《简约之美》

图书基本信息

书名：《简约之美》

13位ISBN编号：9787115302380

10位ISBN编号：7115302383

出版时间：2013-1

出版社：人民邮电出版社

作者：[美] Max Kanat-Alexander

页数：120

译者：余晟

《简约之美》

前言

好程序员和差程序员的区别在于理解能力。差劲的程序员不理解自己做的事情，优秀的程序员则相反。信不信由你，道理就是这么简单。　　写这本书，是为了帮助各位程序员，以适用于各种编程语言、各种项目的广阔视角来理解软件开发。本书以普通人容易理解的方式，讲解了软件开发的科学规律。　　如果你是程序员，这些规律能够说明，为什么有些开发方法有效，另一些无效。这些规则也会指引你在日常工作中做出开发决策，帮助你的团队进行高质量的交流，最终制订出合理的计划。

如果你不是程序员，但身在软件行业，仍然可以享受到本书的价值：　　·它既是提供给初级程序员的优秀教材，又包含对高级程序员相当有 用的知识；　　·它帮助你更深入地理解软件工程师某些行为的原因，以及软件为何 要以某种方式来开发；　　·它帮助你理解优秀的软件工程师做决定的基本原理，让你与开发人 员更顺畅地沟通。　　理想的状态是，软件行业中的每个人都可以阅读并理解这本书，即便他们没有多少编程经验，甚至母语不是英语也无所谓。如果你已经有相当的技术积累，把握书中的概念会更加容易，但是大部分内容不需要编程经验就能理解。　　实际上，本书虽然讲的是软件开发，却没有多少代码。这怎么可能呢？答案是，其中的思想适用于各种软件项目、各种语言。要明白如何运用这些思想，并不需要懂得某一门具体的编程语言。相反，本书中包含了大量的实例和比喻，它们会让你更好地理解所表述的每条原理。　　最重要的是，这本书是为了帮助你而写的，希望能助你在软件开发中保持头脑清醒、遵守秩序、写出简洁代码。我希望它读起来是一种享受，它有助于改善你的生活，你的软件。　　排版约定　　本书中格式约定如下。　　黑体：表示新术语。　　等宽字体：用于代码示例，在段落中使用时，表示与程序有关的部分，比如变量或者函数名。

此图标表示提示、建议或者普通的旁注。　　致谢　　Andy Oram和 Jolie Kanat两位编辑为本书作了巨大的贡献。 Andy的建议和意见深入且充满智慧；Jolie的坚持和支持促成了本书的最后出版，她为早期手稿所做的大量编辑工作尤其值得感谢。　　Rachel Head是本书的文字编辑，做整理和校对的工作，她的才华无与伦比。　　还要感谢的是与我在开源社区中一同工作或讨论过问题的程序员——尤其是在 Bugzilla项目中共事的几位开发人员，有了你们的帮助，我才有清楚的思维，讲解这些年来真实存在的，活生生的软件系统。　　这些年来，我的 blog上收到的评论和反馈，帮我确定了本书的形式和内容。在这里要感谢参与其中的所有人，即使你们仅仅给我鼓励，或者是告诉我你读过我的文章。

从个人来说，我尤其要感谢 Jevon Milan、Cathy Weaver，以及与他们工作过的所有人。确切地说，有了他们，我才能写出这本书。最后，要向我的朋友 Ron致敬，没有他，这本书根本不可能出现。

使用示例代码　　让我们助你一臂之力。也许你要在自己的程序或文档中用到本书中的代码。除非大段大段地使用，否则不必与我们联系取得授权。例如，无需请求许可，就可以用本书中的几段代码写成一个程序。但是销售或者发布 O ’ Reilly图书中代码的光盘则必须事先获得授权。引用书中的代码来回答问题也无需授权。将大段的示例代码整合到你自己的产品文档中则必须经过许可。　　我们非常希望你能标明出处，但并不强求。出处一般含有书名、作者、出版商和 ISBN，例如“ Code Simplicity：The Science of Software Development by Max Kanat-Alexander（O ’ Reilly，2012）版权所有。

如果有关于使用代码的未尽事宜，可以随时与我们取得联系。　　Safari · 在线图书　　Safari在线图书是应需而变的数字图书馆。它能够让你非常轻松地搜索 7500多种技术性和创新性参考书以及视频，以便快速　　地找到需要的答案。　　订阅后就可以访问在线图书馆内的所有页面和视频。可以在手机或其他移动设备上阅读，还能在新书上市之前抢先阅读，也能够看到还在创作中的书稿并向作者反馈意见。复制粘贴代码示例、放入收藏夹、下载部分章节、标记关键点、做笔记甚至打印页面等有用的功能可以节省大量时间。这本书（英文版）也在其中。欲访问本书英文版的电子版，或者由 O ’ Reilly或其他出版社出版的相关图书，请到网站免费注册。　　我们的联系方式　　请把对本书的评论和问题发给出版社。

《简约之美》

内容概要

《简约之美:软件设计之道》将软件设计作为一门严谨的科学，阐述了开发出优雅简洁的代码所应该遵循的基本原则。作者从为什么以前软件设计没有像数学等学科一样成为一门科学开始入手，道出了软件以及优秀的软件设计的终极目标，并给出了具体的指导规则。

《简约之美》

作者简介

Max Kanat-Alexander：开源项目Bugzilla总架构师，Google软件工程师，作家，8岁开始修电脑，14岁开始编程。codesimplicity.com和fedorafaq.org网站维护者，现居北加州。

《简约之美》

书籍目录

目录

《简约之美》

章节摘录

版权页： 程序的代码应当采用什么结构？是程序的速度重要，还是代码容易阅读重要？为满足需求，应该选择哪种编程语言？软件设计与下列问题无关：公司的结构应该是怎样的？什么时候召开团队会议？程序员的工作时间应该如何安排？程序员的绩效如何考核？这些决策与软件本身无关，只与组织有关。显然，保证这类决策的合理性也是重要的——许多软件项目之所以失败，就是管理失当。但是这不是本书的主题，本书关注的是，如何为你的软件制订合理的技术决策。软件系统中任何与架构有关的技术决策，以及在开发系统中所做的技术决策，都可以归到“软件设计”的范畴里。2.1 程序员也是设计师 在软件项目中，每个程序员的工作都与设计有关。首席程序员负责设计程序的总体架构；高级程序员负责大的模块；普通程序员则设计自己的那一小块，甚至只是某个文件的一部分。但是，即便仅仅是写一行代码，也包含设计的因素。哪怕是单干，也离不开设计。有时候你在敲键盘之前，就做了决定，这就是在做设计。有的夜晚，你躺在床上，还在思考要怎样编程。每个写代码的人都是设计师，团队里的每个人都有责任保证自己的代码有着良好的设计。任何软件项目里，任何写代码的人，在任何层面上，都不能忽略软件设计。这不是说设计应该采取民主程序进行。设计决不应该由某个委员会负责，那样的结果必然很差劲。相反，所有的开发人员都应有权在自己的工作中做出良好的设计决策。如果某位开发人员做了糟糕或者平庸的决策，资深开发人员或者首席程序员就应当推翻这些决策，重新来过。对下属的设计，这些人应当拥有否决权。不过，软件设计的责任应当落实在真正写代码的人身上。身为设计师，必须时时愿意聆听建议和反馈，因为程序员大都比较聪明，有不错的想法。但是考虑了所有这些建议和反馈之后，任何决策都必须由单独的个人而不是一群人来做出。2.2软件设计的科学 现在，软件设计仍然不算科学。什么是科学？词典里的定义有点儿复杂，但简单来说，一门学问要成为科学，必须符合下列标准。科学必须包含汇总而来的知识。也就是，它必须包含事实而不是意见，且这些事实必须汇总起来（比如集结成书）。这些知识必须具有某种结构。知识必须能分类，其中的各个部分必须能够依据重要性之类的指标，妥善建立起与其他部分的联系。科学必须包括一般性的事实或者基本的规则。科学必须告诉你在现实世界中如何做一些事情。它必须能够应用到工作或生活中。通常，科学是经由科学方法来发现或证明的。科学方法必须观察现实世界，提炼出关于现实世界的理论，通过验证理论，而且这些实验必须是可重复的。这样才能说明理论是普适的真理，而不仅仅是巧合或者特例。

《简约之美》

编辑推荐

没有人喜欢复杂的东西，所以软件开发的简约之道一定会受读者青睐。本书作者Max Kanat-Alexander创建的关于Linux的简约单页网站Unofficial Fedora FAQ，月访问量超过10万人次。本书作者还是著名的开源Bugzilla Project的首席架构师、社区创始人和发布经理。

《简约之美》

精彩短评

1、有点失望…
2、都是常识，几句话的事情，非得写成一本书。
3、#图灵PDF#
4、应当是写给大众的普及民用版书吧，没有预期那么好,挺爆还那么贵，性价比不高~
5、32开纸只有100页，还真是挺贵的。。。最主要是，排版好难看啊。。。
6、非常多设计的一些原则和方法, 涉及了软件设计, 开发, 代码, 测试等各个方面, 非常有启发.
7、干货不多，一般般
8、2015
9、没啥意思，对于初学者入门还是挺不错的。
10、薄薄的，但“很重”
11、说的比较本质,着重表明软件开发的科学性
12、当当的忠实书友
13、不是令人印象特别深刻的书，说的都是一些基本常识，大概也就是说程序员其实是『设计师』
14、简约之美
15、不错 传授了一种软件设计思想
16、很简单一本书,把最后两页的附录看了就可以了,说的更极端点,
书的标题 如果 深刻理解了, 书都不用看了, 书名已经概括了书要讲的所有东西.
所以非常适合刚刚学编程的人,
就像译者序说的, 就是 一本 <<常识>>
17、价格很便宜，正版！
18、什么是科学，什么是定理，不要重新发明轮子
19、字字珠玑
20、good job~！！
21、非常短，基本上都是已经知道的大实话。还是值得一看。我们这里有不止一个非常好的**反例**。
22、书很小很薄,更像是小册子,但内容肯定值这个价钱.在外面吃个肯德基还要20块呢.对于工作了两三年的工程师来说,有醍醐灌顶的感觉.可以跟<软件简洁之道>一起看,这本书告诉你方法,而<简约之美>告诉你原因.
23、做为软件初级工程师看起来没什么感觉。。。
24、用作者两句话总结：相比降低开发成本，降低维护成本更重要；维护成本正比于系统复杂度。
25、超简短的小册子，新的软件开发观念
26、建议10分钟左右读完
27、99页。简单真的很美，不要再往软件里面加广告了。不敢写书评，怕一不小心写的比本书的字数还多。
28、好棒的一本书~~自己试过重写代码，也试过把代码从单个文件逐渐演化到各个模块的过程，经过这些实践之后，再回过头来看这本书，感觉真不一样....很多很好的原则可以遵循，写代码也是门手艺活~~
29、100页确实说的东西不多，主要是软件可维护性最重要

30、够短的。。把精华收集到附录挺好的，应该再有索引到正文呀。
31、简约之美：软件设计之道
32、没啥意思，全是废话，内容少的可怜，翻来覆去在绕弯子。
33、保持clean code就对了
34、好程序员和差程序员的区别在于理解能力，差劲的程序员不理解自己做的事情，优秀的程序员则相反。
35、多看阅读限免一天，读了下。写的比较简单，属于软件开发常识类。
36、在看从最基本的原理讲解，收获不少。这叫理论指导实践么。

37、这本书非要分一个类的话，就是代码规范类的吧。这种书我已经读的很多了。但是最重要的还是自己的代码感觉，简称码感。
38、书中有些关注值得细细品味
39、看完没什么感觉
40、好的思考，是需要由根本来打好基礎
41、里面的观点都很简单，看起来是个刚入门的同学看的，其实工作了一段时间的同学也应该好好看看，很多东西看起来简单，做起来真的很难。不要忘了开发软件的初心。
42、一般，不妨一看，小有收益
43、很简短的小册子，将的设计相关的道理都很浅显，倒很是印证了近期所在项目的工作内容。中高级码工可以忽略此书。没有耐心的读者只读附录即可。
44、更多的是看书的内容比较实用，包装我觉得都是次要的
45、还不错。虽然是写给程序员的。但是作为设计师。也能够更好的和程序合作。更好的了解软件整体流程中的一些东西
46、废话连篇 看附录的总结就足够了
47、“ 软件的目的是帮助别人 ”是软件设计六条法则之一。另外还有一个软件设计方程式和四条定律（变化定律、缺陷概率定律、简洁定律和测试定律）。
48、书倒是不错，就是太薄了
49、把东西变得简单是一种伟大的能力，当然这种简单并不是狭义上的简单。
50、书中的内容比较浅显，略读就行了
51、挺不错的书，就是内容有点单调了
52、浏览一遍 没啥感觉
53、简约软件设计原则
54、看完后，我首先想到的是，谷歌的里所有的代码都是统一的编程规范带来的开发效率的提升。
55、还没有时间好好看
56、不看也罢，想看类似的书，还不如看那本《架构之美》更有意义
57、好薄的一本书，一晚上就看完了。删掉无用的代码；只根据现有的需求做可扩展设计，不过度设计；考虑软件可维护性很重要；不愧是一本"常识"书。道理很明显，关键是要在实际工作中具体实践。
58、这本书有点浅 不过例子挺多
59、全文就一个核心观点: 不要过度设计, 处理目前已知的明确的需求, 通过迭代的方式进行开发.
60、对于当前的工作有一些触动。书本身也用简洁的叙述自我点题了。非常明快和踏实的引人思考如何让项目简洁起来。
61、经验之谈，值得读一读
62、一切以简为美
63、常识
64、KISS
65、内容很好，值的一看，推荐！
66、过度设计，过犹不及
67、书很短，两小时读完。 可能之前我已经有相关的知识储备，所以收获不大。
对于初学编程者，想要写出设计更好的代码，更推荐 整洁代码之道，以及重构。
68、驾驭业务的复杂性必须依赖设计的简单性，退而求其次的策略是封装复杂性使其尽量少的被感知。
69、简单易懂~~~~~~~
70、书的内容很少完全纲领性的书
71、基本上全是废话
72、http://teawater.coding.me/mindmap/Code_Simplicity.html

欢迎吐槽
73、几句话就可以总结完了，果然是“ 常识 ”

74、通通都是顯而易見的準則，讀完簡直是在浪費我的時間啊幹
75、书很薄很快翻完了，感觉说的太形而上学，总结一下差不多就是中庸之道，要平衡考虑编码的通用性和复杂度……作者要是能多举点案例就好了

《简约之美》

精彩书评

1、* 每个写代码的人都是设计师* 全部软件都有一个相同的目标：帮助其他人* 任何一点改变，其合意程度与其价值成正比，与所付出的成本成反比* 程序员犯的最常见也是最严重的错误，就是在其实不知道未来的时候去预测未来* DRY--Don't Repeat Yourself* 软件设计三大误区：1. 编写不必要的代码2. 代码难以修改3. 过分追求通用
2、总结了很多在实际开发中遇到的问题，当开发过一两个大系统之后再来读，觉得作者说的都在理。摘录部分笔记：https://github.com/onestarshang/Code_Simplicity_The_Science_of_Development是不是评论太少又不可以？是不是评论太少又不可以？是不是评论太少又不可以？是不是评论太少又不可以？是不是评论太少又不可以？是不是评论太少又不可以？是不是评论太少又不可以？是不是评论太少又不可以？是不是评论太少又不可以？是不是评论太少又不可以？
3、总结性的书籍，虽然不是很厚，但是还是比较有内容的书。值得多看看想想，适合比较初级的程序员。额，附录A是一个比较不错的总结，虽然我是按照顺序看的。在我看完想写做一做笔记的时候，发现编者帮我总结好了.....
4、其实整本书说白了就是几句话：代码一定要保持整洁，不要过度设计，也不要不设计，更重要的是考虑后续的维护成本。但是在实际情况下要贯彻落实书中观点是一件很不容易的事情，除了不断实践，不断试错之外，别无他法。只有自己知道痛了才会长记性，光读一两本这种程序员“心灵鸡汤”型的书是远远不够的。书本身内容两分，译者行文流畅度不错加一分。PS：120页的书要25块，这年头...

《简约之美》

章节试读

1、《简约之美》的笔记-第1页

简约之美

2、《简约之美》的笔记-第29页

合意程度（可取程度） desirability 我是没猜出来这个单词。翻译的好别扭。音译：“ 得劲儿程度 ”。讲的是软件的期待的样子。那个梦想中的软件完成时的样子。

3、《简约之美》的笔记-第63页

The ease of maintenance of any piece of software is proportional to the simplicity of its individual pieces.正比，译反了。好像。

4、《简约之美》的笔记-第30页

有时候是很难的。很难得

5、《简约之美》的笔记-第81页

我们不能保证我们写出来的软件没有一点bug，也不能保证它在未来随着环境等一系列条件的变化还能一直正确运行，因此，我们需要测试。The degree to which you know how your software behaves is the degree to
which you have accurately tested it.我们应该尽可能测试详细、周到，这样我们自己心里就对我们的软件的可靠性更有把握。
另外，软件开发人员都知道，任何一点细小的改动都可能引入另外一些bug，因此我们每一次改动都应该仔细斟酌；最好的方法就是每改动一处就将所有的测试用例跑一遍，以确定我们软件的正确性。这又引入了另外一个问题，如果每次都手动测试的话，这是很大的工作量，一般开发人员也没有这么多的时间；因此，我们需要自动化测试。
我觉得应该给每一个开发人员都配一个测试人员，该测试人员应该对软件的需求了解得很透彻，能够编写自动化测试脚本，与开发人员之间最好有良好的默契。

6、《简约之美》的笔记-第1页

好程序员和差程序员的区别在于理解能力。差劲的程序员不理解自己做的事情，优秀的程序员则相反。信不信由你，道理就是这么简单。---该书前言第一句话。

7、《简约之美》的笔记-第70页

anotherNameLikeThat 这怎么还能抄错。

8、《简约之美》的笔记-第1页

简约之美：软件设计之道【美】Max Kanat-Alexander

常识

2014-09-21 18:44:29
用简单的技术构建清晰的架构

2014-09-21 19:17:43
“ 降低维护成本 ”

2.3　为什么不存在软件设计科学

2014-09-22 21:56:16
这些数学家才是计算机科学之父，计算机科学正是对信息处理所做的数学研究。它并不像现在一些人认为的那样，是关于计算机编程的学问。

2014-09-22 21:58:27
其实，缺席的科学分为两种：软件管理的科学，软件设计的科学。

第3章　软件设计的推动力

2014-09-23 08:56:07
总之，在设计软件时，应当将目标——帮助他人——视为应该考虑的最重要因素，这样，我们才能认识并了解软件设计的真正科学。

2014-09-23 09:00:13
确保软件能提供尽可能多的帮助。

2014-09-23 09:00:17
确保软件能持续提供尽可能多的帮助。

2014-09-23 09:00:22
设计程序员能尽可能简单地开发和维护的软件系统，这样的系统才能为用户提供尽可能多的帮助，而

且能持续提供尽可能多的帮助。

2014-09-23 09:03:07
有时候，学习新技术，做一份好的设计，都要花很长时间，但是长期来看，你做出的选择必须确保开发和维护软件都比较简单。

4.1.5　化简方程式

2014-09-23 09:12:41
随着时间的流逝，它会越来越不重要，甚至完全无足轻重。于是，随着时间流逝，这个方程式变成了

2014-09-23 09:12:33
其实，几乎所有软件设计的决策都完全忽略了未来价值与维护成本的对比。

4.1.6　你需要什么，不需要什么

2014-09-23 09:18:49
相比降低实现成本，降低维护成本更加重要。

5.1　真实世界中程序的变化

2014-09-23 09:28:21
另一点值得学习的有趣之处是，回顾某个特定文件修改历史。如果某个文件存在了很长时间，而且你有程序记录每个文件的修改历史，请回顾整个过程中的每次修改。问问自己，最初写这个文件时，你能预测到这些变化吗，是否一开始写好就能够减轻后期的工作量。总的来说，就是要尝试理解每次修改，看看是否能从中得到一些关于软件开发的新的收获。

5.2.1　编写不必要的代码

2014-09-23 09:28:54
如今，软件设计中有一条常见的规则，叫做“ 你不会需要它 ”（You Ain't Gonna Need It），或者简称为YAGNI。

5.3 渐进式开发及设计

2014-09-24 01:06:19
相比开始就建立完整的系统，一次性构建出来，这种开发方法需要时间更少，也不用考虑过多。如果你习惯其他的开发方法，头一次实践可能并不那么容易，但是经过锻炼，用起来就会变容易。

2014-09-24 01:06:10
总的来说，在其中的每个阶段，下一步都只做最容易的事情。

2014-09-24 00:26:44
有些时候，你甚至需要把某个单独的功能拆分为一系列小的、简单的逻辑步骤，然后才可以很方便地实现。

第6章 缺陷与设计

2014-09-24 21:48:51
在程序中新增缺陷的可能性与代码修改量成正比。

2014-09-24 21:49:52
最好的设计，就是能适应外界尽可能多的变化，而软件自身的变化要尽可能少。

6.1 如果这不是问题……

2014-09-24 21:50:49
没错，如果不添加代码，也不修改代码，就不会引入新的缺陷，这是软件设计中的一条主要规律。

2014-09-24 21:51:11
永远不要“修正”任何东西，除非它真的有问题，而且有证据表明问题确实存在。

2014-09-24 21:53:27

在这类问题上，最有名的错误就是所谓的“提前优化”。也就是说，有些开发人员想让速度尽可能快，所以，他们还没弄清楚速度到底慢不慢，就花时间来优化程序。

2014-09-24 21:53:32
在你的程序中，真正需要关注速度的部分，应该局限于你可以证明的、真正让用户体会到有性能问题的那些部分。对程序的其他部分，最主要关心的还是灵活和简洁，而不是速度。

2014-09-24 21:53:37
要违背这条规律有成千上万种办法，不过遵守它的办法很简单：在动手解决之前，真正拿到证据，证明问题确实存在。

6.2　避免重复

2014-09-24 21:53:46
理想情况下，任何系统里的任何信息，都应当只存在一次。

2014-09-24 21:54:52
遵守这条规则的一个强有力的理由，就是缺陷概率定律。如果新增功能时可以重用代码，就不需要写太多代码，引入错误的可能性也就随之减少了。

第7章　简洁

2014-09-24 21:56:02
软件任何一部分的维护难度，反比于该部分的简洁程度。
换句话说，某一部分的代码越简洁，未来进行变化的难度就越低。

2014-09-24 21:59:02
不过，由大模块构成的系统，质量要差得多，而且，将来你得花很多时间去修正错误，结果就是维护的难度越来越高。相反，简单系统的维护难度会越来越低。长期来看，能保证效率的恰恰是简单的系统，而不是复杂的系统。

2014-09-24 21:59:32
落实这条法则的一个好办法，就是第5章讲解的渐进式开发和设计方法。因为每次添加功能之前都有

个“ 重新设计 ”的过程，所以系统能持续简化。即便不用这种方法，你也可以在增添新功能之前，花点时间去化简任何让你或你的同事觉得不够简洁的代码。

7.1　简洁与软件设计方程式

2014-09-24 22:00:47
我们不必预测未来，完全可以只审视自己的代码，如果它足够复杂，就立刻动手简化它。这就是随时间推移降低维护成本的办法——持续不断地让代码变得更简洁。

7.4　保持一致

2014-09-24 22:07:23
要做到简单，保持一致是很重要的工作。如果你在一个地方采用了某种规则，就应当在其他每个地方都遵守这种规则。

2014-09-24 22:08:19
真实世界里或许不存在这样的一致性，但是程序的世界由你负责，所以必须保持程序的简单和一致。

2014-09-24 22:08:41
编程也是这样——缺乏一致性，只会一团糟。有了一致性，世界就很简单。即便你做不到那么简单，至少也要做到：一旦你理解了某种复杂性，就不必再进行重复劳动。

7.5　可读性

2014-09-24 22:09:07
软件开发领域反复强调一点：代码被阅读的次数远多于编写和修改的次数。

2014-09-24 22:10:53
代码可读性主要取决于字母和符号之间的空白排布。

2014-09-24 22:12:22
一般来说，如果某段代码有很多bug，又难以阅读，那么首先要做的是让它更容易阅读。然后，bug在

哪里才能看得更清楚。

7.5.1 命名

2014-09-24 22:12:50
名字应当足够长，能够完整表达其意义或描述其功能，但不能太长，以免影响阅读。

7.5.2 注释

2014-09-24 22:15:30
但事实并非如此，你的代码可能很容易阅读，但系统仍然非常复杂。

7.6 简洁离不开设计

2014-09-24 22:17:10
反过来，如果你的设计非常好，一般却难得听到多少称赞。设计中的缺陷是大家都看得到的，但逐步演变为良好设计的改进过程，却是不熟悉代码的人看不到的。于是，设计师就成了一个费力不讨好的工作。解决重大缺陷为你赢得很多赞誉，但是避免缺陷发生……嗯，没有人会注意到。

第8章 复杂性

2014-09-24 22:19:37
原有功能越多，新增功能的成本就越高。优秀的设计可以尽量避免此类问题，但是每项新功能仍然会有单独的成本。

2014-09-24 22:19:45
有些项目从一启动就设定了繁多的需求，所以永远无法发布第一版。如果遇到这种情况，就应当删减功能。初次发布不应当设定过高的目标，而应当先让程序跑起来，再持续改进。

2014-09-24 22:24:39
相比众多平庸的开发人员，少量精干的开发人员更容易获得成功。

2014-09-24 23:45:45
一般来说，“ 困于糟糕的技术 ”指的是你之前决定了采用某种技术，因为极度依赖它，长期无法摆脱。这里说的“ 糟糕 ”，意思是你深陷其中（未来无法简单地切换到其他技术），不能灵活地适应未来的需求，或是达不到设计简洁软件所需的质量标准。

2014-09-24 23:46:04
程序员不理解自己的工作，就容易设计出复杂的系统。这可能是恶性循环：理解错误导致复杂性，复杂性又进一步加剧理解错误，如此往复。提升设计水平的最主要办法是，确保自己完全理解所用的系统和工具。

2014-09-24 23:46:26
你对它们的理解越到位，对软件开发的一般规律了解越多，你的设计就越简洁。

8.1 复杂性与软件的用途

2014-09-24 23:54:16
你正开发的任何系统，其基本用途应当相当简单。这样开发出来的系统，既满足实际需求，整体来说也是简单的。

2014-09-24 23:54:29
同样重要的是要思考用户的需求。

8.2 糟糕的技术

2014-09-25 00:10:13
好在，开始使用之前，你可以通过三个因素来判断技术是否“ 糟糕 ”：生存潜力、互通性、对品质的重视。

8.3 复杂性及错误的解决方案

2014-09-25 00:15:22
一旦程序里出现了“ 无法解决的复杂性 ”，就说明设计中有些深层次的基本错误。如果问题在这个层面上无法解决，应当回过头去看看产生问题的真正原因是什么。

2014-09-25 00:15:12
其实，程序员经常这么做。你可能会说：这代码太垃圾了，要添加新功能真够麻烦的。那么，深层次的基本问题就是代码太混乱。所以，你应该整理这些代码，让它们变简洁，你就会发现增加新功能也会变简单。

2014-09-25 00:16:00
所以你真正要做的就是，找出自己所处的环境中最好的办法。

2014-09-25 00:16:26
确认你真正理解了问题的方方面面，找到最简单的解决办法。

2014-09-25 00:16:32
不要问“用现有代码怎么解决这个问题”，或者“Anne教授在程序里是怎么解决这个问题的”，而是问问你自己：“通常情况下，在最完美的方案里，这类问题要如何解决？”

8.4 复杂问题

2014-09-25 00:17:06
如果你在解决复杂问题时遇到了麻烦，那么用简单易懂的文字把它写在纸上，或者画出来。

2014-09-25 00:17:26
有些最优秀的程序设计就是在纸上完成的，真的。把它输入到计算机里只是次要的细节。

2014-09-25 00:17:20
大多数麻烦的设计问题，都可以用在纸上画图或写出来的办法找到答案。

8.5 应对复杂性

2014-09-25 00:18:58
身为程序员，你必然要面对复杂性。其他程序员会写复杂的程序，你必须去修正。硬件设计师和语言

设计师会让你的生活更麻烦。

2014-09-25 00:20:14
如果系统中某个部分太过复杂，有个好办法来解决：把它分解成几个独立的小部分，逐步重新设计。

2014-09-25 00:20:29
更常见的做法是在每个步骤中都把一个复杂的部分拆分成若干个简单的部分。

2014-09-25 00:21:30
如果所有的代码都包含在一个巨大的文件里，改进的第一步就是把某个部分保存到单独的文件里。之后改进这个小部分的设计，然后把另一个部分保存到新的文件，再改进这个部分的设计。如此重复下去，最终得到的就是可靠的、可理解的、可维护的系统。

2014-09-25 00:21:55
如果系统非常复杂，这么做的工作量可能相当大，所以必须有耐心。你必须首先做好设计，改进之后的系统要比现在的简单——即便只是简单一点点。然后，朝着这个目标一步步地前进。得到了这个简单的系统之后，就设计下一个更简单的系统，再朝那个目标前进。不必设计“完美”的系统，因为它不存在。你只需要持续不懈地追求比现有系统更好的系统，最终就会得到相当容易管理的简洁系统。

2014-09-25 00:23:38
你甚至可以这样来处理bug：如果发现修改设计之后，某些bug更容易修复，那么先重新设计代码再修复bug。

8.5.1 把某个部分变简单

2014-09-25 00:28:36
不要仅仅因为权威的肯定就机械地生搬硬套某个工具，我们的选择永远是，在当前的环境下，当前的代码中，做合适的事情。

2014-09-25 00:29:08
要怎么做，才可以让事情处理或是理解起来更容易？

8.5.2 不可解决的复杂性

2014-09-25 00:30:16
简化系统时，你可能会发现某些复杂性是无可避免的，可能所使用的硬件就是很复杂的。如果遇到这类不可解决的复杂性，你要做的就是屏蔽这种复杂性。在程序外面妥善包装上一层，让其他程序员更容易使用和理解。

8.6　推倒重来

2014-09-25 00:31:34
你有足够的资源，可兼顾维护原有系统和重新设计系统。绝对不要为了让程序员重写新系统而停止对原有系统的维护。系统只要在使用，都离不开维护。请记住，你自己的精力也是一种资源，必须慎重分配——如果两线作战，你每天有足够的时间分配给原有系统和新系统吗？

第9章　测试

2014-09-25 00:33:41
你对软件行为的了解程度，等于你真正测试它的程度。

2014-09-25 00:33:45
软件最后一次测试的时间距离现在越近，它可以继续正常运行的可能性就越大。

2014-09-25 00:33:53
除非亲自测试过，否则你不知道软件是否能正常运行。

多看笔记 来自多看阅读 for Kindle

9、《简约之美》的笔记-第37页

满身问题 千疮百孔

《简约之美》

版权说明