) {

logger. info (players. get (currentPlayer) + ’’ is the current player"); logger.info("They have rolled a n + rollingNumber);

if (!players.get(currentPlayer).isInPenaltyBox()) {

currentPlayerMovesToNewPlaceAndAnswersAQuestion(rollingNumber); return;

}

boolean isRollingNumberForGettingOutOfPenaltyBox = rollingNumber != 4; if (isRollingNumberForGettingOutOfPenaltyBox) {

players.get(currentPlayer).getOutOfPenaltyBox();

logger. info (players. get (currentPlayer) + •’ is getting out of the penalty box");

currentPlayerMovesToNewPlaceAndAnswersAQuestion(rollingNumber); return;

}

logger.info(players.get(currentPlayer) + " is not getting out of the penalty box"

players.get(currentPlayer).sentToPenaltyBox();

}

private void currentPlayerMovesToNewPlaceAndAnswersAQuestion(int rollingNumber) { players.get(currentPlayer).moveForwardSteps(rollingNumber);

logger.info(players.get(currentPlayer)

+ n's new location is "

+ players.get(currentPlayer).getPlace()); logger•inf〇("The category is "+ players.get(currentPlayer).getCurrentCategory〇}; askQuestion();

}

private void askQuestion() {

if (players.get(currentPlayer).getCurrentCategory() == "Pop”） logger.info(questionMaker.removeFirstPopQuestion()); if (players.get(currentPlayer).getCurrentCategory() == "Science")

logger.info(questionMaker.removeFirstScienceQuestion{)); if (players.get(currentPlayer).getCurrentCategory() == "Sports") logger.info(questionMaker.removeFirstSportsQuestion()); if (players.get(currentPlayer).getCurrentCategory() \*= "Rock") logger.info(questionMaker.removeFirstRockQuestion());

}

// TODO-later: The name of method Game.wasCorrectlyAnswered() should be Game.answeredCorrectly() public boolean wasCorrectlyAnswered() {

if (players.get(currentPlayer).isInPenaltyBox()) { nextPlayer();

boolean theGamelsStillInProgress = true; return theGamelsStilllnProgress;

}

return currentPlayerGetsAGoldCoinAndSelectNextPlayer();

)

private boolean currentPlayerGetsAGoldCoinAndSelectNextPlayer() { logger.info("Answer was correct!!!!"); players.get(currentPlayer).winAGoldCoin();

logger.info(players.get(currentPlayer)

+ " now has "

+ players.get(currentPlayer).countGoldCoins()

+ " Gold Coins.");

boolean isGameStilllnProgress = isGameStilllnProgress()/ nextPlayer();

return isGameStilllnProgress;

)

private void nextPlayer() { currentPlayer++;

if (currentPlayer == players.size()) currentPlayer = 0;

}

// TODO-later: The name of method Game.wrongAnswer() should be Game.answeredWrong() public boolean wrongAnswer() {

logger.info("Question was incorrectly answered");

logger.info(players.get(currentPlayer) + " was sent to the penalty box"); players.get(currentPlayer).sentToPenaltyBox();

nextPlayer(>;

boolean theGamelsStillInProgress = true; return theGamelsStilllnProgress;

private boolean isGameStilllnProgress() {

return i(players.get(currentPlayer).countGoldCoins〇 == NUMBER一OF\_GOLD COINS\_TO WON AND\_GAME OVER);

IMayer类的代码如下所示：

public class Player {

public static final int MAX\_NUMBER\_\_OF\_PLACE = 12; public static final int CATEGORY\_POP\_l = 0; public static final int CATEGORY一POP\_2 = 4; public static final int CATEGORY—POP\_3 = 8/ public static final int CATEGORY—SCIENCE一1 = 1; public static final int CATEGORY\_SCIENCE\_2 = 5; public static final int CATEGORY\_SCIENCE\_3 = 9; public static final int CATEGORY\_SPORTS\_\_l = 2; public static final int CATEGORY\_SPORTS\_2 = 6; public static final int CATEGORY\_SPORTS\_3 = 10; public static final String POP = "Pop"; public static final String SCIENCE = "Science"; public static final String SPORTS = "Sports”； public static final String ROCK >= "Rock"; private String playerName; private int place = 0; private int sumOfGoldCoins = 0; private boolean isInPenaltyBox = false;

public Player(String playerName) { this.playerName = playerName;

}

©Override

public String toString() { return this.playerName;

public void moveForwardSteps(int steps) { this.place += steps;

if (this.place > MAX\_NUMBER\_OF\_PLACE - 1) this.place -= MAX\_NUMBER\_OF\_PLACE;

}

public int getPlace() { return this.place;

==CATEG0RY\_P0P\_1) return POP;

==CATEG0RY\_P0P\_2) return POP;

if

if

if

if

if

if

if

if

if

(this.place (this.place (this.place (this.place (this.place (this.place (this.place (this.place (this.place

==CATEG0RY\_P0P\_3) return POP;

==CATEGORY一SCIENCE」）return SCIENCE; ==CATEG0RY\_SCIENCE\_2) return SCIENCE; ==CATEGORY\_SCIENCE\_3) return SCIENCE; ==CATEGORY—SPORTS一1> return SPORTS;

==CATEGORY一SPORTS一2) return SPORTS;

==CATEGORY一SPORTS一3) return SPORTS;

return ROCK;

public void winAGoldCoin() { this.sumOfGoldCoins++;

public int countGoldCoins() {

return this.sumOfGoldCoins;

public boolean isInPenaltyBox() { return this.isInPenaltyBox;

public void getOutOfPenaltyBox() { this.isInPenaltyBox = false;

public void sentToPenaltyBox() { this.isInPenaltyBox = true;

QuestionMaker类的代码如下所示：

public class QuestionMaker {

public static final int MAX\_NUMBER\_OF\_QUESTIONS = 50; private LinkedList<String> popQuestions = new LinkedList<String>()/ private LinkedList<String> scienceQuestions = new LinkedList<String>()/ private LinkedList<String> sportsQuestions = new LinkedList<String>(); private LinkedList<String> rockQuestions = new LinkedList<String>();

public QuestionMaker() {

for (int i = 0; i < MAX\_NUMBER\_OF\_QUESTIONS; i++) {

addPopQuestion(MPop Question " + i); addScienceQuestion(("Science Question " + i)); addSportsQuestion(("Sports Question " + i)); addRockQuestion("Rock Question " + i);

public void addPopQuestion(String popQuestion) { popQuestions.add(popQuestion);

public void addScienceQuestion(String scienceQuestion) { scienceQuestions.add(scienceQuestion);

}

public void addSportsQuestion(String sportsQuestion) { sportsQuestions.add(sportsQuestion);

public void addRockQuestion(String rockQuestion) { rockQuestions.add(rockQuestion);

public String removeFirstPopQuestion() { return popQuestions.removeFirst();

public String removeFirstScienceQuestion() { return scienceQuestions.removeFirst();

}

public String removeFirstSportsQuestion() { return sportsQuestions.removeFirst();

}

public String removeFirstRockQuestion() { return rockQuestions.removeFirst();

驯服完了 Trivia烂代码，接下来还要操练一下用Mode来为已有代码编写单元测试。不 过在继续操练之前，让我们看一看本章都做了哪些工作。

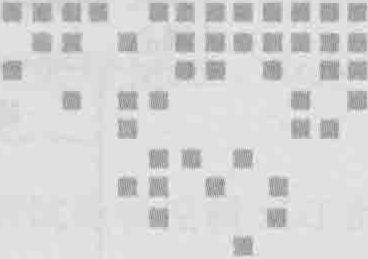
1. 在类的职责划分上基本没有什么大问题的前提下，阅读代码，发现遗留的代码“腐 臭”并用TODO记录下来，以便解决。
2. 用了提取出具有良好命名的解释性的变量的方法解决条件判断语句中条件不易读的 问题。
3. 消除了一些魔法数。
4. 在 Player 类中，清除了成员变量 isGettingOutOfPenaltyBox,消除了原先在 isInPenalty- Box和isGettingOutOfPenaltyBox这两个成员变量之间存在的逻辑上的重复。

5 )为Player类编写用户意图测试。

1. 用提取方法的办法解决了 Game类的构造器中语句不在同一抽象层次上的问题。
2. 使用了卫语句解决了嵌套的条件表达式难以阅读的问题。
3. )将GameTest这个包含全部13个测试的大类，按照测试彼此的相关性，分解出Player- Test 和 QuestionMakerTest 这两 个测试小类。
4. )实现了 Trivia题目的新特性。

10)通过操练我们学到了以下技能：

1. 用要保留下来的成员变量逐步取代要被替换的成员变量，最后再清除后者的定义的做 法，来消除两个成员变量之间在逻辑上的重复。
2. 将不易读的条件判断或返回值提取、命名成易读的解释性变量。
3. 当对生产代码的行为理解不准时，编写出的测试会运行失败，此时需要根据用户意图 和生产代码的实际行为调整测试代码，来达到二者的平衡。
4. 当TODO不是很多且手上也没有正在处理的任务时，可以一发现代码“腐臭”就随 时修改，不必再写TODO。
5. 确保函数中的语句都处于同一抽象层次上。否则就使用提取方法的做法，来确保提取 出来的每个方法中的语句都在同一个抽象层次上。
6. 对于未被使用的方法，可以大胆删除，写好Commit Message,等以后需要时再从版 本管理系统中找出来也不迟。
7. 可以使用卫语句来替代嵌套的条件表达式。
8. 将多次使用的字符串提取成常量以增强可维护性，并能消除重复代码。
9. 将包含多种用户意图的大的测试类分解为只包含一种用户意图的小的测试类。
10. 当一个大类被分解为多个小类后，就可以把原先分布在大类中各处与小类相关的逻 辑，按照小类的职责转移到相应的小类中。比如把Game类的构造器中的问题准备的语句转 移到QuestionMaker类的构造器中。
11. 每一次重构都可能让以前运行通过的测试运行失败。此时要频繁地运行测试，发现失 败就立即解决，以缩小范围，降低査找问题的难度。
12. 当实现新特性时，有时新特性的用户意图会与原有特性的用户意图有冲突，此时 要根据新特性的用户意图更改原有的测试代码，进而更改生产代码，从而实现新特性的用户 意图。



第16章

分而测之——编写Stub及提取接口

唐僧在取经路上相继收了悟空、八戒和沙僧为徒后，继续马不停蹄地向西赶路。黎 山老母、观音、普贤和文殊这4位菩萨觉得应该考验一下师徒四人组成的这个新团队， 看看他们的禅心是否清静。于是4位菩萨化作如花似玉的母女四人，组成一个寡妇人家， 住在一个阔气的大宅院里，迎接路过的师徒四人。在将他们请进屋中喝茶时，黎山老母 化作的母亲就开始向师徒四人提亲，希望唐僧做家长，并希望将3位女儿许配给3位徒 弟，尽享富贵。唐僧、悟空和沙僧皆不动心。唯有八戒在见到母女四人后，贪图美色， 心痒难挠，甚至向黎山老母表白愿同时娶这母女四人为妻妾。最后被黎山老母吊在树上， 作为惩戒。

这个西游记的“四圣试禅心”故事，除了表现唐僧取经的坚定决心之外，还能给我们一 点启示，来解决下面这个在单元测试时经常面临的问题：如何解决被测系统（System Under Test,以下简称SUTe)所依赖的组件（Depended-On Component,以下简称DOC)在测试中 难以控制的问题。

如果把这个故事看成对师徒四人的一个测试，那么4位菩萨就好比是测试工程师，故事 本身就好比是测试代码（Test Code),师徒四人好比是SUT，而向师徒四人提亲时所需要的现 实世界的4位女性就好比是DOC。这个测试的时序图如图16-1所示。

㊀下文中有关 SUT、DOC、Stub、Mock 和 Test Double 的定义，参见 Gerard Meszaros,《太£/所7 你ms:

Refactoring Test Code ), Addison-Wesley, May 31, 2007 〇

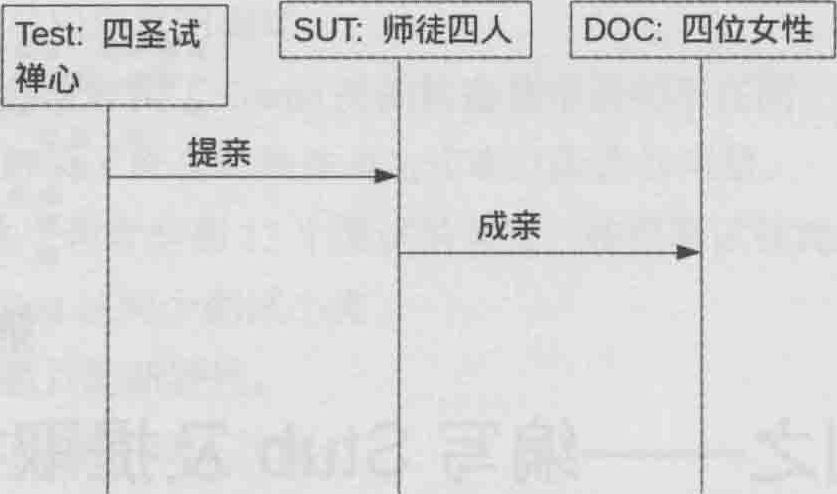


图16-丨安排师徒四人与现实世界的四位女性相亲的时序图

对于4位菩萨来说，尽管他们法力无边，但是要想真的从现实世界中找到4位看起来比 较般配且愿意嫁给师徒四人的女性，并且让她们与师徒四人碰面，也是难于上青天的。好在 4位菩萨可以运用法力，根据现实世界美女的接口，来分别化为母女四人，住在一个大宅院 中，来等候师徒四人。这样一来，菩萨化作的母女四人就好比Test Double (测试替身)。用 菩萨化作的母女四人替代现实世界的4位女性来向师徒四人提亲的时序图如图16-2所示。

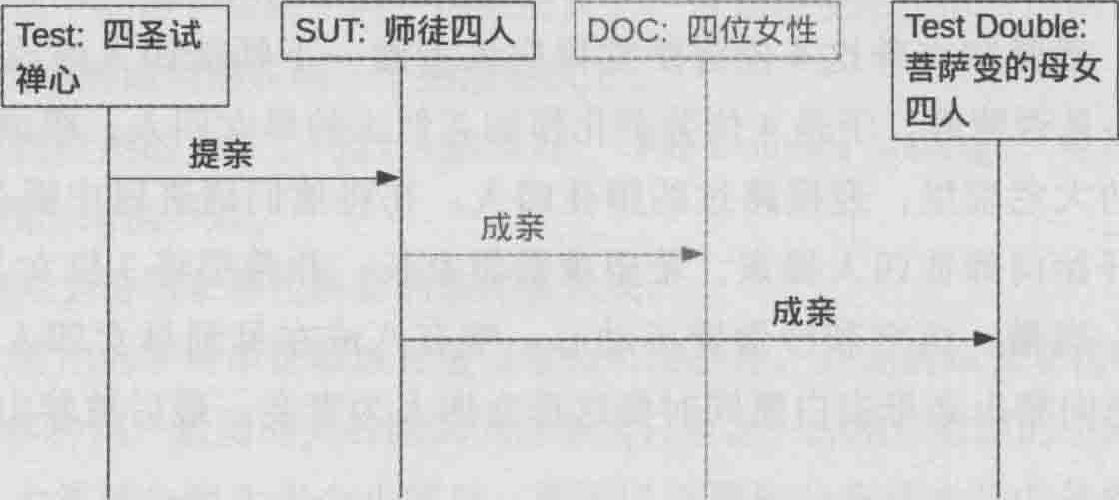


图16-2菩萨在取经必经之地化作母女四人向师徒四人提亲的时序图

只要4位菩萨化作的母女四人能够酷似现实世界的美女，或者用程序员的话说，只要这 母女四人能够实现现实世界美女的所有接口，那么师徒四人就可以把这母女四人当成现实世 界的美女来成亲，而菩萨们就能利用这样的机会来考验师徒四人。

“四圣试禅心”的故事告诉我们，当SUT所依赖的DOC很难在测试中进行控制时，就 可以像4位菩萨那样，根据现实世界的女性这个DOC，提取出美女这个接口，然后让师徒四 人这个SUT不再针对现实世界的女性这个DOC,而是针对美女这个接口来做出反应，并根 据这个美女接口化作母女四人这样易于控制的Test Double的做法，来替代难以控制的DOC, 从而令师徒四人这个SUT把母女四人这个Test Double当成美女接口来看待，进而考验师徒 四人这个SUT。毕竟母女四人这个Test Double就是菩萨变的，所以想怎样控制就可怎样控 制，从而解决了 DOC在测试中难以控制的问题。

现在咱们就操练用提取接口的方法来为DOC编写Stub。

“我没有编写过Stub。但我经常听说在编写单元测试时要写Mock。Stub和Mock到底有 什么区别呢？”

Stub和Mock都属于Test Double。Stub的作用是让测试能够控制SUT的间接输入，以 便于测试能够强制SUT进人正常情况下很难进人的运行路径中；而Mock的作用是让测试能 够验证SUT的间接输出。

在“四圣试禅心”的故事中，4位菩萨化作的母女四人就好比Stub,因为这个Stub可以 作为向师徒四人这个SUT提亲的间接输入，从而强制师徒四人考虑在正常情况下很难有机会 考虑的成亲这件事。在整个故事中，3位女儿除了作为提亲这件事的间接输人，做一些跑龙 套的事情外，并没有做多少有关判断师徒四人禅心的事情，所以3位女儿仅仅是Stub。而黎 山老母化作的母亲，除了像3位女儿一样做间接输人，还下了 “唐僧、悟空、沙僧这3位圣 僧有德不俗，八戒无禅有凡”的判断，这相当于验证了 SUT的间接输出，所以黎山老母化作 的母亲起到了 Mock的作用。

“哦，这下就明白多了。”

咱们这个编写Stub的编程操练题目是Tire Pressure Monitoring Systeme。这个题目是 有关轮胎气压检测系统的。有两个类Alarm和Sensor,其中Sensor类负责获取胎压值，而 Alarm类则检查Sensor传来的胎压值，若在正常范围之外则报警。另外，当前后两次检査胎 压值时，若前一次检査的胎压值在正常范围之外并报警后，后一次检査的胎压值又回到正常 范围，此时应该继续报警，以便于人工干预。本操练要求首先为Alarm类编写单元测试，然 后在测试的保护下实现一个新功能：在前后两次检查胎压值时，若前一次检查的胎压值在正 常范围之外并报警，后一次检查的胎压值又回到正常范围，此时应该停止报警。

“先看一看源代码《吧。”

源代码只有3个类。其中，第1个类Sensor有一个名为popNextPressurePsiValue()的公 共接口，来返回一个由随机数产生的胎压值。

Sensor类的代码如下所示：

public class Sensor {

public static final double OFFSET =\* 16;

public double popNextPressurePsiValue() {

double pressureTelemetryValue;

㊀取自ThougbtWorks Studio的培训师和教练Luca Minudel在2012年设计并上传到GitHub上的编程操练系 列题目 TDD with Mock Objects，参见：<https://github.com/lucaminudel/TDDwithMockObjectsAndDesignPrin> ciples。

© #JaL： <https://github.com/wubin28/tbc-tire-pressure-monitoring-system-java/tree/stub-with-extracting-interface>

pressureTelemetryValue = samplePressure();

return OFFSET + pressureTelemetryValue;

}

private static double samplePressure()

{

// placeholder implementation that simulate a real sensor in a real tire Random basicRandomNumbersGenerator = new Random(42);

double pressureTelemetryValue = 6 \* basicRandomNumbersGenerator.nextDouble() \* basicRandomNumbersGenerator.nextDouble(); return pressureTelemetryValue;

}

}

第2个类Alarm有两个公共接口，一个名为check〇，用来调用Sensor的popNext- PressurePsiValueO方法来获取胎压值，并判断胎压值是否在正常范围内，若不在正常范围内， 则将该类的alarmOii这个私有成员变量赋值为true，即设置为报警状态。另一个公共接口名 为isAlarmOn()，仅仅是返回alarmOn这个成员变量的值。

Alarm类的代码如下所示：

public class Alarm {

private final double LowPressureThreshold =17; private final double HighPressureThreshold = 21;

private Sensor sensor = new Sensor();

private boolean alarmOn = false;

public void check()

{

double psiPressureValue = sensor.popNextPressurePsiValue();

if (psiPressureValue < LowPressureThreshold 丨丨 HighPressureThreshold < psiPressureValue)

{

alarmOn = true;

public boolean isAlarmOn()

{

return alarmOn;

第3个类AlarmTest是一个测试类，里面只有一个测试2+3=5的这个必然通过的测 试，用来验证单元测试框架JUnit是否能正常工作。把光标移动到这个测试类名上，然后按 Ctrl+Shift+FlO组合键，运行一下这个测试。通过。

“代码很易读，看起来也不那么烂呀。”

代码虽然易读，但由于没有测试保护，使其对于代码维护者来说反馈慢，所以还是属于 烂代码。

这个操练题目要求实现一个新功能：若前一次检查的胎压值在正常范围之外并报 警，后一次检査的胎压值又回到正常范围，此时应该停止报警。能否把这个新特性写一个 TODO 呢？

“我刚才读了代码。要实现这个功能，只要Alarm类的check()方法每次检查胎压值之 前，把警报关闭就好了，也就是把alarmOn赋值为false就可以了。”

添加新特性在每次检査胎压前将警报关闭的TODO的代码如Tpj^(CM:AddedTODO- new-feature: the alarm will be turned off before each checking of pressure.)：

public class AlarmTest {

+ // TODO-new-feature: the alarm will be turned off before each checking of pressure

}

我感觉这个TODO写得有些问题。

“什么问题？”

如果用TDD来实现这个新特性的TODO,那么就需要先写测试，然后用测试中的意图 代码来驱动生成生产代码。换句话说，咱们的测试将来就要针对这个TODO来写。而这个 TODO是在每次检查胎压前将警报关闭，这是具体实现。前面咱们讨论过，针对具体实现的 测试代码注定是脆弱的，因为这违反了依赖倒置原则。

“哦，对。确实这个TODO是依赖具体实现了。需要把这个TODO从抽象的用户意图的 角度出发来改一改。”

将针对具体实现的在每次检查胎压前将警报关闭的新特性TODO,改为针对抽象的用户 意图的“跟随在正常范围之外的胎压值之后的正常胎压值应该能让之前所引发的警报停止” 的代码如下所不（CM: Updated the new feature TODO to be of user intent: a normal pressure value after a value outside the range should stop the alarm.)：

public class AlarmTest {

- // TODO-new-feature: the alarm will be turned off before each checking of pressure

+ // TODO-new-feature: a normal pressure value after a value outside the

range should stop the alarm

}

写完了新特性的TODO,现在可以大概读一读这些代码，看看能否看到比较严重的

问题D

“开始我还以为这些代码写得很烂呢，但读完之后发现写得还算比较整洁，没有发现什 么很严重的问题。”

这个题目中的两个类Alarm和Sensor，是谁依赖谁？

“Alarm 类依赖 Sensor 类〇”

对。那么Alarm类是如何依赖Sensor类的？

“在Alarm类中new出了一个Sensor对象〇”

问题就出在new这个操作符上了。通过new操作符创建的Sensor对象，就让Alarm类 依赖一个具体的Sensor对象，而不是一个抽象。这样做就会违反前面提到的依赖倒置原则。 另外由于Alarm类依赖的是new出来的Sensor类的具体的实现，这样就无法用扩展的方法 将其替换为一个测试替身来进行单元测试，无法做到“对修改关闭，对扩展开放”，所以也 违反了开闭原则。

“但是Alarm类确实需要一个具体的Sensor对象来帮它获得胎压值呀。如何才能做到即 能让Alarm类依赖一个抽象，又能让Alarm类获得一个具体的Sensor对象呢？”

咱们可以把上面提到的那条new出一个具体Sensor对象的语句移动到Alarm类之外， 并通过Alarm类的构造器或setter〇方法，把这个在外面new出的具体的Sensor对象注人 Alarm对象中，而Alarm类的构造器或setter〇方法则带有从Sensor类所提取的接口的类型。 这样就可以做到上面那一点。一会通过写意图代码就能看得明白。先把这个问题用TODO记 下来。

在Alarm类中添加有关“ new Sensor();”语句违反依赖倒置和开闭原则的TODO的代码 如下所示（CM: Added TODO: Depending on a concrete Sensor violates the Dependency Inversion Principle and Open-Closed Principle.)：

public class Alarm {

+ //TODO: Depending on a concrete Sensor violates the Dependency Inversion Principle and Open-Closed Principle private Sensor sensor = new Sensor();

写完了这个依赖具体实现的TODO,接下来可以看看该写哪些有关测试的TODO 了。

“需要针对用户意图来编写有关测试的TODO。从用户角度出发，我现在能想到的用户 意图有3个。第一个是检测到正常的胎压值时不应该报警；第二个是检测到正常范围之外的 胎压值时应该报警；第三个是跟随在正常范围之外的胎压值之后的正常胎压值应该不会让之 前所引发的警报停止。”

在AlarmTest类中添加3个有关用户意图的TODO的代码如下所示（CM: Added 3 user intent test TODOs.)：

public class AlarmTest {

+ // TODO-user-intent-test: a normal pressure value should not raise the alarm

+ // TODO-user-intent-test: a pressure value outside the range should raise the alarm

+ // TODO-user-intent-test: a normal pressure value after a value outside the

range should not stop the alarm

+

// TODO-new-feature: a normal pressure value after a value outside the range should stop the alarm

}

现在可以按Alt+6快捷键来列出所有的5个TODO,挑一个最简单的先做。比方说可以 先做a normal pressure value should not raise the alarm (检测到正常的胎压值时不应该报警）这 个TODO。直接把这个TODO的英文单词之间加上下划线，就成了针对这个TODO的测试 的方法名。这样能使测试的命名尽量做到像自然语言那样容易阅读，使得测试能起到文档的 作用。另外按照咱们的老习惯，先写测试的Assert部分。

把检测到正常的胎压值时不应该报警这个TODO标记为working-on,并编写相应的测 试和其 Assert 部分的代码如下所示（CM: Working on TODO: a normal pressure value should not raise the alarm. Wrote a test a\_normal\_pressure\_value\_should\_not\_raise\_the\_alarm() with an Assert part.)：

public class AlarmTest {

- // TODO-user-intent-test: a normal pressure value should not raise the alarm

+ // TODO-user-intent-test-working-on: a normal pressure value should not

raise the alarm @Test

+ public void a\_normal\_pressure\_value\_should\_not\_raise\_the\_alarm() {

+ // Assert

+ assertFalse(alarm.isAlarmOn());

}

Assert部分写好后，就可以从它推出Act部分。要让alarm.isAlarmOnO能获得正确的 值，需要先调用alarm.check〇方法，而调用alarm.check()方法就是测试的Act部分。

编写 a\_normal\_pressure\_valiie\_should\_not\_raise\_the\_alarm()测试方法的 Act 部分的代 码如下所示（CM: Wrote the intention code for the Act and Arrange parts of the test a\_normal\_ pressure\_value\_should\_not\_raise\_the\_alarm().)：

public class AlarmTest {

// TODO-user-intent-test-working-on: a normal pressure value should not raise the alarm @Test

public void a\_normal\_pressure一value一should一not\_raise\_the一alarm(> {

+ //Act

+ alarm.check();

+

// Assert

assertFalse(alarm.isAlarmOn());

}

写好了 Act部分，那么该看alarm这个变量是怎么来的了。alarm需要新创建一个Alarm 对象而被new出来。

在 a\_normal\_pressure\_value\_should\_not\_raise\_the\_alarm()测试方法的 Arrange 部分中 new 出一个Alarm对象的代码如下所示：

public class AlarmTest {

// TODO-user-intent-test-working-on: a normal pressure value should not raise the alarm ©Test

public void a—normal一pressure\_value\_should\_not\_raise\_the\_alarm() {

+ // Arrange

+ Alarm alarm = new Alarm();

alarm.check();

// Assert

assertFalse(alarm.isAlarmOn());

}

“按理说，这个测试代码就能运行了。运行一下试试……通过。不过咱们读了代码就能 知道，这个测试运行的结果并不总是通过的，因为Sensor类所返回的下一个胎压值是一个随 机数，不好控制〇”

这里，AlarmTest类就是测试代码，Alarm类就是SUT, Alarm类所依赖的Sensor类就 是DOC。就如同难以在现实世界中找到4位般配的女性愿意与师徒四人成亲一样，我们也 难以控制Sensor这个依赖于随机数的类，来让它总能返回我们所期望的某一个胎压值，作为 Alarm类的间接输人。

“在这个例子中，我们需要根据Sensor类这个DOC,提取出一个类似于‘美女’这样的 接口，比如叫ISensor,并让SUT针对这个接口编程，从而解决这个问题。”

思路很好，不过ISensor这个接口命名不够好。

“为什么不好？这样不是很方便吗？而且C#语言的命名惯例也是这样的。”

有些惯例其实是有问题的。在一个类名前冠以大写字面I来命名一个接口，看似方便， 姐是将来若把接口重构为抽象类或类时，那这个I就词不达意了。所以最好还是用另一个英 文单词作为接口名。我有一个窍门。可以用查找英文近义词的方法，来为一个具体的类寻找 它的接口的命名。咱们可以用浏览器访问http://translate.google.cn链接，然后在左侧文本框 中输入一个英文词，这样在右边下方就能看到这个词的近义词，从中选一个就好了。用这个 方法，咱们找到了 Sensor的近义词Transducer。这样可以用Transducer作为接口的名字。

用浏览器访问http://translate.google.cn链接，查找英文近义词来用于接口命名的界面如 图16-3所示。

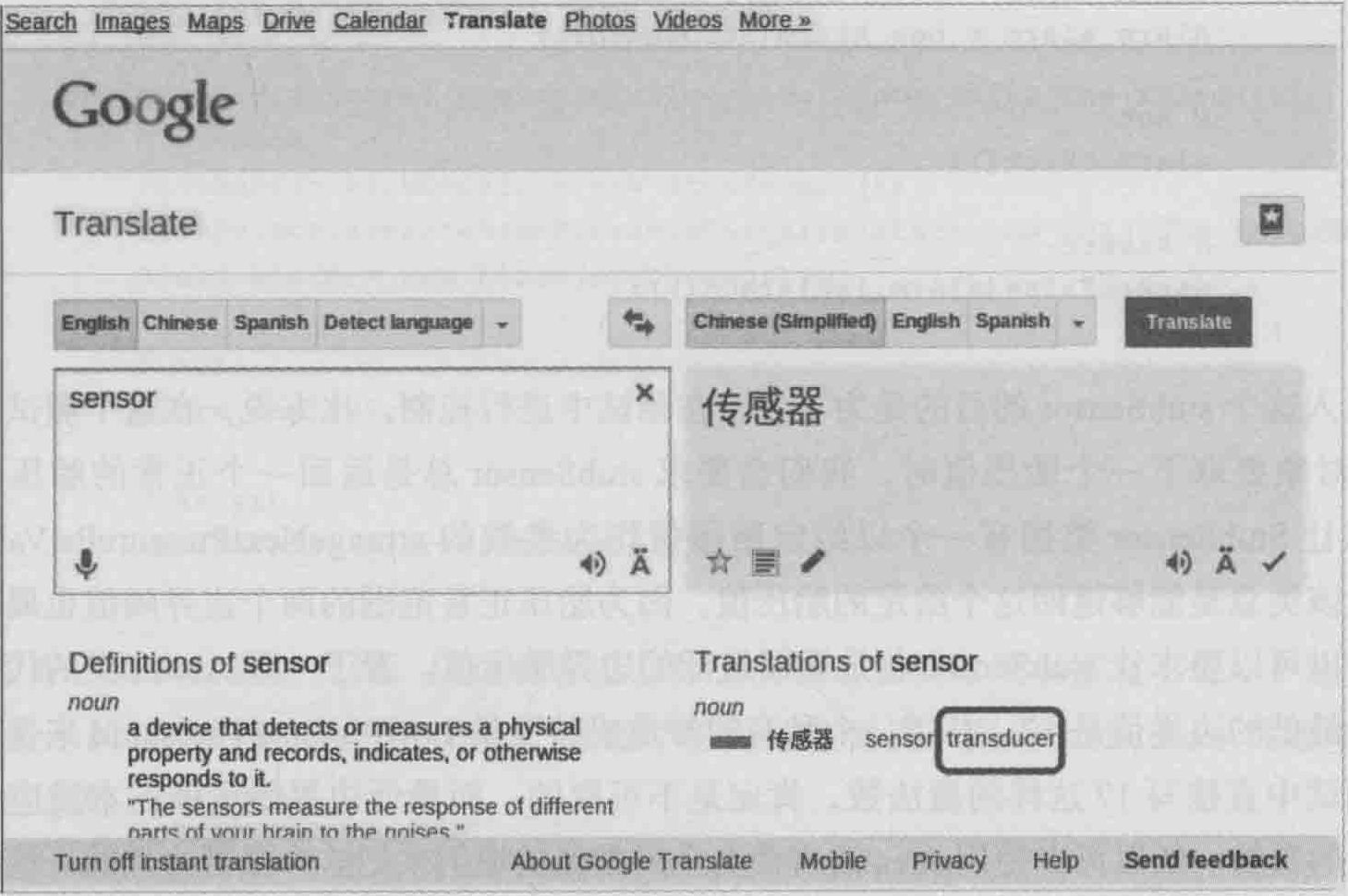


图16-3用浏览器访问http://translate.google.cn链接查找英文近义词的界面

如果有了 Traiisducei•这样的接口，就可以根据这个接口，编写出一个易于控制的 Stub类，不妨命名为StubSensor,来取代难以控制的Sensor类，从而令Alarm类把这个 StubSensor类当成Transducer接口来看待，进而方便地测试Alarm类。

“那么怎样才能让Alarm类把StubSensor类当成Transducer接口来看待呢？”

可以让Alarm类针对抽象的Transducer接口，而不是针对具体的Sensor类来编程。具 体的想法是：先从Sensor类中提取Transducer接口，然后让StubSensor类实现该接口，并通 过Alarm类的带有Transducer类型的构造器，将一个StubSensor对象注入Alarm对象中，从 而让Alarm类把StubSensor类当成Transducer接口来看待。咱们可以使用这个编程意图，在 测试中编写意图代码。先让形参为Transducer的Alarm类的构造器接受StubSensor这个实参 类型为StubSensor的参数。

让形参为Transducer的Alarm类的构造器接受StubSensor这个实参类型为StubSensor的

参数的意图代码如下所示:

public class AlarmTest {

// TODO-user-intent-test-working-on: a normal pressure value should not raise the alarm @Test

public void a\_normal\_pressure\_value\_should\_not\_raise\_the\_alarm() {

// Arrange

Alarm alarm = new Alarm 〇;



Alarm alarm =» new Alarm (stubSensor);

//Act

alarm.check();

// Assert

assertFalse(alarm.isAlarmOn());

引入这个stubSensor的目的是为了便于在测试中进行控制。比方说，在这个测试中，当 Alarm对象要取下一个胎压值时，我们会要求stubSensor总是返回一个正常的胎压值。所 以可以让StubSensor类拥有一个以给定胎压值作为参数的arrangeNextPressurePsiValue()方 法，让该类总是能够返回这个给定的胎压值。因为胎压正常范围的两个边界阈值也属于正常 值，所以可以要求让stubSensor总是返回最低的边界胎压值。看了一眼Alarm类的代码，发 现这个最低的边界值是17,并用一个私有的常量成员变量LowPressureThreshold来保存。要 是在测试中直接写17这样的魔法数，肯定是不可取的。而最低边界胎压值的本质应该是一 个全局的常量。所以可以使用Java语言命名全局常量的惯例来把这个最低边界胎压值命名为 Alarm.LOW\_PRESSURE—THRESHOLD 0

让stubSensor对象通过调用arrangeNextPressurePsiValue()方法来把最低边界胎压值作为 下一个胎压值返回的意图代码如下所示：

public class AlarmTest {

// TODO-user-intent-test-working-on: a normal pressure value should not raise the alarm @Test

public void a\_normal\_pressure\_value\_should\_not\_raise\_the\_alarm() {

// Arrange

+ stubSensor.arrangeNextPressurePsiValue(Alarm.L0W\_PRESSURE\_THRESH0LD);

Alarm alarm \* new Alarm(stubSensor);

//Act

alarm.check();

// Assert

assertFalse(alarm.isAlarmOn());

}

“这个stubSensor对象是怎么来的？需要new出一个StubSensor对象来。” 创建StubSensor对象的代码如下所示：

public class AlarmTest {

// TODO-user-intent-test-working-on: a normal pressure value should not raise the alarm @Test

public void a\_normal\_pressure\_value\_should\_not\_raise\_the\_alarm() {

// Arrange

+ StubSensor stubSensor = new StubSensor();

stubSensor.arrangeNextPressurePsiValue(Alarm.LOW\_PRESSURE\_THRESHOLD); Alarm alarm = new Alarm(stubSensor);

//Act

alarm.check();

// Assert

assertFalse(alarm.isAlarmOn());

}

“写好了测试的意图代码，仔细看看没有什么问题后，就可以根据在IDEA中这些意图 代码所显示的红色的编译错误的指引，来逐个修复这些错误。因为StubSensor类还未创建， 所以是红色的，先修复它。”

创建StubSensor类来修复编译错误的代码如下所7K ( CM: Fixed compiler error: Created class StubSensor.)：

+public class StubSensor {

+ }

“下一步修复Alarm丄OW\_PRESSURE\_THRESHOLD红色的编译错误。”

创建Alarm丄OW\_PRESSURE\_THRESHOLD类常量并用其替换原有私有常量的代码如 下所7K (CM: Fixed compiler error: Created constant field Alarm.LOW PRESSURE THRESHOLD and replaced it with the original private field.)：

public class Alarm {

- private final double LowPressureThreshold = 17/

+ public static final double LOW—PRESSURE一THRESHOLD = 17; private final double HighPressureThreshold = 21;

if (psiPressureValue < LowPressureThreshold || HighPressureThreshold < psiPressureValue)

+ if (psiPressureValue < LOW一PRESSURE一THRESHOLD | | HighPressureThreshold <

psiPressureValue)

{

alarmOn = true;

‘下一步修复调用stubSensor对象的arrangeNextPressurePsiValue()方法的红色编译错误。” 创建StubSensor.arrangeNextPressurePsiValue()方法来修复编译错误的代码如下所7K (CM: Fixed compiler error: Created method StubSensor.arrangeNextPressurePsiValue〇.)：

public class StubSensor {

+ public void arrangeNextPressurePsiValue(double nextPressurePsiValue) {

+

+ }

)

“接下来该修复创建Alarm对象时向构造器传人的实参stubSensor的红色编译错误了。 不过要修复这个错误，需要经过多步才能完成。首先可以创建Alarm类的带有Transducer类 型参数的构造器。”

创建Alarm类的带有Transducer类型参数的构造器来修复编译错误的代码如下所示 (CM: Fixing compiler error: Created constructor Alarm(Transducer).)：

public class Alarm {

+ public Alarm(Transducer transducer) {

+ }

“Transducer由于还未创建，所以是红色的，再修复它。”

创建接口 Transducer来修复编译错误的代码如下所示（CM: Fixing compiler error in test: Created interface Transducer.)：

4-public interface Transducer {

+ }

“由于新创建的带有参数的Alarm类的构造器的形参是接口 Transducer,而传给这个构造 器的实参是StubSensor类型的对象，且StubSensor类尚未实现Transducer接口，所以有这个 编译错误。现在让StubSensor类实现Transducer接口，来修复错误。”

让StubSensor类实现Transducer接口来修复错误的代码如下所示（CM: Fixed compiler error in test: Made class StubSensor implement interface Transducer.)：

-public class StubSensor {

+public class StubSensor implements Transducer {

现在测试里面的编译错误都修复了。运行测试，通过。但是咱们在前面创建的所有类、 接口和方法里面的内容都是空的，所以这个运行通过的测试，还是运行在原有随机数代码逻 辑上的碰巧的运行通过。这个测试TODO尚未完成。接下来就要把原有的代码逻辑切换到新 的逻辑之上，即要修复那个依赖于具体的Sensor从而违反依赖倒置和开闭原则的TODO。把 它标记为working-on。

“咱们在前面创建了一个带有Transducer类型参数的Alarm类的构造器，这就要求咱们 还必须创建Alarm类的默认构造器，以保持原有接口。这需要写一个TODO记下来。”

在Alarm类中添加有关创建Alarm类的默认构造器以保持原有接口的TODO的代码如 下所示（CM: Added TODO: Retain the original interface for the default constructor of Alarm，）：

public class Alarm {

+ //TODO: Retain the original interface for the default constructor of Alarm public Alarm(Transducer transducer) {

“这个TODO比较简单，所以可以先完成它。”

完成在Alarm类中添加有关创建Alarm类的默认构造器以保持原有接口的TODO的 代码如下所示（CM: Finished TODO: Retain the original interface for the default constructor of Alarm.)：

public class Alarm {

- // TODO-working-on: Retain the original interface for the default constructor

of Alarm

+ public Alarm() {

+ this.sensor = new Sensor();

+ }

“现在可以回过头来解决那个依赖于Sensor具体实现的TODO 了。可以先在Alarm 类中编写意图代码。首先把私有成员变量sensor改名为transducer，并且将其类型换成 Transducer。另外把‘new Sensor();’这句话放到Alarm类的默认构造器中。再把所有原先引 用成员变量sensor的地方换成transducer。最后在新增的Alarm类的带有Transducer类型的 构造器中，将传入的实参transducer保存在该类的transducer的成员变量中。”

在Alarm类中编写意图代码来解决那个依赖于Sensor具体实现的TODO的代码如下所 (CM: Wrote intention code for TODO: Depending on a concrete Sensor violates the Dependency Inversion Principle and Open-Closed Principle.)：

public class Alarm {

// TODO-working-on: Depending on a concrete Sensor violates the Dependency Inversion Principle and Open-Closed Principle

* private Sensor sensor = new Sensor();

+ private Transducer transducer = null;

private boolean alarmOn = false;

public Alarm() {

* this.sensor = new Sensor();

+ this.transducer = new Sensor();

}

public Alarm(Transducer transducer) {

+ this.transducer = transducer;

}

public void check()

{

* double psiPressureValue = sensor.popNextPressurePsiValue();

+ double psiPressureValue = transducer.popNextPressurePsiValue();

“这段意图代码写完后，就在Alarm类中出现了两处红色的编译错误。先修复出现在 Alarm类默认构造器中的那一行‘ this.transducer = new Sensor(); ’语句的编译错误。这处错 误的原因是因为Sensor类尚未实现Transducer接口。现在就修复它。”

让Sensor类实现Transducer接口来修复编程错误的代码如下所示（CM: Fixed compiler error: Made class Sensor implement interface Transducer.)：

-public class Sensor {

♦public class Sensor implements Transducer {

“下一个有红色编译错误的地方是调用transducer对象的popNextPressurePsiValue()方法， 错误的原因是Transducer接口还没有定义这个方法，现在就修复它。”

在接口 Transducer中添加popNextPressurePsiValue〇方法的定义来修复编译错误的代码 如下所示（CM: Fixed compiler error: Created method Transducer.popNextPressurePsiValue().):

public interface Transducer {

+ double popNextPressurePsiValue();

}

“现在Alarm类中所有红色的编译错误都修复了。可以按Ctrl+F5快捷键重新运行测试。 出现另一个编译错误：StubSensor类尚未实现刚刚在其所实现的接口 Transducer中新增的方 法 popNextPressurePsiValue〇。现在就修复它。先在 StubSensor 类中创建一^带有 @Override 标注的popNextPressurePsiValue()方法，然后让它返回该类在成员变量nextPressurePsiValue 中所保存的下一个胎压值。而成员变量nextPressurePsiValue中所保存的下一个胎压值又来自 该类的方法arrangeNextPressurePsiValue()所传人的参数，即我们通过Stub来进行控制的期 望的胎压值。”

在 StubSensor 类中实现 popNextPressurePsiValue()方法的代码如下所 7K ( CM: Fixed compiler error: Implemented method StubSensor.popNextPressurePsiValue().)：

public class StubSensor implements Transducer {

+ private double nextPressurePsiValue;

+

public void arrangeNextPressurePsiValue(double nextPressurePsiValue) {

+ this.nextPressurePsiValue « nextPressurePsiValue;

+ }

+ @Override

+ public double popNextPressurePsiValue() {

+ return this.nextPressurePsiValue;

}

}

“运行测试，通过。既然StubSensor类实现Transducer接口时写了 @Override标注，那 么Sensor类实现Transducer接口时也应该写这个标注。”

在Sensor类的popNextPressurePsiValue()方法前面添加@Override标注的代码如下所7K (CM: Added Override notation to method Sensor.popNextPressurePsiValue().)：

public class Sensor implements Transducer {

+ @Override

public double popNextPressurePsiValue()

“运行测试，通过。现在既完成了那个依赖于Sensor具体实现的TODO,又完成了检测 到正常的胎压值时不应报警的用户意图TODC^可以把这两个TODO删除了。”

接下来该处理下一个用户意图的测试TODO 了：检测到正常范围之外的胎压值时应 该报警。这个TODO的测试的意图代码与前一个有关检测到正常范围之内的胎压值不应 报警的用户意图的测试TODO很相似。只不过arrangeNextPressurePsiValue()方法传人的 是Alarm.HIGH\_PRESSURE\_THRESHOLD+l这个正常范围之外的值。另外最后判断alarm. isAlarmOn()用的是 assertTrue()〇

将用户意图的测试TODO “检测到正常范围之外的胎压值时应该报警”标记为working- on, 并编写相关测试的意图代码的代码如下所示 （ CM: Working on TODO: a pressure value outside the range should raise the alarm. Wrote intention code for the Assert, Act and Arrange parts

of the test a\_pressure\_value\_outside\_the\_range\_should\_raise\_the\_alarm〇.)：

public class AlarmTest {

- // TODO-user-intent-test: a pressure value outside the range should raise the alarm

+ // TODO-user-intent-test-working-on: a pressure value outside the range

should raise the alarm + @Test

+ public void a\_pressure\_value\_outside一the一range一should一raise\_the一alarm〇 {

+ // Arrange

+ StubSensor stubSensor \* new StubSensor();

+ stubSensor.arrangeNextPressurePsiValue(Alarm.HIGH\_PRESSURE\_THRESHOLD + 1);

+ Alarm alarm = new Alarm(stubSensor);

+

+ //Act

+ alarm.check();

+

+ // Assert

+ assertTrue(alarm.isAlarmOn());

+ )

“测试的意图代码写好后，出现了一处红色的编译错误：HIGH\_PRESSURE\_ THRESHOLD。这是因为Alarm类尚未定义这个常量。现在就修复这个错误。"

创建Alarm.HIGH\_PRESSURE\_THRESHOLD类常量并用其替换原有私有常量的代码如下 所示（CM: Fixed compiler error: Created constant field Alarm.HIGH\_PRESSURE\_THRESHOLD and replaced it with the original private field.)：

public class Alarm {

public static final double LOW\_PRESSURE一THRESHOLD = 17;

- private final double HighPressureThreshold - 21;

+ public static final double HIGH\_PRESSURE一THRESHOLD = 21;

private Transducer transducer = null;

- if (psiPressureValue < LOW\_PRESSURE\_THRESHOLD 丨丨 HighPressureThreshold <

psiPressureValue)

+ if (psiPressureValue < LOW\_PRESSURE\_THRESHOLD || HIGH一PRESSURE\_THRESHOLD <

psiPressureValue)

{

alarmOn = true;

}

“运行测试，通过。现在完成了 ‘检测到正常范围之外的胎压值时应该报警’的测试意 图TODO,可以将其删除。”

现在该处理“跟随在正常范围之外的胎压值之后的正常胎压值应该不会让之前所引发的

警报停止”这个TODO 了。

将“跟随在正常范围之外的胎压值之后的正常胎压值应该不会让之前所引发的警报停 止”这个TODO标记为working-on，并编写相应测试的Assert部分的代码如下所示（CM: Working on TODO: a normal pressure value after a value outside the range should not stop the alarm. Wrote the Assert part of the test a\_normal\_pressure一value after a一value outside the一 range\_should\_not\_stop\_the\_alarm().)：

public class AlarmTest {

- // TODO-user-intent-test: a normal pressure value after a value outside the

range should not stop the alarm

+ // TODO-user-intent-test-working-on: a normal pressure value after a value

outside the range should not stop the alarm ■f @Test

+ public void a\_normal\_pressure\_value\_after\_a\_value\_outside\_the\_range\_should\_

not\_stop一the一alarm() {

+ // Assert

+ assertTrue(alarm.isAlarmOn());

+ }

“有了前面的测试的铺垫，写这个测试就很容易了。这个测试要求先检测到一个在正 常范围之外的胎压值并报警，然后再检测到一个正常范围的胎压值且不应把之前的报警给 关闭。基于这个要求，咱们可以分两步来实现这个测试的Act部分。第一步，先调用Stub 对象的arrangeNextPressurePsiValue〇方法，令其在被Alarm对象调用时返回Alarm丄0W\_\_ PRESSURE\_THRESH0LD-1这样一个正常范围之外的值，然后调用Alarm类的check〇方法 来检测。第二步，调用Stub对象的arrangeNextPressurePsiValue()方法，令其在被Alarm对 象调用时返回AlamLLOW\_PRESSURE\_THRESHOLD这样一个在正常范围之内的值，然后 调用Alarm类的Check()方法来检测。测试写完后，运行，通过。这样，这个TODO也完成 了，可以将该TODO删除。”

完成“跟随在正常范围之外的胎压值之后的正常胎压值应该不会让之前所引发的警报停 止”这个 TODO 的代码如下所示（CM: Finished TODO: a normal pressure value after a value outside the range should not stop the alarm.)：

public class AlarmTest {

// TODO-user-intent-test-working-on: a normal pressure value after a value outside the range should not stop the alarm @Test

public void a\_normal\_pressure一value\_after\_a\_value一outside\_the\_range一should not\_stop\_the\_alarm() {

+ // Arrange

+ StubSensor stubSensor = new StubSensor();

+ Alarm alarm = new Alarm(stubSensor);

+

+ //Act

+ stubSensor.arrangeNextPressurePsiValue(Alarm.LOW\_PRESSURE\_THRESHOLD - 1);

+ alarm.check();

+

+ stubSensor•arrangeNextPressurePsiValue(Alarm.LOW一PRESSURE一THRESHOLD);

+ alarm.check();

+

// Assert

assertTrue(alarm.isAlarmOn());

}

“现在该处理最后一个TODO,即那个新特性‘跟随在正常范围之外的胎压值之后的正 常胎压值应该能让之前所引发的警报停止’的TODO 了。因为这个新特性的TODO与刚刚 编写的那个用户意图TODO在功能上正好相反，所以可以把前一个TODO的测试改造一下， 改一下测试名字，将Assert部分的assertTrue()方法改为assertFalse()方法，就成为现在正在 处理的TODO的测试。运行测试，失败。原因是在A丨arm类的check〇方法中每次检测胎压 值前清除警报，即alarmOn这个成员变量没有在每次检测胎压值之前赋值为false。在Alarm 类的check()方法开头加上一句‘alarmOn = false;’就好了。”

完成新特性“跟随在正常范围之外的胎压值之后的正常胎压值应该能让之前所引发的 警报停止”的 TODO 的代码如下所示（CM: Finished the new feature TODO: a normal pressure value after a value outside the range should stop the alarm.)：

public class AlarmTest {

* // TODO-new-feature: a normal pressure value after a value outside the range

should stop the alarm @Test

* public void a\_normal\_pressure\_value\_after\_\_a\_value\_outside\_the\_range\_

should一not一stop一the\_alarm() {

+ public void a\_normal一pressure\_value\_after一a\_value一outside一the\_range一should一 stop一the\_alarm() {

// Arrange

StubSensor stubSensor = new StubSensor();

Alarm alarm = new Alarm(stubSensor);

// Assert

assertTrue(alarm.isAlarmOn()); assertFalse(alarm.isAlarmOn());

public class Alarm {

public void check()

+ alarmOn = false;

double psiPressureValue = transducer.popNextPressurePsiValue();

至此，这个有关胎压检测系统的编程操练就告一段落。为了便于和重构前的代码进行比 较，下面列出重构后的测试代码和生产代码。

重构后AlarmTest类的代码如下所示：

public class AlarmTest {

@Test

public void a\_normal\_pressure\_value\_should\_not\_raise\_the\_alarm () {

// Arrange

StubSensor stubSensor = new StubSensor();

stubSensor.arrangeNextPressurePsiValue(Alarm.LOW\_PRESSURE一THRESHOLD); Alarm alarm = new Alarm(stubSensor);

//Act

alarm.check();

// Assert

assertFalse(alarm.isAlarmOn());

}

@Test

public void a一pressure一value一outside\_the一range\_should一raise一the\_alarm() {

// Arrange

StubSensor stubSensor = new StubSensor();

stubSensor.arrangeNextPressurePsiValue(Alarm.HIGH\_PRESSURE\_THRESHOLD + 1); Alarm alarm = new Alarm(stubSensor);

//Act

alarm.check ();

// Assert

assertTrue(alarm.isAlarmOn());

}

@Test

public void a\_normal\_pressure\_value一after\_a\_value\_outside\_the一range一should一 stop\_the\_alarm() {

// Arrange

StubSensor stubSensor = new StubSensor();

Alarm alarm = new Alarm(stubSensor);

//Act

stubSensor.arrangeNextPressurePsiValue(Alarm.LOW一PRESSURE一THRESHOLD - 1); alarm.check ();

stubSensor.arrangeNextPressurePsiValue(Alarm.LOW\_PRESSURE\_THRESHOLD); alarm.check();

// Assert

assertFalse(alarm.isAlarmOn());

重构后Sensor类的代码如下所示：

public class Sensor implements Transducer { public static final double OFFSET = 16;

@Override

public double popNextPressurePsiValue〇

{

double pressureTelemetryValue; pressureTelemetryValue = samplePressure();

return OFFSET + pressureTelemetryValue;

}

private static double samplePressure()

{

// placeholder implementation that simulate a real sensor in a real tire

Random basicRandomNumbersGenerator = new Random(42);

double pressureTelemetryValue = 6 \* basicRandomNumbersGenerator.

nextDouble() \* basicRandomNumbersGenerator.nextDouble(); return pressureTelemetryValue;

重构后Transducer接口的代码如下所示：

public interface Transducer {

double popNextPressurePsiValue();

}

重构后StubSensor类的代码如下所示：

public class StubSensor implements Transducer { private double nextPressurePsiValue;

public void arrangeNextPressurePsiValue(double nextPressurePsiValue) {

this.nextPressurePsiValue = nextPressurePsiValue;

}

©Override

public double popNextPressurePsiValue() {  
return this.nextPressurePsiValue;

重构后Alarm类的代码如下所示：

public class Alarm {

public static final double LOW一PRESSURE—THRESHOLD = 17; public static final double HIGH\_PRESSURE一THRESHOLD = 21;

private Transducer transducer = null;

private boolean alarmOn = false;

public Alarm() {

this.transducer = new Sensor();

}

public Alarm(Transducer transducer) {  
this.transducer = transducer;

}

public void check()

{

alarmOn = false;

double psiPressureValue = transducer.popNextPressurePsiValue();

if (psiPressureValue < LOW一PRESSURE—THRESHOLD || HIGH\_\_PRESSURE\_THRESHOLD < psiPressureValue)

{

alarmOn = true;

public boolean isAlarmOn()

{

return alarmOn;

}

}

在操练了 Stub编写和接口提取的编写单元测试的方法后，咱们可以再操练一下Mock编 写和子类化并覆写方法的方法。不过在继续操练前，咱们看看这个操练都做了哪些工作。

1)阅读轮胎气压检测系统的代码，包括Sensor类和Alarm类。

1. )从用户意图而不是具体实现的角度编写新特性的TODO。

3)记录了使用new操作符创建的Sensor对象，从而违反依赖倒置和开闭原则的TODO。

1. )记录了 3个用户意图测试的TODO。
2. 先写了测试的Assert部分的意图代码，然后再推导出Act和Arrange部分的测试意 图代码，并通过修复意图代码的编译和测试运行错误，来驱动出生产代码，从易到难地完成 上述TODO。
3. 使用提取接口的方法，让SUT针对抽象的接口，而不是针对具体的DOC来编程。 先从DOC中提取接口，然后编写Stub类实现该接口，并通过SUT的带有上述接口类型的构 造器，将一个Stub对象注人SUT对象中，从而实现让SUT把易于控制的Stub类当成上述 接口来看待，对SUT进行测试。
4. 通过操练我们学到了以下技能：
5. 代码虽然易读，但由于没有测试保护，使其对于代码维护者来说反馈慢，所以这样的 代码还是属于烂代码。
6. 当SUT所依赖的DOC很难在测试中进行控制时，可以根据DOC提取出一个接口， 然后让SUT不再针对DOC,而是针对这个接口编程，并根据这个接口编写易于控制的Test Double，来替代那个难以控制的DOC，从而令SUT把Test Double当成那个接口来看待，进 而对SUT进行测试，来解决DOC在测试时难以控制的问题。
7. Stub和Mock都属于Test Double。前者的作用是让测试能够控制SUT的间接输入， 以便于测试能够强制SUT进入正常情况下很难进人的运行路径中；而后者的作用是让测试能 够验证SUT的间接输出。
8. 有关测试TODO应该针对抽象的用户意图来编写，而不要针对具体的编程实现逻辑 来编写，以便让测试变得强壮而不是变得脆弱。
9. 使用像ISensor这样命名接口的方式并不理想，比较好的方法是用translate.google.cn 来寻找一个类名的英文近义词来为该类接口命名。

第 17 章；

分而测之一编写Mock及子类化并覆写方法

为了解决SUT所依赖的DOC在测试中难以控制的问题，我们可以使用刚刚操练的编写 Stub及提取接口的方法。除此之外，我们还可以使用另一种方法，即编写Stub及子类化并 覆写的方法。现在咱们就开始做Ticket Dispenser0这个题目，来操练一下后者。

这个题目是有关自动取号系统的，类似于去银行办事进门时取号的那种取号机。有3个 类：TumTicket类表示从取号机上所取的票，票上印着号码；TumNumberSequence类用于产 生所有票上的号码；而TicketDispenser类则表示取号机，根据从TumNumberSequence类获 得的号码来出票。这个题目要求在有多个TicketDispenser的情况下，两个人分别在两台不同 的取号机上取号，不能取到同样号码的票。操练这个题目首先要为TicketDispenser类编写测 试，然后在测试的保护下实现两个新特性：VIP客户的票号从1001开始；普通（Regular)客 户的票号从2001开始。

“看一下现有的代码0卩巴。”

源代码有4个类。第1个是TumTidcet类，它有两个公共接口方法，一个是带有整型参 数tumNumber的构造器，用来把票号保存到成员变量中；另一个是方法getTumNumber(), 用来获取票号。

㊀取自ThoughtWorks Studio的培训师和教练Luca Minude丨在2012年设计并上传到GitHub上的编程操练系 列题目 TDD with Mock Objects,参见：<https://github.cora/lucaminudel/TDDwithMockObjectsAndDesignPrin> ciples〇

㊁ Ticket Dispenser 操练题目原有代码参见：<https://github.com/wubin28/tbc-ticket-dispenser-java/tree/exercise> ; 本书针对该题目的操练步骤代码参见：[https://github.com/wubin28/tbc-ticket-dispenser-java/tree/subclass-and- override-method](https://github.com/wubin28/tbc-ticket-dispenser-java/tree/subclass-and-override-method) 〇

TumTicket类的代码如下所示：

public class TurnTicket {

private final int turnNumbere­public TurnTicket(int turnNumber)

{

this.turnNumber = turnNumber;

}

public int getTurnNumber()

{

return turnNumber;

>

}

第2个类是TumNumberSequence，它只有一个公共接口方法，就是静态的getNext- TumNumber()方法，返回静态的成员变量turnNumber的值，然后将其增1。 TumNumberSequence类的代码如下所示：

public class TumNumberSequence {

private static int 一turnNumber = 0;

public static int getNextTurnNumber()

{

return 一turnNumber++;

}

}

第3个类是TicketDispenser,它也只有一个公共接口方法，即getTumTicket()方法。这 个方法从TumNumberSequence那里得到下一张票的票号，然后用这个票号创建一张票，并 返回该票。

TicketDispenser类的代码如下所示：

public class TicketDispenser {

public TurnTicket getTurnTicket(>

(

int newTurnNumber = TumNumberSequence. getNextTurnNumber ();

TurnTicket newTurnTicket = new TurnTicket(newTurnNumber);

return newTurnTicket;

最后一个类是TicketDispenserTest测试类，里面只有一个测试，来判断2+3等于5，用 来测试JUnit能否正常运行。把光标移动到这个测试类名上，然后按Ctrl+Shift+FlO组合键，

运行一下这个测试。通过。

读完代码后，现在可以写一些TODO,来记录一下接下来要做的事情。

“和上一个操练题目类似，我发现在TicketDispenser类中的getTumTicket()方法里， ‘ TumNumberSequence.getNextTumNumber(); ’ 这条语句调用 i" TumNumberSequence 类的 getNextTumNumber()静态方法，属于针对具体实现编程，而不是针对抽象编程，违反了依赖 倒置和开闭原则。”

没错。写个TODO记录下来。

在TicketDispenser类中添加依赖静态方法进行编程的TODO代码如下所不（CM: Added TODO: Depending on a static method violates the Dependency Inversion Principle and Open- Closed Principle.)：

public class TicketDispenser {

public TurnTicket getTurnTicket()

{

十 //TODO: Depending on a static method violates the Dependency Inversion

Principle and Open-Closed Principle, int newTurnNumber = TurnNumberSequence.getNextTurnNumber();

可以再写两个TODO来记录两个新特性：VIP客户的票号从1001开始，和普通 (Regular)客户的票号从2001开始。

在TicketDispenserTest类中添加两个有关新特性的TODO代码如下所7K (CM: Added two new feature TODOs for the different sequence starting numbers of VIP and regular customers.)：

public class TicketDispenserTest {

+ // TODO-new-feature: the turn number sequence of the vip customers starts from 1001

+ // TODO-new-feature: the turn number sequence of the regular customers

starts from 2001

}

能想出有什么用户意图的测试TODO吗？

“我能想到的有两个。一个是新出的票的票号要比前一张票的票号要大，另一个是新出 的票的票号要比在另一台取号机上所取的前一张票的票号要大。”

好的，写两个用户意图测试的TODO。

在TicketDispenserTest类中添加两个有关用户意图的测试TODO的代码如下所不（CM: Added 2 user intent test TODOs.)：

public class TicketDispenserTest {

+ // TODO-user-intent-test: a new ticket should have the turn number subsequent

to the previous ticket

+ // TODO-user-intent-test: a new ticket should have the turn number subsequent

to the previous ticket from another dispenser

我再添加一个需要做单元测试的TODO:如果给取号机11这个票号，那么取号机应该 能打印出一张印有11这个号码的票。

在TicketDispenserTest类中添加单元测试的TODO的代码如下所示（CM: Added an unit test TODO for class TicketDispenser.)：

public class TicketDispenserTest {

+ // TODO-unit-test: the ticket dispenser should dispense the ticket number

11 if give a turn number 11 to it

TODO写得差不多了，下面该挑一个简单的先做。那个“新票比前一张的票号要大”的 TODO看起来简单，先做它。写一个这个TODO的测试方法，再写它的Assert部分的意图代 码，来判断新票号要比前一张票号大1。

将新票比前一张的票号要大的TODO标记为working-on，并编写该TODO的测试和 Assert 部分的意图代码如下所7K ( CM: Working on TODO: a new ticket should have the turn number subsequent to the previous ticket. Wrote an assertion for its test.)：

public class TicketDispenserTest {

- // TODO-user-intent-test: a new ticket should have the turn number subsequent

to the previous ticket

+ // TODO-user-intent-test-working-on: a new ticket should have the turn number

subsequent to the previous ticket + @Test

+ public void a\_new\_ticket\_should\_have\_the\_turn\_number\_subsequent\_to\_the\_

previous一ticket(> {

+ // Assert

+ assertEquals(X, newTicket,getTurnNumber() - previousTicket.getTurnNumber());

+ }

在这行assertEquals()语句中，newTicket和previousTicket是有编译错误的红色，因为它 们都未创建。先把光标定位在newTicket上，按Alt+Enter快捷键来创建这个本地变量，类型 是TumTicket。它的值来自调用一个名为ticketDispenser的取号机对象的getTumTicket〇方法 的返回值。类似地再创建previousTicket这个本地变量。最后创建ticketDispenser这个本地变 量，它的类型是TicketDispenser,它的值来自new出来的一个TicketDispenser对象。写完代 码后，按Ctrl+F5快捷键运行测试，通过。这个TODO就完成了，可以删除。

完成新票比前一张的票号要大的TODO的代码如下所示（CM: Finished TODO: a new ticket should have the turn number subsequent to the previous ticket.)：

public class TicketDispenserTest {

@Test

- // TODO-user-intent-test-working-on: a new ticket should have the turn

number subsequent to the previous ticket public void a\_new\_ticket\_should\_have\_the\_turn\_number\_subsequent\_to\_the\_ previous\_ticket() {

+ // Arrange

+ TicketDispenser ticketDispenser = new TicketDispenser();

+ TurnTicket previousTicket = ticketDispenser.getTurnTicket();

+

+ //Act

+ TurnTicket newTicket = ticketDispenser.getTurnTicket()/

+

// Assert

assertEquals(1, newTicket.getTurnNumber() - previousTicket.getTurnNumber());

}

现在可以再做那个“新票比另一台取号机上出的前一张票的票号要大”的TODO 了。把 它标记为working-on,并编写相应的测试和Assert部分的意图代码。

将新票比另一台取号机上出的前一张票的票号要大的TODO标记为working-on,并编 写相应的测试和Assert部分的意图代码如下所示（CM: Working on TODO: a new ticket should have the turn number subsequent to the previous ticket from another dispenser. Wrote an assertion for its test.)：

public class TicketDispenserTest {

- // TODO-user-intent-test: a new ticket should have the turn number subsequent

to the previous ticket from another dispenser + // TODO-user-intent-test-working-on: a new ticket should have the turn number

subsequent to the previous ticket from another dispenser + @Test

+ public void a\_new\_ticket\_should\_have\_the\_turn\_number\_subsequent\_to\_the\_

previous\_ticket\_from\_another\_dispenser() {

+ // Assert

+ assertEquals(1, newTicket.getTurnNumber() - previousTicket.getTurnNumber());

+ }

在意图代码中，还是newTicket和previousTicket这两个变量是编译错误的红色。可以 用类似前面的做法推导出测试的Act和Arrange部分，只不过这一次previousTicket要出自 另一台取号机。写完代码后，按Ctrl+F5快捷键运行测试，通过。这个TODO就完成了，可 以删除。

完成新票比另一台取号机上出的前一张票的票号要大的TODO的代码如下所示（CM: Finished TODO: a new ticket should have the turn number subsequent to the previous ticket from another dispenser.)：

public class TicketDispenserTest {

- // TODO-user-intent-test-working-on: a new ticket should have the turn number

subsequent to the previous ticket from another dispenser @Test

public void a\_new\_ticket\_should\_have\_the\_turn\_number\_\_subsequent\_to\_the\_ previous—ticket一from\_another\_ciispenser () {

+ // Arrange

+ TicketDispenser anotherTicketDispenser = new TicketDispenser();

+ TicketDispenser ticketDispenser = new TicketDispenser();

+

+ //Act

+ TurnTicket previousTicket = anotherTicketDispenser.getTurnTicket();

+ TurnTicket newTicket = ticketDispenser.getTurnTicket();

+

// Assert

assertEquals(1, newTicket,getTurnNumber() - previousTicket.getTurnNumber());

}

现在该做那个单元测试的TODO 了：给取号机11就能打印出一张号码是11的票。还是 编写这个TODO的测试的Assert部分，断言出的票的号码是11。

将单元测试“给取号机11就能打印出一张号码是11的票”的TODO标记为working- on，并编写相应测试的Assert部分的代码如下所示（CM: Working on TODO ’the class Ticket­Dispenser should dispense the ticket number 11 if give a turn number 11 to it' and wrote the Assert part of its test.)：

public class TicketDispenserTest {

■ // TODO-unit-test: the class TicketDispenser should dispense the ticket number 11

if give a turn number 11 to it

+ // TODO-unit-test-working-on: the class TicketDispenser should dispense the

ticket number 11 if give a turn number 11 to it + @Test

+ public void the\_class\_TicketDispenser\_should\_dispense\_the\_ticket\_number\_

1l\_if\_give\_a\_turn\_number\_l() {

+ // Assert

+ assertEquals(11, ticket.getTurnNumber());

+ }

接下来要从测试的Assert部分推导出Act和Arrange部分了。Act部分和前面类似， ticket本地变量是TurnTicket类型，其值来自取号机对象ticketDispenser的getTumTicket〇方

法的返回值。而ticketDispenser本地变量是TicketDispenser类型，其值来自该类new出来的 一个对象。

这里，TicketDispenser 类是 SUT，而 TicketDispenser 类所依赖的 TumNumberSequence 类就是DOC。由于这个DOC中所保存的下一个票号在每一次获取票号后就自动增1，所 以在测试中难以控制让这个DOC就打印出一个票号为11的票。前面讲过，当SUT所依赖 的DOC很难在测试中进行控制时，就可以根据DOC提取一个接口，然后让SUT不再针对 DOC,而是针对这个接口来编程，并根据这个接口编写易于控制的Test Double,比如让它能 打印出11这个票号，来替代这个难以控制的DOC,从而令SUT把Test Double当成那个接 口来看待，进而测试这个SUT,从而解决DOC在测试中难以控制的问题。

前面咱们操练了接口提取的方法，现在咱们换一种方法来操练。这种方法叫子类化并 覆写方法。具体说来，后者不像前者那样需要根据DOC来提取接口，而是把DOC就当成 接口，让SUT根据DOC这个接口编程。然后编写DOC这个接口一个子类，不妨命名为 MockTumNumberSequence,并把 MockTuraNumberSequence 类当成易于控制的 Test Double， 且在这个子类中覆写其DOC父类中难以在测试中控制的方法。最后用这个子类来替代这个 难以控制的DOC父类。

“这个题目的DOC是TumNumberSequence类是一个具体的类，不是一个接口呀？”

这里TumNumberSequence类确实被定义为一^具体的类，而不是一个接口，但可以被 当成接口来用。在一个符合里氏替换原则e的父类和子类的继承关系中，父类就可以被看作 是子类的接口。

虽然咱们现在是在操练Mock,但Mock和Stub都属于Test Double,所以可以仿照前面 操练Stub的做法来做。现在要做的是，让TicketDispenser类把MockTumNumberSequence类 当成TumNumberSequence这个接口来看待。

虽然现在TicketDispenser类已经针对TumNumberSequence这个类编程了，但 TumNumberSequence这个类里面全是静态方法和静态成员变量，需要改造一下才能成为 一个好用的接口。在根据父类TumNumberSequence编写了子类MockTumNumberSequence 后，需要在SUT中让子类的对象替换掉父类的对象。所以需要通过TicketDispenser类的 带有TumNumberSequence类型的构造器，将一个MockTumNumberSequence对象注人 TicketDispenser 对象中。

咱们可以使用这个编程意图，来在测试中编写意图代码。先让形参为TumNumber­Sequence 的 TicketDispenser 类 的构造 器接受 mockTumNumberSequence 这个实 参类型 为 MockTumNumberSequence 的参数，然后让 mockTumNumberSequence 对象调用它

㊀里氏替换原则指的是，子类型必须能够替换掉它们的基类型.使得替换后程序行为不变。参见Robert C.

Martin所著《敏捷软件开发：原则、模式与实践》一书。

的arrangeNextTumNumber()方法，来将期望的票号11传进去。到此为止，除了子类化 和变量命名中有Mock之外，其他步骤都和前面Stub的操练很类似。但前面咱们讲到 了 Stub和Mock的区别。Mock要比Stub多做一件事，即验证SUT的间接输出。在这 里，我们可以让mockTurnNumberSequence这个Mock对象帮助验证，测试运行结束前，在 TurnNumberSequence这个接口中的getNextTumNumber()方法被调用了一次。所以在测试的 最后让 mockTurnNumberSequence 对象调用 verifyMethodGetNextTurnNumberCalledOnce()方 法，来验证这个SUT的间接输出。

完成 the\_dass\_TicketDispenser\_should\_dispense\_the\_ticket\_number\_l l\_if\_give\_a\_tum\_ number\_ll\_to\_it()测试方法的意图代码如下所示（CM: Wrote intention code for the test the\_ classTicketDispensershoulddispensetheticketnumberllifgiveaturnnumberl l\_to\_

public class TIcketDispenserTest {

// TODO-unit-test-working-on: the class TicketDispenser should dispense the ticket number 11 if give a turn number 11 to it QTest

public void the\_class\_TicketDispenser\_should\_dispense\_the\_ticket\_number\_ll\_ if\_give\_a\_turn\_number\_ll\_to\_it () {

+ // Arrange

+ MockTurnNumberSequence mockTurnNumberSequence = new MockTurnNumberSequence ();

+ mockTurnNumberSequence.arrangeNextTurnNumber(11);

+ TicketDispenser ticketDispenser = new TicketDispenser(mockTurnNumberSequence);

+

+ //Act

+ TurnTicket ticket = ticketDispenser.getTurnTicket();

+

// Assert

assertEquals(11, ticket.getTurnNumber());

+ mockTurnNumberSequence.verifyMethodGetNextTurnNumberCalledOnce();

}

写好了这个单元测试的意图代码后，看到一些红色的编译错误。先修复MockTurn- NumberSequence 这个红色的编译错误。创建 MockTurnNumberSequence 类。

创建MockTurnNumberSequence类以修复编程错误的代码如下所示（CM: Fixed compiler error: Created class MockTurnNumberSequence.)：

+public class MockTurnNumberSequence {

+ }

接下来修复arrangeNextTuraNumber()方法的红色编译错误，在MockTumNumber- Sequence类中创建该方法。

在 MoclcTumNumberSequence 类中创建方法 arrangeNextTumNumber()以修复编译错 误的代码如下所示（CM: Fixed compiler error: Created method MockTumNumberSequence. arrangeNextTurnNumber().)：

public class MockTumNumberSequence {

+ public void arrangeNextTurnNumber(int nextTurnNumber) {

+

+ )

}

下面该修复TicketDispenser类的构造器参数mockTumNumberSequence有一条红色下划 波浪线的编译错误了。在TicketDispenser类中创建带有TumNumberSequence类型参数的构 造器。

在TicketDispenser类中创建带有TumNumberSequence类型参数的构造器以修复编译错 误的代码如下所示（CM: Fixing compiler error: Created constructor TicketDispenser(TumNumbe rSequence).)：

public class TicketDispenser {

+ public TicketDispenser(TumNumberSequence turnNumberSequence) {

+

+ }

创建了一个带有参数的TicketDispenser类的构造器后，前面写的测试代码中“new TicketDispenserO;”语句中括号下面就出现了表示编译错误的红色波浪线。这需要咱们 再显示定义TicketDispenser类的默认构造器，来修复错误。在默认构造器中，可以调 用刚刚创建的带有TumNumberSequence类型参数的构造器，并传入一个new出来的 TumNumberSequence 对象 〇

创建TicketDispenser默认构造器以修复编译错误的代码如下所示（CM: Fixing compiler error: Created constructor TicketDispenser().)：

public class TicketDispenser {

+ public TicketDispenser() {

+ this(new TurnNumberSequence());

+ )

现在，在测试代码中，TicketDispenser类的构造器参数mockTumNumberSequence下面的 红色下划波浪线依然存在。用鼠标指向它看看编译错误，发现是MockTumNumberSequence 类尚未继承TumNumberSequence类。现在修复它。

让MockTuraNumberSequence类继承TumNumberSequence类以修复编译错误的代码如下 0T7Jn ( CM: Fixed compiler error: Made class MockTumNumberSequence extend class TumNumber- Sequence.)：

-public class MockTumNumberSequence {

♦public class MockTumNumberSequence extends TurnNumberSequence {

最后在测试代码中，就剩下尚未创建MockTumNumberSequence类的verifyMethodGetN extTuraNumberCalledOnce()方法这个编译错误了。修复它。

创建 MockTumNumberSequence 类的 verifyMethodGetNextTumNumberCalledOnce()方 法以修复编译错误的代码如下所示（CM: F"ixed compiler error: Created method MockTum- NumberSequence.verifyMethodGetNextTumNumberCalledOnce().)：

public class MockTumNumberSequence extends TurnNumberSequence {

+ public void verifyMethodGetNextTurnNumberCalledOnce() {

+

+ )

}

测试代码中的红色编译错误都没有了。可以按CtrRF5快捷键运行下测试。前两个测 试通过，最后这个正在写的测试失败，有断言错误。错误信息中说，期望值是11，但实际 值是4。测试失败的原因是咱们刚才修复编译错误的代码都是内部没有具体实现的空方法。 咱们可以从上往下检査一遍这个单元测试的TODO的测试代码，把那些空方法都实现完整 了〇 先实现 MockTumNumberSequence 类的 arrangeNextTumNumber()方法，把该方法传人的 nextTumNumbei•参数的值保存到该类的成员变量中。

实现 MockTumNumberSequence 类的 arrangeNextTumNumber〇 方法以修复断言错误 的代码如下所示（CM: Fixing assertion error: Saved the parameter nextTumNumber of method MockTumNumberSequence.arrangeNextTumNumberO into the field nextTuraNumber of class MockTumNumberSequence.)：

public class MockTumNumberSequence extends TurnNumberSequence {

+ private int nextTumNumber;

public void arrangeNextTurnNumber(int nextTumNumber) {

+ this .nextTumNumber = nextTumNumber;

接下来继续实现这个测试代码中下一个空方法 TicketDispenser类的带MockTum-

NumberSequence类型参数的构造器，把构造器传进的参数tumNumberSequence的值保存在 该类的成员变量中。

实现TicketDispenser类的带MockTumNumberSequence类型参数的构造器以修复断言 错误的代码如下所示（CM: Fixing assertion error: Saved the parameter tumNumberSequence of constructor TicketDispenser(TumNumberSequence) into the field tumNumberSequence of class TicketDispenser.)：

public class TicketDispenser {

+ private TumNumberSequence tumNumberSequence;

public TicketDispenser (TumNumberSequence tumNumberSequence) { + this. tumNumberSequence = tumNumberSequence;

测试代码接下来就该调用TicketDispenser类的getTurnTicket()方法了。这个方法虽然不 是空方法，但是里面有咱们先前写的依赖静态方法进行编程的TODO。现在也到了处理这个 TODO的时候了。把调用TumNumberSequence类的静态方法getNextTumNumber()的语句， 改为调用tumNumberSequence对象的成员方法getNextTumNumber()这样的意图代码。另夕卜， 在TumNumberSequence类中，把getNextTurnNumber()方法前面的static修饰去掉，从而让 其子类可以对其进行覆写。这样就完成了处理依赖静态方法进行编程的TODO。

完成依赖静态方法进行编程的TODO以修复断言错误的代码如下所示（CM: Fixing assertion error: Finished TODO: Depending on a static method violates the Dependency Inversion Principle and Open-Closed Principle.)：

public class TicketDispenser {

public TurnTicket getTurnTicket()

{

* //TODO: Depending on a static method violates the Dependency Inversion

Principle and Open-Closed Principle.

* int newTurnNumber « TumNumberSequence.getNextTurnNumber()/

+ int newTurnNumber = tumNumberSequence. getNextTurnNumber ();

public class TumNumberSequence {

private static int \_turnNumber = 0;

- public static int getNextTurnNumber〇

+ public int getNextTurnNumber()

{

return turnNumber++;

最后，MockTumNumberSequence 这个子类尚未覆写父类 TumNumberSequence 的 get- NextTumNumber()方法，现在给加上，让它直接返回MockTumNumberSequence类成员变量 中保存的下一张票号nextTuraNumber。这一步操作就是‘子类化并覆写方法’命名的由来吧。

MockTumNumberSequence 类覆写父类 TumNumberSequence 的 getNextTumNumber()方 法的代码如下所7K (CM: Fixed assertion error: Overrode method getNextTumNumber() in class MockTumNumberSequence.)：

public class MockTumNumberSequence extends TumNumberSequence {

+

+ @Override

+ public int getNextTurnNumber() {

+ return this.nextTurnNumber;

+ }

}

按Ctrl+F5快捷键运行测试，通过。现在完成了那个单元测试的TODO,可以把它删除。 接下来可以处理第一个新特性的TODO: VIP客户的票号从1001开始。可以把这个TODO 标记为working-cm，并写下其对应的测试的Assert部分，判断取出的票号为1001。

完成给出11就打印出11号票的单元测试的TODO,并把新特性TODO V1P客户的票号 从1001开始标记为working-on，且编写其测试的Assert部分的代码如下所7K ( CM: Finished TODO: the class TicketDispenser should dispense the ticket number 11 if give a turn number 11 to it. Working on TODO: the turn number sequence of the vip customers starts from 1001. Wrote a test with the Assert part of it.)：

public class TicketDispenserTest {

* // TODO-unit-test-working-on: the class TicketDispenser should dispense the

ticket number 11 if give a turn number 11 to it

* // TODO-new-feature: the turn number sequence of the vip customers starts

from 1001

+ // TODO-new-feature-working-on: the turn number sequence of the vip customers

starts from 1001 + @Test

+ public void the—turn一nunber\_sequence一of\_the一vip\_custoroers一starts一from\_1001 () { + // Assert

+ assertEquals(1001, ticket.getTurnNumber());

+ }

下面接着写这个测试的Act和Arrange的意图代码。

“等等，MockTumNumberSequence 类的 verifyMethodGetNextTumNumberCalledOnce()方

法还没实现呢。”

哦，对，忘记实现了。顺便写个TODO记下来。

完成 the\_tum\_number\_sequence\_of\_the\_vip\_customers\_starts\_from\_1001()测试的 Act 和 Arrange 部分的意图代码并添力口实现 MockTumNumberSequence 类的 verifyMethodGetNextTur nNumberCalledOnce()方法的 TODO 的代码如下所示（CM: Wrote the intention code of Act and Arrange parts of the test the turn—number sequence of the vip customers starts from l 001 (). Added TODO: Finish the implementation of method MockTumNumberSequence.verifyMethodGet NextTumNumberCalledOnce〇.)：

public class TicketDispenserTest {

// TODO-new-feature-working-on: the turn number sequence of the vip customers starts from 1001 @Test

public void the^turn\_number\_sequence\_of\_the\_vip\_customers\_starts\_from\_1001() { + II Arrange

+ TurnNumberSequence vipCustomerTumNumberSequence = new TurnNumberSequence(1001);

+ TicketDispenser vipCustomerTicketDispenser = new TicketDispenser

(vipCustomerTumNumberSequence);

+

+ //Act

+ TurnTicket ticket = vipCustomerTicketDispenser.getTurnTicket();

+

// Assert

assertEquals(1001f ticket.getTurnNumber());

}

public class MockTurnNumberSequence extends TurnNumberSequence {

+ //TODO: Finish the implementation of method MockTurnNumberSequence. verifyMethodGetNextTurnNumberCalledOnce(). public void verifyMethodGetNextTurnNumberCalledOnce() {

}

在刚刚写的测试的意图代码中，“newTumNumberSequence(lOOl);”语句中1001下面 有表示编译错误的红色波浪下划线，因为TurnNumberSequence类还没有整型参数的构造器。 现在修复它，在这个构造器中，把传人的firstNumber参数值赋值给TurnNumberSequence类 的turaNumber静态成员变量。

在类TurnNumberSequence类中创建带有整型参数的构造器以修复编译错误的代码如下所 7^ (CM: Fixed compiler error: Created and implemented constructor TumNumberSequence(int).)：

public class TurnNumberSequence {

private static int \_turnNumber = 0;

+ public TurnNumberSequence(int firstNumber) {

+ this.\_turnNumber = firstNumber;

+ }

运行测试，发现编译错误。原因是刚刚新增了带有int类型参数的TurnNumberSequence 类的构造器后，忘记显式地定义默认的TurnNumberSequence类的构造器了。现在加上，让 默认的构造器调用带有int类型参数的构造器，并传递参数0。

创建并实现TurnNumberSequence类的默认构造器以修复编译错误的代码如下所示（CM: Fixed compiler error: Created and implemented constructor TumNumberSequence().)：

public class TurnNumberSequence {

+ public TurnNumberSequence() {

+ this (0);

+ }

运行测试，通过。现在完成了新特性TODO VIP客户的票号从1001开始，可以把它删 除。接下来处理刚刚新加上实现MockTumNumberSequence类的verifyMethodGetNextTum NumberCalledOnce()方法的 TODO，把它标记为 working-on。在 verifyMethodGetNextTum NumberCalledOnceO方法中，可以先编写意图代码，即如果记录方法调用次数的成员变量 callsCount 不是 1 的话，就拋 IllegalStateException 异常。

在 MockTumNumberSequence 类的 verifyMethodGetNextTumNumberCalledOnce()方法 中，编写意图代码的代码如下所示（CM: Wrote intention code for method MockTumNumber- Sequence.verifyMethodGetNextTumNumberCalledOnce().)：

public class MockTurnNumberSequence extends TurnNumberSequence {

// TODO-working-on: Finish the implementation of method MockTurnNumberSequence.

verifyMethodGetNextTurnNumberCalledOnce(). public void verifyMethodGetNextTurnNumberCalledOnce() {

+ if (this.callsCount != 1) {

+ throw new IllegalStateException (\*\*The method getNextTurnNumber ()

should be called once.")；

+ }

在这段意图代码中，callsCount是有编译错误的红色，因为尚未创建该成员变量。现在 就创建，并初始化为0。

在MockTurnNumberSequence类中创建并初始化成员变量callsCount以修复编译错误

的代码如下所本（CM: Fixed compiler error: Created and initialized field callsCount in class MockTumNumberSequence.)：

public class MockTumNumberSequence extends TurnNumberSequence { private int nextTurnNumber;

+ private int callsCount = 0;

运行测试，失败，发现 IllegalStateException 异常。原因是在 MockTumNumberSequence 类中，方法getNextTumNumber()被调用时，尚未将callsCount这个成员变量增1。现在修复 它。再运行测试，通过。现在就完成了实现MockTumNumberSequence类的verifyMethodGet NextTumNumberCalledOnce()方法的 TODO，可以删除掉。

完成实现 MockTumNumberSequence 类的 verifyMethodGetNextTumNumberCalledOn ce〇 方法的 TODO 的代码如下所示（CM: Fixed IllegalStateException: Made callsCount plus one in method MockTurnNumberSequence.getNextTurnNumber(). Finished TODO: Finish the implementation of method MockTumNumberSequence.verifyMethodGetNextTuraNumberCalledO nce().)：

public class MockTumNumberSequence extends TurnNumberSequence {

- // TODO-working-on: Finish the implementation of method MockTumNumberSequence.

verifyMethodGetNextTurnNumberCalledOnce().

public void verifyMethodGetNextTurnNumberCalledOnce() { if (this.callsCount != 1) {

throw new IllegalStateException("The method getNextTurnNumber() should be called once.;

©Override

public int getNextTurnNumber() { + this.callsCount++;

return this.nextTurnNumber;

类似地，咱们可以编写下一个新特性TODO普通客户的票号从2001开始的测试。运行 测试，通过。可以删除这个TODO 了。

新特性TODO普通客户的票号从2001开始的代码如下所示（CM: Finished TODO: the turn number sequence of the regular customers starts from 2001.)：

public class TicketDispenserTest {

// TODO-new-feature: the turn number sequence of the regular customers starts from 2001

+ @Test

+ public void the\_turn\_number\_sequence\_of\_the\_regular\_customers\_starts\_from\_2001 () {

+ // Arrange

+ TurnNumberSequence regularTurnNumberSequence = new TurnNumberSequence(2001)/

+ TicketDispenser regularCustomerTicketDispenser = new TicketDispenser

(regularTurnNumberSequence);

+

+ //Act

+ TurnTicket ticket = regularCustomerTicketDispenser.getTurnTicket();

+

+ // Assert

+ assertEquals(2001, ticket.getTurnNumber());

+ }

}

“我发现在TurnNumberSequence类中，成员变量\_tumNumber的命名以下划线开头，不 大符合Java代码命名规范，可以把这个下划线去掉。”

去掉TurnNumberSequence类中成员变量tumNumber的命名中开始部分的下划线的代码 如下所不（CM: Renamed field tumNumber of class TurnNumberSequence to be tumNumber.):

public class TurnNumberSequence {

* private static int 一turnNumber = 0;

+ private static int turnNumber = 0;

public TurnNumberSequence(int firstNumber) {

* this•一turnNumber = firstNumber;

+ this.turnNumber = firstNumber;

public int getNextTurnNumber()

{

- return \_turnNumber++;

+ return turnNumber++;

运行测试，通过。

“咱们最后写的那两个新特性的测试里面，分别出现了两次1001和2001这两个魔法数。 一方面每个数重复了两遍，另一方面魔法数本身不好理解其意。最好能提取出一个常量。” 没错。可以把这两个常量放到TurnNumberSequence类中，并且起一个描述性强的名字。 消除1001和2001这两个魔法数的代码如下所示（CM: Eliminated magic number code smell for VIP customer first number 1001 and regular customer first number 2001.)：

public class TurnNumberSequence {

+ public static final int REGULAR\_CUSTOMER\_FIRST一NUMBER = 2001;

+ public static final int VIP\_CUSTOMER\_FIRST\_NUMBER = 1001;

public class TicketDispenserTest {

@Test

public void the一turn一number一sequence一of\_the\_vip一customers—starts\_from\_1001 () { // Arrange

- TumNumberSequence vipCustomerTurnNumberSequence = new TurnNumberSequence (1001);

+ TurnNumberSequence vipCustomerTurnNumberSequence = new TurnNumberSequence

(TurnNumberSequence.VIP\_CUSTOMER\_FIRST\_NUMBER);

// Assert

- assertEquals(1001r ticket.getTurnNumber());

+ assertEquals(TurnNumberSequence.VIP一CUSTOMER—FIRST一NUMBER, ticket.

getTurnNumber());

@Test

public void the\_turn\_number\_sequence\_of\_the\_regular\_customers\_starts\_from\_2001 () { // Arrange

- TurnNumberSequence regularTurnNumberSequence = new TurnNumberSequence(2001);

+ TurnNumberSequence regularTurnNumberSequence = new TurnNumberSequence

(TurnNumberSequence.REGULAR\_CUSTOMER\_FIRST\_NUMBER);

// Assert

- assertEquals(2001, ticket.getTurnNumber());

+ assertEquals(TurnNumberSequence.REGULAR一CUSTOMER一FIRST一NUMBER, ticket.

getTurnNumber());

这个针对Mock和子类化并覆写的操练就可以告一段落了。

“咱们前面做的Mock操练是手写Mock。现在有很多有关Mock的开源框架，能方便地 写Mock。要不咱们也操练一下？”

当然可以。咱们不妨试试Mockito。因为咱们现在已经把Mock手写完毕了，为了操练 Mockito，咱们需要执行一个git reset-hard命令，来回退到手写Mock之前的代码版本，即刚 刚完成“新票比另一台取号机上出的前一张的票号要大”的TODO的代码版本。然后在这个 代码版本下，用git checkout -b命令再新创建一个名为using-mockito-error-fixing-oriented分 支0，这样咱们操练Mockito的代码就不会与前面的代码发生冲突。

好了，现在已经回退到刚刚完成“新票比另一台取号机上出的前一张的票号要大”的 TODO的代码版本。然后开始处理那个单元测试“给取号机11就能打印出一张号码是11的

㊀针对 Mockito 的操练代码参见：https://github\_com/wubin28/tbc-ticket-dispenser-java/tree/using-mockito-eiT〇r- fixing-oriented。

票”的TODO 了。将其标记为working-on,并编写相应测试的Assert部分的意图代码。这些 步骤和前面是一样的。

将“给取号机11就能打印出一张号码是11的票”的TODO标记为working-on,并编写 相应测试的Assert部分的意图代码如下所示（CM: Working on TODO: the class TicketDispenser should dispense the ticket number 11 if give a turn number 11 to it. Wrote an assertion for its test.)：

public class TicketDispenserTest {

- // TODO-unit-test: the class TicketDispenser should dispense the ticket

number 11 if give a turn number 11 to it

+ // TODO-unit-test-working-on: the class TicketDispenser should dispense the

ticket number 11 if give a turn number 11 to it

+ @Test

+ public void the\_class\_TicketDispenser\_should\_dispense\_the\_ticket\_nuiriber\_ll\_ if\_give\_a\_turn\_number\_ll\_to\_it () {

+ // Assert

+ assertEquals(11, ticket.getTurnNumber());

+ )

接下来就该写这个测试的Act和Arrange部分的意图代码了。Act部分和前面的一样， ticket本地变量是TumTicket类型，其值来自取号机对象ticketDispenser的getTuraTicket()方 法的返回值。而ticketDispenser本地变量是TicketDispenser类型，其值来自该类new出来的 一个对象。

因为咱们要使用Mockito,所以接下来的代码就开始和以前有些不一样了，但是总体思 路是一样的。咱们现在使用Mockito的编程意图，来在测试中编写意图代码，并与前面手写 Mock的代码进行对比。

现在，形参为TumNumberSequence的TicketDispenser类的构造器接受的mockTum- NumberSequence 实参，不再是 new 出来的 MockTurnNumberSequence 对象，而是从 Mockito 框架中调用 mock(TumNumberSequence.class)所创建的 Mock 对象。

接下来将期望的票号11传进Mock对象的方式，不再是让mockTumNumberSequence 对象调用它的arrangeNextTumNiimber()方法，而是调用Mockito框架的when(mock- TuraNumberSequence.getNextTumNumber()).thenRetum(ll)方法来完成。来将期望的票号 11 传进去。

最后在测试运行结束前，验证TumNumberSequence这个接口中的getNextTumNumber() 方法被调用了一次这个验证SUT的间接输出的方式，不再是让mockTumNumberSequence 对象调用 verifyMethodGetNextTumNumberCalledOnceO 方法，而是调用 Mockito 框架的 verify(mockTumNumberSequence).getNextTumNumber()方法。

完成 the\_class\_TicketDispenser\_should\_dispense\_the\_ticket\_number\_l l」f\_give\_a tum number—

ll」o\_it()测试方法的有关Mockito的意图代码如下所示（CM: Wrote the intention code for the Act, Arrange and Assert parts of the test the\_class\_TicketDispenser\_should\_dispense\_the\_ticket\_ numberl l\_if\_give\_a\_tum\_number\_l l\_to\_it().)：

public class TicketDispenserTest {

// TODO-unit-test-working-on: the class TicketDispenser should dispense the ticket number 11 if give a turn number 11 to it @Test

public void the\_class一TicketDispenser一should\_dispense一the\_ticket一number一11— if\_give\_a\_turn\_number\_ll\_to\_it () {

+ // Arrange

+ TurnNumberSequence mockTurnNumberSequence = mock(TurnNumberSequence.class);

+ when(mockTurnNumberSequence.getNextTurnNumber()).thenReturn(11);

+ TicketDispenser ticketDispenser = new TicketDispenser(mockTurnNumberSequence);

+

**+ //Act**

+ TurnTicket ticket = ticketDispenser.getTurnTicket();

+

// Assert

assertEquals(11, ticket.getTurnNumber());

+ verify(mockTurnNumberSequence).getNextTurnNumber();

}

写完测试的意图代码后，发现代码中mock、when和verify这些方法名都是编译错误的 红色。这是因为Mockito的Jar包还没有通过Maven引人。需要修改pom.xml文件，并把这 些方法静态import进来。

在pom.xml文件中添加Mockito的依赖，并在TicketDispenserTest类中静态import相 关方法的代码如下所示（CM: Added mockito dependency in the pom.xml and imported related methods statically.)：

<project>

〈dependencies〉

+ <dependency>

+ <groupId>org.mockito</groupId>

+ <artifactId>mockito-core</artifactId>

+ <version>l.9.5</version>

+ <scope>test</scope>

+ </dependency>

〈/dependencies〉

</project>

+import static org.mockito.Mockito.mock;

♦import static org.mockito.Mockito.verify;

•♦•import static org. mockito. Mockito .when;

写完上面的代码，在IDEA中单击在右上角随即出现的Import Changes按钮，代码中 mock、when和verify这些方法名的编译错误的红色就消失了。现在，只剩下new TicketDis penser(mockTumNumberSequence);语句中 mockTuraNumberSequence 下面的红色下划波浪线 的编译错误了。这是由于带有TumNumberSequence类型参数的TicketDispenser类的构造器 还未创建。现在就创建它。

创建带有TumNumberSequence类型参数的TicketDispenser类的构造器以修复编译错误 的代码如下所示（CM: Fixed compiler error: Created constructor TicketDispenser(TumNuniberSe quence).)：

public class TicketDispenser {

+ public TicketDispenser(TumNumberSequence turnNumberSequence) {

+

按Ctrl+F5快捷键运行测试，有编译错误d发现刚刚添加的带有参数的TicketDispenser 类的构造器，让原先调用默认的TicketDispenser类的构造器的语句无法工作。现在就显式定 义默认的TicketDispenser类的构造器。

显式定义默认的TicketDispenser类的构造器来修复编译错误的代码如下所示（CM: Fixed compiler error: Created constructor TicketDispenser().)：

public class TicketDispenser {

+ public TicketDispenser() {



现在编译错误没有了，按Ctrl+F5快捷键运行测试。遇到了 MissingMethodlnvocationEx ception，意思是说 “ when(mockTumNumberSequence.getNextTumNumber〇).thenRetum( 11); ” 这条语句中，when()方法中的参数必须是一个能够在Mock对象上调用的方法。把光标移动 到when〇方法中的getNextTuraNumberO上，按Ctrl+Alt+G组合键来看这个方法的定义，看 到TumNumberSequence类中的getNextTumNumber()是一个静态方法，所以无法被Mock对 象调用。现在就把getNextTumNumber()方法前面的static修饰去掉。

将TumNumberSequence类中的getNextTumNumber()方法改为非静态的以修复 MissingMethodlnvocationException 异常的代码如下所示（CM: Fixed MissingMethodlnvocati onException: Made method TumNumberSequence.getNextTumNumber() non-static so that mock

framework can call it.)：

public class TurnNumberSequence {

* public static int getNextTurnNumber()

+ public int getNextTurnNumber()

{

return 一turnNumber++;

}

按CtrH~F5快捷键运行测试。发现编译错误，原因是刚刚把TumNumberSequence类中 的 getNextTuraNumber()方法改为非静态的了，所以在 TicketDispenser 类的 getTumTicket() 方法中对它的调用就不能是直接调用静态方法了，需要使用tumNumberSequence这个对 象来调用这个方法，而这个tumNumberSequence对象需要在TicketDispenser类中创建 TumNumberSequence类型的成员变量〇

把在 TicketDispenser 类的 getTumTicket()方法中对 TumNumberSequence 类中的 getNextTumNumber()方法的调用改为非静态的代码如下所示（CM: Fixed compiler error: Used an object to call a non-static member method.)：

public class TicketDispenser {

+ private TumNumberSequence tumNumberSequence;

public TurnTicket getTurnTicket()

{

// TODO: Depending on a static method violates the Dependency Inversion Principle and Open-Closed Principle.

* int newTurnNumber = TumNumberSequence.getNextTurnNumber();

+ int newTurnNumber = tumNumberSequence. getNextTurnNumber ();

再次运行测试，在 TicketDispenser 类的 getTuraTicket()方法中的 tumNumberSequence. getNextTumNumber()语句中发现了空指针异常。这是由于咱们的TicketDispenser(TuraNumb erSequence)构造器是空的，还未保存从参数传进的tumNumberSequence。现在就修复它。

实现TicketDispenser(TumNumberSequence)构造器以修复空指针异常的代码如下所示 (CM: Fixed null pointer exception: Implemented constructor TicketDispenser(TumNumberSequence).)：

public class TicketDispenser {

private TumNumberSequence tumNumberSequence;

public TicketDispenser (TumNumberSequence tumNumberSequence) {

+ this.tumNumberSequence - tumNumberSequence;

运行测试。“给取号机11就能打印出一张号码是11的票”的TODO的测试运行通过了。

前面两个新票票号比前一张票号大1的测试拋出空指针异常，抛出异常的语句和前面是一样 的。这是因为TicketDispenser的默认构造器还未实现，需要从这个默认构造器中调用那个带 参数的构造器，并传递一个新new出来的TumNumberSequence对象。

实现TicketDispenser〇默认构造器以修复空指针异常的代码如下所示（CM: Fixednull pointer exception: Implemented constructor TicketDispenser(). )：

public class TicketDispenser {

public TicketDispenser() {

+ this(new TurnNumberSequence());

}

运行测试，通过。这样，这个有关Mockito的操练中关于依赖静态方法的TODO和“给 取号机11就能出一张号码是11的票”的TODO都完成了。可以把它们删除。

至此，这个有关自动取号系统的编程操练就告一段落。为了便于和重构前的代码进行比 较，下面列出重构后的测试代码和生产代码e。

手写Mock的TicketDispenserTest类在重构后的代码如下所

public class TicketDispenserTest {

@Test

public void a\_new\_ticket一should\_have一the\_turn\_number一subsequent一to\_the一 previous一ticket() {

// Arrange

TicketDispenser ticketDispenser = new TicketDispenser();

TurnTicket previousTicket = ticketDispenser.getTurnTicket();

//Act

TurnTicket newTicket = ticketDispenser.getTurnTicket();

// Assert

assertEquals(1, newTicket.getTurnNumber(> - previousTicket.getTurnNumber());

}

@Test

public void a\_new\_ticket\_should\_have\_the\_turn\_number\_subsequent\_to\_the\_ previous\_ticket\_from\_another\_dispenser() {

// Arrange

TicketDispenser anotherTicketDispenser = new TicketDispenser(); TicketDispenser ticketDispenser = new TicketDispenser();

//Act

TurnTicket previousTicket = anotherTicketDispenser.getTurnTicket();

© TurnTicket类在重构中未发生变化，故略去。

TurnTicket newTicket = ticketDispenser.getTurnTicket();

II Assert

assertEquals(1, newTicket.getTurnNumber() - previousTicket.getTurnNumber());

}

@Test

public void the\_class\_TicketDispenser\_should\_dispense\_the\_ticket\_number\_l1\_ if\_give\_a\_turn\_number\_ll\_to\_it () {

// Arrange

MockTurnNumberSequence mockTurnNumberSequence = new MockTurnNumberSequence(); mockTurnNumberSequence.arrangeNextTurnNumber(11);

TicketDispenser ticketDispenser = new TicketDispenser(mockTurnNumberSequence); //Act

TurnTicket ticket - ticketDispenser.getTurnTicket();

// Assert

assertEquals(11, ticket.getTurnNumber{));

mockTurnNumberSequence.verifyMethodGetNextTurnNumberCalledOnce();

}

@Test

public void the\_turn\_number一sequence一of\_the一vip\_customers\_starts\_from一1001 () { // Arrange

TurnNumberSequence vipCustomerTurnNumberSequence = new TurnNumberSequence (TurnNumberSequence.VIP\_CUSTOMER\_FIRST\_NUMBER);

TicketDispenser vipCustomerTicketDispenser = new TicketDispenser (vipCustomerTurnNumberSequence);

//Act

TurnTicket ticket = vipCustomerTicketDispenser.getTurnTicket();

// Assert

assertEquals(TurnNumberSequence.VIP一CUSTOMER一FIRST\_NUMBER, ticket. getTurnNumber());

}

@Test

public void the\_turn\_number\_sequence\_of\_the\_regular\_customers\_starts\_from\_2001 () { // Arrange

TurnNumberSequence regularTurnNumberSequence = new TurnNumberSequence (TurnNumberSequence.REGULAR\_CUSTOMER\_FIRST\_NUMBER);

TicketDispenser regularCustomerTicketDispenser = new TicketDispenser (regularTurnNumberSequence);

//Act

TurnTicket ticket = regularCustomerTicketDispenser.getTurnTicket();

II Assert

assertEquals(TurnNumberSequence.REGULAR\_CUSTOMER\_FIRST\_NUMBER, ticket. getTurnNumber());

使用Mockito编写Mock的TicketDispenserTest类在重构后的代码如下所示:

public class TicketDispenserTest {

@Test

public void the\_class\_TicketDispenser\_should\_dispense\_the\_ticket\_number\_ll\_ if\_give\_a—turn—number」l\_to一it (} {

// Arrange

TurnNumberSequence mockTurnNumberSequence = mock(TurnNumberSequence.class); when(mockTurnNumberSequence.getNextTurnNumber()).thenReturn(11); TicketDispenser ticketDispenser = new TicketDispenser(mockTurnNumberSequence);

//Act

TurnTicket ticket = ticketDispenser.getTurnTicket();

// Assert

assertEquals(11, ticket.getTurnNumber());

verify(mockTurnNumberSequence).getNextTurnNumber();

// TODO-new-feature: the turn number sequence of the vip customers starts from 1001

// TODO-new-feature: the turn number sequence of the regular customers starts from 2001

}

重构后的TurnNumberSequence的代码如下所示：

public class TurnNumberSequence {

public static final int REGULAR\_CUSTOMER一FIRST一NUMBER = 2001; public static final int VIP一CUSTOMER\_FIRST一NUMBER = 1001; private static int turnNumber = 0;

public TurnNumberSequence(int firstNumber) { this.turnNumber = firstNumber;

public TurnNumberSequence() { this (0);

public int getNextTurnNumber()

return turnNumber++;

重构后的MockTumNumberSequence类的代码如下所示：

public class MockTurnNumberSequence extends TurnNumberSequence { private int nextTurnNumber; private int callsCount = 0;

public void arrangeNextTurnNumber(int nextTurnNumber) { this.nextTurnNumber = nextTurnNumber;

}

public void verifyMethodGetNextTurnNumberCalledOnce() { if (this.callsCount != 1) {

throw new IllegalStateException(wThe method getNextTurnNumber〇 should be called once.n);

QOverride

public int getNextTurnNumber{) {

this.callsCount++; return this.nextTurnNumber;

重构后的TicketDispenser类的代码如下所示：

public class TicketDispenser {

private TurnNumberSequence turnNumberSequence;

public TicketDispenser(TurnNumberSequence turnNumberSequence) { this.turnNumberSequence = turnNumberSequence;

public TicketDispenser() {

this(new TurnNumberSequence());

}

public TurnTicket getTurnTicket()

{

int newTurnNumber = turnNumberSequence.getNextTurnNumber(); TurnTicket newTurnTicket = new TurnTicket(newTurnNumber);

return newTurnTicket;

在操练了 Mock编写和子类化并覆写的编写单元测试的方法后，咱们可以最后再做一个 有关如何针对涉及文件系统操作和使用第三方类库进行单元测试的操练题目，因为这两个问 题在编程中很常见。不过在继续操练前，咱们看看这个操练都做了哪些工作。

1)阅读自动取号系统的源代码，包含TumTicket类、TumNumberSequence类和Ticket- Dispenser 类0

1. )记录了调用静态方法从而违反依赖倒置和开闭原则的TODO。
2. )记录了两个新特性的TODO。

4)记录了两个用户意图测试的TODO。

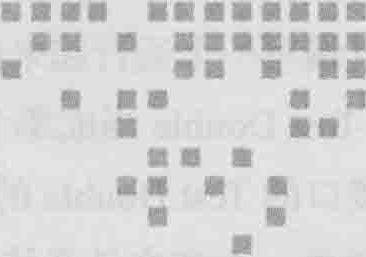
5 )记录了一个单元测试的TODO。

1. 先写了测试的Assert部分的意图代码，然后再推导出Act和Arrange部分的测试意 图代码，并通过修复意图代码的编译和测试运行错误，来驱动出生产代码，从易到难地完成 上述TODO。
2. 使用子类化并覆写的方法，把DOC就当成接口，让SUT根据DOC这个接口编程， 然后编写DOC这个接口一个子类，并把该子类当成易于控制的Mock,且在这个Mock子 类中覆写其DOC父类中难以在测试中控制的方法，最后用这个子类来替代这个难以控制的 DOC父类。

8 )在手工编写Mock完成单元测试后，又使用了 Mockito这个Mock开源框架来编写Mock。

1. 通过操练我们学到了以下技能：
2. —个在面向对象编程语言中所定义的具体的类，虽然在形式上不是一个接口，但如果 该类有一个符合里氏替换原则的子类，那么在子类眼中，父类就可以被当成接口来使用。
3. Mock除了完成Stub所做的为SUT在测试中的运行提供间接输人外，还要额外做验 证SUT在测试中的间接输出的事情，所以Mock类中一般都有verify()这样的验证方法。
4. 使用先编写意图代码，然后通过频繁运行测试，编写最少量的代码（如能让编译通过 的尚未实现的空方法，和能让测试运行通过的上述空方法的最少量的实现代码）以修复编译 和测试运行错误，来驱动出生产代码的开发方式，既能让我们在编写意图代码时进行适当的 设计，并能编写最少量的能让测试运行通过的生产代码，以减少浪费，又能让我们逐步熟悉 从各种编译和测试运行的错误信息中发现导致错误的原因，从而不再惧怕看到错误信息，并 把这些信息当成指路的向导，避免盲目地编写生产代码。
5. 与手工编写Mock相比，用Mockito框架编写Mock有下列好处，不必编写额外的 Mock类，减少工作量；凭借Mockito框架的诸如mock()、when〇和verifyO这些方法的精心 设计，能够让Mock代码可读性更高。
6. 与用Mockito框架编写Mock相比，手工编写Mock也有下列好处，能够完全控制 Mock的编写过程；熟悉手工编写Mock的过程后，有利于理解Mock框架的使用方法。

真正的单元测试



**第18章**

/S

现在中国人出国旅游，要买点称心的纪念品带回来，难度可比以前大多了。

“可不是！我的一个球迷朋友前一阵刚刚参加完世界杯旅游，从巴西回来。他跟我说， 无论是在里约热内卢的世界杯纪念品官方旗舰店里，还是街头的小贩手里，像吉祥物‘福 来哥’、球队队服、球迷围巾这些抢手的纪念品，后面都印着Made in China或Fabricadona China e。搞得他一直在纠结是不是该把这些中国造的东西再千里迢迢地背回国去。”

是呀，要买这些纪念品，最经济快捷的办法是上淘宝，足不出户就能搞定。

与人们用是否经济快捷的标准来衡量购买世界杯纪念品是值还是不值相似，测试也可以 根据是否经济快捷这一点，来划分为真正的单元测试和非单元测试。Michael C. Feathers干脆 把经济快捷这一点浓缩为一个字：“快”。

单元测试运行得快。运行得不快的不是单元测试。一个需要耗时0.1秒才能执行完的单元测试就已算是一个 慢的单元测试了。

有些测试容易跟单元测试混淆起来^比如下面这些测试就不是单元测试：

1>跟数据库有交互；

1. 进行了网络间的通信；
2. 调用了文件系统；
3. 需要对环境做特定的准备（如编辑配置文件）才能运行起来。®

“哦，这样看起来我以前曾经写过的涉及文件系统和数据库的测试，都不能算单元测试， 而只能算集成测试了。那么如何才能把这些集成测试转变为单元测试呢？”

单元测试要测的仅仅是SUT里面的软件行为，而不是测试SUT与诸如数据库、网络和

©葡萄牙语中国制造的意思。

㊁Michael C. Feathers著，刘未鹏译，《修改代码的艺术>,人民邮电出版社，2007年11月第1版，第12页。文件系统这些DOC之间是否能正常交互。可以先找到DOC的接口，然后让SUT针对这个 接口编程，并且编写一个运行起来经济快捷的Test Double来实现这个接口，并注人SUT中， 让SUT把这个Test Double当成那个接口来使用。这样就能把依赖于DOC的集成测试，转换 为依赖于实现接口的Test Double的单元测试了。

咱们现在就做一个和文件系统相关的编程题目，来操练一下。这个题目是网页文本转换 系统' 只有一个类UnicodeFileToHtmlTextConverter,用来将一个Unicode的纯文本文件转换 为HTML编码（HTML-encoded)格式的文件。这个操练要求在为这个类编写单元测试的基础 上，实现一个新特性：能够将一个Unicode的纯文本字符串转换为HTML编码格式的字符串。

在这个题目中，网页文本转换系统是SUT，转换前用的有关文件的FikReader对象就 是DOC。根据前面的讨论，与文件发生交互的测试不是单元测试。要编写单元测试，可以 从DOC提取一个接口，即可以利用现成的Java语言的Reader抽象类，让SUT针对Reader 编程，而不再针对FileReader具体实现编程。新特性要实现字符串的网页文本转换，所以就 可以使用有关字符串的、实现了 Reader接口的StringReader对象，来充当Test Double注人 SUT中，来进行单元测试。

看一下代码e。代码有两个类。第一个类是UnicodeFileToHtmlTextConverter，它有两 个公共接口方法，一个是带有字符串类型的构造器，用来保存文件完整的路径•，另一个是 convertToHtml〇方法，用来读取文件，并把文件中的Unicode文本内容转换为HTML编码格 式的文本内容，最后将转换后的字符串返回。

UnicodeFileToHtmlTextConverter 类的代码如下所示：

public class UnicodeFileToHtmlTextConverter {  
private String fullFilenameWithPath;

public UnicodeFileToHtmlTextConverter(String fullFilenameWithPath)

{

this.fullFilenameWithPath = fullFilenameWithPath;

}

public String convertToHtml() throws IOException{

BufferedReader reader = new BufferedReader(new FileReader(fullFilenameWithPath));

String line = reader.readLine();

㊀取自ThoughtWorks Studio的培训师和教练Luca Minude丨在2012年设计并上传到GitHub上的编程操练系 列题目 TDD with Mock Objects,参见：<https://github.com/lucaminudel/TDDwithMockObjectsAndDesignPrin> ciples0

㊁ Unicode File to HTML Text Converter 操练题目原有代码参见：[https://github.com/wubin28/tbc-imicode-ftle- tohtml-text-converter-java/tree/exercise](https://github.com/wubin28/tbc-imicode-ftle-tohtml-text-converter-java/tree/exercise) ;本书针对该题目的操练步骤代码参见：<https://github.com/wubin28/> tbc-unicode-file-to-html-text-converter-java/tree/no-wrapper。

String html = while (line != null)

{

html += StringEscapeUtils.escapeHtml(line);

html += "<br />";

line = reader.readLine();

}

return html;

另一个类UnicodeFileToHtmlTextConverterTest是一个测试类，里面只有一个测试2+3=5 的这个必然通过的测试，用来验证单元测试框架JUnit是否能正常工作。把光标移动到这个 测试类名上，然后按Ctrl+Shift+F10组合键，运行一下这个测试。通过。

还是像以前一样，先读一读代码，看看有没有什么严重的代码“腐臭”。

“和前面两个题目类似，在convertToHtml()方法里面的语句‘ BufferedReader reader = new BufferedReader(new FileReader(fbllFilenameWithPath)); ’，也是依赖于一个 new 出来 的FileReader的具体实现，而不是依赖于接口编程，这违反了依赖倒置和开闭原则。写个 TODO记下来吧。”

添加依赖于FileReader的具体实现的TODO的代码如下所示（CM: Added TODO: Depending on the file system violates the Dependency Inversion Principle and Open-Closed Principle.)：

public class UnicodeFileToHtmlTextConverter {

public String convertToHtral() throws IOException{

+ //TODO: Depending on the file system violates the Dependency Inversion

Principle and Open-Closed Principle BufferedReader reader = new BufferedReader(new FileReader(fullFilenameWithPath));

“另外，‘StringEscapeUtils.escapeHtml(line);’这条语句直接使用了第三方类库，这也是 一种不明智的行为。因为当程序中遍布对这个第三方类库的直接调用时，一旦该类库有问题 需要更换，就会带来很大的工作量。这也违反了依赖倒置和开闭原则。”

对。比较好的做法是在第三方类库的外面外覆一个类，然后让咱们的生产代码针对这个 外覆类的接口编程，而让第三方类库在这个外覆类中实现这些接口。可以写一个有关直接依 赖第三方类库的TODO。

在convertToHtml〇方法中添加直接依赖第三方类库的TODO的代码如下所7K ( CM: Added TODO: Depending on the third party library violates the Dependency Inversion Principle

and Open-Closed Principle.)：

public class UnicodeFileToHtmlTextConverter {

while (line != null)

{

+ // TODO: Depending on the third party library violates the Dependency

Inversion Principle and Open-Closed Principle html += StringEscapeUtils.escapeHtml(line);

下面可以把咱们要完成的新特性写成一个TODO。

在测试类中添加新特性“能够将一个Unicode的纯文本字符串转换为HTML编码 格式的字符串”的TODO的代码如下所示（CM: Added TODO-new-feature: Make the UnicodeFileToHtmlTextConverter working for not only a file but also a string.)：

public class UnicodeFileToHtmlTextConverterTest {

+ // TODO-new-feature: Make the UnicodeFileToHtmlTextConverter working for not

only a file but also a string

}

“能想到有什么典型的有关HTML编码转换的用户意图测试吗？”

可以考虑转换“&字符”、大于号与小于号和行分隔符这3种情况，可以分别写3个用 户意图TODO。

在测试类中添加转换“&字符”、大于号与小于号和行分隔符这3个用户意图TODO的 代码如下所示（CM: Added 3 user intent test TODOs.):

public class UnicodeFileToHtmlTextConverterTest {

// TODO-new-feature: Make the method convertToHtml() working for not only a file but also a string

+

+ // TODO-user-intent-test: should convert ampersand

+ // TODO-user-intent-test: should convert greater than and less than

+ // TODO-user-intent-test: should add a line break for a new line

}

先处理转换“&”字符的TODO,需要调用一个名为converter的转换器对象的convert- ToHtml()方法。

在测试类中将转换“&”字符的TODO标记为working-on,并编写相应测试的Assert 部分的代码如下所示（CM: Working on TODO: should convert ampersand. Wrote test should\_ convert ampersandQ for it with the Assert part.)：

public class UnicodeFileToHtmlTextConverterTest {

+ // TODO-user-intent-test-working-on: should convert ampersand

@Test

+ public void should一convert一ampersand<) {

+ // Assert

+ assertEquals ("•’，converter.convertToHtml ());

}

“怎么在上面这个assertEqualsO方法中的第一个参数是空字符？这应该是期望的字符 串呀。”

对，这是有意这样写的。这里要运用前面咱们谈到过的特征测试。即可以故意在测试的 断言中填写一个不正确的期望结果，然后让测试运行到这里。一般测试到这里会运行出错， 并在出错信息中给出实际值。此时可以把出错信息中的实际值，填写到测试断言中的期望值 那里，来让测试运行通过。

“不错。这种方法尤其适用于不了解SUT的实际行为的测试。”

接着写这个测试的Act和Arrange部分的意图代码。Act部分其实和Assert部分是 合在一起的。在Arrange部分中，名为converter的转换器对象应该来自一个new出来的 UnicodeFileToHtmlTextConverter 对象。我希望在 new 这个对象的时候，UnicodeFile-ToHtml- TextConverter类的构造器能够接受一个StringReader对象，而这个StringReader对象在new 的时候被传人“H&M”这样一个字符串。

“为什么要这样写意图代码呢？”

其实我是想让上面那个UnicodeFileToHtmlTextConverter类的构造器，能够针对 Readerfc'这个抽象来编程。上面那个StringReader是Reader的子类，而咱们这个题目的 UnicodeFileToHtmlTextConverter 类原先所依赖的 FileReader,是 I叩utStreamReader 类的子 类，而InputStreamReader类又是Reader类的子类。所以这个参数类型为Reader的Unicode- FileToHtmlTextConverter构造器，既能接受这个题目的新特性所要求的StringReader类型的 对象，又能接受原有的FileReader类型的对象。

完成测试should\_convert\_ampersand()的Act和Arrange部分的意图代码的代码如下所7K (CM: Finished the intention code of test should一convert 一 ampersand().):

public class UnicodeFileToHtmlTextConverterTest {

// TODO-user-intent-test-working-on: should convert ampersand @Test

public void should\_convert\_ampersand() {

- // Assert

+ // Arrange

㊀在Java语言中，java.io.Reader是一个抽象类。

+ UnicodeFileToHtinlTextConverter converter = new UnicodeFileToHtmlTextConverter

(new StringReader(wH&Mn));

+

+ ff Act & Assert

assertEquals (•"’，converter.convertToHtml ());

}

“写完意图代码后，‘ new UnicodeFileToHtmlTextConverter(new StringReader("H&M")); ’ 语句中的new StringReader("H&M")下方出现了表示编译错误的红色下划波浪线。这是因为 还未创建带有Reader类型参数的UnicodeFileToHtmlTextConverter构造器。现在就创建。”

创建带有Reader类型参数的UnicodeFileToHtmlTextConverter构造器以修复编译错误的 代码如下所示（CM: Fixed compiler error: Created constructor UnicodeFileToHtmlTextConverter (Reader).)：

public class UnicodeFileToHtmlTextConverter {

+ public UnicodeFileToHtmlTextConverter(Reader reader) {

“修复了上面那个编译错误，接下来看converter.convertToHtmK)语句的红色编译错误。 查看错误信息，发现是没有处理IOException。按Alt+Enter快捷键在测试方法的签名中添加 抛出异常的语句。”

在测试方法should\_convert\_ampersand〇的签名中添加抛出异常的语句以修复编译错误的 代码如下所不（CM: Fixed compiler error: Added Exception to method signature.):

public class UnicodeFileToHtmlTextConverterTest {

// TODO-user-intent-test-working-on: should convert ampersand @Test

- public void should一convert\_ampersand() {

+ public void should\_convert\_ampersand() throws IOException {

“编译错误没有了。现在可以按Ctrl+F5快捷键运行测试。在‘newBufferedReader(new FileReader(ftiUFilenameWithPath)); ’语句中出现了空指针异常，原因是测试中没有给ftill- FilenameWithPath赋值，所以它是空指针。解决的办法可以是让这行代码针对Reader这个抽 象编程，而不是针对new出来的FileReader对象这个具体实现来编程。也就是现在可以处理 那个依赖文件系统的TODO，把这个TODO标记为working-on。”

接下来就可以根据刚刚说到的意图，把“ new BufferedReader(new FileReader(fiillFilename- WithPath)); ” 改为 “ new BufferedReader(this.reader); ”，也就是说希望 UnicodeFileToHtml-

TextConverter类有一个名为reader的成员变量，弁把这个成员变量所保存的Reader对象值 传递到BufferedReader构造器中〇

在convertToHtmlO方法中编写意图代码来让其依赖Reader这个抽象以修复空指针异常 的代码如下所示（CM: Fixing null pointer exception: Wrote intention code in method Unicode- FileToHtmlTextConverter.convertToHtml() to depend on the interface Reader instead of a concrete FileReader object.)：

public class UnicodeFileToHtmlTextConverter {

public String convertToHtml() throws I〇Exception{

// TODO-working-on: Depending on the file system violates the Dependency Inversion Principle and Open-Closed Principle

- BufferedReader reader = new BufferedReader(new FileReader(fullFilenameWithPath));

+ BufferedReader reader = new BufferedReader(this.reader);

“写完意图代码后，在this.reader中的reader成为有编译错误的红色。原因是reader这个 成员变量尚未创建，现在就创建它。”

创建类型为Reader的成员变量reader以修复编译错误的代码如下所示（CM: Fixed compiler error: Created a Reader field in class UnicodeFileToHtmlTextConverter.)：

public class UnicodeFileToHtmlTextConverter { private String fullFilenameWithPath;

+ private Reader reader;

“没有编译错误了，再次运行测试，还是在同样位置出现的空指针异常。这是因为咱们 在测试中通过带有Reader类型参数的UnicodeFileToHtmlTextConverter构造器所传进来的 reader尚未被保存并传递到空指针异常的语句中。现在就修复这个异常。”

实现构造器UnicodeFileToHtmlTextConverter(Reader)以修复空指针异常的代码如下所示 (CM: Fixed null pointer exception: Implemented constructor UnicodeFileToHtmlTextConverter(Re ader).)：

public class UnicodeFileToHtmlTextConverter {

public UnicodeFileToHtmlTextConverter(Reader reader) { + this.reader = reader;

“运行测试，发现了比较失败ComparisonFailure,期望的是空串，但实际值是‘H&amp; M<br/>’，可以把这个实际值复制到测试的assertEqualsO语句的那个空串的位置，作为期望

值。再次运行测试，通过。”

“因为现在UnicodeFileToHtmlTextConverter类已经针对Reader来编程了，所以原先的构 造器 UnicodeFileToHtmlTextConverter(Striiig)也需要修改一下。”

对。具体做法是在构造器UnicodeFileToHtmlTextConverter(String)中，把传递进来的文 件路径转换为Reader。原先的成员变量flillFilenameWithPath就没有用处了，可以删除。

修改原先的构造器UnicodeFileToHtmlTextConverter(String)以适应针对Reader编程的代 码如下所示（CM: Updated the original constructor UnicodeFileToHtmlTextConverter(String) to keep the original function for FileReader working.)：

public class UnicodeFileToHtmlTextConverter {

* private String fullFilenameWithPath; private Reader reader;
* public UnicodeFileToHtmlTextConverter(String fullFilenameWithPath)
* {

- this.fullFilenameWithPath = fullFilenameWithPath;

+ public UnicodeFileToHtmlTextConverter(String fullFilenameWithPath) throws FileNotFoundException {

+ this.reader = new FileReader(fullFilenameWithPath);

}

“运行测试，通过。这样，咱们就完成了依赖文件系统的TODO和转换‘&’字符的 TODO,可以把它们删除。，’

有了这个测试作为基础，就能很容易地编写出针对转换大于号与小于号和行分隔符这两 个用户意图TODO的测试。

完成转换大于号与小于号的用户意图TODO的测试代码如下所示（CM: Finished TODO: should convert greater than and less than.)：

public class UnicodeFileToHtmlTextConverterTest {

- // TODO-user-intent-test: should convert greater than and less than

+ @Test

+ public void should\_convert\_greater\_than\_and\_less\_than() throws IOException {

+ // Arrange

+ UnicodeFileToHtmlTextConverter converter = new UnicodeFileToHtmlTextConverter

(new StringReader{">\_<|||"));

+

+ // Act & Assert

+ assertEqualsI I|<br />", converter.convertToHtml());

+ }

运行测试，通过。

完成转换行分隔符的用户意图TODO的测试代码如下所示（CM: Finished TODO: should

add a line break for a new line.)：

public class UnicodeFileToHtmlTextConverterTest {

- // TODO-user-intent-test: should add a line break for a new line

+ @Test

+ public void should\_add\_a\_line一break一for\_a\_new一line() throws IOException {

+ // Arrange

+ UnicodeFileToHtmlTextConverter converter = new UnicodeFileToHtmlTextConverter

(new StringReader("CheersXnBen Wun));

+

+ //Act & Assert

+ assertEquals(nCheers<br />Ben Wu<br />", converter.convertToHtml());

+ }

运行测试，通过。因为咱们现在已经实现了将字符串转换为HTML编码的格式，所以 也完成了那个新特性的TODO,可以删掉该TODO。

“现在只剩下那个依赖第三方类库的TODO 了，把它标记为woridng-cm。要完成这 个TODO,需要先写一个测试来表达使用外覆类的对象的意图。比如我们可以添加一个转 换‘ & ’字符的测试，然后在UnicodeFileToHtmlTextConverter的构造器中，除了传入new StringReader("H&M"),再传人一个stringEscaper对象。这个对象的类型就是用来外覆第三方 类库的StringEscaper类。另外，我们还需要在convertToHtml()方法中，将StringEscapeUtils. escapeHtml(line)替换为 this.stringEscaper.escapeHtml(line)，使得该方法不再直接依赖第三方 类库，而是依赖咱们的外覆类StringEscaper。”

编写使用外覆类StringEscaper的测试和convertToHtml()方法的意图代码如下所示 (CM: Added test should\_convert\_ampersand\_using\_StringEscaper() with intention code using StringEscaper which will wrap StringEscapeUtils.escapeHtml(). Wrote intention code in class UnicodeFileToHtmlTextConverter to make it depend on StringEscaper instead of third party library StringEscapeUtils.)：

public class UnicodeFileToHtmlTextConverterTest {

+ @Test

+ public void should一convert一ampersand一using\_StringEscaper() throws IOException { + // Arrange

+ StringEscaper stringEscaper = new StringEscaper();

+ UnicodeFileToHtmlTextConverter converter = new UnicodeFileToHtmlTextConverter

(new StringReader("H&M"), stringEscaper);

+

+ //Act & Assert

+ assertEquals ('\*H&amp;M<br />", converter.convertToHtml ());

+ }

public class UnicodeFileToHtmlTextConverter {

while (line != null)

{

// TODO-working-on: Depending on the third party library violates the Dependency Inversion Principle and Open-Closed Principle - html +— StringEscapeUtils.escapeHtml(line);

+ html += this.stringEscaper.escapeHtml(line)/

html += "<br />"; line = reader.readLine();

)

“写完这些意图代码后，就需要修复那些红色的编译错误。先看那个新写的测试。其中 StringEscapei•是编译错误的红色，因为尚未创建该类。现在就创建。”

创建类StringEscaper以修复编译错误的代码如下所示（CM: Fixed compiler error: Created class StringEscaper.)：

+public class StringEscaper {

+ }

“接下来在修复这个测试中 new UnicodeFileToHtmlTextConverter(new StringReader(MH&M")， stringEscaper)里面两个带有红色下划波浪线的参数的编程错误。错误的原因是尚未创建 带有 Reader 和 StringEscaper 这两个参数的 UnicodeFileToHtmlTextConverter 构造器，现在就 创建。”

创建带有 Reader 和 StringEscaper 这两个参数的 UnicodeFileToHtmlTextConverter 构造器 以修复编译错误的代码如下所示（CM: Fixed compiler error: Created constructor UnicodeFileTo HtmlTextConverter(Reader, StringEscaper).)：

public class UnicodeFileToHtmlTextConverter {

+ public UnicodeFileToHtmlTextConverter(Reader reader, StringEscaper stringEscaper) {

+

+ )

“现在测试里面的编译错误已经被修复了。再看看UnicodeFileToHtmlTextConverter类中 的编译错误。这里，this.stringEscaper.escapeHtml(line)中的 stringEscaper 是红色，因为该成 员变量尚未创建。现在就创建。”

在UnicodeFileToHtmlTextConverter类中创建stringEscaper成员变量以修复编程错误的 代码如下所示（CM: nxed compiler error: Created field stringEscaper in class UnicodeFileToHtm ITextConverter.)：

public class UnicodeFileToHtmlTextConverter {

private Reader reader;

十 private StringEscaper stringEscaper;

“ stringEscaper的红色消失了，但后面的escapeHtml()又变红了。因为StringEscaper类 的escapeHtm()方法尚未创建，现在就创建。”

创建StringEscaper类的escapeHtm()方法以修复编译错误的代码如下所75 (CM: Fixed compiler error: Created method StringEscaper.escapeHtml().)：

public class StringEscaper {

+ public String escapeHtml(String originalstring) {

+ return null;

+ }

}

“编译错误都修复了。运行新写的使用StringEscaper类的测试，发现在convertToHtml() 方法中的new BufferedReader(this.reader)语句有空指针异常。原因是带有两个参数的构造器 UnicodeFileToHtmlTextConverter(Reader，StringEscaper)是空的，尚未实现。现在就实现它〇” 实现构造器 UnicodeFileToHtmlTextConverter(Reader，StringEscaper)以修复编译错误的代 码如下所示（CM: Fixed null pointer exception: Implemented constructor UnicodeFileToHtmlText Converter(Reader, StringEscaper).)：

public class UnicodeFileToHtmlTextConverter {

public UnicodeFileToHtmlTextConverter(Reader reader, StringEscaper stringEscaper) {

+ this.reader \* reader;

+ this.stringEscaper = stringEscaper;

}

“编译错误都修复了。再次运行新写的使用StringEscaper类的测试，发现比较失败 ComparisonFailure。期望值是‘H&amp;M<br/>’，实际值是‘null<br/>’。这是因为咱们新 增的外覆类StringEscaper中的escapeHtml()方法是空的，尚未实现。现在就实现它。”

实现外覆类StringEscaper中的escapeHtml()方法以修复比较失败的代码如下所7K ( CM: Fixed comparision failure: Implemented method StringEscaper.escapeHtml().)：

public class StringEscaper {

public String escapeHtml(String originalstring) {

- return null;

+ return StringEscapeUtils.escapeHtml(originalstring);

}

“运行所有测试，新写的使用StringEscaper类的测试通过，而前面写的那3个测试，都

在convertToHtml()方法中的this.stringEscaper.escapeHtml(line)语句中拋出空指针异常。这是 因为构造器 UnicodeFileToHtmlTextConverter(Reader)尚未对新增的 stringEscaper 成员变量赋 值，造成了空指针。现在就修复它。顺便把原先的构造器UnicodeF"UeToHtm丨TextConverter(St ring)也一并修改来对stringEscaper成员变量赋值。”

让构造器 UnicodeFileToHtmlTextConverter(Reader)和 UnicodeFileToHtmlTextConverte r(String)对stringEscaper成员变量赋值以修复空指针异常的代码如下所示（CM: Fixed null pointer error: Updated constructor UnicodeFileToHtmlTextConverter(String) and UnicodeFileToHt mlTextConverter(Reader) to call UnicodeFileToHtralTextConverter(Reader, StringEscaper).)：

public class UnicodeFileToHtmlTextConverter {

public UnicodeFileToHtmlTextConverter(String fullFilenameWithPath) throws FileNotFoundException {

* this.reader = new FileReader(fullFilenameWithPath);

+ this(new FileReader(fullFilenameWithPath), new StringEscaper());

public UnicodeFileToHtmlTextConverter(Reader reader) {

* this.reader = reader;

+ this(reader, new StringEscaper());

}

“运行全部测试，通过。现在有关依赖第三方类库的TODO也完成了，可以将其删除。” 至此，这个有关网页文本转换系统的编程操练就告一段落。为了便于和重构前的代码进 行比较，下面列出重构后的测试代码和生产代码。

重构后 UnicodeFileToHtmlTextConverterTest 类的代码如下所不：

public class UnicodeFileToHtmlTextConverterTest {

@Test

public void should\_convert\_ampersand() throws IOException {

// Arrange

UnicodeFileToHtmlTextConverter converter = new UnicodeFileToHtmlTextConverter (new StringReader);

// Act & Assert

assertEquals("H&arap;M<br />", converter.convertToHtml());

}

QTest

public void should\_convert\_ampersand\_using\_StringEscaper () throws IOException { // Arrange

StringEscaper stringEscaper = new StringEscaper();

• UnicodeFileToHtmlTextConverter converter = new UnicodeFileToHtmlTextConverter

(new StringReader("H&M")/ stringEscaper);

//Act & Assert

assertEquals("H&amp;M<br />", converter.convertToHtml());

@Test

public void should一convert一greater\_than一and一less\_than() throws IOException { // Arrange

UnicodeFileToHtmlTextConverter converter = new UnicodeFileToHtmlTextConverter (new StringReaaer(">\_<|||M));

//Act & Assert

assertEqualsI I I<br />n, converter.convertToHtml());

)

QTest

public void should\_add\_a\_line一break一for一a\_new\_line() throws IOException {

II Arrange

UnicodeFileToHtmlTextConverter converter = new UnicodeFileToHtmlTextConverter (new StringReader("Cheers\nBen Wu"));

//Act & Assert

assertEquals("Cheers<br />Ben Wu<br />", converter.convertToHtml());

}

重构后StringEscaper类的代码如下所示：

public class StringEscaper {

public String escapeHtml(String originalstring) {

return StringEscapeUtils.escapeHtml(originalstring);

重构后UnicodeFileToHtmlTextConverter类的代码如下所示：

public class UnicodeFileToHtmlTextConverter { private Reader reader; private StringEscaper stringEscaper;

public UnicodeFileToHtmlTextConverter(String fullFilenameWithPath) throws FileNotFoundException {

this(new FileReader(fullFilenameWithPath), new StringEscaper());

}

public UnicodeFileToHtmlTextConverter(Reader reader) { this(reader, new StringEscaper());

)

public UnicodeFileToHtmlTextConverter(Reader reader, StringEscaper stringEscaper) { this.reader = reader; this.stringEscaper = stringEscaper;

}

public String convertToHtml() throws IOException{

BufferedReader reader = new BufferedReader(this.reader);

String line = reader.readLine();

String html = ,,M; while (line null)

{

html += this.stringEscaper.escapeHtml(line);

html += "<br />";

line = reader.readLine();

}

return html;

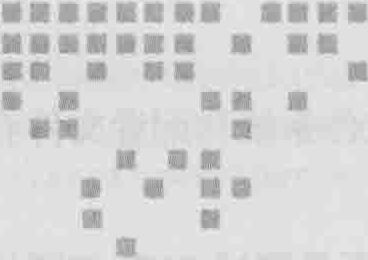
至此，我和亲爱的读者您之间的结对编程就告一段落了。下一章将总结驯服烂代码的步 骤。在总结之前，让我们看看本章都做了哪些工作。

1. 阅读网页文本转换系统的代码，包含UnicodeFileToHtmlTextConverter类。
2. 记录了依赖于new出来的FileReader的具体实现，从而违反了依赖倒置和开闭原则 的 TOD00
3. 记录了直接使用第三方类库，从而违反了依赖倒置和开闭原则的TODO。
4. )记录了一个要完成的新特性的TODO。
5. )记录了 3个用户意图测试的TODO。
6. )先写了测试的Assert部分的意图代码，然后再推导出Act和Arrange部分的测试意 图代码，并通过修复意图代码的编译和测试运行错误，来驱动出生产代码，从易到难地完成 上述TODO。
7. 用外覆类包装了针对第三方类库的使用，即在第三方类库的外面外覆一个类，然后 让生产代码针对这个外覆类的接口编程，而让第三方类库在这个外覆类中实现这些接口，以 便于将来更换第三方类库。
8. 当不大了解要测试的SUT的实际行为时，使用了特征测试的方法来进行测试。即可 以故意在测试的断言中填写一个不正确的期望结果，然后让测试运行到这里。一般测试到这 里会运行出错，并在出错信息中给出实际值。此时可以把出错信息中的实际值填写到测试断

言中的期望值那里，来让测试运行通过。

9 )通过操练我们学到了以下技能：

1. 运行速度慢的不是单元测试。与数据库、网络、文件系统和配置文件有交互的测试不 是单元测试。
2. 单元测试要测的仅仅是SUT里面的软件行为，而不是测试SUT与诸如数据库、网络 和文件系统这些DOC之间是否能正常交互。
3. 要把SUT与诸如数据库、网络、文件系统这些的DOC之间进行交互的集成测试，转 变为针对SUT的单元测试，可以先找到DOC的接口，让SUT针对这个接口编程，并且编写 一个运行起来经济快捷的Test Double来实现这个接口，并注入SUT中，让SUT把这个Test Double当成那个接口来使用。这样就能把依赖于DOC的集成测试，转换为依赖于实现接口 的Test Double的单元测试。



第19章

驯服烂代码的步骤：lePpTr

前面给烂代码下了定义后，让我们看看驯服烂代码的可操作的步骤。从前面所得出的在 拜唐僧为师时的孙悟空是一段烂代码的结论可以看出，烂代码好像一个人，而且是一个有正 常行为能力的、同时有一身小毛病或正在学习和修行的、还没有最终觉悟、在修正自身缺陷 和学习修行上进步与反馈迟缓的人。

在字典里，“驯服”的意思是“使顺从”。那么如何才能使烂代码“顺从”呢？既然烂 代码像人，那么我们可以换个问法，如何才能使一个人“顺从”呢？孙子曰：“知己知彼者, 百战不殆。”要想使人顺从，首先要做的是了解这个人。那么，如何才能了解一个人呢？

距今2500多年前的一天上午，孔子在学校里溜达，忽然发现他的学生宰予趴在课桌上 睡大觉。孔子看了很生气，回到课堂上，对弟子们说：“宰予这孩子竟敢骗我！以前他曾跟 我信誓旦旦地保证，要改掉白天睡觉这个坏毛病。谁知道今天上午，我又碰到他趴在桌子上 睡大觉！真是朽木不可雕也，垃圾做的墙壁不可粉刷也！对宰予这孩子，我还能说他什么 好呢？”过了会，老先生余怒未消，接着说：“以前，我听一个人所说的话，就会相信他的 行为。今天宰予的事情，让我从今往后，在了解一个人时，不仅要听他所说的话，还要看 他所做的事。e”

上面孔子的那段话，告诉我们，要了解一个人，不仅要听其言，更要观其行。前面说 过，烂代码像一个人，那么要了解一段烂代码，也不仅要听其言，即看代码本身、文档及注 释来理解代码的意图，更要观其行，即用运行测试代码的方式来验证代码是否言行一致。

在用“听其言而观其行”的方式了解了一段代码之后，我们又该如何使代码“顺从”呢？

㊀出自《论语》，公冶长。 老子说：“ ‘道’经常是处于无名和质朴的状态。f道’虽然很小，但是天下没有谁能够征服 ‘道’的。天下的诸侯若能坚守这个‘道’，那么世间的万物都将自己会顺从。6”看来咱们 只要在写代码时遵守这个“道”，那么代码自己就会顺从于咱们。那么这个“道”到底在哪 里呢？

在距今500多年前的一天，一位明朝的官员，因为反对宦官而被朝廷打了板子，随后 被降职并发配到生活条件十分艰苦的贵州龙场。这位官员也在日日夜夜苦苦思索这个问题: “‘道’到底在哪里？”终于，在一个夜晚，他大喊一声：“找到啦！ ‘道’就在你、我的心里！ 圣人之道，吾性自足！ ”这就是历史上著名的“龙场悟道”，这位官员，就是中国历史上罕有 的“三不朽©”的人物之 王阳明。

我们每一个人，都来自大自然，同时也是大自然的一部分。大自然中蕴涵的“道”，也 同样蕴涵在我们每一个人的内心。如此说来，编写代码的“道”，也蕴涵在每一位程序员的 内心之中。

或许有人会问：“既然‘道’在每个程序员的内心中，那么程序员就不应该写出那么多 不遵守‘道’的烂代码。如果是这样的话，那么现在充斥在项目中的大量烂代码又该如何解 释呢？”

对于这个问题，王阳明最得意的门生徐爱曾说过这样一段话：“人的内心就像一面镜子。 圣人内心的镜子十分明亮，而常人内心的镜子则比较昏暗。如果按照朱熹的‘格物致知’的 方法来寻求存在于每一个物件之中的‘道’，那么就好比用镜子照物时在‘照’上下工夫，而 全然不顾自己的内心是不是昏暗，这样照镜子能照得好才怪呢。而王阳明先生的‘格物就 好比把镜子擦亮，是在‘擦’上下工夫。等把镜子擦亮了，自然就能照镜子照得好了。®”

这样看起来，目前世上现存的所有烂代码，都是程序员内心的“镜子”没有被擦亮而导 致的。而如果程序员把内心的“镜子”擦亮会怎样？就会“照”出像设计模式、面向对象的 SOLID设计原则®、重构、测试驱动开发（Test-Driven Development, TDD)、敏捷软件开发、 精益这些软件开发的“道”。

©出自老子<道德经》第三十二章：“道常无名，朴。虽小，天下莫能臣\_。侯王若能守之，万物将自宾 ©我国伦理思想史上的一个命题。春秋时鲁国大夫叔孙豹称“立德"、“立功"、“立言”为“三不朽' “立德"’， 即树立高尚的道德；“立功即为国为民建立功绩立言"，即提出具有真知灼见的言论。此三者是虽久 不废，流芳百世的。一般认为我国历史上能够做到真三不朽的只有两个半人，分别是大成至圣先师孔子、 王文成公王守仁（别号阳明）和曾文正公曾国藩（半个)。——引自百度百科 ㊂出自《传习录》。

㊃面向对象的SOLID设计原则，即下面5种设计原则的英文首字母：单一职责原则SRKSingle-Responsibility Principle)、开闭原则 0CP ( Open/Closed Principle)、里氏替换原则 LSP ( Liskov Substitution Princip丨e)、接 口隔离原则丨SP (丨nterface Segregation Principle)和依赖倒置原则 DIP (Dependency-丨nversion Principle)，参 见Robert C. Martin所著《敏捷软件开发：原则、模式与实践》一书。

这里需要注意的是，千万不可把用昏暗的“镜子”所照出的那些“反模式e ”当成软件 开发之“道”。在编写代码中最典型的反模式就是为图方便而使用快捷键Ctrl+C和Ctrl+V来 复制和粘贴代码的“CV大法”，另外在《重构》一书所列出的22种代码“腐臭”中也能找 到其他的反模式。

人的内心除了可以比作镜子，还可以比作玉石。玉石在雕琢前，被称为“璞玉”或“玉 原石”。玉原石在开采时，粗糙不平的原石表面一定覆盖着厚厚的尘土，这好比人内心上覆 盖着的名与利的滚滚红尘。要把这块玉原石雕琢成一块美玉，我们首先要除去尘土，然后切 开玉石（切)、修制成型（蹉）和雕琢花纹（琢)，最后打磨抛光（磨）。而正处于修制成型和 雕琢花纹阶段的玉石，就好比正在不断学习和修行的人的内心；经过最后打磨抛光后的美玉， 就好比大彻大悟最终得道的圣人的内心。

一旦我们掌握了烂代码的所言和所行，我们就能了解烂代码。在了解了烂代码后，我们 程序员还需要时时刻刻把自己内心的那面“镜子”擦亮，或把自己内心的那块璞玉磨亮，去 照出并坚守软件开发之“道”，这样我们就可以驯服烂代码。

驯服烂代码不仅需要“听其言、观其行、守其道”，还需要一个可以操作的具体步骤。 我们固然可以把前面用TDD开发方法做编程操练的过程归纳一下，形成一个可供参考的步 骤，但生动的比喻胜过枯燥的归纳。在归纳之前，我们不妨继续看这个用作比喻的观音菩萨 助唐僧教化悟空的故事，看看驯服悟空的过程是个什么样子。

从石头中蹦出来的悟空，身上有许多毛病和bug。而对于唐僧和观音菩萨来说，悟空最 大的bug便是一味行凶杀生。如果回味一下前面描述的观音菩萨助唐僧教化孙悟空的过程， 我们会发现观音采用的方法是这样的：首先，在依如来佛旨意去找取经人路过两界山时，规 劝悟空不再行凶、皈依佛法和保护取经人，并听悟空将这些劝告的意图都答应下来，这就是 “听其言'然后让唐僧收悟空为徒，看到悟空行凶并撇下师父后，给唐僧准备了能令悟空信 守承诺的衣服和帽子，并教其念紧箍咒，让猴子在好奇心的驱使下自己穿戴上，而帽子中就 隐藏着用于矫正悟空bug行为的金箍。观音菩萨观察悟空的一举一动，并做出一系列能让前 面悟空答应的事情得到落实的行为，就是“观其行'最后，令唐僧每当需要依照佛法教诲犯 错的悟空时，就默念紧箍咒，疼得猴子伏地求饶，死心塌地地皈依佛法，这就是“守其道”。

如果把悟空换做烂代码，那么驯服烂代码的步骤如下：

1)听其言，维护TODO列表s、编写用户意图测试和意图代码。TODO列表就是意图列

©反模式（Anti-pattern or Antipattem)是一种针对经常发生的问题所作出的通常无效的常见应对方法，这种 方法会带来高度的起反作用的风险。 引自wikipedia.org

©维护TODO列表的想法受Kent Beck的著作《Test-Driven Development By Example》一书的启发。Beck 在书中使用了笔和便签来维护要做的任务，我把这种方式转变为在代码中编写T0D0注释，并用IDE来 管理TODO列表。用TODO注释的方式来记录任务有两个好处：一方面可以把T0D0列表纳人版本管理 系统进行保存，避免便签丢失的情况；另一方面，把TODO以注释的形式写在相关的代码边上，便于程序 员找到与这个TODO相关的代码。

表。程序员可以收集并审查所有当前和今后要做的诸如用户意图测试、bug修复和代码“腐 臭”治理这样的任务，形成一个用意图来表达的TODO列表。然后根据当时的情况，或者选 择一个条件已经具备且有信心完成的用户意图TODO作为下一步要开发的任务，把它标记为 working-on,根据用户意图做出合理的分析和设计，根据设计意图编写用户意图测试或意图 代码。可以把每条TODO都以代码注释的方式写到代码相关位置，并用IDE的TODO管理 界面加以管理。对于已经完成的TODO,就可以从列表中删除。

1. 观其行，以修复编译错误和测试运行错误为指引编写恰好够用的生产代码，直至测 试运行通过。首先以上一步所编写的意图代码的红色编译错误为指引，当信心弱时可以编写 尽量少的生产代码（即可以写空类和空方法)，当信心强时可以写自认为合理的生产代码e, 来让测试代码和生产代码编译通过。然后运行测试，当发现测试运行中的诸如空指针这样的 测试运行错误后，以这些错误为指引，同样当信心弱时编写尽量少的生产代码（可以写假数 据)，当信心强时可以写自认为合理的生产代码，让测试运行通过。
2. 守其道，全面重构TODO列表、测试代码和生产代码。首先要“嗅一嗅”上一步令 测试运行通过的生产代码中，是否有代码“腐臭”。如果有，就要在经常运行测试的情况下 小步重构生产代码，以消除“腐臭”。然后根据诸如面向对象设计的SOLID原则、已知的或 更新后的产品特性和设计模式这些软件开发的“道”的层面的概念，来审查目前所有的测试 代码和生产代码，来找出其中诸如不再合理的测试、代码“腐臭”、被遗漏的User Story或刚 刚从测试工程师手里接到的bug报告等内容，并以TODO的形式记录下来，补充到“听其 言”步骤中的TODO列表中，来进人下一个迭代。

在做上面第3步“守其道”时，要注意这里的“全面重构”，扩展了 Martin Fowler有关 “重构”的定义。Martin Fowler把重构定义为这样一种过程：“在不改变代码外在行为的前提 下，对代码做出修改，以改进程序的内部结构。”这个定义描述了在用测试代码固定生产代 码外在行为的前提下，改进生产代码的内部结构的过程。它适合于代码外在行为符合预期的 情况。但随着软件开发的不断进展，很多以前认为是合理的软件外在行为的预期，渐渐变得 不再合理，而需要调整这些软件外在行为。而对软件外在行为的调整，意味着要跟着调整相 应的测试，以固化新的软件外在行为。为了在重构中应对上述软件外在行为的变化，需要对 该定义进行扩展。

不妨把这个扩展了的重构，称为“全面重构”（Total Refactoring)。本书对全面重构的定 义如下：在遵从诸如面向对象设计的SOLID原则、当下更新后的产品特性和消除代码“腐臭” 等这些软件开发的“道”的层面的概念前提下，对记录未完成任务的TODO列表、测试代码 和生产代码进行修改，以改进代码的内部结构的过程。

e感谢微博网友@pkuxkxjason有关“TDD小步前进因人而异，一味强调写尽量少的代码是低效的”观点。

全面重构的具体步骤如下：

1. 审视并更新TODO列表，删除已经完成的任务，添加新发现的任务，确保该列表反 映了当下所有已知的尚未完成的任务。
2. 根据程序员的经验，从TODO列表中挑选一个自认为最合理的任务作为下一个任务3
3. 编写或修改与上述选出的任务相关的测试代码，一方面确保测试代码固化了当下合 理的软件外在行为，另一方面确保测试代码调用了当下合理的生产代码的接口。
4. 在频繁运行上述测试代码的前提下，对生产代码做出修改，以改进程序的内部结构。

上面讨论了驯服像悟空这样有bug的烂代码的步骤。但如果回顾一下前面用TDD开发

方法做的那个酒店世界时钟编程操练的步骤，就可以看出，上述驯服有bug的烂代码的步 骤，同样也适用于开发一个尚未编写任何代码的全新的软件项目。虽然上述步骤对这两种情 况都适用，但二者还是有些区别。主要区别在于，前者在编写意图代码时，需要不断地参考 已有代码的现有接口；而后者由于生产代码尚未编写，所以在编写意图代码时可以自由定义 生产代码所需提供的接口。由此看来，使用测试先行的TDD开发方法编写一个全新的功能， 比起先写生产代码再后补测试的测试后行的开发方法的难度要小得多，且代码的可测性也会 天然地好很多。

在统一了驯服已有烂代码和编写全新项目的新代码的开发步骤后，我们不妨把这种开发 步骤起一个名字：丨ePpTr。如果把TDD开发方法比作一个接口，那么IePpTr就是TDD的一 种实现。

IePpTr 表示 Intent-ErrorProductionPassed-TotalRefactoring。字母组合 TR 前面的横线，表 示这个TR所代表的全面重构，不仅需要实施到前面的字母P所代表的Production Code上， 而且还需要实施到前面的字母I所代表的TODO、用户意图测试和意图代码上。

TDD的一种实现：IePpTr。

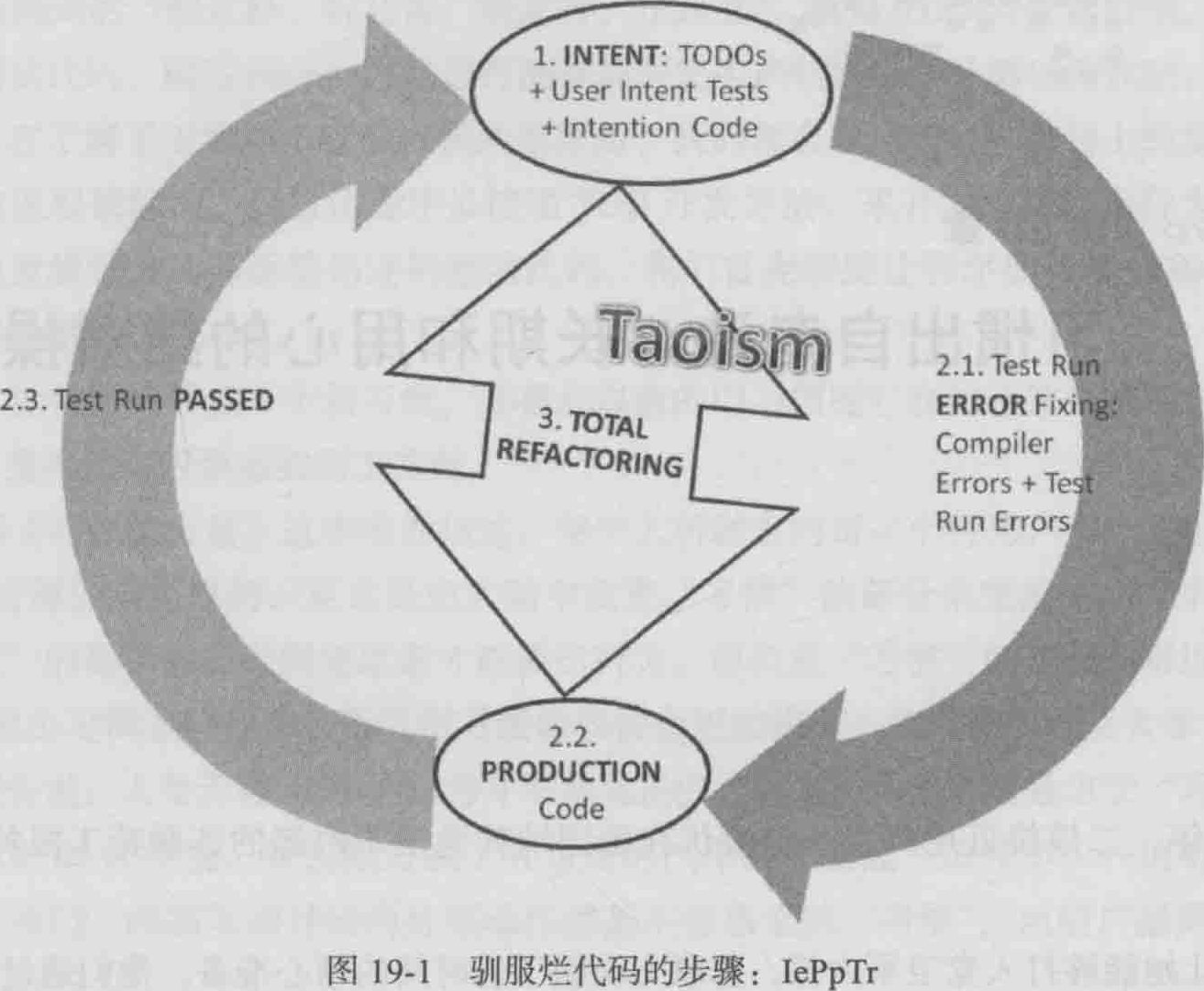
1. )听其言：Intent (TODOs + User Intent Tests 十 Intention Code)
2. )观其行：Test Run Error Fixing (Compiler Errors + Test Run Errors), Production Code, Test Run Passed
3. )守其道：Total Refactoring

IePpTr的示意图如图19-1所示

在了解了驯服烂代码的步骤后，我们需要把这些步骤化为习惯，才能让它们在我们的曰 常工作和编程操练中发挥作用。在讨论如何形成习惯之前，先回顾一下本章的内容：

1)驯服烂代码的IePpTr方法是TDD开发方法的一种实现，可以分为3步：听其言（维 护TODO列表、编写用户意图测试和意图代码)、观其行（以修复编译错误和测试运行错误 为指引编写恰好够用的生产代码，直至测试运行通过)、守其道（全面重构TODO列表、测 试代码和生产代码)。

An Implementation of TDD: IePpTr



2) Total Refactoring全面重构，是为了适应软件开发过程中软件外在行为会逐渐发生变 化的情况，而对传统重构概念所做的扩展。这种扩展强调在遵从软件开发的“道”的前提下， 对记录未完成任务的TODO列表、测试代码和生产代码进行修改，以改进代码的内部结构的 过程。

於：第 20 章

习惯出自专注、长期和用心的结对操练

1945年，二战接近尾声。一位潜伏在德国纳粹党卫军内部的苏联特工凯特快要生孩 子了 〇

为了让她能够打人党卫军内部，苏联方面做了长时间的精心准备。他们通过特殊训练， 让凯特学会了德国一个村庄的方言，然后轰炸了这个村庄，令全体村民无一幸免。这样， 凯特就能以这个村庄唯一幸存者的身份，说着流利的、带有这个村庄的口音的德语，打入 党卫军。

凯特的领导施季里茨提醒她：“生孩子时要到别处去生，或者记得用德语喊叫，不要暴 露你是俄国人。”不久，凯特的住所遭到了轰炸。凯特被人从瓦砾中解救出来，并送往医院 去生孩子。在医院里，凯特的身份被暴露了。因为她在生孩子的极度疼痛中，不由自主地喊 出了俄语〜

为什么凯特在习惯了讲德语后，却在生孩子过程中因疼痛叫喊时，又操起了讲俄语的老 习惯呢？

Charles Duhigg在《习惯的力量》e—书中讲到的有关习惯的一个原理，能够解释上面凯 特的故事：“习惯是不能被消除的，而只能被代替。”换句话说，凯特讲俄语的习惯，虽然能 被通过特殊训练而掌握的讲德语的新习惯所替代，但是它并没有被消除，而是潜伏在意识深 处。一旦人所能承受的外在压力增大到超过一定限度时，旧习惯就有可能会取代新习惯而伺

㊀故事来自莫斯科电影制片厂1973年出品的电视剧《春天里的17个瞬间>,参考了《冬吴相对论》第427 期“习惯的力量”和百度百科。

© CharlesDuhigg著，吴奕俊、陈丽丽、曹婢译，《习惯的力量》，中信出版社，2013年4月，第1版。

机冒出。

上面提到的“察且择、听其言、观其行、守其道”，就是TDD开发新习惯。而程序员原 先不写测试代码、编写main()方法进行测试或写完生产代码后再补测试等作法，都是一些旧 的习惯。在了解了上面那个有关习惯的原理后，我们就会知道，程序员身上的那些旧习惯是 不可能被连根拔除的。要想让程序员使用TDD开发方法，来开发出在代码行为理解、问题 感知和质量维护方面都反馈迅速的整洁代码，我们首先需要让程序员养成TDD开发的新习 惯，来替代原来的旧习惯。

那么如何才能养成一个新习惯，并替代以前的旧习惯呢？在讨论这个问题之前，咱们先 看看什么是习惯和习惯是如何工作的。

根据《习惯的力量》这本书的描述，每个人所做出的每一个行为，要么是由大脑中负责 “决定”的部分来完成的，要么是由大脑中负责“习惯”的部分来完成的。其中，大脑中负 责“决定”的部分需要时间来思索才能做出行为，而负责“习惯”的部分则可以不假思索地 “自动”做出习惯了的行为，所以用习惯做事情会更加快速。该书援引杜克大学2006年发布 的研究报告说：人每天有40%的行为并不是真正由“决定”促成，而是出于“习惯”。

这就好比11年前，我刚刚习惯了开自家的汽车后，就能一边开车，一边听广播。我在 开车时踩油门、踩刹车和开转向灯等动作都是不假思索的“习惯”，而听广播则是需要思索 的“决定”。

根据《习惯的力量》一书的结论，我们大脑的“习惯”的工作方式是“一个由3步组 成的回路。第1步，存在着一个暗示，能让大脑进入某种自动行为模式，并决定使用哪种习 惯。第2步，存在一个惯常行为，这可以是身体、思维或情感方面的。第3步则是奖赏，这 让您的大脑辨别出是否应该记下这个回路，以备将来之用”。

我们还是拿用旧习惯编程来打比方，比如我们打开计算机，启动IDE准备编程，这就是 上面第1步中的那个暗示，以便让大脑决定进入某种编程习惯；接着我们用旧的惯常行为不 假思索地首先编写生产代码，就是第2步；然后我们完成生产代码的编写，之后再编写一个 main()方法来测试一下，当看到main()方法的输出结果符合我们的预期时，就是第3步里面 的那个奖赏，来让我们的大脑记录下来这个回路，以备后用。

在了解了习惯的工作方式之后，我们可以看出，如果我们想要养成一个新习惯，那么只 要把上述旧习惯的回路中的第1步中的“暗示”和第3步中的“奖赏”保留不动，然后用一 个新的惯常行为替换掉第2步中旧的惯常行为，那么大脑就可以在第1步的结尾处选择我们 的新习惯，从而养成这个新的习惯。这就是在《习惯的力量》一书中阐明的利用已经存在的 旧习惯来将其改变的秘密。之所以这样做，是因为如果新行为模式的开头和结尾存在一个人 所熟悉的东西，那么就能更容易地说服这个人接受新的行为模式。

比如对于TDD开发方法这个新习惯，我们可以保留第1步打开IDE准备编程这个暗示， 和第3步看到符合预期的结果这个奖赏，然后把中间先写生产代码的旧的惯常行为，替换 为TDD开发方法。这样保留第1步和第3步，我们就可以最大限度地让程序员接受这个新 习惯。

既然我们要在第2步中，把原先旧的惯常行为替换为更好的新的惯常行为，那么，怎样 才能让新的惯常行为更好，甚至达到世界级水平呢？

Malcolm Gladwell 在他写的 Outliers0—书中，提出了 “10000 小时法则”（The 10,000- HourRule)。Gladwell在书中写道：“各种研究表明，为了达到各种领域的世界级专家的水平， 需要10000小时的实践。神经学家Daniel Levitin这样写道，‘我们对下面各种专家进行了一 个又一个的研究，这些专家包括作曲家、篮球运动员、小说家、滑冰运动员、钢琴演奏家、 国际象棋棋手、犯罪大师，以及其他您能想到的专家，我们发现10000这个数字一次又一次 地出现。当然这并不是解决为什么一些人会比其他人能从他们的实践中获取更多技能这个问 题。但是迄今为止没有一个人能够发现下面这种情况，即真正的世界级的专业知识能够在少 于10000小时的时间内被掌握。这看起来似乎是大脑要花10000小时这样长的时间来消化所 有它需要知道的东西，来到达真正的精通程度。’”

看来，如果想让我们新的惯常行为达到世界级的专业水平，需要花至少10000小时来实 践，即每天花3小时，需要连续进行10年的实践。这提示我们，要形成更好的新的惯常行 为，我们需要长期的实践。

有人会说：“既然实践10000小时就能达到世界水平，那我从小到大走路的时间都已经 超过10000小时了，为何还不是竞走的世界冠军呢？”

瑞典心理学家、佛罗里达州立大学心理学教授、举世公认的在“专业知识与技能”领 域的理论和试验领先的研究者K. Anders Ericsson,在他1993年发表的文章The Role of Deliberate Practice in the Acquisition of Expert Performance 中指出的下面结论，或许能够解释 上面的问题：“人们相信，因为专业高手的表现与正常人的表现具有本质的不同，所以这些 专业级别的表演者一定天生就具有在本质上与正常的成年人完全不同的禀赋。这一观点阻碍 了科学家们用一般的心理学的规律和原则，对专业高手进行系统考察，并研究他们的表现。 我们认同专业高手的表现与正常人的表现具有本质的不同，我们甚至认同专业高手具备在本 质上与正常的成年人完全不同的，或者至少超出了正常成年人范围的禀赋和能力。然而，我 们否认由于天生禀赋的原因，使得这种不同一成不变。只有少数最极端的例外情况是由基因 所决定的。我们认为，在专业高手与正常成年人之间存在的差异，可以用下面一点来反映出 来，即是否在某一特定领域用一生的时间来刻意地完善自己的表现。”

Ericsson在这段有关专业高手的结论中有3个要点：第一个是“在某一特定领域”，即专 注；第二个是“用一生的时间”，即长期；第三个是“刻意地完善自己”，即用心。

㊀ Malcolm Gladwell，《o/Sttccaw》，Back Bay Books，June 7, 2011 〇

专注、长期和用心这三者要同时具备，才有可能做好我们新的惯常的行为，进而达到专 业高手的水平。

我们正常的成年人天天在走路，但没有几个能成为竞走世界冠军，这是因为我们虽然能 够做到长期，但是我们的主要精力会发散到诸如上班、照顾孩子、娱乐等其他事情上，做不 到专注和用心，使得我们的技能达不到专业水平。

很多人上学时在考试的压力下，能够专注和用心地把外语说得很好，可后来到了一家不 用外语的单位工作几年后，由于做不到长期坚持，使得以前学到的外语“全还给老师了”。 这说明即使我们暂时能够做到专注和用心，但如果没有长期坚持，以前能够做好的惯常行 为，到后来也会渐渐被我们荒废了。

但是，即使专注、长期和用心这3个要素都具备了，比如像在本章开头的故事中的凯 特，通过专门的、长期的和刻意的训练，伪装成德国人而打人党卫军内部，但是在生孩子的 疼痛这样的外在压力下，她养成的说德语的新习惯还是被以前说俄语的旧习惯所取代。那 么，有没有可能即使在外在压力很大的情况下，把经过新的惯常行为改造后的习惯回路永久 化呢？

对于上述问题，《习惯的力量》一书给出的回答是：“信仰本身。”“一旦人们学会信仰某 种东西，这种信仰就会扩展到生活的其他方面，直到他们开始相信自己能改变。”"大多数彻 底改变了自己生活的人，并没有遇到意义重大的事件或致命的灾难，而仅仅是因为加人了 团体，这个团体让他们相信改变是可能的，有时这个团体即使只有两个人，也会有同样的 效果。”

这里所说的信仰，不一定是像宗教那样的信仰，而是指任何人们所能长期相信的东西。 下面将描述的我个人的切身经历表明：任何能够使人们发生改变的团体，哪怕只有两个人, 都同样能够帮助人们识别和验证他以前所相信的东西，进而将其确定为他的信仰，并使得他 改变后的新习惯永久化。

7年前的我，尽管已经37岁，并且工作了 14年，但还是保持了我30年前上小学以来养 成的老习惯：腼腆胆小，缺乏自信，说话口吃。那时的我，即使在熟人面前也不敢讲话，更 不用说在生人面前了。

正在我为自己这个令我烦恼的老习惯发愁时，一位曾去加拿大留学的同事胡涛，介绍我 去参加一个非营利的教育机构——国际演讲会e—的一个俱乐部的会议。当我第一次走进这 样的团体时，在会议中看到讲台上用英语充满自信地侃侃而谈的演讲者，听着他们讲述自己 如何在俱乐部的演讲活动中找到自信，改变自己的故事，我被强烈地震撼了。这正是我所需 要的。很快，我就加入了国际演讲会。

我至今还对我在国际演讲会上的第一次演讲记忆犹新。演讲的头天晚上，我一夜未眠，

㊀参见：[www.toastmasters.orgo](http://www.toastmasters.orgo) 在床上辗转反侧，脑子里想的全是第二天演讲的事情：在讲台上我的手该放在哪里？眼睛该 看谁？要是忘词儿该怎么办？

第二天，我紧张万分地上了讲台，把我在网上找到的一篇英文文章背完后，就低头回到 了自己的座位。

很快，为我做点评的会员Vivian Kong走上讲台，她竟然夸我的肢体语言做得很好，并 且说我的语音语调能够有效地烘托气氛，而且在讲台上没有使用讲稿。她还希望我下次要把 声音再放大一些，再仔细按照国际演讲会的手册来准备演讲，以便讲得更好。

Vivian这种既给出表扬又提出建议的点评，给了我莫大的鼓励，令我相信自己可以发生 改变，来做好下一次演讲。

很快，我也学会了 Vivian这种“表扬加建议”的点评方式，除了自己做演讲，也开始为 其他会员提供点评。就这样一直参加俱乐部的会议，到如今已经7年了。

上个月，我的倶乐部负责教育的副会长Hubert Lin请我做倶乐部新会员培训的主持人。 我担心我的英语还不够好，而且到会的会员中没有外国人，所以我想用中文主持培训。后来 会员们反馈说都愿意讲英文，我就只好用英语主持了俱乐部新会员的培训。培训结束后，会 员们说我主持得很自信和流畅。

在20多个人面前用英文来主持培训，而且还担心自己英文不够好，这就是一个压力。 而在这个压力面前，我以前腼腆胆小的旧习惯没有跳出来，却仍然保持了自信流畅的新习 惯，这归功于7年来俱乐部这个团体给我的信心，让我确立了自己可以被改变的信仰。

顺便说一句，上个月培训前的那天晚上，我睡得很好。

国际演讲会这个大团体，让我确立了“我的演讲能力可以被改变”这个信仰。相似地， 结对编程这个两人的小团体，也让我相信我的编程能力也是可以被改进的。2013年4月，受 国外程序员做编程道场和办软件重构小组的启发，我也创办了一个公益的编程操练社区“北 京设计模式学习组”。我一方面自己组织了 10多次结对编程道场的活动，在其中分享自己的 TDD经验并组织大家结对编程，另一方面，我也参加其他的编程道场活动，和程序员结对 编程。无论是我组织的编程道场这样的大团体，还是我和另一位程序员一对一结对编程这样 的小团体，每次都能让我学习到新的技能和知识，并对自己的编程技能加以改进，这一切都 给了我自信，让我确立了“我能够改善我的编程技能”这个信仰，使得我在过去的一年中20 多次的技术大会和编程道场上的现场编程的压力下，始终保持了新养成的TDD的开发习惯。

前两年我的孩子学钢琴，孩子的妈妈给他找了一位钢琴老师来教他，这就是钢琴陪练。 11年前我学开车，找了一位师傅上路教我，这就是驾车陪练。陪练中的两个人，就是一个小 团体，会让学习者产生信心，确立信仰，进而将新学到的习惯永久化。我时常在想，编程也 像弹琴和开车那样，是门手艺，也需要结对编程这样的小团体来建立信仰，把新的编程习惯 永久化，但为何见不到编程陪练呢？或许将来在国内会有这样的编程陪练诞生呢。

再回到本章开头凯特的故事，她之所以在生孩子的疼痛面前没有保持讲德语的新习惯, 就是因为她的特工身份使得她无法找到一个能帮助她建立信心改变自己习惯的团体，进而形 成不了信仰，不能让她的新习惯永久化，最终让她的旧习惯冒了出来。

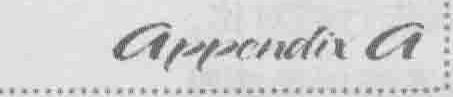
“天下没有不散的筵席。”在我们先用传统的设计驱动的开发方法和TDD的开发方法从 零开始做了酒店世界时钟的结对编程操练后，我们一起讨论了测试后行与测试先行的对比， 定义了烂代码，又一起继续做了驯服诸如答题闯关游戏Trivia、轮胎压力检测系统、自动取 号系统和网页文本转换系统这些已有的烂代码的结对编程操练，最后一起讨论了驯服烂代码 的步骤和习惯的养成方法。现在终于到了要和亲爱的读者说再见的时候了。在和您说临别赠 言之前，让我们回顾一下本章的内容：

1 )习惯是不能被消除的，而只能被代替。

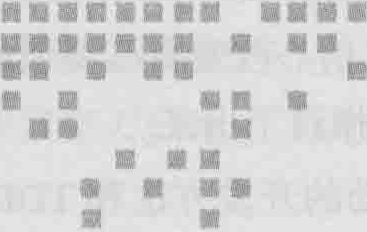
1. 如果把旧习惯的回路中的第1步中的“暗示”和第3步中的“奖赏”保留不动，然 后用一个新的惯常行为替换掉第2步中旧的惯常行为，那么大脑就可以在第一步的结尾处选 择我们的新习惯，从而养成这个新的习惯。
2. 如果想让我们新的惯常行为达到世界级的专业水平，我们需要花至少丨0000小时来 实践，即每天花3小时，连续进行10年的实践。
3. 只有专注、长期和用心地去做这10000小时的实践，才有可能让我们新的惯常的行 为达到专业高手的水平。
4. 只要我们加入一个能让我们相信改变是可能的团体，即使这个团体只有两个人，也 能让我们的新习惯永久地固化下来，并杜绝旧习惯的复发。

最后，我给亲爱的读者的临别赠言是：编程是门手艺，好的编程手艺就是编程匠艺，而 编程匠艺出自专注、长期和用心的结对编程操练。希望咱们能有机会结对编程，共同成为编 程艺匠。

编程操练简介



钃



附录A

编程操练，即英文的Code Kata,是《程序员修炼之道：从小工到专家》(《The Pragmatic Programmer: From Journeyman to Master》）一书的合著者、美国程序员 Dave Thomas 在 2003 年前后创造的字眼，表示一个编程练习，程序员可以通过反复地操练该练习来提高自身的编 程技能。Kata是一个日语片假名力、亡的英译，对应的汉字是“型”或者“形”，表示供单人 或双人进行操练的、经过仔细编排的动作模式e。

编程操练，说白了其实就是程序员练功时对一个编程题目进行练习。说起练功，可以很 自然地联想到京剧演员练习压腿和踢腿，相声演员练习绕口令和开声，和习武之人的独自站 粧和与人过招切磋。与之相比，程序员的练功似乎就没有那么讲究。自20世纪80年代面向 对象的编程语言出现以来至今，这30多年的时间里，国内的绝大部分程序员的所谓“练功”， 仅仅停留在读一些技术书籍和博客上，最多再照着示例代码写一些程序，运行一下而已。即 使在程序员编程水平很髙的国外，直到2004年5月，法国程序员Laurent Bossavit才写了一 篇有关多位程序员在一起做编程操练的“编程道场”的博客。这里的“编程道场”是英文 coding dojo的中译，其中的dojo同样也来自于日语，是片假名匕3的英译，对应的汉 字是“道场”，指一个正式的训练场所，来供学习日本武术的学生聚在一起进行操练。

编程道场意指多位程序员聚在一起，用两人结对或单人演示的形式，来做编程操练的 过程。编程操练和编程道场在国外已经发展了近10年，其影响力不断扩大，到2009年又出 现了编程静修e (code retreat)的新形式，即几十个程序员聚在一起，用一整天的时间来在编

㊀参见：[http://en.wikipedia.org/wiki/Kata。](http://en.wikipedia.org/wiki/Kata%e3%80%82)

㊁感谢来自上海的软件开发咨询师姚若舟向笔者建议“静修”这个译法。程道场中轮流结对做编程操练。编程操练、编程道场和编程静修这几年在国内也陆续得到一 些发展，比如出生于澳大利亚墨尔本的程序员Mike Long于2011年12月3日，在北京发起 了“编程静修全球日”（Gl〇balDay〇fC〇deretreate)北京站的活动，从那以后到撰写本书时， Mike每年12月都在北京举办一次编程静修的活动。笔者有幸参加了其中2012年和2013年 的活动。另外，在撰写本书时，笔者受《重构与模式》（《Refactoring to Patterns》）一书的作 者Joshua Kerievsky于1995年在美国纽约创办设计模式学习小组的启发，于2013年4月在 北京创办了 “bjdp.org北京设计模式学习组”，到撰写本书时，已举办18次活动，每次能吸 引8 ~ 20位程序员来进行结对操练编程技艺。这一切似乎都在表明，编写程序不再仅是按 照既定的软件架构或框架，来像垒砖那样被动地“填”代码，而是像唱京戏、说相声、练武 术那样，更加强调人的创造性，是一门需要反复操练才能悟道出师的手艺。

既然编程操练是供程序员在没有工作压力的情况下练功时所使用的，为了能够让程序员 们在操练时获得更有趣的体验，编程操练需要设计得“有趣”，即除了题目的内容可以是生 活中有意思的场景外，最好还能通过实现这个操练，练习一些有挑战性的技能，比如结对编 程和设计模式。

㊀参见：[http://globalday.coderetreat.org/。](http://globalday.coderetreat.org/%e3%80%82)

怎样在Windows系统中搭建编程操练环境

下面以Windows 7简体中文版（x64位版）为例来说明如何在Windows系统中搭建编程 操练环境。

1. )下载e并安装 JDK 8 (Java SE 8u20 )。

运行下载的安装文件jdk-8u20-windows-x64.exe。在安装JDK 8过程中，将JDK 1.8安 装在目录 C:\Program Files\Java\jdkl.8.0\_20 下，将 JRE 1.8 安装在目录 C:\Program Files\Java\ jrel\_8.0\_20 下〇

1. )下载s并解压 Maven (Apache Maven 3.2.3 )。

将下载下来的 Maven 的 ZIP 文件解压到目录 C:\Program Files (x86)\apache-maven-3\_2.3 下。

1. )能在命令行上运行Maven。

首先配置环境变量，新建JAVA\_HOME和M3\_HOME环境变量，分别指向JDK和 Maven的安装目录，然后修改Path环境变量，添加上面两个安装目录下的bin目录，使得能 在命令行上运行bin目录下的命令。

新建环境变量的方法：控制面板—查找env—选择“编辑系统环境变量”—系统变量 新建。

新建环境变量 JAVA\_HOME 为 C:\ProgramFiles\Java\jdkl .8.0\_20。

再新建环境变量 M3\_HOME 为 C:\Program Files (x86)\apache-maven-3.2.3。

修改环境变量PATH,在其值前面添加字符串“％JAVA\_HOME%\bin;%M3\_HOME%\bin;”。

㊀ JDK 8 下载地址：http://www.oracle,com/technetwork/es/java/javase/downloads/index.html0 ㊁ Maven 3 下载地址：[http://maven.apache.org/down丨oad.cgi。](http://maven.apache.org/down%e4%b8%a8oad.cgi%e3%80%82)

打开一个命令行窗口，运行命令“mvii-v”。如果能看到类型下面的输出，就表示JDK： 和Maven已经安装完成。

Apache Maven 3.2.3 (33f8c3el027c3ddde99d3cdebad2656a31e8fdf4; 2014-08-12T04:58:1

0+08:00)

Maven home: C:\Program Files (x86)\apache-maven-3.2.3

Java version: 1.8.0一20, vendor: Oracle Corporation

Java home: C:\Program Files\Java\jdkl.8.0\_20\jre

Default locale: zh\_CN, platform encoding: GBK

OS name: "windows 7", version: "6.1", arch: "amd64", family: "dos"

1. 下载e并安装 Git (Git-1.9.4-preview20140815)o

运行下载的Git的EXE文件，将其安装在C:\Program Files (x86)\Git目录下。

在Git安装过程中，勾选On the Desktop选项。这样安装完毕后，Git Bash命令行工具 就能出现在桌面上。

1. )从GitHub上复制源代码。

打开Git Bash命令行工具，运行“cd”命令，进人当前用户的home目录（以/c/Users/ ben为例)。再运行“mkdirkatas”命令，在当前目录下创建目录katas。然后运行“cdkatas” 命令进入该目录。

打开一个浏览器，访问 <https://github.com/wubin28/tbc-ticket-dispenser-java.git> 页面， 该页面显示了本书第17章中有关Mock的编程操练的题目源代码。在页面左上侧，单击 branch: master按钮，能显示一个下拉菜单，内有这个编程操练题目的一些分支（branch)。其 中，exercise分支是该编程题目的原题，subclass-and-override-method分支是第17章中介绍 的有关这个题目的解决方案。选择相应的分支，再单击上面的nn commits,就能在页面上浏 览历次提交的代码。

下面要把GitHub上的源代码从网上复制到本地，再用IntelliJ IDEA来编写并运行代码。

单击页面中下方右侧HTTPS clone URL右下方的“Copy to clipboard”按钮，将代码 URL复制到剪贴板上。

然后回到Git Bash命令行工具窗口中，输人命令“gitclone”，并选用该命令行窗口左上 角的相应菜单将刚刚复制的URL粘贴到命令行中，最终命令如下所示：

git clone <https://github.com/wubin28/tbc-ticket-dispenser-java.git>

按回车运行该命令后，如果命令行有类似下面的输出，则表示代码从网上成功复制到 本地。

Cloning into ftbc-ticket-dispenser-java\*... remote: Counting objects: 1167, done.

remote: Total 1167 (delta 0}, reused 0 (delta 0)

Receiving objects: 100% (1167/1167), 81.84 KiB | 68.00 KiB/s, done.

Resolving deltas: 100% (342/342), done.

Checking connectivity... done.

此时，运行“cdtbc-ticket-dispenser-java”命令，进入刚刚创建的目录。运行“gitbranch” 命令显示本地的分支，应该只能看到master分支。

1. )将exercise分支从网上复制到本地。

接着运行“git branch-a”命令显示所有的分支，如下所示：

\* master

remotes/origin/HEAD -> origin/master remotes/origin/bjdp-18-jd remotes/origin/exercise remotes/origin/extract-interface remotes/origin/master

remotes/origin/subclass-and-override-method remotes/origin/using-mockito

remotes/origin/using-mockito-error-fixing-oriented

此时可以看到remotes/origin/exercise分支是红色的，表示它还在网上，没有被复制到 本地。

运行下面的命令，将exercise分支从网上复制到本地，并将当前分支切换为该分支。

git checkout -b exercise origin/exercise

如果命令运行成功，会有类似下面的输出：

Branch exercise set up to track remote branch exercise from origin. Switched to a new branch 'exercise'

此时运行命令“gitbranch”，会有下面的输出：

$ git branch \* exercise master

这里exercise左边有一个星号，表示它是当前的分支。

1. )下载e并安装 IntelliJ IDEA 集成开发环境 Community 版（ideaIC-13.1.4b)〇 运行下载的IDEA的EXE安装文件，将软件安装到目录CAProgram Files (x86)\JetBrains\ IntelliJ IDEA Community Edition 13.1.4 下。

打开IDEA,在Quick Start窗口中选择Open Project,然后打开下面的pom.xml文件。

C:\Users\ben\katas\tbc-ticket-dispenser-java\td\pom.xml

此时，在IDEA底部的状态条中会显示“ Downloading plugins for td.,.”，表示IDEA正 在下载插件。待其消失后，表示IDEA已经安装好了。

1. )用IDEA运行exercise分支中的测试类。

将光标定位到左上部的Project窗口中文件夹td/src/test/java/kata.td下的TicketDispenser- Test类，然后按快捷键Ctrl+Shift+F10来运行这个测试类。

此时会显示一个对话框，显示如下错误：

Error: Cannot start compiler: the SDK is not specified for module "td". Specify the SDK at Project Structure dialog.

这个错误表示JDK还未在IDEA中配置好。单击该对话框中的OK按钮后，就会出现 Project Structure 对话框。其中 Module SDK:右边显示 <No Project SDK>,表示 JDK 未配好。 单击“New...”按钮，出现Set up Module SDK下拉菜单。从中选择JDK。再选择JDK的 Home Directory 为 C:\Program Files\Java\jdkl.8.0\_20。单击 OK 按钮。当出现 “ Set up created SDK on project?”时，选择Yes。然后单击OK按钮。

然后再次将光标定位到左上部的Project窗口中td/src/test/java/kata.td文件夹下的 TicketDispenserTest类，然后按快捷键Ctrl+Shift+F丨0,来运行这个测试类。

此时IDEA底部的状态条会显示一个旋转的光标，表示正在编译运行该测试类。等运行 完毕后，在中下方会出现测试运行成功的绿条。现在就可以在IDEA里的这个编程操练题目 的原题上进行编程了。

1. )根据代码提交的 Commit Message (CM)来使用 Git 阅读 subclass-and-override-method 分支中的代码。

下面来说明如何根据本书每段示例代码前所标出的CM来使用Git阅读代码。比如现 在想将代码恢复到书中所标出的CM的状态，即CM: Added TODO: Depending on a static method violates the Dependency Inversion Principle and Open-Closed Principle。

先运行下面的命令将subclass-and-override-method分支从网上复制到本地，并将其设置 为当前分支。

git checkout -b subclass-and-override-method origin/subclass-and-override-method

执行命令“gitbranch”，应该可以看到下面的输出：

exercise

master

\* subclass-and-override-method

此时可以运行命令“git log”来査看以往提交的所有CM。再运行“git log >git-log- subclass-and-override-method.txt”命令，将以往所有的CM提交记录都保存到一个TXT文

件中。

用一个文本编辑器打开这个TXT文件，查找“ CM: Added TODO: Depending on a static method violates the Dependency Inversion Principle and Open-Closed Principle.”，找到该字符 串所在行上方的一长串字母数字组成的SHA1值：dal69b7dbb5cel5ebe5c31a91a5b3403b6b 5b445。将该值复制下来。

然后运行 “git reset—hard dal69b7dbb5cel5ebe5c31a91a5b3403b6b5b445” 命令。此时能 看到类似下面这样的输出：

HEAD is now at dal69b7 Added TODO: Depending on a static method violates the Dependency Inversion Principle and Open-Closed Principle.

最后运行“gitk&”命令，能够看到一个图形化的Git工具。从中能够直观地看到本次 提交下代码的变化。

切换到IDEA,能看到其中的代码也相应地发生了变化，反映出当前这个CM的代码 状态。

■鼸■!

附录c

怎样在〇S )(系统中搭建编程操练环境

下面以OS X 10.9.4版为例来说明如何在Mac OS X系统中搭建编程操练环境。

1. )下载e并安装 JDK 8 (Java SE 8ull )。

在Finder的Downloads文件夹中，选中下载的文件jdk-8ull-macosx-x64.dmg，然后按 Command+O快捷键来打开这个文件，并进行安装。

1. )用 Homebrew ㊁来安装 Maven (Apache Maven 3.2.2 )〇

首先按快捷键Ctrl+Command+Space，打开右上角Spotlight输入框。在其中输人 terminal，来打开一个命令行窗口。将下述命令复制到该命令行窗口中，来安装Homebrew。

ruby -e "$ (curl -fsSL <https://raw.github.com/Homebrew/homebrew/go/install)n>

接下来运行命令“brew install mave”，来安装Maven。

1. )在命令行上运行Maven。

在命令行窗口运行命令“mvn-v”。如果能看到类型下面的输出，就表示JDK和Maven 已经安装完成。

Apache Maven 3.2.2 (45f7c06d68e745d05611f7fdl4efb6594181933e; 2014-06-17T21:51:42+08:00) Maven home: /usr/local/Cellar/maven/3.2.2/libexec Java version: 1.8.0一11, vendor: Oracle Corporation

Java home: /Library/Java/JavaVirtualMachines/jdkl.8.0\_11.jdk/Contents/Home/jre Default locale: en\_US, platform encoding: UTF-8

OS name: "mac os x", version: T, 10.9.4nr arch: "x86\_64", family: "mac" ㊀ JDK 8 下载地址：<http://www.oracle.com/technetwork/es/java/javase/downloads/index.html>, ㊁Homebrew是OS X系统下的一个软件包管理工具，参见：http://brew.sh。

1. 下载㊀并安装 Git (git-2.0.l-intel-universal-snow-leopard)〇

运行下载的 Git 的 DMG 文件 git-2.0.1-intel-universal-snow-leopard.dmg 进行安装。

将下面一行添加到文件/etc/paths的头部：

/usr/local/git/bin

在命令行上运行命令“git-version”，能看到类似下面这样的输出：

git version 2.0.1

5 )从GitHub上克隆源代码。

在命令行窗口中，运行“cd”命令，进入当前用户的home目录（以/Users/ben为例)。 再运行“mkdir katas”命令，在当前目录下创建目录katas。然后运行“cd katas”命令进人 该目录。

打开一个浏览器，访问 <https://github.com/wubin28/tbc-ticket-dispenser-java.git> 页面， 该页面显示了本书第17章中有关Mock的编程操练的题目源代码。在页面左上侧，单击 “branch: master”按钮，能显示一个下拉菜单，内有这个编程操练题目的一些分支（branch)。 其中，exercise分支是该编程题目的原题，subclass-and-override-method分支是第17章中介 绍的有关这个题目的解决方案。选择相应的分支，再单击上面的nn commits,就能在页面上 浏览历次提交的代码。

下面要把GitHub上的源代码从网上复制到本地，再用IntelliJ IDEA来编写并运行代码。

单击页面中下方右侧HTTPS clone URL右下方的“Copy to clipboard”按钮，将代码 URL复制到剪贴板上。

然后回到命令行窗口中，敲入命令“git clone”，并用快捷键Command+V将刚刚复制的 URL粘贴到命令行中，最终命令如下所示：

git clone https ://github. com/wubin28/tbc-ticket-dispenser-java . git

按回车运行该命令后，如果命令行有类似下面的输出，则表示代码从网上成功复制到 本地。

Cloning into 'tbc-ticket-dispenser-java\*...

remote: Counting objects: 1167, done.

remote: Total 1167 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (1167/1167), 81.84 KiB | 68.00 KiB/s, done.

Resolving deltas: 100% (342/342), done.

Checking connectivity... done.

此时，运行“cdtbc-ticket-dispenser-java”命令，进入刚刚创建的目录。运行“gitbranch”

命令显示本地的分支，应该只能看到master分支。

1. )将exercise分支从网上复制到本地。

接着运行“git branch-a”命令显示所有的分支，如下所示：

* master

remotes/origin/HEAD -> origin/master remotes/origin/bjdp-18-jd remotes/origin/exercise remotes/origin/extract-interface remotes/origin/master

remotes/origin/subclass-and-override-method remotes/origin/using-mockito

remotes/origin/using-mockito-error-fixing-oriented

此时可以看到remotes/origin/exercise分支是红色的，表示它还在网上，没有被复制到 本地。

运行下面的命令，将exercise分支从网上复制到本地，并将当前分支切换为该分支。

git checkout -b exercise origin/exercise

如果命令运行成功，会有类似下面的输出：

Branch exercise set up to track remote branch exercise from origin.

Switched to a new branch 'exercise \*

此时运行“git branch”命令，会有下面的输出：

$ git branch

* exercise master

这里exercise左边有一个星号，表示它是当前的分支。

1. )下载e并安装 IntelliJ IDEA 集成开发环境 Community 版（idealC-13.1.4b)。

运行下载的IDEA的DMG安装文件ideaIC-13.1.4b.dmg进行安装。

打开IDEA，在Open窗口中打开下面的pom.xml文件。

/ Users / ben / katas / tbc-ticket-dispenser-java / td / pom.xml

此时，在IDEA底部的状态条中会显7K “ Downloading plugins for td...”，表示IDEA正 在下载插件。待其消失后，表示IDEA已经安装好了。

1. )用IDEA运行exercise分支中的测试类。

将光标定位到左上部的Project窗口中文件夹td/src/test/java/kata.td下的TicketDispenser- Test类，然后按快捷键Ctri+ShifHFlO来运行这个测试类。

此时会显示一个对话框，显示如下错误：

Error: Cannot start compiler: the SDK is not specified for module "td". Specify the SDK at Project Structure dialog.

这个错误表示JDK还未在IDEA中配置好。单击该对话框中的OK按钮后，就会出现 Project Structure 对话框。其中 Module SDK:右边显示 <No Project SDK>，表示 JDK 未配好。 单击“New...”按钮，出现Set up Module SDK下拉菜单。从中选择JDIC。再选择JDK的 Home Directory 为 /Library/Java/JavaVirtualMachines/jdk 1.8.0\_1 l.jdk/Contents/Home。单击 0K 按钮。当出现“Set up created SDK on project?”时，选择Yes。然后单击OK按钮。

然后再次将光标定位到左上部的Project窗口中td/src/test/java/kata.td文件夹下的 TicketDispenserTest类，然后按快捷键Ctrl+Shift+FlO，来运行这个测试类。

此时IDEA底部的状态条会显示一个旋转的光标，表示正在编译运行该测试类。等运行 完毕后，在中下方会出现测试运行成功的绿条。现在就可以在IDEA里的这个编程操练题目 的原题上进行编程了。

1. )根据代码提交的 Commit Message (CM)来使用 Git 阅读 subclass-and-override-method 分支中的代码。

下面来说明如何根据本书每段示例代码前所标出的CM来使用Git阅读代码。比如现 在想将代码恢复到书中所标出的CM的状态，即CM: Added TODO: Depending on a static method violates the Dependency Inversion Principle and Open-Closed Principle.

先运行下面的命令将subclass-and-override-method分支从网上复制到本地，并将其设置 为当前分支。

git checkout -b subclass-and-override-method origin/subclass-and-override-method

执行命令“gitbranch”，应该可以看到下面的输出：

exercise

master

★ subclass-and-override-method

此时可以运行命令“git log”来查看以往提交的所有CM。再运行“git log > git-log- subclass-and-override-method.txt”命令，将以往所有的CM提交记录都保存到一个TXT文 件中。

用一个文本编辑器打开这个TXT文件，査找“ CM: Added TODO: Depending on a static method violates the Dependency Inversion Principle and Open-Closed Principle.”，找到该字符串 所在的行的上方的一长串字母数字组成的SHA1值：dal6%7dbb5cel5ebe5c31a91a5b3403b6b 5b445。将该值复制下来。

然后运行命令 “ git reset —hard dal69b7dbb5cel5ebe5c31a91 a5b3403b6b5b445”。此时能 看到类似下面这样的输出：

HEAD is now at dal69b7 Added TODO: Depending on a static method violates the Dependency Inversion Principle and Open-Closed Principle.

最后运行“gitk&”命令，能够看到一个图形化的Git工具。从中能够直观地看到本次 提交下代码的变化。

切换到IDEA,能看到其中的代码也相应地发生了变化，反应出当前这个CM的代码 状态。

附录D

怎样在Linux系统中搭建编程操练环境

下面以Ubuntu 14.04.1 LTS版为例来说明如何在Linux系统中搭建编程操练环境。

1. )下载e并安装 JDK 8 (Java SE 8u5 )〇

在Terminal命令行窗口中，用管理员的权限将下载下来的安装文件jdk-8u5-linux-i586. tar.gz 复制到/opt 目录下。然后运行 “sudotarxvzfjdk-8u5-linux-i586.tar.gz” 命令将该文件 解压到该目录下。

1. )下载并安装 Maven (Apache Maven 3.2.1 )〇

在Terminal命令行窗口中，用管理员的权限，将下载下来的安装文件apache-maven- 3.2.1-bin. tar.gz 复制至/opt 目录下。然后运行 “ sudo tar xvzf apache-maven-3.2.1-bin. tar.gz” 命令将该文件解压到该目录下。

1. )能在命令行上运行Maven。

需要配置一下环境变量才能在命令行中运行Maven命令。

打开文件〜/.bashrc，在文件结尾处添加下面几行：

JAVA\_HOME=/opt/jdkl.8.0\_05; export JAVA一HOME M3\_HOME=/opt/apache-maven-3.2.1/ export M3\_H0ME PATH=$JAVA\_HOME/bin:$M3\_H0ME/bin:$PATH; export PATH

重新打开一个命令行窗口，运行“mvn-v”命令。如果能看到类型下面的输出，就表示 JDK和Maven已经安装完成。

Apache Maven 3.2.1 (ea8b2b07643dbblb84b6dl6elf08391b666bcle9; 2014-02-15T01:37:52+08:00)

Maven home: /opt/apache-maven-3.2.1

Java version: 1.8.0一05, vendor: Oracle Corporation

Java home: /opt/jdkl.8.0\_05/jre

Default locale: en\_US, platform encoding: UTF-8

OS name: "linux", version: "3.13.0-32-generic", arch: "i386", family: "unix"

1. 下载e并安装 Git (git 1.9.1)。

在命令行窗口中运行“apt-get install git”命令来进行安装。

在命令行上运行“git-version”命令，能看到类似下面这样的输出：

git version 1.9.1

5 )从GitHub上复制源代码。

在命令行窗口中，运行“cd”命令，进入当前用户的home目录（以/home/ben为例）。 再运行“mkdir katas”命令，在当前目录下创建目录“katas”。然后运行“cd katas”命令进 人该目录。

打开一个浏览器，访问 <https://github.com/wubin28/tbc-ticket-dispenser-java.git> 页面, 该页面显示了本书第17章中有关Mock的编程操练的题目源代码。在页面左上侧，单击 “branch: master”按钮，能显示一个下拉菜单，内有这个编程操练题目的一些分支（branch)。 其中，exercise分支是该编程题目的原题，subclass-and-override-method分支是第17章中介 绍的有关这个题目的解决方案。选择相应的分支，再单击上面的nn commits,就能在页面上 浏览历次提交的代码。

下面要把GitHub上的源代码从网上复制到本地，再用IntelliJ IDEA来编写并运行代码。

单击页面中下方右侧HTTPS done URL右下方的“Copy to clipboard”按钮，将代码 URL复制到剪贴板上。

然后回到命令行窗口中，输人“gitclone”命令，并用快捷键Shift+Ctrl+V将刚刚复制 的URL粘贴到命令行中，最终命令如下所示：

git clone <https://github.com/wubin28/tbc-ticket-dispenser-java.git>

按回车运行该命令后，如果命令行有类似下面的输出，则表示代码从网上成功复制到 本地。

Cloning into 'tbc-ticket-dispenser-java'...

remote: Counting objects: 1167, done.

remote: Total 1167 (delta 0), reused 0 (delta 0)

Receiving objects: 100% (1167/1167), 81.84 KiB | 68.00 KiB/sf done. Resolving deltas: 100% (342/342), done.

Checking connectivity... done.

此时，运行“cdtbc-ticket-dispenser-java”命令，进人刚刚创建的目录。运行“gitbranch” 命令显示本地的分支，应该只能看到master分支。

1. )将exercise分支从网上复制到本地。

接着运行“git branch-a”命令显示所有的分支，如下所示：

* master

remotes/origin/HEAD -> origin/master

remotes/origin/bjdp-18-jd

remotes/origin/exercise

remotes/origin/extract-interface

remotes/origin/master

remotes/origin/subclass-and-override-method remotes/origin/using-mockito

remotes/origin/using-raockito-error-fixing-oriented

此时可以看到remotes/origin/exercise分支是红色的，表示它还在网上，没有被复制到 本地。

运行下面的命令，将exercise分支从网上复制到本地，并将当前分支切换为该分支。

git checkout -b exercise origin/exercise

如果命令运行成功，会有类似下面的输出：

Branch exercise set up to track remote branch exercise from origin.

Switched to a new branch 'exercise'

此时运行“gitbranch”命令，会有下面的输出：

$ git branch

* exercise master

这里exercise左边有一个星号，表示它是当前的分支。

1. )下载a并安装丨ntelliJ IDEA集成开发环境Community版（ideaIC-13.1.4b)。

在Termina丨命令行窗口中，用管理员的权限将下载下来的安装文件ideaIC-13.1.4b.tar.gz 复制到/opt目录下。然后运行“sudotarxvzfidea丨C-13.1.4b.tar.gz”命令将该文件解压到该目 录下。

需要配置一下环境变量才能在命令行中运行IDEA。

打开文件~/上&5111^,在前面所添加的M3\_H0ME —行下面，添加下面一行：

IDEAjHOME=/opt/idea-IC-135.1230; export IDEA\_HOME

再修改PATH的那行，将$lDEA\_HOME/bin添加进去：

〆

㊀ IntelliJ IDEA 下载地址：[http://www.jetbrains.com/idea/(iowiiload/〇](http://www.jetbrains.com/idea/(iowiiload/%e3%80%87)

PATH=$JAVA\_H0ME/bin:$M3\_H0ME/bin:$IDEA\_H〇ME/bin:$PATH; export PATH

新打开一个命令行窗口，运行“idea.sh&”命令打开IDEA,在Open窗口中打开下面 的pom.xml文件。

/ home / ben / katas / tbc-ticket-dispenser-java / td / pom.xml

此时，在IDEA底部的状态条中会显示“ Downloading plugins for td...”，表示IDEA正 在下载插件。待其消失后，表示IDEA已经安装好了。

8 )用IDEA运行exercise分支中的测试类。

将光标定位到左上部的Project窗口中文件夹td/src/test/java/kata.td下的 TicketDispenserTest类，然后按快捷键Ctrl+Shift+F10来运行这个测试类。

此时会显示一个对话框，显示如下错误：

Error: Cannot start compiler: the SDK is not specified for module "td". Specify the SDK at Project Structure dialog.

这个错误表示JDK还未在IDEA中配置好。单击该对话框中的OK按钮后，就会出现 Project Structure 对话框。其中 Module SDK:右边显示 <No Project SDK>,表示 JDK 未配好。 单击“New...”按钮，出现Set up Module SDK下拉菜单。从中选择JDK。再选择JDK的 Home Directory 为/opt/jdkl.8.0\_05。单击 0K。当出现 “Set up created SDK on project?” 时， 选择Yes。然后单击OK按钮。

然后再次将光标定位到左上部的Project窗口中td/src/test/java/kata.td文件夹下的 TicketDispenserTest类，然后按快捷键Ctrl+Shift+F10,来运行这个测试类。

此时IDEA底部的状态条会显示一个旋转的光标，表示正在编译运行该测试类。等运行 完毕后，在中下方会出现测试运行成功的绿条。现在就可以在IDEA里的这个编程操练题目 的原题上进行编程了。

9 )根据代码提交的 Commit Message (CM)来使用 Git 阅读 subclass-and-override-method 分支中的代码。

下面来说明如何根据本书每段示例代码前所标出的CM来使用Git阅读代码。比如现 在想将代码恢复到书中所标出的CM的状态，即CM: Added TODO: Depending on a static method violates the Dependency Inversion Principle and Open-Closed Principle。

先运行下面的命令将subclass-and-override-method分支从网上复制到本地，并将其设置 为当前分支。

git checkout -b subclass-and-override-method origin/subclass-and-override-method

执行“git branch”命令，应该可以看到下面的输出：

exercise

master

\* subclass-and-override-method

此时可以运行“ git log”命令来査看以往提交的所有CM。再运行“ git log > git-log- subclass-and-override-method.txt”命令，将以往所有的CM提交记录都保存到一个TXT文 件中。

用一个文本编辑器打开这个TXT文件，查找“ CM: Added TODO: Depending on a static method violates the Dependency Inversion Principle and Open-Closed Principle. ”，找至(I该字符 串所在行上方的一长串字母数字组成的SHA1值：dal69b7dbb5cel5ebe5c31a91a5b3403b6b 5b445。将该值复制下来。

然后运行命令 “ git reset —hard dal69b7dbb5cel5ebe5c31a91a5b3403b6b5b445”。此时能 看到类似下面这样的输出：

HEAD is now at dal69b7 Added TODO: Depending on a static method violates the Dependency Inversion Principle and Open-Closed Principle.

最后运行“gitk&”命令，能够看到一个图形化的Git工具。从中能够直观地看到本次 提交下代码的变化。

切换到IDEA,能看到其中的代码也相应地发生了变化，反应出当前这个CM的代码 状态。

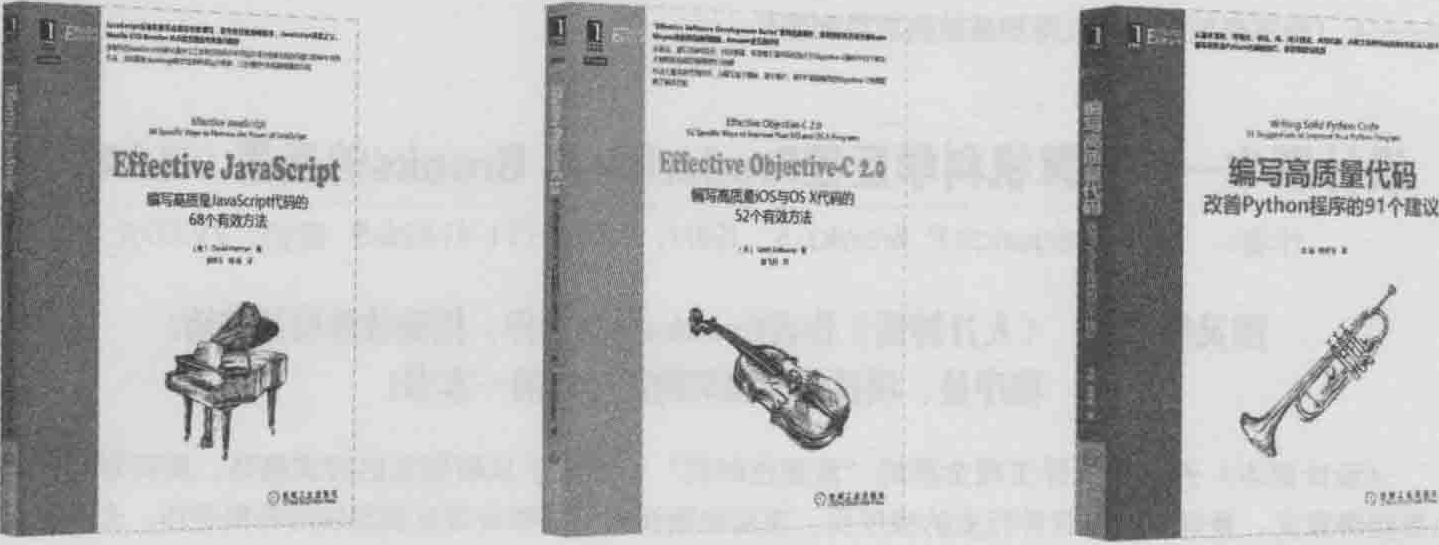
推荐阋读

r

编写高质量代码

改善Java程序的151个置IS鏞辟团置代n

**S8c#g9Ki57** 个鼸



|\*1 TomO«M»rw T1imilhyU\*H» ■ «S «H 31

,....,•'父.

t

\_

人件

(原书第3版>

推荐阋读

I

，is

设计原本

ltd机料字巨S Frederick P. Brooks的反思

，rrfr\_i,....…+’一 ：

I l. j I: iy

**« EBXM «**

**nn**：：**k:j**

Eaojrs from 〇 Comiwlw Sdeofel

ytJiiiiii

©8A4IAy

人件（原书第3版）

作者:(美）Tom DeMarco 等丨SBN: 978-7-1U-47436-4 定价：69.00元

公认对软件行业影响最大最具价值的著作之一，历时15年全面更新  
与《人月神话》共同被誉为软件图书领域最为璀璨的“双子星”，近30年全球畅销不衰

在软件管理领域.很少有著作能够与本书媲美。全书从管理人力资源、创建健康的办公环境、雇用并留 用正确的人、高效团队形成、改造企业文化和快乐工作等多个角度阐释了如何思考和管理软件开发的最大问 题一人（而不是技术），以得到高效的项目和团队。

设计原本 计算机科学巨匠Frederick P. Brooks的反思（经典珍藏）

作者：（美）Frederick P, Brooks,Jr. ISBN: 978-7-11 卜41626-5 定价：79.00元

图灵奖得主、《人月神话》作者Brooks封笔之作，揭秘软件设计神话！

程序员、项目经理和架构师必读的一本书！

《设计原本》开启了软件工程全新的“后理性时代”，完成了从破到立的圆满循环，具有划时代的重 大里程碑意义，是每位从事软件行业的程序员、项目经理和架构师都应该反复研读的经典著作。全书以设计 理念为核心，从对设计模型的探讨入手，讨论了有关设计的若干重大问题：设计过程的建立、设计协作的规 划、设计范本的固化、设计演化的管控，以及设计师的发现和培养。

作为一个程序员，如何才能提升编程技艺、驯服烂代码，从而写出高质量的代码？ Kent Beck、Martin Fowler、Michael C. Feathers、Robert C. Martin、Joshua Kerievsky和 Gerard Meszaros这些大师们从编程态度、思想、方法、技艺和习惯等方面给出了大量的建 议，井以著作的方式记录下来。大师们的著作在全球范围内广为传播和传承，是人类软件 开发史上的宝贵财富。

虽然大师们的著作字字珠机、如同珍宝，但是编程技艺的提升绝不是一日之功，不仅 要有大师们的指导，还需要大量的编程操练，在实践中去领会和贯通大师们的建议，从而 真正实现编程技艺的提升。大师们的珠玉在前，本书则完全从实践角度切入，以编程操练 的方式讲述了识别和驯服烂代码的过程、思想和方法，希望能帮助广大的程序员们真正领 悟和吸收大师们的智慧。

本书的写作同样遵循和实践了驯服烂代码的原则和方法，书中所有内容和素材均来源 于作者组织的多达18次的编程操练，本书的写作方式和内容组织方式也在写作过程中根据 编程操练中收集到的反馈信息不断重构和优化，以确保为读者带来“高内在质量”的内容 和阅读体验。

投稿热线：（010)88379604

客服热线：（010)88378991 88361066

购书热线：（010)68326294 88379649 68995259



**T\_M TI** 關**D [\_E**

-Discerning the 1^0 in the Code Katas

华章网站：[www.hzbook.com](http://www.hzbook.com) 网上购书：www.china\_pub.com 数字阅读：[www.hzmedia.com.cn](http://www.hzmedia.com.cn)



上架指导：计爾机/程序设计

ISBN 978-7-111-48155-3



定价：69.00元

1. e为节省篇幅，这里仅列出部分代码。

   ㊀单一职责原则SRP( Single-Responsibility Principle)指的是一个类发生变化的原因只能有一个；开闭原则 OCP (Open/Closed Principle)指的是软件的实体（如类、模块和函数等）应该对扩展开放，对修改关闭。 参见RobertC.Martin所著《敏捷软件开发：原则、模式与实践》一书。 [↑](#footnote-ref-1)