## 深入理解简单设计

测试驱动开发遵守了测试—开发—重构的闭环。测试设定了新功能的需求期望，并为功能实现提供了保护；开发让实现真正落地，满足产品功能的期望；重构则是为了打磨代码质量，降低软件的维护成本。期望—实现—改进的螺旋上升态势，为测试驱动开发闭环提供了源源不断的动力。缺少任何一个环节，闭环都会停滞不动。没有期望，实现就失去了前进的目标；没有实现，期望就成为了空谈；没有改进，前进的道路就会越来越窄，突破就会变得愈发地艰难。

### 重构改进的目标

若已有清晰的用户需求，为其设定期望然后寻求实现，这并非难事。但是改进的标准却是模糊的。要达到什么样的目标才符合重构的要求？Martin Fowler 的回答是让代码消除坏味道，他甚至编写了专著《重构：改善既有代码的设计》来总结他在多年开发和咨询工作中遇见的所有坏味道，如下表所示：

|  |  |
| --- | --- |
| 坏味道 | 问题 |
| 重复代码 | 函数包含相同代码；两个子类包含相同代码；两个不相干类包含相同代码 |
| 过长函数 | 函数越长越难以理解 |
| 过大的类 | 一个类要做的事情太多，可能导致重复代码 |
| 过长参数列表 | 参数列表过长会导致难以理解，太多参数会造成前后不一致、不易使用，会导致频繁修改 |
| 发散式变化 | 一个类因为不同原因在不同的方向发生变化，指“一个类受多种变化的影响” |
| 霰弹式修改 | 与发散式修改相反，因为一个变化导致多个类都需要作出修改 |
| 依恋情结 | 函数对某个类的兴趣高过对自己所处类的兴趣 |
| 数据泥团 | 两个类中相同的字段、许多函数签名中相同的参数。这些总是绑在一起出现的数据应该拥有属于它们自己的对象 |
| 基本类型偏执 | 基本类型不能很好地表现某个概念 |
| Switch 语句 | Switch 语句的问题在于重复 |
| 平行继承体系 | 每当你为某个类增加一个子类，必须也为另一个类相应增加一个子类 |
| 冗余类 | 一个类的所得不值其身价，就是冗余的，应该去掉 |
| 夸夸其谈未来 | 过度设计，过度抽象 |
| 临时字段 | 某个实例变量仅为某种特定情况而设 |
| 消息链条 | 如果你看到用户向一个对象请求另一个对象，然后再向后者请求另一个对象，然后再请求另一个对象，这就是消息链条 |
| 中间人 | 过度运用委托，例如某个类接口有一半的函数都委托给其他类 |
| 狎昵关系 | 两个类过于亲密，花费太多时间去探究彼此的 private 成分 |
| 异曲同工的类 | 如果两个函数做同一件事，却有着不同的签名 |
| 数据类 | 拥有一些字段以及访问这些字段的函数，除此之外没有其他方法 |
| 被拒绝的馈赠 | 子类应该继承超类的函数和数据，但如果它们不想或不需要继承，意味着继承体系设计错误 |
| 过多的注释 | 一段代码有着长长的注释，但这些注释之所以存在是因为代码很糟糕 |

数一数，Martin Fowler 一共提出了二十一种代码坏味道。要记住所有的坏味道并非易事，更何况我们还得保持嗅觉的敏感性，一经察觉代码的坏味道，就得尽量着手重构。经科学家研究证明，人类的短时记忆容量大约为 7±2，如果一时无法记住所有的坏味道，则可遵循 Kent Beck 的简单设计原则。我在 1-11《领域实现模型》中已经简单地介绍了简单设计原则，内容为：

* 通过所有测试（Passes its tests）
* 尽可能消除重复（Minimizes duplication）
* 尽可能清晰表达（Maximizes clarity）
* 更少代码元素（Has fewer elements）
* 以上四个原则的重要程度依次降低。

最后一个原则说明前面四个原则是依次递进的：功能正确，减少重复，代码可读是简单设计的根本要求。一旦满足这些要求，就不能创建更多的代码元素去迎合未来可能并不存在的变化，避免过度设计。

### 简单设计的量化标准

在满足需求的基本前提下，简单设计其实为代码的重构给出了三个量化标准：重复性、可读性与简单性。重复性是一个客观的标准，可读性则出于主观的判断，故而应优先考虑尽可能消除代码的重复，然后在此基础上保证代码清晰地表达设计者的意图，提高可读性。只要达到了重用和可读，就应该到此为止，不要画蛇添足地增加额外的代码元素，如变量、函数、类甚至模块，保证实现方案的简单。

第四个原则是“奥卡姆剃刀”的体现，更加文雅的翻译表达即“如无必要，勿增实体”。人民大学的哲学教授周濂在解释奥卡姆剃刀时，如是说道：

作为一个极端的唯名论者，奥卡姆的威廉（William of Occam，1280—1349）主张个别的事物是真实的存在，除此之外没有必要再设立普遍的共相，美的东西就是美的，不需要再废话多说什么美的东西之所以为美是由于美，最后这个美，完全可以用奥卡姆的剃刀一割了之。

这个所谓“普遍的共相”就是一种抽象。在软件开发中，那些不必要的抽象反而会产生多余的概念，实际会干扰代码阅读者的判断，增加代码的复杂度。因此，简单设计强调恰如其分的设计，若实现的功能通过了所有测试，就意味着满足了客户的需求，这时，只需要尽可能消除重复，清晰表达了设计者意图，就不可再增加额外的软件元素。若存在多余实体，当用奥卡姆的剃刀一割了之。

### 一个实例

让我们通过重构一段 FitNesse 代码来阐释简单设计原则。这段代码案例来自 Robert Martin 的著作《代码整洁之道》。Robert Martin 在书中给出了对源代码的三个重构版本，这三个版本的演化恰好可以帮助我们理解简单设计原则。

重构前的代码初始版本是定义在 HtmlUtil 类中的一个长函数：

public static String testableHtml(PageData pageData, boolean includeSuiteSetup) throws Exception {  
 WikiPage wikiPage = pageData.getWikiPage();  
 StringBuffer buffer = new StringBuffer();  
 if (pageData.hasAttribute("Test")) {  
 if (includeSuiteSetup) {  
 WikiPage suiteSetupPage = PageCrawlerImpl.getInheritedPage(SuiteResponder.SUITE\_SETUP\_NAME, wikiPage);  
 if (suiteSetupPage != null) {  
 WikiPagePath pagePath = wikiPage.getPageCrawler().getFullPath(suiteSetupPage);  
 String pagePathName = PathParser.render(pagePath);  
 buffer.append("\n!include -setup .")  
 .append(pagePathName)  
 .append("\n");  
 }  
 }  
 WikiPage setupPage = PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);  
 if (setupPage != null) {  
 WikiPagePath setupPath = wikiPage.getPageCrawler().getFullPath(setupPage);  
 String setupPathName = PathParser.render(setupPath);  
 buffer.append("\n!include -setup .")  
 .append(setupPathName)  
 .append("\n");  
 }  
 }  
 buffer.append(pageData.getContent());  
 if (pageData.hasAttribute("Test")) {  
 WikiPage teardownPage = PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);  
 if (teardownPage != null) {  
 WikiPagePath tearDownPath = wikiPage.getPageCrawler().getFullPath(teardownPage);  
 String tearDownPathName = PathParser.render(tearDownPath);  
 buffer.append("\n")  
 .append("!include -teardown .")  
 .append(tearDownPathName)  
 .append("\n");  
 }  
 if (includeSuiteSetup) {  
 WikiPage suiteTeardownPage = PageCrawlerImpl.getInheritedPage(SuiteResponder.SUITE\_TEARDOWN\_NAME, wikiPage);  
 if (suiteTeardownPage != null) {  
 WikiPagePath pagePath = wikiPage.getPageCrawler().getFullPath(suiteTeardownPage);  
 String pagePathName = PathParser.render(pagePath);  
 buffer.append("\n!include -teardown .")  
 .append(pagePathName)  
 .append("\n");  
 }  
 }  
 }  
 pageData.setContent(buffer.toString());  
 return pageData.getHtml();  
 }

假定这一个函数已经通过了测试，按照简单设计的评判步骤，我们需要检查代码是否存在重复。显然，在上述代码的 6~13 行、15~22 行、26~34 行以及 36~43 行四个地方都发现了重复或相似的代码。这些代码的执行步骤像一套模板：

* 获取 Page
* 若 Page 不为 null，则获取路径
* 解析路径名称
* 添加到输出结果中

这套模板的差异部分可以通过参数差异化完成，故而可以提取方法：

private static void includePage(WikiPage wikiPage, StringBuffer buffer, String pageName, String sectionName) {  
 WikiPage suiteSetupPage = PageCrawlerImpl.getInheritedPage(pageName, wikiPage);  
 if (suiteSetupPage != null) {  
 WikiPagePath pagePath = wikiPage.getPageCrawler().getFullPath(suiteSetupPage);  
 String pagePathName = PathParser.render(pagePath);  
 buildIncludeDirective(buffer, sectionName, pagePathName);  
 }  
 }  
  
 private static void buildIncludeDirective(StringBuffer buffer, String sectionName, String pagePathName) {  
 buffer.append("\n!include ")  
 .append(sectionName)  
 .append(" .")  
 .append(pagePathName)  
 .append("\n");  
 }

在提取了 includePage() 方法后，就可以消除四段几乎完全相似的重复代码。重构后的长函数为：

public static String testableHtml(PageData pageData, boolean includeSuiteSetup) throws Exception {  
 WikiPage wikiPage = pageData.getWikiPage();  
 StringBuffer buffer = new StringBuffer();  
  
 if (pageData.hasAttribute("Test")) {  
 if (includeSuiteSetup) {  
 includePage(wikiPage, buffer, SuiteResponder.SUITE\_SETUP\_NAME, "-setup");  
 }  
 includePage(wikiPage, buffer, "SetUp", "-setup");  
 }  
 buffer.append(pageData.getContent());  
 if (pageData.hasAttribute("Test")) {  
 includePage(wikiPage, buffer, "TearDown", "-teardown");  
 if (includeSuiteSetup) {  
 includePage(wikiPage, buffer, SuiteResponder.SUITE\_TEARDOWN\_NAME, "-teardown");  
 }  
 }  
 pageData.setContent(buffer.toString());  
 return pageData.getHtml();  
 }

从重复性角度看，以上代码已经去掉了重复。当然，也可以将 pageData.hasAttribute(“Test”) 视为重复，因为该表达式在第 5 行和第 12 行都出现过，表达式用到的常量 “Test” 也是重复。不过，你若认为这是从代码可读性角度对其重构，也未尝不可：

private static boolean isTestPage(PageData pageData) {  
 return pageData.hasAttribute("Test");  
 }

重构后的 testableHtml() 方法的可读性仍有不足之处，例如方法的名称，buffer 变量名都没有清晰表达设计意图，对 Test 和 Suite 的判断增加了条件分支，给代码阅读制造了障碍。由于 includePage() 方法是一个通用方法，未能清晰表达其意图，且传递的参数同样干扰了阅读，应该将各个调用分别封装为表达业务含义的方法，例如定义为 includeSetupPage()。当页面并非测试页面时， pageData 的内容无需重新设置，可以直接通过 getHtml() 方法返回。因此，添加页面内容的第 11 行代码还可以放到 isTestPage() 分支中，让逻辑变得更加紧凑：

public static String renderPage(PageData pageData, boolean includeSuiteSetup) throws Exception {  
 if (isTestPage(pageData)) {  
 WikiPage testPage = pageData.getWikiPage();  
 StringBuffer newPageContent = new StringBuffer();  
  
 includeSuiteSetupPage(testPage, newPageContent, includeSuiteSetup);  
 includeSetupPage(testPage, newPageContent);  
 includePageContent(testPage, newPageContent);  
 includeTeardownPage(testPage, newPageContent);  
 includeSuiteTeardownPage(testPage, newPageContent, includeSuiteSetup);  
  
 pageData.setContent(buffer.toString());  
 }   
 return pageData.getHtml();  
 }

无论是避免重复，还是清晰表达意图，这个版本的代码都要远胜于最初的版本。Robert Martin 在《代码整洁之道》中也给出了他重构的第一个版本：

public static String renderPageWithSetupsAndTeardowns(PageData pageData, boolean isSuite) throws Exception {  
 boolean isTestPage = pageData.hasAttribute("Test");  
 if (isTestPage) {  
 WikiPage testPage = pageData.getWikiPage();  
 StringBuffer newPageContent = new StringBuffer();  
 includeSetupPages(testPage, newPageContent, isSuite);  
 newPageContent.append(pageData.getContent());  
 includeTeardownPages(testPage, newPageContent, isSuite);  
 pageData.setContent(newPageContent.toString());  
 }  
 return pageData.getHtml();  
 }

对比我的版本和 Robert Martin 的版本，我认为 Robert Martin 的当前版本仍有以下不足之处：

* 方法名称过长，暴露了实现细节
* isTestPage 变量不如 isTestPage() 方法的封装性好
* 方法体缺少分段，不同的意图混淆在了一起

最关键的不足之处在于第 7 行代码。对比第 7 行和第 6、8 两行代码，虽然都是一行代码，但其表达的意图却有风马牛不相及的违和感。这是因为第 7 行代码实际暴露了将页面内容追加到 newPageContent 的实现细节，第 6 行和第 8 行代码却隐藏了这一实现细节。这三行代码没有处于同一个抽象层次，违背了“单一抽象层次原则（SLAP）”。

Robert Martin 在这个版本基础上，继续精进，给出了重构后的第二个版本：

public static String renderPageWithSetupsAndTeardowns(PageData pageData, boolean isSuite) throws Exception {  
 if (isTestPage(pageData))  
 includeSetupAndTeardownPages(pageData, isSuite);  
 return pageData.getHtml();  
 }

该版本的方法仍然定义在 HtmlUtil 工具类中。对比 Robert Martin 的两个重构版本，后一版本的主方法变得更加简单了，方法体只有短短的三行代码。虽然方法变得更简短，但提取出来的 includeSetupAndTeardownPages() 方法却增加了不必要的抽象层次。 **封装需要有度，引入太多的层次反而会干扰阅读。** 尤其是方法，Java 或大多数语言都不提供“方法嵌套方法”的层次结构（Scala 支持这一语法特性）。如果为一个方法的不同业务层次提取了太多方法，在逻辑上，它存在递进的嵌套关系，在物理上，却是一个扁平的结构。阅读这样的代码会造成不停的跳转，不够直接。正如 Grady Booch 所述：“整洁的代码从不隐藏设计者的意图，充满了 **干净利落** 的抽象和 **直截了当** 的控制语句。”干净利落，直截了当，可以破除对过度细粒度方法的迷信！与其封装一个用超长名称才能表达其意图的 includeSetupAndTeardownPages() 方法，不如直接“敞开”相同层次的代码细节，如：

includeSuiteSetupPage(testPage, newPageContent, includeSuiteSetup);  
includeSetupPage(testPage, newPageContent);  
includePageContent(testPage, newPageContent);  
includeTeardownPage(testPage, newPageContent);  
includeSuiteTeardownPage(testPage, newPageContent, includeSuiteSetup);

这五行代码不正是直截了当地表达了包含的页面结构吗？因此，我觉得 Robert Martin 提取出来的 includeSetupAndTeardownPages() 方法违背了简单设计的第四条原则，即增加了不必要的软件元素。事实上，如果一个方法的名称包含了 and，就说明该方法可能违背了“一个方法只做一件事情”的基本原则。

我并不反对定义细粒度方法，相反我很欣赏合理的细粒度方法，如前提取的 includePageContent() 方法。一个庞大的方法往往缺少内聚性，不利于重用，但什么才是方法的合适粒度呢？不同的公司有着不同的方法行限制，有的是 200 行，有的是 50 行，有的甚至约束到 5 行。最关键的不是限制代码行，而在于一个方法只能做一件事。

若发现一个主方法过长，可通过提取方法使它变短。当提取方法的逻辑层次嵌套太多，彼此的职责又高内聚时，就需要考虑将这个主方法和提取出来的方法一起委派到一个专门的类。显然，testableHtml() 方法的逻辑其实是一个相对独立的职责，根本就不应该将其实现逻辑放在 HtmlUtil 工具类，而应按照其意图独立为一个类 TestPageIncluder。提取为类还有一个好处就是可以减少方法之间传递的参数，因为这些方法参数可以作为单独类的字段。重构后的代码为：

public class TestPageIncluder {  
 private PageData pageData;  
 private WikiPage testPage;  
 private StringBuffer newPageContent;  
 private PageCrawler pageCrawler;  
  
 private TestPageIncluder(PageData pageData) {  
 this.pageData = pageData;  
 testPage = pageData.getWikiPage();  
 pageCrawler = testPage.getPageCrawler();  
 newPageContent = new StringBuffer();  
 }  
  
 public static String render(PageData pageData) throws Exception {  
 return render(pageData, false);  
 }  
  
 public static String render(PageData pageData, boolean isSuite) throws Exception {  
 return new TestPageIncluder(pageData).renderPage(isSuite);  
 }  
  
 private String renderPage(boolean isSuite) throws Exception {  
 if (isTestPage()) {  
 includeSetupPages(isSuite);  
 includePageContent();  
 includeTeardownPages(isSuite);  
 updatePageContent();   
 }  
 return pageData.getHtml();  
 }  
  
 private void includeSetupPages(boolean isSuite) throws Exception {  
 if (isSuite) {  
 includeSuitesSetupPage();  
 }  
 includeSetupPage();  
 }  
 private void includeSuitesSetupPage() throws Exception {  
 includePage(SuiteResponder.SUITE\_SETUP\_NAME, "-setup");  
 }  
 private void includeSetupPage() throws Exception {  
 includePage("SetUp", "-setup");  
 }  
 private void includeTeardownPages(boolean isSuite) throws Exception {  
 if (isSuite) {  
 includeSuitesTeardownPage();  
 }  
 includeTeardownPage();  
 }  
 private void includeSuitesTeardownPage() throws Exception {  
 includePage(SuiteResponder.SUITE\_TEARDOWN\_NAME, "-teardown");  
 }  
 private void includeTeardownPage() throws Exception {  
 includePage("TearDown", "-teardown");  
 }  
  
 private void updateContent() throws Exception {  
 pageData.setContent(newPageContent.toString());  
 }  
  
 private void includePage(String pageName, String sectionName) throws Exception {  
 WikiPage inheritedPage = PageCrawlerImpl.getInheritedPage(pageName, wikiPage);  
 if (inheritedPage != null) {  
 WikiPagePath pagePath = wikiPage.getPageCrawler().getFullPath(inheritedPage);  
 String pathName = PathParser.render(pagePath);  
 buildIncludeDirective(pathName, sectionName);  
 }  
 }  
 private void buildIncludeDirective(String pathName, String sectionName) {  
 buffer.append("\n!include ")  
 .append(sectionName)  
 .append(" .")  
 .append(pathName)  
 .append("\n");  
 }  
}

引入 TestPageIncluder 类后，职责的层次更加清晰了，分离出来的这个类承担了组装测试页面信息的职责，HtmlUtil 类只需要调用它的静态方法 render() 即可，避免了因承担太多职责而形成一个上帝类。通过提取出类和对应的方法，形成不同的抽象层次，让代码的阅读者有选择地阅读自己关心的部分，这就是清晰表达设计者意图的价值。

对比 Robert Martin 给出的重构第二个版本以及这个提取类的最终版本，我赞成将该主方法的逻辑提取给专门的类，但不赞成在主方法中定义过度抽象层次的 includeSetupAndTeardownPages() 方法。二者同样都增加了软件元素，我对此持有的观点却截然不同。我也曾就 Robert Martin 给出的两个版本做过调查，发现仍然有一部分人偏爱第二个更加简洁的版本。这一现象恰好说明简单设计的第三条原则属于主观判断，不如第二条原则那般具有客观的评判标准，恰如大家对美各有自己的欣赏。但我认为，一定不会有人觉得重构前的版本才是最好。即使不存在重复代码，单从可读性角度判断，也会觉得最初版本的代码不堪入目，恰如大家对美的评判标准，仍具有一定的普适性。

Robert Martin 在《代码整洁之道》中也给出了分离职责的类 SetupTeardownIncluder。两个类的实现相差不大，只是 TestPageIncluder 类要少一些方法。除了没有 includeSetupAndTeardownPages() 方法外，我也未曾定义 findInheritedPage() 和 getPathNameForPage() 之类的方法，也没有提取 isSuite 字段，因为我认为这些都是不必要的软件元素，它违背了简单设计的第四条原则，应当用奥卡姆的剃刀一割了之。

下一章

还没有评论



评论

## 更多资源下载交流请加微信：Morstrong,加入永久会员,网盘更新更快捷！

# 本资源由微信公众号：光明顶一号，提供支持