

TURING

图灵原创

大话 重构

范钢 著

明白真正的**专业级**软件开发是如何进行的
明白真正的**重构**具体是一步步怎么做的

高效可行的重构七步。面对实际重构，不会卡壳。
超越代码级重构，渗透系统与设计的各个层面。

第一次真正理解那些最熟悉的陌生技术，
惊讶于它们各就各位榫卯成强韧的整体。



人民邮电出版社
POSTS & TELECOM PRESS

原创经典

大 话 重 构

你听过或没听过的那些术语和概念，多少明白或完全不明白的技术和方法，知道却没用过或完全不知道的工具和软件，这些之前各玩各的的独立散碎，在这本书中被榫卯成一个强韧的整体。你会明了它们中每一个的作用，应被安插到的位置，并见识它们各就各位时所发挥出的能量。头脑从未有过的清醒，你理解了以前所不理解的。

你看到这里有好些不乖的代码，那里也有数百头粗野的程序。所有这些显然都需要重构，但应该先改哪处呢，要从哪儿开始呢？在看了那么多重构书后，你发现你竟然在重构的第一步就卡壳了。因为没有书告诉你第一步该做什么。

图灵社区：iTuring.cn
热线：(010) 51095186 转 600

分类建议 计算机 / 程序设计

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-34885-2

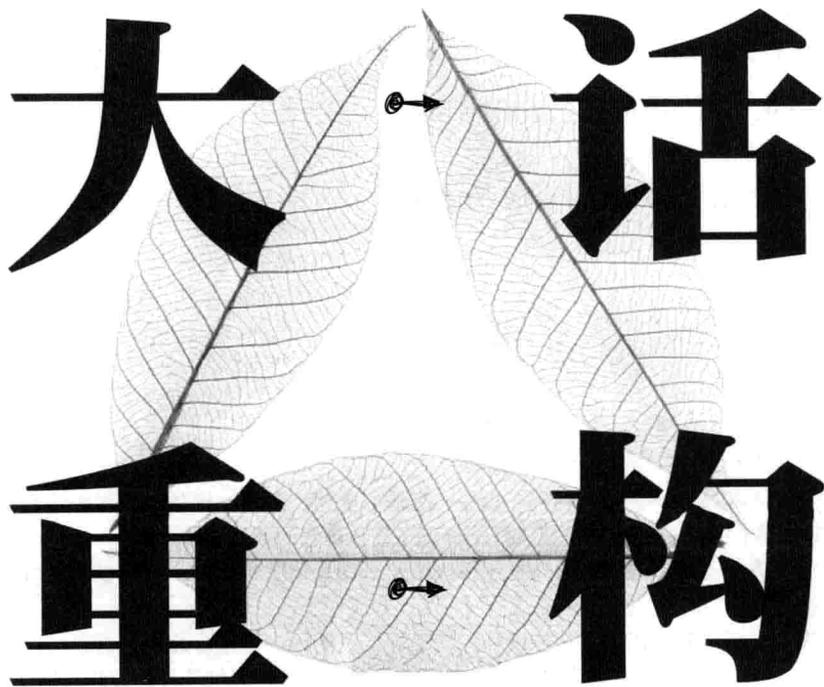


9 787115 348852 >

ISBN 978-7-115-34885-2

定价：45.00元

TURING
图灵原创



大 话
重 构

The title '大话重' is rendered in large, bold, black characters. The characters are partially overlaid by three detailed, light-colored leaf illustrations. Two small black arrows point from the stems of the leaves towards the right, one pointing to the top-right character '话' and the other to the bottom-right character '构'.

范钢 ©著

人民邮电出版社
北京

图书在版编目(CIP)数据

大话重构 / 范钢著. — 北京: 人民邮电出版社,
2014.5

(图灵原创)

ISBN 978-7-115-34885-2

I. ①大… II. ①范… III. ①软件设计 IV.
①TP311.5

中国版本图书馆CIP数据核字(2014)第041929号

内 容 提 要

本书的价值在于两点:

- 一、让你明白真正的专业级软件开发是如何进行的;
- 二、让你明白真正的重构具体是一步步怎么做的。

本书运用大量源于实践的示例,从编码、设计、组织、架构、测试、评估、应对需求变更等方面,深入而多角度地讲述了我们应该如何重构,建设性地提出了高效可行的重构七步。

读完本书,实践重构不再卡壳,需求变更不再纠结。全面领悟重构之美,遗留系统不再是梦魇,自动化测试原来可以这样做。

本书帮助程序员告别劣质代码步入精妙设计,让遗留系统的维护者逐步改善原有设计,指导重构实践者走出困惑步步坚定。同时,也为管理者加强软件质量的管理与监督,提供了好的方法与思路。

◆ 著 范 钢

策划编辑 陈 冰

责任编辑 朱 巍

责任印制 焦志炜

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号

邮编 100164 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

三河市海波印务有限公司印刷

◆ 开本: 800×1000 1/16

印张: 16.75

字数: 314千字

2014年5月第1版

印数: 1-5 000册

2014年5月河北第1次印刷

定价: 45.00元

读者服务热线: (010)51095186转600 印装质量热线: (010)81055316

反盗版热线: (010)81055315

广告经营许可证: 京崇工商广字第 0021 号

编辑的话

当你面对一本书，你最想知道的应该是这本书究竟可以给你带来什么。

对于这本书，它最大的价值在于两点：

一、让你明白真正的专业级软件开发是如何进行的。

打个比方，你是摄影新手，拽得很，但无论你怎么摆弄你的数码单反，就是照不出专业摄影师似乎轻轻松松就搞出来的东西。他们使用的东西，你没见过，瞧不出是干嘛用的，自然也不知道为什么要用它。他们所做的至关重要的措施，你也不知道有什么意义。所有这些你看不明白的过程，造就了最终一目了然的差距。

你在软件开发上投入了不少时间，攻克过不少难关，解决了一堆问题，但做出来的东西、写出来的代码、设计出的架构还是透着或淡或浓的业余味。原因在于有些重要和基本的东西你还不知道。在得到专业之前，你必须见多识广。

本书带你领教立竿见影的专业级软件开发的过程。你听过或没听过的那些术语和概念，多少明白或完全不明白的技术和方法，知道却没用过或完全不知道的工具和软件，这些之前各玩各的的独立散碎，在这本书中被榫卯成一个强韧的整体。你会明了它们中每一个的作用，应被安插到的位置，并见识它们各就各位时所发挥出的能量。头脑从未有过的清醒，你理解了以前所不理解的。

二、让你明白真正的重构具体是一步步怎么做的。

重构的书，市面上有一些了。但这些书瞄准的均是代码级的重构。把这行代码改成这样会好些，把这段程序改成这样会更合适些。遇到这样的代码，你要这么改；面对那样的情况，你得那样处理。一切都没有问题，直到你面对实际中的重构。

当你在实际中真正要去做一次重构时，你发现你面对的不是一行或一段代码，而是整个软件。你看到这里有好些不乖的代码，那里也有数百头粗野的程序。所有这些显然都需要重构，但应该先改哪处呢，要从哪儿开始呢？在看了那么多重构书后，你发现你竟然在重构的第一步就卡壳了。因为没有书告诉你第一步该做什么。

本书明确地告诉你第一步要做什么，那就是分解大函数，这是软件退化的重灾区，也是重构过程的不二起点。本书不仅告诉你第一步该做什么，更将看似漫无头绪繁复冗长的软件重构清楚明白地划分成七个步骤，告诉你整个重构七步中的每一步都该做些什么，并详细且

通俗易懂地讲解了具体你该如何去做。

本书所讲解的重构远远超越了代码级，充分渗透到软件系统与设计的各个层面，涵盖从代码、函数、类与对象，直至设计模式、分层架构、领域模型、软件测试的整个过程。

在阅读本书的过程中，你经常会为读到精彩之处而喜悦（头脑豁然开朗），而读完本书后，你会成竹在胸。以往在精神和肉体上折磨你很久的客户需求变更，或是因前期考虑不周而引起的设计变化，都不会再让你感到纠结，因为你可以通过重构润物细无声地容纳这些变化，而且你清楚如何去做。

再来说说这本书的一些趣事。本书的作者范钢对软件开发有着无比的热忱，做事极为认真，这份书稿是我收到过的交稿中错别字最少的一部，在进行文稿加工时，常常连续多页都没有一处加工痕迹。本书语言通俗易懂，逻辑清晰，读起来很轻松。唯一的缺点是个别之处有些啰嗦，这些啰嗦之处已被我剔除，以获得更好的阅读体验。

做为本书的策划编辑，我是带着喜悦的心情读完本书的，相信你也会的。

陈冰

2014年2月6日

前 言

我常常感到幸运，我们现在所处的是一个令人振奋的时代，我们进入了软件工业时代。在这个时代里，我们进行软件开发已经不再是一个一个小作坊，我们在进行着集团化的大规模开发。我们开发的软件不再是为某个车间、某个工序设计的辅助工具，它从某个单位走向整个集团，走向整个行业，甚至整个社会，发挥着越来越重要的作用。一套软件所起到的作用与影响有多大，已经远远超越了所有人的想象，成为一个地区、一个社会，乃至整个国家不可或缺的组成部分。慢慢地，人们已经难以想象没有某某软件或系统的生活和工作会是怎样的。这就是软件工业时代的重要特征。

然而，在这个令人振奋的软件工业时代，处于时代中心的各大软件企业却令人沮丧。在软件规模越来越庞大，软件结构越来越复杂的同时，却是软件质量越来越低下，软件维护变得越来越困难，以至于每个小小的变更都变得需要伤筋动骨。研发人员为此手足无措，测试人员成为唯一的救星，每个小小的变更都需要付出巨大代价进行测试。软件企业在这样一种恶性循环中苦苦支撑。毫无疑问，这也成为这个令人振奋的时代的另一个特征。

是的，面对软件工业时代我们并没有做好准备。过去，一套软件的生命周期不过 2~3 年时间，随着软件需求的变化，我们总是选择将软件推倒了重新开发，但是现在这样的情况在发生着改变。随着软件规模的扩大，软件数据的积累，软件影响力的提升，我们，以及我们的客户，都真切地感受到，要推倒一套软件重新开发，将变得越来越困难且不切实际。这样的结果就是，我们的软件将不停地修改、维护、再修改、再维护……直到永远。这是一件多么痛苦的事情！

一套软件，当它第一次被开发出来的时候，一切都十分清晰：清晰的业务需求、清晰的设计思路、清晰的程序代码。但经历了几次需求变更与维护以后，一切就变得不那么清晰了。业务需求文档变得模糊不清，设计思路已经跟不上变更的脚步，程序代码则随着业务逻辑的复杂而臃肿不堪。程序员开始读不懂代码，软件开发工作变得不再是一种乐趣。

随着时间的推移，软件经过数年、数十次的变更与维护，情况变得越来越糟。最初的程序员已经不愿再看到自己的代码而选择离去。他的继任者们变得更加无所适从，由于看不懂程序，代码的每一次修改如同在走钢丝。测试人员变成了唯一的希望，开发人员的每一次修改都意味着测试人员需要把所有程序测试一遍。继任者们开始质问最初的设计者们，程序是

怎么设计的。如果此时恰巧又有什么新技术出现，就会更显得原有系统的破旧与不堪。

相信这就是软件工业时代的所有企业都不得不面对的尴尬境地。难道真的是我们最初的设计错了吗？是的，我们都这样质问过我们自己，因此我们开始尝试在软件设计之初投入更多的精力。我们开始投入更多的时间作需求调研，考虑更多可能的需求变化，做更多的接口，实现更加灵活但复杂的设计。然后呢，解决了我们的问题了吗？显然是没有。需求并没有像我们想象的那样发生变更：我们之前认为可能发生的变更并没有发生，使我们为之做出的设计变成了摆设；我们之前没有考虑到的变更却发生了，让我们猝不及防，软件质量开始下降，我们被打回了原形。难道真的是无药可解了吗？在我看来，如果我们没有看明白软件开发的规律与特点，那么我们永远找不到那味向往已久的解药。现在，让我们真正静下心来分析分析软件开发的规律与特点。

软件，特别是管理软件，其实质是对真实世界的模拟。我们通过对真实世界的模拟，实现计算机的信息化管理，来提高我们的生产效率。然而，真实的世界复杂而多变的，我们认识世界却是一个由简单到复杂循序渐进的过程，这是一个我们无法改变的客观规律。因此，毫无疑问，遵循着这样一个客观规律，我们的软件开发过程必然也是一个由简单到复杂循序渐进的过程。

最初，我们开发的是一个对真实世界最简单、最主要、最核心部分的模拟。因为简单，我们的思路变得清晰而明了。但是，我们的软件不能永远只是模拟那些最简单、最主要、最核心的部分。我们的客户在使用软件的过程中，如果遇到那些不那么简单、不那么主要、不那么核心的情况时，我们的软件就无法处理了，这是客户所不能接受的。因此，当软件的第一个版本交付客户以后，客户的需求就开始变更。

客户的需求永远不会脱离真实世界，也就是说，真实世界不存在的事物、现象、关系永远都不可能出现在软件需求中。但是，真实世界的事物、规则与联系并不是那么地简单与清晰的。随着我们的软件对它模拟得越来越细致，程序的业务逻辑开始变得不再清晰而易于理解，这就是软件质量下降最关键的内因。

任何一个软件的设计，总是与软件的复杂度有密切的关系。举例来说吧，客户资料是许多系统都必须记录的重要信息。起初，我们程序简单，客户资料只记录了一些简单的信息，如客户名称、地址、电话等等，但随着程序复杂度的增加，客户资料开始变得复杂。比如，起初“地址”字段就仅仅需要一个字符串就可以了，但随着需求的变更，它开始有了省份、城市、地区、街道等信息。随后还会有邮政编码、所属社区、派出所等信息。起初增加两个字段时我们还可以在“客户信息表”里凑合一下，但后来我们必须要及时调整我们的设

计，将地址提取出来单独形成一个“地址信息表”。如果不及时予以调整，“客户信息表”将越来越臃肿，由 10 来个字段，变成 50 个、80 个、上 100 个……

信息表尚且如此，业务操作更是如此。起初的业务操作是如此地简单而明了，以至于我们不需要花费太多的类就可以将它们描述清楚。比如开票操作，最初的需求就是将已开具的票据信息读取出来，保存，并统计出本月开票量及金额。这样一个简单操作，设计成一个简单的“开票业务类”合情合理。但随后的业务逻辑变得越来越复杂，我们要检查客户是否存在、开票人是否有权限、票据是否还有库存，等等。起初的开票方式只有一种，但随着非正常开票的加入，开票方式不再单一，而统计方式也随之变化……随着业务的不断增加，软件代码的规模也在发生着质的变化。如果这时我们不及时调整我们的设计，而是将所有的程序都硬塞进“开票业务类”，那么程序质量必然会退化。“开票业务类”由原有的数十行，激增到数百行，甚至上千行。这时的代码将难于阅读，维护它将变成一种痛苦，毫无乐趣可言。

面对这样的状况，我们应当怎样走出困境呢？毫无疑问，就是重构。开票前的校验真的属于“开票业务类”吗？它们是否应当被提取出来，解耦成一个一个的校验类。正常开票与非正常开票真的应该写在一起吗？是否我们应当把“开票业务类”抽象成接口，以及正常开票与非正常开票的实现类。这就是我给大家的良方：当软件因为需求变更而开始渐渐退化时，运用软件重构改善我们的结构，使之重新适应软件需求的变化。

范钢

2014 年元旦

目 录

第一部分 基础篇

第 1 章 重构：改变既有代码的一剂良药	2
1.1 什么是系统重构	2
1.2 在保险索上走钢丝	3
1.3 大布局与小步快跑	5
1.4 软件修改的四种动机	6
1.5 一个真实的谎言	9
第 2 章 重构方法工具箱	10
2.1 重构是一系列的等量变换——第一次 HelloWorld 重构	10
2.2 盘点我们的重构工具箱——对 HelloWorld 抽取类和接口	13
第 3 章 小步快跑的开发模式	19
3.1 大布局你伤不起	19
3.2 小设计而不是大布局	20
3.3 小步快跑是这样玩的——HelloWorld 重构完成	22
第 4 章 保险索下的系统重构	30
4.1 你不能没有保险索	30
4.2 自动化测试——想说爱你不容易	31
4.3 我们是这样自动化测试的——JUnit 下的 HelloWorldTest	33
4.4 采用 Mock 技术完成测试	37

第二部分 实践篇

第 5 章 第一步：从分解大函数开始	44
5.1 超级大函数——软件退化的重灾区	44

5.2	抽取方法的实践	51
5.3	最常见的问题	54
第 6 章	第二步：拆分大对象	57
6.1	大对象的演化过程	57
6.2	大对象的拆分过程——抽取类与职责驱动设计	60
6.3	单一职责原则（SRP）与对象拆分	61
6.4	合久必分，分久必合——类的归并	63
第 7 章	第三步：提高代码复用率	66
7.1	顺序编程的烦恼	66
7.2	代码重复与 DRY 原则	67
7.3	提高代码复用的方法	69
7.3.1	当重复代码存在于同一对象中时——抽取方法	69
7.3.2	当重复代码存在于不同对象中时——抽取类	71
7.3.3	不同对象中复用代码的另一种方法——封装成实体类	72
7.3.4	当代码所在类具有某种并列关系时——抽取父类	75
7.3.5	当出现继承泛滥时——将继承转换为组合	76
7.3.6	当重复代码被割裂成碎片时——继承结合模板模式	78
7.4	代码重复的检查工具	79
第 8 章	第四步：发现程序可扩展点	80
8.1	开放-封闭原则与可扩展点设计	81
8.2	过程的扩展与放置钩子——运用模板模式增加可扩展点	85
8.3	面向切面的可扩展设计	89
8.4	其他可扩展设计	93
第 9 章	第五步：降低程序依赖度	98
9.1	接口、实现与工厂模式	98
9.1.1	彻底理解工厂模式和依赖反转原则	98
9.1.2	工厂模式在重构中的实际运用	102
9.2	外部接口与适配器模式——与外部系统解耦	106
9.3	继承的泛滥与桥接模式	109
9.4	方法的解耦与策略模式	112

9.5	过程的解耦与命令模式	116
9.6	透明的功能扩展与设计——组合模式与装饰者模式	119
第 10 章	第六步：我们开始分层了	128
10.1	什么才是我们需要的分层	128
10.2	怎样才能拥抱需求的变化	131
10.3	贫血模型与充血模型	136
10.4	我们怎样面对技术的变革	139
第 11 章	一次完整的重构过程	143
11.1	第一步：分解大函数	143
11.2	第二步：拆分大对象	145
11.3	第三步：提高复用率	147
11.4	第四步：发现扩展点	148
11.5	第五步：降低依赖度	151
11.6	第六步：分层	151
11.7	第七步：领域驱动设计	153

第三部分 进阶篇

第 12 章	什么时候重构	156
12.1	重构是一种习惯	156
12.2	重构让程序可读	158
12.3	重构，才好复用	159
12.4	先重构，再扩展	161
12.5	变更任务紧急时，又该如何重构	163
第 13 章	测试驱动开发	166
13.1	测试驱动开发（TDD）vs.后测试开发（TAD）	167
13.2	测试驱动开发与重构	170
13.3	遗留系统怎样开展 TDD	178
第 14 章	全面的升级任务	182
14.1	计划式设计 vs.演进式设计	182
14.2	风险驱动设计	184

14.3	制定系统重构计划	188
第 15 章	我们怎样拥抱变化	190
15.1	领域才是软件系统的“心”——工资软件的三次设计演变	190
15.2	领域模型分析方法	197
15.3	原文分析法	199
15.4	领域驱动设计——使用领域模型与客户一起设计	203
15.5	在遗留系统中的应用	209
第 16 章	测试的困境	213
16.1	重构初期的困局	213
16.2	解耦与自动化测试	215
16.3	开发人员，还是测试人员	219
16.4	建立自动化测试体系	223
第 17 章	系统重构的评价	225
17.1	评价软件质量的指标	225
17.2	怎样评价软件质量呢	228
结束语	重构改变了世界	233
附录		235



第一部分 基础篇

第1章 重构：改变既有代码的一剂良药

前面我们提到了，面对软件工业时代的到来，我们的软件企业陷入了一种更深的迷茫之中，一种“后有追兵，前有悬崖，进退两难”的境地。

后有追兵：面对维护了数十年之久的大型遗留系统，我们到底改还是不改？不改，面对越来越多的需求变更，我们维护的成本越来越高，变更变得越来越困难；面对不断涌现的新技术，我们的系统显得越来越丑陋与落后；面对越来越多的竞争者，我们面临着被市场淘汰的风险。

前有悬崖：原本运行得好好的软件系统，凑合一下还可以运行几年。一不小心改出问题了，企业立马就歇菜儿了，面对大量的用户投诉，企业四处救火，竞争对手趁火打劫，这是任何软件企业都不能承受的巨大风险。

难道真的“鱼与熊掌不能兼得”吗？真的没有一种方法，能够既保证我们的系统可以技术改造，又能有效地避免改造过程的风险吗？有，那就是系统重构。

1.1 什么是系统重构

提到重构，许多人都讳莫如深、敬而远之。那么什么是系统重构呢？大家可能有很多不同的看法：

1. 系统重构是那些系统架构师、技术大牛玩的高端玩意儿，咱普通屌丝不懂，跟咱没啥关系。
2. 系统重构就是改代码，大改特改那种，整个重来一遍那种，这个比较邪恶，比较容易改出事儿，还是不要轻易尝试为妙。
3. 我知道系统重构，也知道它能改善遗留系统，但我还是不敢轻易尝试，因为改出问题来怎么办，还是算了吧。

然而我认为，现在我们对系统重构有太多的误解，以至于我们还不怎么了解它，就已经将它拒之门外。什么是系统重构？它是一套严谨而安全的过程方法，它通过一系列行之有效

的方法与措施，保证软件在优化的同时，不会引入新的 BUG，保证软件改造的质量。这一点在我后面一步一步的拆解中，你可以慢慢体会到。

我们先看看系统重构的概念。系统重构，就是在不改变软件的外部行为的基础上，改变软件内部的结构，使其更加易于阅读、易于维护和易于变更^①。

系统重构中一个非常关键的前提就是“不改变软件的外部行为”，这个前提非常重要，它保证了我们在改造原有系统的同时，不会为原系统带来新的 BUG，以确保改造的安全。这里，什么是“为原系统带来新的 BUG”？我们必须为其做出一个严格的定义，那就是“改变了软件原有的外部行为”。也许你对此有些不太赞同，改变了软件原有的外部行为，怎么就能武断地认为，是为原系统带来了新 BUG 呢？为此我们来举个例吧。

假如一个系统的报表查询功能，原来在表格里的返回结果中，日期是这样表示的：“2013-2-18”。经过系统改造以后变成这样了：“2013-2-18 00:00:00”。这是 BUG 吗？作为开发人员你可能认为这算什么 BUG，但作为客户那就是 BUG，因为它让表格变得难看，使用不再方便了。系统重构，对于客户来说应当是完全透明的。我们为之做了很多工作，而他们应当完全感觉不到它的存在。如果我们的重构做到了这一点，那么我们的重构就必然是安全的、可靠的、没有风险的。

更广泛一些来说，如果我们打开软件内部，保证系统中的每个接口与改造前是等价的，也就是说，其输入输出在改造前后都是一致的。当我们的每个改造都是这样进行的，则必然不会为系统带来新的 BUG。这就是我们进行改造的保险索，它也是我现在所说的重构与以往那种拿着代码一阵瞎改的根本区别。

总而言之，系统重构不是那种冒着极大风险进行的代码修改，而是必须保证修改前后输入输出的一致，这就是我们说的“不改变外部行为”。为此，贯穿整个重构过程的是不断地测试。起初这种测试是手工测试，随后逐渐转变为自动化测试。每修改一点点就进行一个测试，再修改一点点。测试，就是系统重构的保险索。

1.2 在保险索上走钢丝

当我们开始系统重构的时候，不是着手去修改代码，而是首先建立测试机制。不论什么程序，只要是被动我们修改了，理论上就可能引入 BUG，因此我们就必须要进行测试。既然

^① Refactoring: a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. ——引自 *Refactoring: Improving the Design of Existing Code* 一书

是测试就必须要有个正确与否的评判标准。以往的测试，其评判的标准就是是否满足业务需求。因此，测试人员往往总是拿着需求文档测试系统。

与以往的代码修改不同，重构没有引入任何新的需求，系统原来什么功能，重构以后还是这些功能。因此，重构的测试标准就只有一个，就是与之前的功能完全保持一致，仅此而已。

然而在许多经典的重构书籍中，大师们总是建议我们首先建立自动化测试机制，这已经被我在无数次实践中证明是不现实的。要知道我们现在重构的是一个遗留系统，它往往设计混乱，接口不清晰，程序相互依赖。所有这些都使得最初的自动化测试变得非常困难而不切实际。

因此，从一开始就实现自动化测试是不切实际的，我们所能采用的还是手工测试。在重构之前首先将系统启动起来，执行相应的功能，得到各自相应的输出。然后开始重构，每次重构对代码的修改量不要太大，花费的时间不要过长。因为，修改量太大，花费时间太长，一旦测试不通过，很难定位错误的原因。在这种情况下，我们只能还原代码，将此次修改的工作完全作废。如果此次修改已经花费了你数天甚至数月的时间，这样的损失就实在太大了。

每做一次重构，修改一点儿代码，然后提交，对系统进行一次测试。如果系统依然保持与以往一样的功能，重构成功，继续进行下一次重构。如果不是，你不得不还原回来重新重构。频繁地测试着实让你挺烦，但却有效减少了重构失败带给你的损失。一个折中的办法就是，平时频繁测试的时候，测试项目少一些，只测试主要项目，定期再进行全面的测试。录制 QTP^①脚本也是一个不错的方式，它虽然有诸多问题，但却可以在系统重构初期有效地建立自动化测试，系统级别的自动化测试。随着系统重构的不断深入，我们的程序开始在改善，耦合变得越来越松散，程序变得越来越内聚，接口变得越来越清晰。这时候，当自动化测试条件成熟时，我们才可以逐渐开始自动化测试，而这种自动化测试才是建立在代码级别的真正的自动化测试。

一旦某个修改测试不通过，则还原回来。这种一次一小步的修改模式，我们形象地称之为“小步快跑”。在测试集成工具的不断监督下一点一点地修改程序，是系统重构异于以往的另外一个特点。通过小步快跑可以使我们在重构的过程中，以最快的速度发现修改的问题，将因修改错误带来的损失减到最小，毕竟是人都不可能避免犯错。

^① QTP, Quicktest Professional的简称，是一种自动测试工具。使用QTP的目的是想用它来执行重复的手动测试，主要是用于回归测试和测试同一软件的新版本。

1.3 大布局与小步快跑

以往我们在重新设计一个系统时，总是喜欢大布局。全面地整理系统需求，全面地分析系统功能，再全面地设计系统、开发、测试。这样一个过程往往会持续数月，花费大量的工作量。但是，不到最后设计出来，谁都不知道会不会存在问题。这就是“大布局”的弊端。

正因为如此，软件大师在讲述系统重构时总是强调，系统重构应当避免大设计，而尽量采用一个一个连续不断的小设计。这就是我们所说的“小步快跑”的设计模式。

小步快跑体现出了敏捷软件开发的特点——简单与快速反馈。不要想得太多了，活在今天的格子里，你永远不可能预知今后会发生什么。所以，做今天的设计，解决今天的问题，完成今天的重构，让明天见鬼去吧。要知道，简单对于我们是多么地重要。当我们的头脑开始思考各种复杂的问题时，就开始充血，然后就是梦游，最后的结果就是顾此失彼。既然如此，我们为何不选择一种更加简单的生活方式呢？

非常遗憾的是，很多时候我们不能选择这种简单的生活方式，因为我们害怕明天，害怕明天会出现那些我们处理不了的业务场景，因此我们开始过度设计。我不是批判我们不应当有预见，预见性设计与过度设计往往就是一线之隔。有限的预见是可以的，但不要想得过于遥远。当明天真的需求变更了，我们当前的设计不能满足了，怎么办呢？没什么大不了的，因为我们有重构。通过安全而平稳的重构方法先重构我们的系统，使之可以应付那个需求，然后再添加代码，实现新需求。这个过程被称为“两项帽子”：一项是只重构而不新增功能，一项是增加新的功能实现新需求。正因为如此，我们在设计时思考当下就可以了。

另外一个问题就是及时反馈，落实到此处就是及时测试。只有测试通过了，此次重构才算成功，我们才能继续往下重构，否则我们必须还原。从这里我们不难看出，重构的周期是多么地重要。在我以往的重构工作中，一次重构的周期也就在10分钟到1小时。重构的周期越长，说明你考虑的问题越复杂，最终出错的概率也就越大。所以，我们一定要习惯“小步快跑”的工作方式，让自己只活在当下。

说了那么多重构，相信一定有一个疑问开始在你脑中萦绕。既然系统重构对于客户来说是透明的，客户完全感觉不到它的存在，毫无疑问，它对于客户来说就是毫无价值的。这下疑问就来了：既然重构对于客户来说毫无价值，我们做它还有什么意义呢？要说明白这个问题，我们需要首先谈一谈软件修改的四种动机。

1.4 软件修改的四种动机

软件，自从被我们开发出来并交付使用以后，如果它运行得好好的，我们是不会去修改它的。我们要修改软件，万变不离其宗，无非就是四种动机：

1. 增加新功能；
2. 原有功能有 BUG；
3. 改善原有程序的结构；
4. 优化原有系统的性能^①。

第一种和第二种动机，都是源于客户的功能需求，而第四种是源于客户的非功能需求。

软件的外部质量，其衡量的标准就是客户对软件功能需求与非功能需求的满意度。它涉及一个企业、一个软件的信誉度与生命力，因此为所有软件企业所高度重视。但是，就在所有企业高管把软件外部质量放在高于一切的高度的同时，软件内部质量却长期为人所漠视。企业没有保证软件内部质量的措施，甚至因为需要赶工而肆意地降低内部质量的标准。这样带来的长期恶果就是，程序编写越来越烂，运行效率越来越低，程序结构越来越让人看不懂。当一群群刚刚毕业的大学生游走在这堆写得很烂的代码中时，他们开始沉沦，开始天真地以为代码就是这样写的（真是毁人不倦呀）。当我们的软件企业培养出一批批质量不高的开发人员，开发出一个个质量低下的软件系统时，它们开始发现软件维护成本越来越高，最终不得不收获自己种下的恶果。

要提高软件内部质量，毫无疑问就是软件修改的第三个动机：改善原有程序的结构。它的价值是隐性的，并不体现在某一次或两次开发中，而是逐渐体现在日后长期维护的软件过程中。高质量的软件，可以保证开发人员（即使是新手）能够轻易看懂软件代码，能够保证日后的每一次软件维护都可以轻易地完成（不论软件经历了多少次变更，维护了多少年），能够保证日后的每一次需求变更都能够轻易地进行（而不是伤筋动骨地大动）。要做到这几点并不容易，它需要我们持续不断地对系统内部质量进行优化与改进。这，就是系统重构的价值。

为了有效提高软件的内部质量，我们在系统重构中应当做哪些事情呢？首先是提高软件的可读性，让它易于阅读。软件要可读，并不是添加几行注释那么简单的事儿，首先是软件的业务逻辑要清晰。一个业务流程可能其处理过程非常复杂，如果在一个函数中顺序地一行一行写下来，可能需要写数百行，甚至上千行。比如，我们要实现这样一个需求：像 Spring

^① 引自 *Working Effectively with Legacy Code*，中文版译为《修改代码的艺术》。

那样从一个或者数个配置文件中读取 XML，然后根据 XML 依次去创建每一个 bean，将每一个 bean 放到 beanFactory 中。如果没有经过精心地设计，而是随性地一行一行写，要实现这个功能没个几百上千行代码，想想都难。但如果我们换个思路，在入口函数里仅仅调用几个顶级方法，比如 findXmlFile()、readXmlStream()、buildFactory()，然后依次去实现这几个顶级方法，程序结构就会变得清晰而易读。请看这段示例代码：

```
public abstract class XmlBuildFactoryTemplate {
    /**
     * 初始化工厂。根据路径读取XML文件，将XML文件中的数据装载到工厂中
     * @param path XML文件的路径
     */
    public void initFactory(String path){
        //寻找XML文件，读取数据流
        InputStream inputStream = findXmlFile(path);
        //解析XML文件，返回根
        Element root = readXmlStream(inputStream);
        //根据XML文件创建类，放入工厂中
        buildFactory(root);
    }
    /**
     * 读取一个XML的文件，输出其数据流
     * @param path XML文件的路径
     * @return InputStream文件输入流
     */
    protected InputStream findXmlFile(String path) {
        ...
    }
    /**
     * 读取并解析一个XML的文件输入流，以Element的形式获取XML的根，返回之
     * @param inputStream 文件输入流
     * @return ElementXML的根
     */
    protected Element readXmlStream (InputStream inputStream) {
        ...
    }
    /**
     * 用从一个XML的文件中读取的数据构建工厂
     * @param root 从一个XML的文件中读取的数据的根
     */
    protected abstract void buildFactory(Element root);
}
```

在实现这几个顶级函数的时候，我们还会将一些比较独立的功能分解出去，形成类与接口，比如这里的 `findXmlFile()`，它可能会以各种方式去寻找 XML 文件，这时把这些功能提取出来，形成 `Resource` 接口和它的多个实现（如图 1.1 所示），为 `findXmlFile()` 所调用。如果我们将这个复杂的功能设计成这样，则毫无疑问，系统可读性将大大提高。

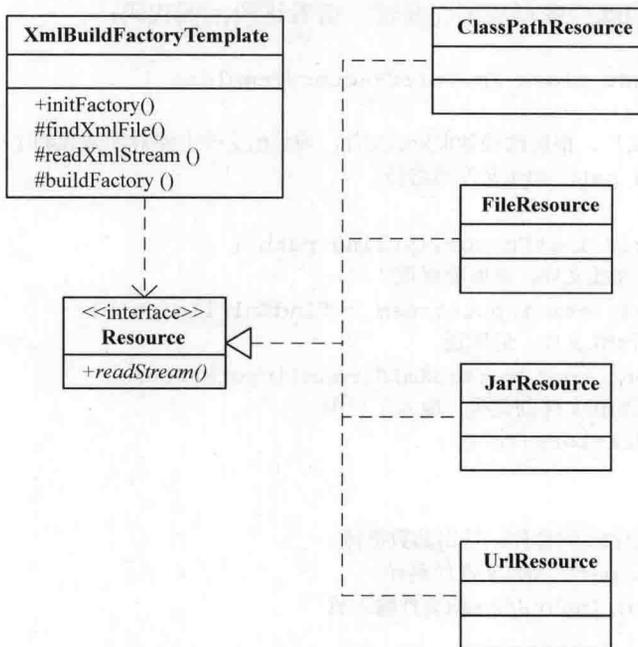


图1.1 工厂类模板的设计图

此外，在面向对象的世界里，我们设计的类、方法、关联，应当与现实世界中的事物、行为，及其相互的关系对应起来。现实世界有什么事物，这些事物都应当有什么行为，相互之间是什么关系，则我们在软件世界里就应当设计什么类、什么方法和它们之间的关联关系。只有这样的设计才是最易于为人所理解的设计，这就是“领域驱动设计”的思想^①。在系统重构中，我们将使用“抽取方法”来分解难于阅读与维护的大函数，用“抽取对象”来分解无所不能的大对象。

系统重构要干的另一件事情就是使软件易于维护、易于变更。软件需求总是变得越来越复杂，这是无法改变的客观事实。在添加新功能的同时，我们既要保证原有代码的有效性，

^① 领域驱动设计（DDD: Domain-Driven Design），源自软件理论大师Eric Evans在其2010年发表的同名著作中提出来的设计思想，本书将大量提及。

又必须有效地复用原有的代码。这是多么矛盾的一件事儿啊！怎么办呢？看我在后面一步一步为你分解……

1.5 一个真实的谎言

经过前面的一番讲解，相信你已经对系统重构有了一些初步的认识了。一切的一切仿佛在告诉我们，系统重构总是与需求变更无关。但此时，我不得不告诉你这是真实的谎言。

我们的软件系统总是处于一种变化之中，并且往往是一种由浅入深、由易到难的过程。但是，当系统复杂程度发生变化时，我们应当及时调整我们的设计，来适应新的变化。然而我们没有做到这一点，所以我们的系统维护变得越来越困难。要解决我们的问题，必须通过系统重构去优化我们的程序，使之重新适应业务需求。毫无疑问，需求变更才是我们去重构的主要动因。

然而，系统重构要求我们不能改变软件的外部行为，这意味着在重构的过程中不能为软件添加任何新功能，重构仿佛与变更无关。这就有些让人搞不懂了，我们因为变更才尝试重构，但重构却不让我们去变更！破解这个真实的谎言的关键就是，系统重构并不是禁止我们变更，而是应当以“两项帽子”的方式进行变更。就是当我们需要变更系统时，应该将变更的过程分为两个步骤：首先是不添加任何功能先重构我们的系统，使之适应新的需求；然后开始变更，实现新的需求。

让我们先剖析一番以往我们是怎么添加新功能的。当用户来了一个新需求时，我们修改代码的首要原则，是对新需求的修改不能影响以往的功能。我们过去如何做到这一点的呢？其实没有什么好的办法，一个直观的想法就是，原有代码改得越少，修改出错的风险就会越小。正因为如此，即使原程序已经不适应新的需求，程序员也不愿去修改程序的结构，而是就着现有代码不断地往里添加程序。随着时间的流逝，添加的程序越来越多、越来越乱，因而系统维护成本越来越高。

采用系统重构完全就不是这样一个概念了。这里当原程序不适应新的需求时，我们采用的是一种“糟糕设计零容忍度”的策略，即先重构系统使之首先适应新的需求，再顺理成章去实现这些需求。由于“不改变软件外部行为”，我们可以很容易地建立测试机制，在测试机制的不断监督下，有质量地保证重构的成功。然后我们再实现新功能时，就可以保证易于实现，并且使得可读性、可维护性和易变更性得以保障。这样的过程避免了那些因变更而带来的糟糕设计，从而避免了软件质量的下滑。

第2章 重构方法工具箱

毫无疑问，系统重构是一件如履薄冰、你必须时时小心应对的工作。你就像走在钢丝上的人，每一步都必须要保证正确，一个不经意的失误就可能造成巨大麻烦。尽管如此，只要你掌握了正确的方法，即使站在钢丝上也能如履平地，而这个正确的方法，就是那些被证明是正确的重构方法。说了那么多，你一定开始好奇，系统重构到底都是一些什么方法呢？行了，我也就不卖关子了，我们来看看重构方法工具箱里都有些什么东东。

2.1 重构是一系列的等量变换——第一次HelloWorld重构

系统重构要求我们对代码的每一步修改，都不能改变软件的外部行为，因此在系统重构中的所有方法，都是一种代码的等量变换。重构的过程，就好像在做数学题，一步一步地进行算式的等量变换。经过一系列等量变换，最终的结果虽然在形式上与原式不一样，但通过计算可以得到与原式完全相同的结果。

然而，等量变换不等于原地踏步。正如矩阵通过等量变换可以得到方程组的解，微积分可以通过等量变换计算最终的结果，重构通过等量变换，在保证代码正确的同时，可以使程序结构得到优化。为了说明系统重构中的这种等量变换，我们来看一个简单的例子，原始程序是这样的：

```
public class HelloWorld {
    public String sayHello(Date now, String user){
        Calendar c;
        int h;
        String s = null;
        c = Calendar.getInstance();
        c.setTime(now);
        h = c.get(Calendar.HOUR_OF_DAY);
        if(h>=6 && h<12){
            s = "Good morning!";
        }else if(h>=12 && h<19){
            s = "Good afternoon!";
        }
    }
}
```

```
        }else{
            s = "Good night!";
        }
        s = "Hi, "+user+". "+s;
        return s;
    }
}
```

这是一个非常简单的 HelloWorld 程序，写得简单是为了大家更容易看懂程序的变换过程。这个程序虽然简单却符合遗留系统的许多特点：没有注释、顺序编程、没有层次、聚合度低，等等。因此我们进行了初步重构，增加注释、调整顺序、重命名变量、进行分段：

```
/**
 * The Refactoring's hello-world program
 * @author fangang
 */
public class HelloWorld {
    /**
     * Say hello to everyone
     * @param now
     * @param user
     * @return the words what to say
     */
    public String sayHello(Date now, String user){
        //Get current hour of day
        Calendar calendar = Calendar.getInstance();
        calendar.setTime(now);
        int hour = calendar.get(Calendar.HOUR_OF_DAY);

        //Get the right words to say hello
        String words = null;
        if(hour>=6 && hour<12){
            words = "Good morning!";
        }else if(hour>=12 && hour<19){
            words = "Good afternoon!";
        }else{
            words = "Good night!";
        }
        words = "Hi, "+user+". "+words;
    }
}
```

```

        return words;
    }
}

```

然后将两段注释中的代码分别提取出来形成 `getHour()`与 `getSecondGreeting()`函数:

```

/**
 * The Refactoring's hello-world program
 * @author fangang
 */
public class HelloWorld {
    /**
     * Say hello to everyone
     * @param now
     * @param user
     * @return the words what to say
     */
    public String sayHello(Date now, String user){
        int hour = getHour(now);
        return "Hi, "+user+". "+getSecondGreeting(hour);
    }

    /**
     * Get current hour of day.
     * @param now
     * @return current hour of day
     */
    private int getHour(Date now){
        Calendar calendar = Calendar.getInstance();
        calendar.setTime(now);
        return calendar.get(Calendar.HOUR_OF_DAY);
    }

    /**
     * Get the second greeting.
     * @param hour
     * @return the second greeting
     */
    private String getSecondGreeting(int hour){
        if(hour>=6 && hour<12){

```

```
        return "Good morning!";
    }else if(hour>=12 && hour<19){
        return "Good afternoon!";
    }else{
        return "Good night!";
    }
}
}
```

通过这个例子我们可以看到，将没有先后顺序的语句调整编写顺序是一种等量变换，将语句中某段相对独立的语句提取出来形成一个函数，而让原语句调用这个函数，也是一种等量变换。除此之外，调整函数名称、修改变量名称等等，都是等量变换。等量变换，程序还是那些程序，执行的结果还是那些结果，但程序组织结构发生了变化，变得更加可读、可维护、易变更了，这就是重构的意义。

将密密麻麻的程序代码按照功能划分在数个函数中，可以有效地提高代码的可读性；将程序中各种各样的变量和函数合理地予以命名，并在函数头或定义处适时地进行注释，也是在提高代码的可读性；将各种各样品种繁多的函数恰当地分配到各自的对象中合理地组织起来，则是在有效提高系统的可维护性与易变更性。这些对于一个遗留系统的日常维护与生命延续都是非常有帮助的。

2.2 盘点我们的重构工具箱——对HelloWorld抽取类和接口

下面我们来盘点一下系统重构工具箱里都有什么，也就是看一看系统重构到底都有哪些方法。系统重构总是在不同层次上调整我们的代码，因此重构方法也就分为了多个层次。从总体上看，重构方法分为以下几个层次：方法的重构、对象的重构、对象间的重构、继承体系间的重构、组织数据的重构与体系架构的重构。

前面那个例子我们可以清楚地看到方法的重构过程。方法的重构往往发生在一个对象的内部，是对一个对象内部的优化。从这个例子中，我们首先看到了增加注释、调整顺序、重命名变量、进行分段等操作，这些虽然不是什么重构方法，却是我们进行前期准备的常用技法。随后我们将通过注释分段存放的各个代码段提取出来，形成单独的函数。这种重构方法被称为“抽取方法”（Extract Method）。

随后我们继续重构。仔细观察这个程序我们发现，这个程序的内聚性不好。最好用一个

DateUtil 管理诸如 getHour() 的方法，用 Greeting 管理各种问候语及其规则，因此我们进行了如下重构：

```
/**
 * A utility about time.
 * @author fangang
 */
public class DateUtil {
    /**
     * Get hour of day.
     * @param date
     * @return hour of day
     */
    public int getHour(Date date){
        Calendar calendar = Calendar.getInstance();
        calendar.setTime(date);
        return calendar.get(Calendar.HOUR_OF_DAY);
    }
}

/**
 * All kinds of greeting.
 * @author fangang
 */
public class Greeting {
    /**
     * Get the first greeting.
     * @param user
     * @return Hi, {user}.
     */
    public String getFirstGreeting(String user){
        return "Hi, "+user+". ";
    }

    /**
     * Get the second greeting.
     * @param hour
     * @return if in the morning, say "Good morning!",
     * if in the afternoon, say "Good afternoon!",

```

```
* else say "Good night!".
*/
public String getSecondGreeting(int hour){
    if(hour>=6 && hour<12){
        return "Good morning!";
    }else if(hour>=12 && hour<19){
        return "Good afternoon!";
    }else{
        return "Good night!";
    }
}

/**
 * The Refactoring's hello-world program
 * @author fangang
 */
public class HelloWorld {
    /**
     * Say hello to everyone
     * @param now
     * @param user
     * @return the words what to say
     */
    public String sayHello(Date now, String user){
        DateUtil dateUtil = new DateUtil();
        int hour = dateUtil.getHour(now);

        Greeting greeting = new Greeting();
        return greeting.getFirstGreeting(user)+
            greeting.getSecondGreeting(hour);
    }
}
```

这里我们将 `getHour()` 从 `HelloWorld` 类中抽取出来，放到了 `DateUtil` 类中，使 `HelloWorld` 类中仅保留一个引用。同时，我们将 `getFirstGreeting()` 与 `getSecondGreeting()` 从 `HelloWorld` 类中抽取出来，放到了 `Greeting` 类中，使原程序变为引用。这样的技法我们称之为“抽取类”（Extract Class）。

再仔细观察这一段程序：

```
/**
 * Get the second greeting.
 * @param hour
 * @return if in the morning, say "Good morning!",
 * if in the afternoon, say "Good afternoon!",
 * else say "Good night!".
 */
public String getSecondGreeting(int hour){
    if(hour>=6 && hour<12){
        return "Good morning!";
    }else if(hour>=12 && hour<19){
        return "Good afternoon!";
    }else{
        return "Good night!";
    }
}
```

除了 morning、afternoon、night 以外，如果我们再增加 evening，程序的可扩展性就不好了。因此我们将它们提取出 GreetingRule 接口，并分别编写了 morning、afternoon、night 和 evening 的实现类：

```
/**
 * Greeting rules interface
 * @author fangang
 */
public interface GreetingRule {
    /**
     * @param hour
     * @return whether the rule is right
     */
    public boolean isRight(int hour);

    /**
     * @return the greeting words
     */
    public String getGreeting();
}
```

```
/**
 * The greeting in the morning
 * @author fangang
 */
public class MorningGreeting implements GreetingRule {

    /* (non-Javadoc)
     * @see org.refactoring.helloWorld...GreetingRule#getGreeting()
     */
    @Override
    public String getGreeting() {
        return "Good morning! ";
    }

    /* (non-Javadoc)
     * @see org.refactoring.helloWorld...GreetingRule#isRight(int)
     */
    @Override
    public boolean isRight(int hour) {
        if(hour>=6 && hour<12){
            return true;
        }
        return false;
    }
}
```

其他几个实现类我就不累赘了，最后将 `getSecondGreeting()` 方法改成这样：

```
/**
 * Get the second greeting.
 * @param hour
 * @return if in the morning, say "Good morning!",
 * if in the afternoon, say "Good afternoon!",
 * if in the evening, say "Good evening! ",
 * else, say "Good night!".
 */
public String getSecondGreeting(int hour){
    for(GreetingRule greetingRule : greetingRules){
        if(greetingRule.isRight(hour)){
```

```
        return greetingRule.getGreeting();
    }
}
throw new RuntimeException("Error when greeting! ");
}
```

这种将相似的，或者同类型的代码抽取出来形成接口，以及接口下的多个实现，我们称之为“抽取接口”（Extract Interface）。

看了这些例子你可能会会有一个疑问，这样简单的程序搞成这样，是否值得？是的，不错，程序的结构应当与需求的复杂度相适应。简单的需求编写简单的程序，复杂的需求编写复杂的程序。如果将简单的需求，为了玩技术，搞成了复杂的程序，那就是“过度设计”。但这里为了更加清楚地向大家展示重构，又能够使大家不要因为复杂的程序而分心，故而将简单程序过度设计了一把。但在后面我们可以看到，当业务需求逐渐复杂时，我们进行的以上这些重构是值得的。

书后附录列出了所有的重构方法，它们都来源于重构经典书籍《重构：改善既有代码的设计》，日后我们将反复运用这些方法。

正如武侠大师金庸所说的“无招胜有招”，如此多的重构方法不是要让你去生搬硬套，而是应该对其进行深刻理解以后，最终变成你自己的重构方法。我们在实际工作中不要过于介意用了什么重构方法，哪次重构是用的哪个方法，只要是合适的设计就 OK。但是，在无招胜有招之前，我们必须要有招，即学会了、理解了各个招式，在实际工作中你才能想起这些招式，去运用这些招式。

然而，系统重构经典书籍不少，指导我们实践的书籍却不多。相信有许许多多有志之士，在看过重构的书籍以后，激情洋溢、热血沸腾，但回到现实世界中，回到实际工作中却无所适从，经过一番苦苦挣扎之后，从此作罢。因此，本书将从实践出发，用实际工作中的示例来向大家展示，系统重构是怎样指导我们工作的，让大家真正地用起来。

第3章 小步快跑的开发模式

作为一名优秀的开发人员，重构应当成为一种习惯。但作为重构初学者来说，这并不容易，即使你已经从业很多年。

所谓改变一种习惯，就好像习惯了左手吃饭的人，突然被要求改成右手，你必然会感到许多的不适。首先的不适是一种心理上的障碍，即如何能迈出重构代码的第一步，这需要一种决心。初次尝试重构的开发人员总是要经历一段很长时间的纠结，纠结对既有的代码是隐忍还是重构。是的，走出重构的第一步并不容易，它对于你来说是一小步，但对于这个项目甚至这个机构来说却是一大步。

另一种不适则是来源于一种思维的定式，是小设计还是大布局。来看看我曾经经历过的一个故事吧。

3.1 大布局你伤不起

一个已经维护了很多年的大型软件项目，典型的遗留系统。经过多年的维护，代码已经凌乱不堪，开发人员已经换了很多茬，维护工作变得越来越困难。这时，甲乙双方经过仔细的沟通达成了一致，要对其进行一次大规模的改造。

毫无疑问，改造工作总是从会议开始的。经过许多轮会议，起初是跟客户谈，然后是闭门会议，项目经理、资深开发人员、系统架构师坐在一起，探讨的就一个事儿，系统改造的行动方案。这时，一位颇有经验的系统架构师说话了：“系统改造嘛我以前做过。我们首先应当对现有系统进行梳理，梳理它的业务功能。维护了那么多年，业务需求文档肯定不齐全。功能业务都没有梳理清楚，怎么改造呀？”说得还挺中肯。

“功能梳理清楚了，就开始梳理系统架构：怎么分层，怎么制定接口。这是一个非常细致的工作，分层一定要合理，要反复论证。然后，所有的接口都是梳理之后设计出来的，要形成设计文档。”

“制定这个设计文档非常重要，没有设计文档怎么开发呀？设计错了谁来负责呀？岂不是会乱成一锅粥了。所以设计文档也应当论证，反复论证，最后才开始开发、测试、交付。整个项目搞下来怎么也得好几个月吧。”

听了系统架构师的发言，大家一致觉得可行。再一想到他之前做过，有经验，项目经理就这样拍板了。随后就是持续数月的分析、整理、开发、测试，几十号人干得不亦乐乎。最后到了该结尾的时候了，和谐社会嘛应当是一个相当和谐的结局，系统改造成功，新系统顺利上线，甲乙双方十分满意。但非常遗憾的是，这个故事却没有得到那个和谐的结局。原系统经过多年的维护，发现并解决了许多问题，这些问题都体现在程序代码中非常细节的设计，但却为之后的分析设计师们所忽略。因此在新系统上线试运行问题此起彼伏，新系统仿佛就是数年前那个旧系统的翻版。许多在旧系统曾经发现并早已修复的问题都在新系统中再次出现。当最终系统试运行结束时，我们花费了数月的辛苦劳动打了水漂，客户再也不提这个改造后的新系统了。

听了这个故事你作何感想呢？你有过类似的经历吗？正是有了那么多系统改造惨痛的教训，才使得那么多项目经理对系统重构讳莫如深。人总是喜欢学会遗忘，遗忘那些曾经刺痛过我们心灵的经历。但遗忘永远不可能让我们进步。所以，让我们正视问题，分析其错误的根源。这个根源就是持续数月后才得到的反馈。

分析整个项目的过程，我们惊奇地发现，从整理需求、设计接口、着手开发、测试，最后交付，在整整数月的时间，我们都无法知道，我们做得对还是不对。最后的结局因此就变成了这场赌博，要么成功，要么失败，各 50% 的几率，这就是问题的关键。什么是成功的项目管理？就是要将失败的几率降至最低，而这里我们没有做到。

怎样才能将失败的几率降至最低呢？如果我们每工作一个月就可以知道这个月的工作对不对，则错误给我们的损失就是一个月；如果我们一周检查一次，错误的损失就是一周；如果是一天，损失的就是一天。总之，错误发现得越早，我们的损失也就越小。

假如时光可以倒转，当我们重新回到以往那个会议室重新规划那个系统改造方案时，我们应当怎样做呢？不要再去布那么大的局了，大布局你伤不起！不要去规划那个遥远而无法掌控的未来，它只会让你头昏脑胀。小设计可以让你获得成功。

3.2 小设计而不是大布局

开车的朋友一定深有体会，驾驶汽车其实就是在不断矫正汽车行驶方向的一个过程。在整个驾驶过程中，你必须全神贯注地紧盯前方，通过方向盘不断矫正方向，否则即使行驶在直线路段也可能偏离车道。那些疲劳驾驶的司机，因为进入睡眠状态，无法再矫正方向，车辆就会越来越偏离航向。这种情况下，即使数秒钟的小盹，也能造成车毁人亡的严重后果。

重构与驾车虽然属于完全不同的领域，但其道理是相同的。我们在运用重构方法修改代码的过程中也是经常会犯错的（是人就会犯错）。犯错就如同偏离了航向一样，因此我们需要不断去矫正。既然是矫正，就必须有一个正确的标准，以及判断是否正确的方法。驾车过程中正确的标准是车辆是否正确行驶在自己的车道上，判断是否正确的方法则是司机朋友的目测。同样地，重构过程中正确的标准是我们的软件是否保持重构前的外部行为，判断的方法则是测试，不论是手工测试还是自动化测试。

说起来这个道理很简单，但问题是你不可能随时都在测试，随时都在矫正，这是不可能的。因此，我们必须要有个周期，即间隔多长时间测试并矫正一次。周期越长，出大问题的几率就越大；周期越短，我们所需要付出的成本就越高，因为每进行一次测试都是需要我们去付出成本的。周期的长短是需要我们去不断权衡的。

然而，与驾车不同，重构的测试周期还要取决于完成一次重构并提交代码的周期。因为软件重构总是这样一个过程，首先修改一部分代码，使程序处于一个错误而无法编译运行的状态，然后完成其他所有相关代码的修改，使程序恢复编译可运行的状态。在这个中间状态中，我们是无法测试的，或者说测试是无意义的。只有当相关代码都修改完成，程序恢复到可运行状态时，测试才变得有意义。每完成这样一个过程，我们称之为“完成一次重构”。而完成一次重构所花费的时间，是决定我们测试周期的关键因素。

既然如此，我们完成一次重构到底要花费多少时间呢？这听起来像是在进行数学推导，但其中的道理就是这样。一次重构花费的时间，是由我们的设计决定的。小设计，我们对代码的修改量少，我们完成重构的时间就短；大设计，我们对代码的修改量大，我们完成重构的时间必然长。完成重构时间长，测试的周期就长，我们发现错误的时间就晚，可能给我们造成的损失必然就大。

大布局为什么我们伤不起？漫长的业务整理，漫长的设计与开发，持续数月之久。在这数月中我们一直都无法评估自己是否正确。当辛苦数月之后才被告知我们犯错了，一切都为时已晚，项目不得不滑入了失败的深渊。这就是前面那个故事中系统改造失败的根本原因。

我们说，大布局不可以，但大设计同样不可能。我开始要重构了，我们思考了很多问题，运用了很多重构手法，来完成一次重构。这样的重构，代码修改量必然多，重构周期必然长，出错的风险就大。因此，我们的重构应当是一个一个小设计。运用一个重构手法，解决一个问题，完成一次重构，测试通过，再运用一个重构手法，解决另一个问题，完成另一次重构……如此往复。

但是一个我必须要澄清的概念就是，判断是否是大设计的衡量标准并不是代码行数。举

一个简单例子，我们将一段上千行的代码从一个函数，原封不动地挪到另一个函数中，而这段代码本身改动很小，这属于不属于大设计呢？显然不属于。因为我们真正修改的代码量很少。后面我们将看到，重构过程会经常进行这种代码的搬移。

小步快跑让我们每次重构的时候只关注一个问题，运用一个重构手法去解决这一个问题。这样就使我们每次在修改问题时不会想得太多太远。但是，小步快跑并不是要我们完全没有远期规划，不是这样的。你可以有远期规划，但这种规划不要做得太早，过早做出这些规划往往容易顾此失彼。先工作一段时间，做一些基础的工作，让我们对系统的整体有了一个比较全面的认识，然后再规划，这样将得到更好的效果。

同时，它要求我们对远期的规划不要过细。越近期的计划越细，越远期的规划越粗。一些人之所以急急匆匆地去做细致的远期规划，是因为担心今天做出来的东西，到了今后发现不对，得改，所以今天多想想，多规划一些。但问题是，今天规划的就正确吗？如果是当然很好，而现实常常是否。不正确的规划却常常适得其反，让我们陷入一种困境，一种既不能解决当前问题，又不得不为错误设计埋单的困境之中。而重构的思想却完全打破了这种思维习惯。重构不惧怕修改，因为它让明天的代码修改是安全的，而不再是走钢丝。它认为，今天的任务就是改今天的，即使到了明天会被认为是错误的，也是明天再改去。

3.3 小步快跑是这样玩的——HelloWorld重构完成

说了那么多，相信你对小步快跑的概念有了一个初步的印象，但理解还不是很深。让我们来看一个实际工作中的例子，亲身感受一下什么是大布局，什么是大设计，什么是小设计。

还是回到前面那个 Hello World 的例子，起初的需求总是简单而清晰的。当用户登录一个网站时，网站往往需要给用户打一个招呼：“hi, XXX!”。同时，如果此时是上午则显示“Good morning!”，如果是下午则显示“Good afternoon!”，除此以外显示“Good night!”。对于这样一个需求，我们在一个 HelloWorld 类中写了十来行代码：

```
/**
 * The Refactoring's hello-world program
 * @author fangang
 */
public class HelloWorld {
    /**
     * Say hello to everyone
```

```
* @param now
* @param user
* @return the words what to say
*/
public String sayHello(Date now, String user){
    //Get current hour of day
    Calendar calendar = Calendar.getInstance();
    calendar.setTime(now);
    int hour = calendar.get(Calendar.HOUR_OF_DAY);

    //Get the right words to say hello
    String words = null;
    if(hour>=6 && hour<12){
        words = "Good morning!";
    }else if(hour>=12 && hour<19){
        words = "Good afternoon!";
    }else{
        words = "Good night!";
    }
    words = "Hi, "+user+" ". +words;
    return words;
}
}
```

如果需求没有变更，一切都是美好的。但事情总是这样，当软件第一次提交，变更就开始了。系统总是不能直接获得用户名称，而是先获得他的 `userId`，然后通过 `userId` 从数据库中获得用户名。后面的问候可能需要更加精细，如中午问候“Good noon!”、傍晚问候“Good evening!”、午夜问候“Good midnight!”。除此之外，用户希望在一些特殊的节日，如新年问候“Happy New Year!”、情人节问候“Happy Valentine’s Day!”、三八妇女节问候“Happy Women’s Day!”，等等。除了已经列出的节日，他们还希望临时添加一些特殊的日子，因此问候语需要形成一个库，并支持动态添加。不仅如此，这个问候库应当支持多语言，如选择英语则显示“Good Morning!”，而选择中文则显示“上午好!”……总之，各种不同的需求源源不断地被用户提出来，因此我们的设计师开始头脑发热、充血，开始思维混乱。是的，如果你期望自己能一步到位搞定所有这些需求，你必然会感到千头万绪、顾此失彼，进而做出错误的设计。但如果你学会了“小步快跑”的开发模式，一切就变得没有那么复杂了。

首先，我们观察原程序，发现它包含三个相对独立的功能代码段，因此我们采用重构中的“抽取方法”，将它们分别抽取到三个函数 `getHour()`、`getFirstGreeting()`、`getSecondGreeting()`

中，并让原函数对其引用：

```

/**
 * The Refactoring's hello-world program
 * @author fangang
 */
public class HelloWorld {
    /**
     * Say hello to everyone
     * @param now
     * @param user
     * @return the words what to say
     */
    public String sayHello(Date now, String user){
        int hour = getHour(now);
        return getFirstGreeting(user)+getSecondGreeting(hour);
    }

    /**
     * Get current hour of day.
     * @param now
     * @return current hour of day
     */
    private int getHour(Date now){
        Calendar calendar = Calendar.getInstance();
        calendar.setTime(now);
        return calendar.get(Calendar.HOUR_OF_DAY);
    }

    /**
     * Get the first greeting.
     * @param user
     * @return the first greeting
     */
    private String getFirstGreeting(String user){
        return "Hi, "+user+". ";
    }

    /**
     * Get the second greeting.
     * @param hour
     * @return the second greeting

```

```
*/
private String getSecondGreeting(int hour){
    if(hour>=6 && hour<12){
        return "Good morning!";
    }else if(hour>=12 && hour<19){
        return "Good afternoon!";
    }else{
        return "Good night!";
    }
}
}
```

这次重构虽然使程序结构发生了较大变化，但其中真正执行的代码却没有变化，还是那些代码。随后，我们核对需求发现，用户需求分成了两个不同的分支：对用户问候语的变更，和关于时间的问候语变更。为此，我们再次对 HelloWorld 的程序进行了分裂，运用重构中的“抽取类”，将对用户问候的程序分裂到 GreetingToUser 类中，将关于时间的问候程序分裂到 GreetingAboutTime 类中：

```
/**
 * The Refactoring's hello-world program
 * @author fangang
 */
public class HelloWorld {
    /**
     * Say hello to everyone
     * @param now
     * @param user
     * @return the words what to say
     */
    public String sayHello(Date now, String user){
        GreetingToUser greetingToUser = new GreetingToUser(user);
        GreetingAboutTime greetingAboutTime = new GreetingAboutTime(now);
        return greetingToUser.getGreeting() + greetingAboutTime.getGreeting();
    }
}

/**
 * The greeting to user
 * @author fangang
 */
```

```

public class GreetingToUser {
    private String user;
    /**
     * The constructor with user
     * @param user
     */
    public GreetingToUser(String user){
        this.user = user;
    }
    /**
     * @return greeting to user
     */
    public String getGreeting(){
        return "Hi, "+user+". ";
    }
}

/**
 * The greeting about time.
 * @author fangang
 */
public class GreetingAboutTime {
    private Date date;
    public GreetingAboutTime(Date date){
        this.date = date;
    }
    /**
     * @param date
     * @return the hour of day
     */
    private int getHour(Date date){
        Calendar calendar = Calendar.getInstance();
        calendar.setTime(date);
        return calendar.get(Calendar.HOUR_OF_DAY);
    }
    /**
     * @return the greeting about time
     */
    public String getGreeting(){
        int hour = getHour(date);
        if(hour>=6 && hour<12){

```

```
        return "Good morning!";
    }else if(hour>=12 && hour<19){
        return "Good afternoon!";
    }else{
        return "Good night!";
    }
}
}
```

系统重构到这一步，我们来看看用户关于时间问候语部分的变更需求：问候需要更加精细，如中午问候“Good noon!”。除此之外，用户希望在一些特殊的节日，如新年问候“Happy New Year!”等等。此时我们发现，我们对时间问候语的变更不再需要修改 `HelloWorld` 或其他什么类，而是仅仅专注于修改 `GreetingAboutTime` 就可以了，这就是因重构带来的改善。

同时，我们发现，过去只需 `getHour()` 就足够，而现在却需要 `getMonth()` 与 `getDay()`。随着程序复杂度的提升，我们适时进行了一次重构，将与时间相关的程序抽取到一个新类 `DateUtil` 中，就可以顺利地改写原有的时间问候语程序：

```
/**
 * The utility of time
 * @author fangang
 */
public class DateUtil {
    private Calendar calendar;
    /**
     * @param date
     */
    public DateUtil(Date date){
        calendar = Calendar.getInstance();
        calendar.setTime(date);
    }
    /**
     * @return the hour of day
     */
    public int getHour(){
        return calendar.get(Calendar.HOUR_OF_DAY);
    }
    /**
     * @return the month of date
     */
}
```

```

public int getMonth(){
    return calendar.get(Calendar.MONTH)+1;
}
/**
 * @return the day of month
 */
public int getDay(){
    return calendar.get(Calendar.DAY_OF_MONTH);
}
}
/**
 * The greeting about time.
 * @author fangang
 */
public class GreetingAboutTime {
    private Date date;
    public GreetingAboutTime(Date date){
        this.date = date;
    }
    /**
     * @return the greeting about time
     */
    public String getGreeting(){
        DateUtil dateUtil = new DateUtil(date);
        int month = dateUtil.getMonth();
        int day = dateUtil.getDay();
        int hour = dateUtil.getHour();

        if(month==1 && day==1) return "Happy New year! ";
        if(month==2 && day==14) return "Happy Valentine's Day! ";
        if(month==3 && day==8) return "Happy Women's Day! ";
        if(month==5 && day==1) return "Happy Labor Day! ";
        .....

        if(hour>=6 && hour<12) return "Good morning!";
        if(hour==12) return "Good noon! ";
        if(hour>=12 && hour<19) return "Good afternoon! ";
        if(hour>=19 && hour<22) return "Good evening! ";
        return "Good night! ";
    }
}

```

最后，我们建立 user 表存放用户信息，创建 UserDao 接口及其实现类，为 GreetingToUser 提供用户信息访问的服务；我们用 greetingRule 表存放各种问候语，创建由 GreetingRuleDao 接口及其实现类，为 GreetingAboutTime 提供一个可扩展的、支持多语言的问候语库（如图 3.1 所示）。所有这一切都是在现有基础上，通过小步快跑的方式一步一步演变过来的。

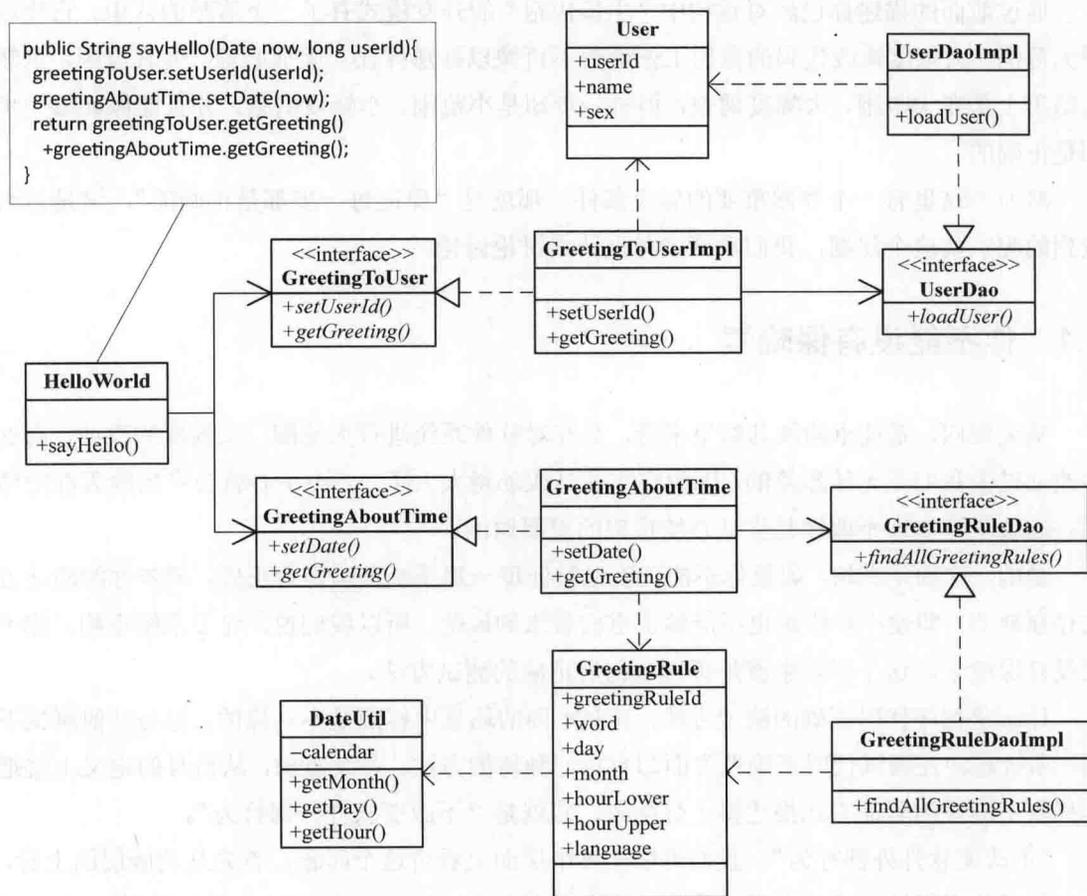


图3.1 HelloWorld的设计图

小步快跑是一种逐步进化式的程序优化过程，它是重构思想的重要核心。后面我们还会用更多实际工作中的示例，让你真实体会到小步快跑的开发过程。

第4章 保险索下的系统重构

通过前面的描述你已经对重构中“小步快跑”的开发模式有了一个清楚的认识。它让这种大范围、大幅度修改代码的重构工作变得不再像以往那样让人胆战心惊。究其原因，虽然从结果上是在大范围、大幅度调整，但每一步却是小范围、小幅度调整，并且能保证每一步都是正确的。

然而，这里有一个非常重要的假设条件，那就是“保证每一步都是正确的”，这是怎么做到的呢？就这个问题，我们需要展开来认真讨论讨论。

4.1 你不能没有保险索

毫无疑问，系统重构就其结果来看，是在对软件系统进行大范围、大幅度的改动，而这种改动过去我们是无法想象的，因为它实在是改动得太大了，一不小心就会产生毁灭性的结果。这就是许多人不敢轻易尝试系统重构的重要原因。

是的，这就是真相，如果你不能正确地验证每一步系统重构是否正确，或者你的验证方式存在缺陷，即使小步快跑也不能解决重构带来的风险。所以我们说，对于系统重构，你不能没有保险索。这个保险索就是每次重构后正确的测试方法。

什么是程序代码正确的测试方法，其在不同的场景中标准是不一样的。但与其他测试不同，系统重构在测试代码正确性方面却有自己独特的方法。系统重构，从自身的定义上就把其代码正确性的验证方式描述得十分明白，那就是“不改变软件外部行为”。

“不改变软件外部行为”，我们可以从多个层面上看待这个问题。首先从功能层面上看，重构前系统提供给用户什么样的功能，重构后也应当提供同样的功能。说得更具体一些，重构前，用户从界面上输入什么内容，发出什么请求，得到什么结果，那么重构后用户应当得到同样的结果。这种结果可能体现在前端界面所展现的结果中，也可能体现在此处操作后存入数据库的数据中，即重构前后系统存入数据库的数据也应当是一致的。存入数据库的数据同样是测试中系统输出的一种。

从功能层面再往下钻就到了代码层面。打开我们的软件系统，有各个层次、各项接口和各种函数。我们进行系统重构往往是在某个层次、某项接口或某个函数上进行的重构。比如，

对某项接口进行重构，那么我们可能会修改这个接口的实现类、方法函数、底层调用类，但不论怎么修改，这个接口必须保持不变。也就是说，对于代码层面的重构，“不改变软件外部行为”是针对的这个接口。你可以重构接口内部，但必须保证接口外部是不变的。

所以，系统重构的测试可以从两个层面来进行：系统测试与单元测试。系统测试往往是一种手工测试，当然也可以使用 QTP 等工具进行一些简单的录制。重构前我们首先通过需求文档确认系统现有功能，根据现有功能设计测试用例。然后进入系统，确保每个测试通过，并录制成 QTP 脚本。这样，我们对重构的准备工作就完成了。在整个测试过程中，我们每完成一次重构就去手工执行这些测试，或者执行 QTP 脚本，保证每个测试通过。如果测试不通过，则要么寻找问题原因并解决，要么就恢复到上次测试的版本，此次重构宣布失败。

反复地手工进行同样的测试是一件比较烦人的事，因此你可以有两个选择：录制 QTP 脚本，以及在接口层面建立自动化测试程序。这里所说的自动化测试程序是指那些基于 JUnit 编写的自动化测试程序，它实际上是在代码层面进行的单元测试。建立自动化测试的关键在于我们在哪个层面建立测试。代码有许多层次，有函数级别、有类级别、有接口或抽象类级别，从系统分层结构来说有底层、DAO 层、BUS 层、Web 层。每次重构都是站在某个层次进行重构，因此自动化测试代码也应当在这个层次上写。比如最初的重构是在函数层次上进行的，就应当对函数写测试代码；后来在对象层次上进行重构，就对对象写测试代码；再到接口层次……

与手工测试一样，在系统重构前，我们首先要保证原有代码自动化测试通过。然后执行重构，保证每次重构都能够测试通过，如此往复。但重构中的自动化测试有一个问题就是，测试接口不太稳定。起初是在函数级别进行的重构，我们写了针对函数的测试代码。但随着重构的深入，我们开始在对象层面进行重构了，可能要调整原有的函数。这意味着原有的针对函数编写的测试代码将失效而必须要遗弃，这不得不说是一种浪费。因此我给大家的建议是，不要过早编写自动化测试代码，它不一定能节省我们的时间，甚至可能花费更多。最初的重构可以采用手工测试或 QTP 测试的方式进行。

4.2 自动化测试——想说爱你不容易

正如许多事情都有其两面性一样，测试方法也是这样。要保证测试方法正确，最简单、最直观的想法就是多写些测试用例，从更多的角度去测试。但这必然增加我们的测试成本。小步快跑要求我们频繁进行测试，假如我们重构的周期是 20 分钟，但测试却要花掉 10 分钟，

那么这样的成本就实在太大了。假如这种测试还是开发人员手工测试，每天都要对同样的测试反复执行数十遍，那么开发人员估计就要疯掉了。

你可能立即就想到自动化测试了。是的，在许多重构的书籍中，大师们都建议我们在重构开始前，首先建立自动化测试机制。但遗憾的是，我经过多年的实践总结出来的经验是，这几乎不可能实现。每次重构，我们面临的都是一个遗留系统。大多数遗留系统都有一些共同的特征：代码凌乱，没有清晰的接口；代码间耦合度高，相互依赖严重；Web层、业务层、数据访问层往往没有清晰的界限，代码相互掺杂其中。在这样的情况下，编写自动化测试代码是几乎不可完成的任务。当然，这里所说的自动化测试代码，是指那些基于JUnit编写的自动化测试程序。

举一个简单的例子：假如你现在要测试一个开票类，想编写它的测试代码。本来这个开票类并不复杂，业务也很清晰。但是在函数传递参数时，其中一个参数是Web容器中的Request、Response或Session。这下麻烦了，为了测试一个简单的函数，我们必须启动整个Web应用，这是我们不可接受的。

随后你可能会说了，我们为什么非要传递一个真正的Request、Response或Session呢？我们Mock（指在测试过程中，对于某些不容易构造或不容易获取的对象，用一个虚拟对象来替代以使测试得以继续的方法）一个假的嘛！想法不错，但你真正去尝试Mock时会发现这也是一个不可完成的任务。Request、Response或Session有许多的状态，属性变量中又有对象，又有属性变量。除此还有大量集合变量，集合变量里都有什么对象，天才知道。因此，即使你费尽千辛万苦Mock出来，也可能因某些属性不对而使得测试失败。

另一个写自动化测试程序比较忌讳的就是访问数据库。比如你这次执行的插入操作成功了，并不意味着下次执行就可以成功。下次执行会报“主键冲突”错误，出现这个错误并不是被测程序错了，而是测试程序错了。上次执行一个查询产生的结果集，不一定是下一次执行同样一个查询产生的结果。查询结果变了，并不意味着被测程序错了，而是测试程序不对。自动化测试程序之所以能够自动化执行，必须要保证测试过程是可以反复执行的，并且不论什么时候执行都有一个确定的结果。

总之，自动化测试不是银弹，并不是所有代码都适合自动化测试。与Web容器或其他设备驱动相关的代码是不适合自动化测试的，因为我们在测试的时候不希望去启动Web容器或其他设备。因此，我们在做自动化测试程序前，首先应当确保要测试的程序已经与Web容器或其他设备驱动相关的代码充分解耦。一个比较好的办法就是分离出Web层与BUS层，Web层负责从Web容器中获取数据，并打包传递给BUS层，而BUS层则完成真正需要测试

的业务逻辑。

另一个不适合自动化测试的就是要访问数据库的程序，因为它们执行的结果总是与数据库状态有关，无法获得稳定而可以不断复现的结果。所以，我们解决它的最好办法就是将访问数据库的部分 Mock 掉。如何 Mock 呢？你不能 Mock 一个 JDBC，也不能 Mock 一个 Hibernate，因为那都过于复杂了，你唯一可以做的就是将 DAO 层 Mock 掉。这就要求我们对系统重构的时候，要将数据库访问的代码从业务代码中脱离出来，写入到 DAO 层。最后，被 Mock 的 DAO 层代码并不真正去访问数据库。每当客户程序传入一个参数时，它首先作为测试程序去验证这个参数是否与预期一致，然后返回一个确定的结果。

4.3 我们是这样自动化测试的——JUnit下的HelloWorldTest

说了那么多，让我们用示例看看，系统重构是应该怎样做自动化测试的。还是回到前面那个 HelloWorld 的例子（详见 3.3 节“小步快跑是这样玩的”），该类中有一个 sayHello() 方法，只要我们输入当前的时间与用户名，就返回对该用户的问候语。如果当前时间是上午，则返回“Hi, XXX. Good morning!”；如果是下午，则返回“Hi, XXX. Good afternoon!”；如果是晚上，则返回“Hi, XXX. Good night!”，这是 HelloWorld 这个程序实现的功能。

然后我们开始为这段程序编写测试代码（如果采用测试驱动开发，应当先写测试代码再写程序）。我们首先建立一个 test 源程序目录，然后建立与被测程序对应的包和测试程序。这就是说，如果被测程序在“org.refactoring.helloWorld.resource”包中，则测试程序应当建立“test.org.refactoring.helloWorld.resource”包与之对应；如果被测程序叫“HelloWorld”，则建立“HelloWorldTest”类与之对应，这个类是一个 JUnit 测试程序。

下面就是编写这个测试程序执行测试了。由于被测程序有三个分支，即当前时间是上午、下午、晚上，因此我们分别为之建立了三个测试用例，测试程序如下：

```
/**
 * Test for {@link org.refactoring.helloWorld.resource.HelloWorld}
 * @author fangang
 */
public class HelloWorldTest {

    private HelloWorld helloWorld = null;
    /**
```

```
* @throws java.lang.Exception
*/
@Before
public void setUp() throws Exception {
    helloWorld = new HelloWorld();
}

/**
 * @throws java.lang.Exception
 */
@After
public void tearDown() throws Exception {
    helloWorld = null;
}

/**
 * Test method for {@link org...HelloWorld#sayHello(java.util.Date,
 * java.lang.String)}.
 */
@Test
public void testSayHelloInTheMorning() {
    Date now = DateUtil.createDate(2013, 9, 7, 9, 23, 11);
    String user = "鲍晓妹";
    String result = "";
    result = helloWorld.sayHello(now, user);
    assertThat(result, is("Hi, 鲍晓妹. Good morning!"));
}

/**
 * Test method for {@link org...HelloWorld#sayHello(java.util.Date,
 * java.lang.String)}.
 */
@Test
public void testSayHelloInTheAfternoon() {
    Date now = DateUtil.createDate(2013, 9, 7, 15, 7, 10);
    String user = "关二锅";
    String result = "";
    result = helloWorld.sayHello(now, user);
    assertThat(result, is("Hi, 关二锅. Good afternoon!"));
}
```

```
/**
 * Test method for {@link org...HelloWorld#sayHello(java.util.Date,
 *               java.lang.String)}.
 */
@Test
public void testSayHelloAtNight() {
    Date now = DateUtil.createDate(2013, 9, 7, 21, 30, 10);
    String user = "IT攻城狮";
    String result = "";
    result = helloWorld.sayHello(now, user);
    assertThat(result, is("Hi, IT攻城狮. Good night!"));
}
}
```

这段程序采用的是 JUnit4（JUnit 是 Java 世界中知名度最高的单元测试工具，主要用于白盒测试和回归测试）编写的，其中在 `assertThat(result, is("Hi, IT 攻城狮. Good night!"))`；中，第一个参数是被测程序执行的结果，而第二个参数是根据期望结果进行验证。如果执行结果与预期结果相同，则测试通过，否则测试失败。

随后我们运行该测试程序，得到如下结果：

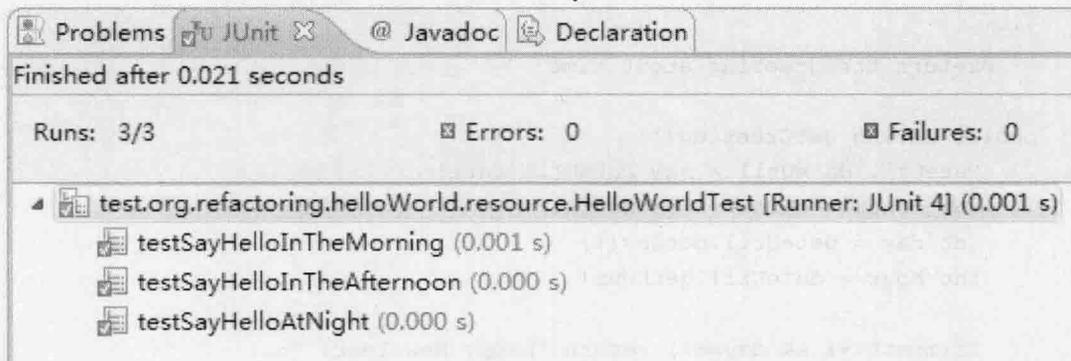


图4.1 JUnit测试结果

三项测试用例全部通过，测试成功！

现在我们为原程序编写了测试用例并全部测试通过，为重构所做的准备工作就一切就绪了。然后，我们开始进行第一次重构。如前面所述，第一次重构我们调整了程序的顺序，进行了分段，增加了注释，并修改了相应的变量，使其更加利于阅读。这是一个小步快跑的过

程，我们完成此次重构只花费了三五分钟。当重构完成，程序重新回到可编译运行状态时，我们执行它的这个测试程序，测试通过。测试通过意味着，虽然程序内部的代码有所修改，但程序对外的功能没有变化，即程序的外部行为没有变化，则重构成功，我们可以继续后面的工作。

第二次重构，我们运用“抽取方法”，从 sayHello()函数中抽取出了 getFirstGreeting()、getSecondGreeting()和 getHour()三个方法。之后我们再次执行测试程序，测试通过。

第三次重构，我们运用“抽取类”，将 getFirstGreeting()与 getSecondGreeting()分别抽取出来形成了 GreetingToUser 和 GreetingAboutTime。完成之后执行测试通过。

第四次重构，我们的需求发生了变化，问候语不仅随一天中的上午、下午、晚上等进行变化，还需要根据不同的日期判断是否是节日。在这种情况下，我们采用“两顶帽子”的方式进行开发：首先不引入新的需求，仅仅修改原程序，使之适应新需求。为此我们从 GreetingAboutTime 类中提炼出 DateUtil，使之不仅有 getHour()，还有 getMonth()与 getDate()。完成重构以后测试通过。

关于“两顶帽子”的设计方式，也是系统重构中另一个不同以往的地方，我们还将后面详细地进行讨论。随后我们开始添加新需求，使 GreetingAboutTime 中的 getGreeting()写成这样：

```
/**
 * @return the greeting about time
 */
public String getGreeting(){
    DateUtil dateUtil = new DateUtil(date);
    int month = dateUtil.getMonth();
    int day = dateUtil.getDay();
    int hour = dateUtil.getHour();

    if(month==1 && day==1) return "Happy New Year! ";
    if(month==2 && day==14) return "Happy Valentine's Day! ";
    if(month==3 && day==8) return "Happy Women's Day! ";
    if(month==5 && day==1) return "Happy Labor Day! ";
    ...

    if(hour>=6 && hour<12) return "Good morning!";
    if(hour==12) return "Good noon! ";
    if(hour>=12 && hour<19) return "Good afternoon! ";
```

```

    if(hour>=19 && hour<22) return "Good evening! ";
    return "Good night! ";
}

```

之后我们的测试不能通过：

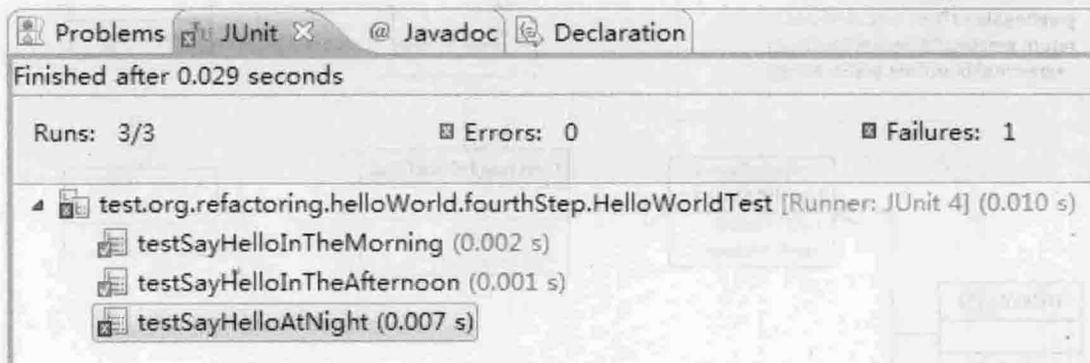


图4.2 测试用例不能通过

为什么 `testSayHelloAtNight` 测试不能通过呢？仔细查看被测程序，我们发现它的功能发生了变化，变为：如果当前时间是1月1日，则返回“Hi, XXX. Happy New Year!”；如果是2月14日，则返回“Hi, XXX. Happy Valentine’s Day!”……如果当前时间都不是这些节日，若是上午则返回“Hi, XXX. Good morning!”，是中午则返回“Hi, XXX. Good noon!”，是下午则返回“Hi, XXX. Good afternoon!”，是傍晚则返回“Hi, XXX. Good evening!”，否则才返回“Hi, XXX. Good night!”。正因为如此，我们需要调整这个测试程序，为每一个分支编写测试用例。测试修改好后，测试通过。

4.4 采用Mock技术完成测试

第五次重构我们引入了数据库的设计，用户信息要从数据库中读取，问候语库存储在数据库中，并支持添加与更新。数据库的引入使自动化测试变得困难了，因为数据状态总是变化着的，而这种变化使得测试过程不能复现，这是我们所不愿看到的。因此，我们在设计时将业务与数据库访问分离，形成了 `UserDao` 与 `GreetingRuleDao`。此时，我们的设计应当遵从“依赖反转”原则，即将 `UserDao` 与 `GreetingRuleDao` 设计成接口，并编写它们的实现 `UserDaoImpl` 与 `GreetingRuleDaoImpl`。这样设计就为我们Mock掉 `UserDao` 与 `GreetingRuleDao`

的实现类创造了条件。

这是我们的设计：

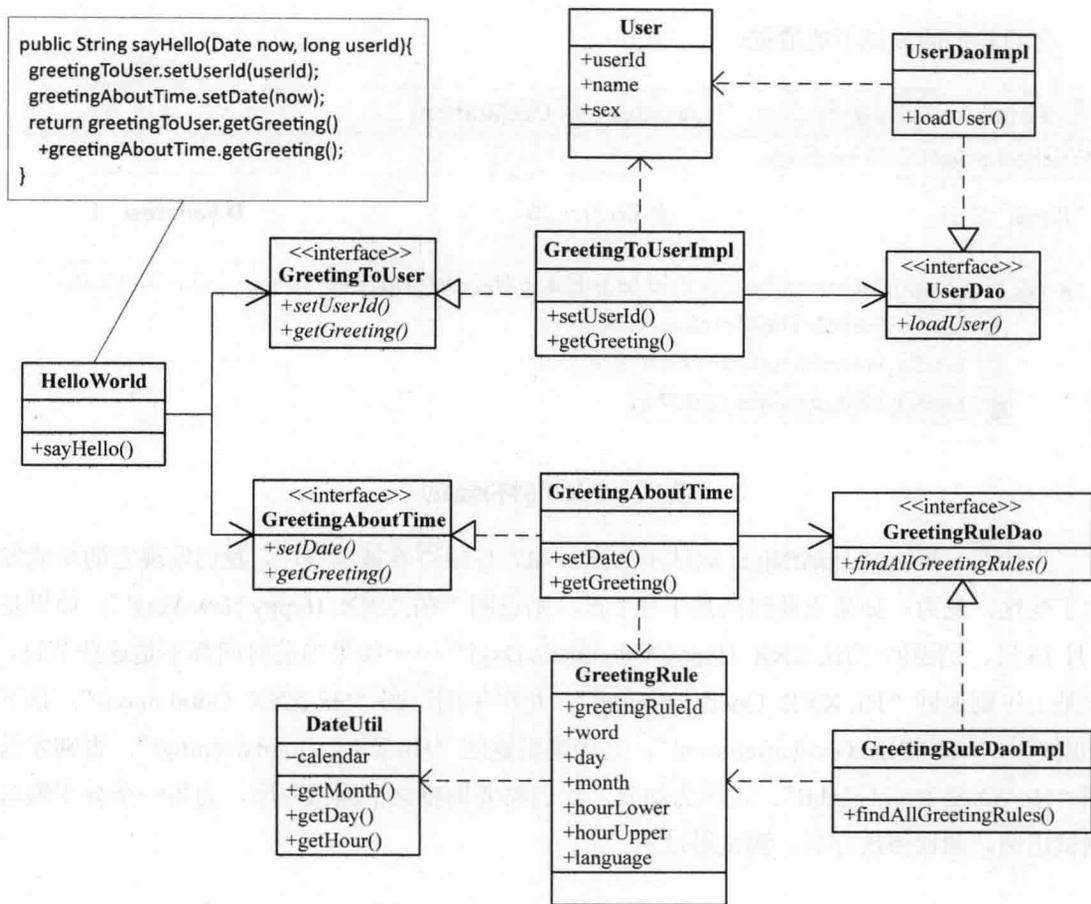


图4.3 HelloWorld的设计图

为此，我们编写了这样的测试程序：

```

private HelloWorld helloWorld = null;
private GreetingToUserImpl greetingToUser = null;
private GreetingAboutTimeImpl greetingAboutTime = null;
private final static List<GreetingRule> GREETING_RULES = getRules();

/**

```

```
* @throws java.lang.Exception
*/
@Before
public void setUp() throws Exception {
    helloWorld = new HelloWorld();
    greetingToUser = new GreetingToUserImpl();
    greetingAboutTime = new GreetingAboutTimeImpl();
    helloWorld.setGreetingToUser(greetingToUser);
    helloWorld.setGreetingAboutTime(greetingAboutTime);
}

/**
 * @throws java.lang.Exception
 */
@After
public void tearDown() throws Exception {
    helloWorld = null;
    greetingToUser = null;
    greetingAboutTime = null;
}

/**
 * Test method for {@link org...HelloWorld#sayHello(java.util.Date,
 java.lang.String)}.
 */
@Test
public void testSayHelloInTheMorning() {
    final Date now = DateUtil.createDate(2013, 9, 7, 9, 23, 11);
    final long userId = 2013090701;

    UserDao userDao = createMock(UserDao.class);
    GreetingRuleDao greetingRuleDao = createMock(GreetingRuleDao.class);
    expect(userDao.loadUser(userId)).andAnswer(new IAnswer<User>() {
        @Override
        public User answer() throws Throwable {
            User user = new User();
            user.setUserId(userId);
            user.setName("鲍晓妹");
            return user;
        }
    });
}}
```

有关此电子图书的说明

本人由于一些便利条件，可以帮您提供各种中文电子图书资料，且质量均为清晰的 PDF 图片格式，质量要高于网上大量传播的一些超星 PDG 的图书。方便阅读和携带。只要图书不是太新，文学、法律、计算机、人文、经济、医学、工业、学术等方面的图书，我都可以帮您找到电子版本。所以，当你想要看什么图书时，可以联系我。我的 QQ 是：**89039855**，大家可以在 QQ 上联系我。

此 PDF 文件为本人亲自制作，请各位爱书之人尊重个人劳动，敬请您不要修改此 PDF 文件。因为这些图书都是有版权的，请各位怜惜电子图书资源，不要随意传播，否则，这些资源更难以得到。