



下载APP



26 | 简单设计：难道一开始就要把设计做复杂吗？

2020-07-27 郑晔

软件设计之美

[进入课程 >](#)



讲述：郑晔

时长 13:28 大小 12.34M



你好！我是郑晔。

从专栏开始到现在，关于软件设计，我们已经聊了很多。在学习设计原则和模式这个部分时，我们看着每次的代码调整，虽然结果还不错，但不知道你脑子之中有没有闪过这样的疑问：

如果我的每段代码都这么写，会不会把设计做复杂了呢？

确实，几乎每个人在初学设计的时候，都会有用力过猛的倾向。如何把握设计的度，是个做设计的人需要耐心锤炼的。所以，行业里有人总结了一些实践原则，给了我们一些启发性的规则，帮助我们把握设计的度。



我把这些原则放到这个部分的最后来讲，是因为它们并不是指导你具体如何编码的原则，它们更像是一种思考方法、一种行为准则。

好，我们就来看看这样的原则有哪些。

KISS

KISS 原则，是 “Keep it simple, stupid” 的缩写，也就是保持简单、愚蠢的意思。它告诫我们，对于大多数系统而言，和变得复杂相比，**保持简单能够让系统运行得更好**。

很多程序员都知道这条原则，然而，很少人知道这条原则其实是出自美国海军。所以，它的适用范围远比我们以为的程序员社区要广泛得多。无论是制定一个目标，还是设计一个产品，抑或是管理一个公司，我们都可以用 KISS 作为一个统一的原则指导自己的工作。

这个原则看起来有点抽象，每个人对它都会有自己理解的角度，所以，每个人都会觉得它很有道理，而且，越是资深的人越会觉得它有道理。因为资深的人通常都是在自己的工作领域中，见识过因为复杂而引发的各种问题。比如说，堆了太多的功能，调整起来很费劲这样的情况。我们在专栏前面讲过的各种问题，很多时候都是由于复杂引起的。

所以，对资深的人来说，保持简单是一个再好不过的指引了。其实，每个人都可以针对自己的工作场景给出自己的阐释，比如：

如果有现成的程序库，就不要自己写；

能用文本做协议就别用二进制；

方法写得越小越好；

能把一个基本的流程打通，软件就可以发布，无需那么多的功能；

.....

这种级别的原则听上去很有吸引力，但问题是，你并不能用它指导具体的工作。因为，怎么做叫保持简单，怎么做就叫复杂了呢？这个标准是没办法确定的。所以，有人基于自己的理解给出了一些稍微具体一点的原则。比如，在软件开发领域，你可能听说过的 YAGNI 和 DRY 原则。

YAGNI

YAGNI 是 “You aren’ t gonna need it” 的缩写，也就是，你用不着它。这个说法来自于极限编程社区（Extreme Programming，简称 XP），我们可以把它理解成：**如非必要，勿增功能**。

我们在开篇词里就说过，软件设计对抗的是需求规模。一方面，我们会通过自己的努力，让软件在需求规模膨胀之后，依然能有一个平稳的发展；另一方面，我们还应该努力地控制需求的规模。

YAGNI 就告诫我们，其实很多需求是不需要做的。很多产品经理以为很重要的功能实际上是没什么用的。人们常说二八原则，真正重要的功能大约只占 20%，80% 的功能可能大多数人都用不到。做了更多的功能，并不会得到更多的回报，但是，做了更多的功能，软件本身却会不断地膨胀，变得越发难以维护。

所以，在现实世界中，我们经常看到一些功能简单的东西不断涌现，去颠覆更复杂的东西。比如，虽然 Word 已经很强大了，但对于很多人而言，它还只是一个写字的工具，甚至它的重点排版功能都用得非常少。

于是，这就给了 Markdown 一个机会。它可以让我们专注写内容，而且简单的排版标记在日常沟通中也完全够用。至少，我已经不记得自己上一次用 Word 写东西是什么时候了。

我在 [🔗 《10x 程序员工作法》](#) 里写的大部分内容，实际上就是告诉你，什么样的做法可以规避哪些的不必要功能。通过这里的介绍，我们不难发现，YAGNI 是一种上游思维，就是尽可能不去做不该做的事，从源头上堵住。从某种意义上说，它比其他各种设计原则都重要。

DRY


DRY 是 “Don’ t repeat yourself” 的缩写，也就是，**不要重复自己**。这个说法源自 Andy Hunt 和 Dave Thomas 的《程序员修炼之道》（The Pragmatic Programmer）。这个原则的阐述是这样的：

在一个系统中，每一处知识都必须有单一、明确、权威地表述。

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.


每个人对于 DRY 原则的理解是千差万别的，最浅层的理解就是“不要复制粘贴代码”。不过，两个作者在二十年后的第二版特意强调，这个理解是远远不够的。**DRY 针对的是你对知识和意图的复制。**它强调的是，在两个不同地方的两样东西表达的形式是不同的，但其要表达的内容却可能是相同的。

我从《程序员修炼之道》中借鉴了一个例子，看看我们怎么在实际的工作中运用 DRY 原则。下面是一段打印账户信息的代码，这种写法在实际的工作中也非常常见：

 复制代码

```
1 public void printBalance(final Account account) {
2     System.out.printf("Debits: %10.2f\n", account.getDebits());
3     System.out.printf("Credits: %10.2f\n", account.getCredits());
4     if (account.getFees() < 0) {
5         System.out.printf("Fees: %10.2f-\n", -account.getFees());
6     } else {
7         System.out.printf("Fees: %10.2f\n", account.getFees());
8     }
9
10    System.out.printf(" ----\n");
11
12    if (account.getBalance() < 0) {
13        System.out.printf("Balance: %10.2f-\n", -account.getBalance());
14    } else {
15        System.out.printf("Balance: %10.2f\n", account.getBalance());
16    }
17 }
```

然而，在这段代码中，隐藏着一些重复。比如，对负数的处理显然是复制的，可以通过增加一个方法消除它：

 复制代码

```
1 String formatValue(final double value) {
2     String result = String.format("%10.2f", Math.abs(value));
3     if (value < 0) {
4         return result + "-";
5     } else {
6         return result + " ";
7     }
8 }
9
10 void printBalance(final Account account) {
11     System.out.printf("Debits: %10.2f\n", account.getDebits());
12     System.out.printf("Credits: %10.2f\n", account.getCredits());
```

```
13 System.out.printf("Fees:%s\n", formatValue(account.getFees()));
14 System.out.printf(" ----\n");
15 System.out.printf("Balance:%s\n", formatValue(account.getBalance()));
16
```

还有，数字字段格式也是反复出现的，不过，格式与我们抽取出来的方法是一致的，所以，可以复用一下：

[复制代码](#)

```
1 String formatValue(final double value) {
2     String result = String.format("%10.2f", Math.abs(value));
3     if (value < 0) {
4         return result + "-";
5     } else {
6         return result + " ";
7     }
8 }
9
10 void printBalance(final Account account) {
11     System.out.printf("Debits: %s\n", formatValue(account.getDebits()));
12     System.out.printf("Credits: %s\n", formatValue(account.getCredits()));
13     System.out.printf("Fees:%s\n", formatValue(account.getFees()));
14     System.out.printf(" ----\n");
15     System.out.printf("Balance:%s\n", formatValue(account.getBalance()));
16 }
```

再有，这里的打印格式其实也是重复的，如果我要在标签和金额之间加一个空格，相关的代码都要改，所以，这也是一个可以消除的重复：

[复制代码](#)

```
1 String formatValue(final double value) {
2     String result = String.format("%10.2f", Math.abs(value));
3     if (value < 0) {
4         return result + "-";
5     } else {
6         return result + " ";
7     }
8 }
9
10 void printLine(final String label, final String value) {
11     System.out.printf("%-9s%s\n", label, value);
12 }
13
14 void reportLine(final String label, final double value) {
15     printLine(label + ":", formatValue(value));
16 }
```

```
16 }
17
18 void printBalance(final Account account) {
19     reportLine("Debits", account.getDebits());
20     reportLine("Credits", account.getCredits());
21     reportLine("Fees", account.getFees());
22     System.out.printf(" ----\n");
23     reportLine("Balance", account.getBalance());
24 }
```

经过这样的修改，如果我们要改金额打印的格式，就去改 `formatValue` 方法；如果我们要改标签的格式，就去改 `reportLine` 方法。

可能对于有的人来说，这种调整的粒度太小了。不过，我想说的是，如果你的感觉是这样的话，证明你看问题的粒度太大了。

如果仔细品味这个修改，你就能从中感觉到它与我们之前说的分离关注点和单一职责原则有异曲同工的地方，没错，确实是这样的。在讲分离关注点和单一职责原则的时候，我强调的重点也是**粒度要小**。这个例子从某种程度上说，也是为它们增加了注脚。

虽然我们在这里讲的是代码，但 DRY 原则并不局限于写代码，比如：

注释和代码之间存在重复，可以尝试把代码写得更清晰；

内部 API 在不同的使用者之间存在重复，可以通过中立格式进行 API 的定义，然后用工具生成文档、模拟 API 等等；

开发人员之间做的事情存在重复，可以建立沟通机制降低重复；

.....

所有这些努力都是在试图减少重复，同时也是为了减少后期维护的成本。

简单设计

上面说的这三个原则都是在偏思维方式的层面，而下面这个原则稍稍往实际的工作中靠了一些，它就是简单设计（Simple Design）原则。

这个原则来自极限编程社区，它的提出者是 Kent Beck（这个名字在我的两个专栏中已经出现了很多次，由此可见，他对现代软件开发的影响很大）。

简单设计之所以叫简单设计，因为它只包含了 4 条规则：

通过所有测试；

消除重复；

表达出程序员的意图；

让类和方法的数量最小化。

这 4 条规则看起来很简单，但想做到，对于很多人来说，是一个非常大的挑战。Kent Beck 是极限编程这种工作方式的创始人，所以，想满足他提出的简单设计原则，最好要做到与之配套的各种实践。

我们来逐一地看下每条规则。第 1 条是**保证系统能够按照预期工作**，其实，这一点对于大多数项目而言，已经是很高的要求了。怎么才能知道系统按照预期工作，那就需要有配套的自动化测试。大多数项目并不拥有自己的自动化测试，更何况是在开发阶段使用的单元测试，尤其是还得保证测试覆盖了大多数场景。

在 XP 实践中，想要拥有这种测试，最好是能够以测试驱动开发（Test Driven Development，简称 TDD）的方式工作。而你要想做好 TDD，最根本的还是要懂设计，否则，你的代码就是不可测的，想给它写测试就是难上加难的事情。

后面 3 条规则其实说的是**重构的方向**，而重构也是 XP 的重要实践。第 2 条，消除重复，正如前面讲 DRY 原则所说的，你得能够发现重复，这需要你对分离关注点有着深刻的认识。第 3 条，表达出程序员的意图，我们需要编写有表达性的代码，这也需要你对“什么是有表达性的代码”有认识。我们在讲 DSL 曾经说过，代码要说明做什么，而不是怎么做。

第 4 条，让类和方法的数量最小化，则告诉我们不要过度设计，除非你已经看到这个地方必须要做一个设计，比如，留下适当的扩展点，否则，就不要做。

但是，有一点我们需要知道，能做出过度设计的前提，是已经懂得了设计的各种知识，这时才需要用简单设计的标准对自己进行约束。所以，所谓的简单设计，对大多数人而言，并不“简单”。

我们前面说了，简单设计的理念来自于极限编程社区，这是一个重要的敏捷流派。谈到敏捷，很多人以为做敏捷是不需要设计的，其实这是严重的误解。在敏捷实践的工程派，也就是 XP 这一派中，如果单看这些实践的步骤，你都会觉得都非常简单，无论是 TDD 也好，抑或是重构也罢，如果你没有对设计的理解，任何一个实践你都很难做好。

没有良好的设计，代码就没有可测试的接口，根本没有办法测试，TDD 也就无从谈起。不懂设计，重构就只是简单的提取方法，改改名字，对代码的改进也是相当有限的。

简单设计，是 Kent Beck 这样的大师级程序员在经历了足够的积累，返璞归真之后提出的设计原则，它确实可以指导我们的日常工作，但前提是，我们需要把基础打牢。片面地追求敏捷实践，而忽视基本功，往往是舍本逐末的做法。

总结时刻

今天，我给你讲了一些启发性的编程原则，这些设计原则更像是一种思考方式，让我们在软件设计上有更高的追求：

KISS 原则，Keep it simple, stupid，我们要让系统保持简单；

YAGNI 原则，You aren't gonna need it，不要做不该做的需求；

DRY 原则，Don't repeat yourself，不要重复自己，消除各种重复。

我们还讲了一个可以指导我们实际工作的简单设计原则，它有 4 条规则：

通过所有测试；

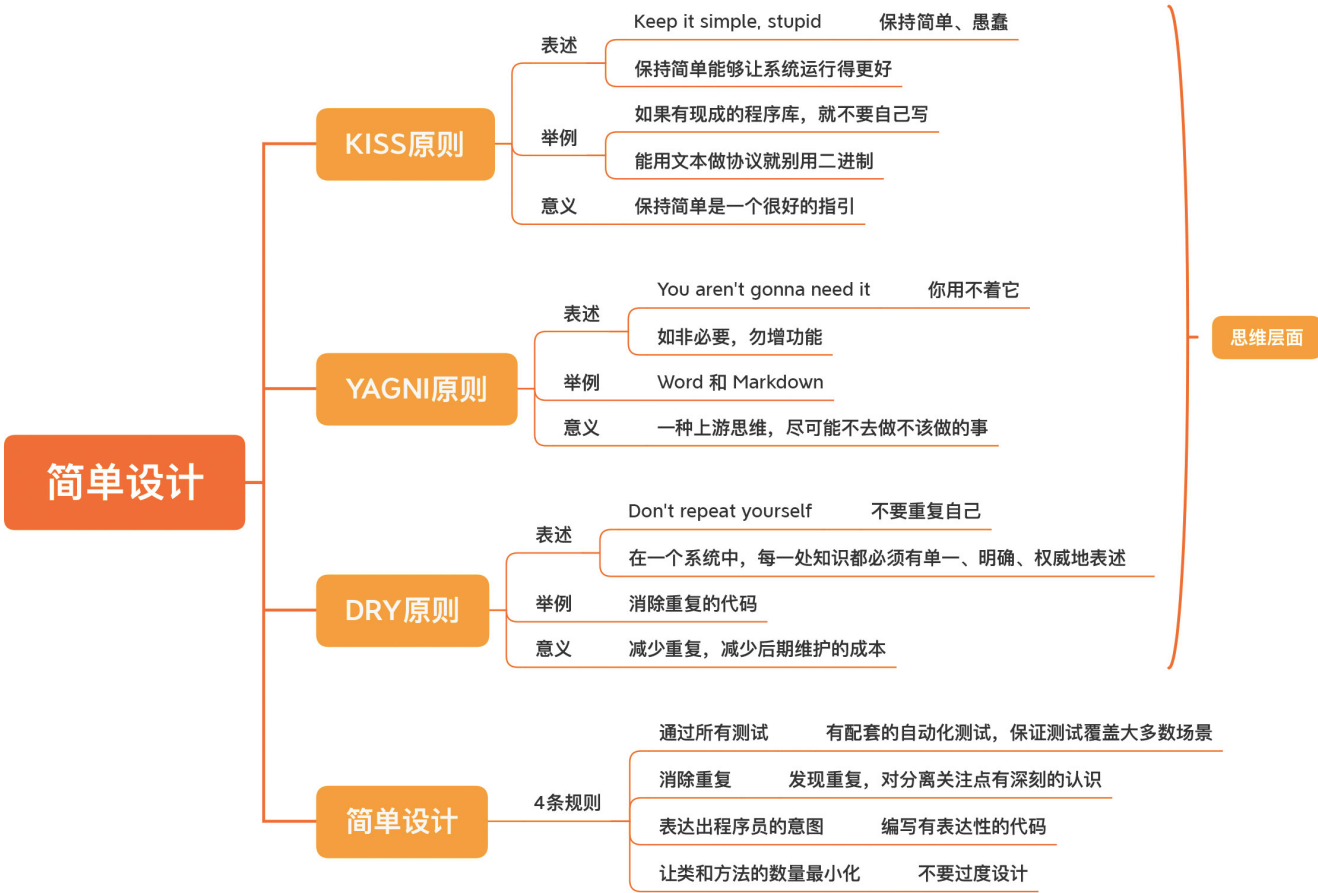
消除重复；

表达出程序员的意图；

让类和方法的数量最小化。

软件设计相关的基础内容，到这里，我已经全部给你讲了一遍。然而，你可能会疑问，有了这些东西之后，我该如何用呢？从下一讲开始，我们来聊聊，如果有机会从头开始的话，该如何设计一个软件。

如果今天的内容你只能记住一件事，那请记住：**简单地做设计。**



思考题

最后，我想请你分享一下，你还知道哪些让你受益匪浅的设计原则，欢迎在留言区写下你的想法。

感谢阅读，如果你觉得这一讲的内容对你有帮助的话，也欢迎把它分享给你的朋友。

更多课程推荐

设计模式之美

前 Google 工程师手把手教你写高质量代码

王争

前 Google 工程师

《数据结构与算法之美》专栏作者



涨价倒计时 🕒

限时秒杀 **¥149**，7月31日涨价至 **¥299**

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 25 | 设计模式：每一种都是一个特定问题的解决方案

下一篇 27 | 领域驱动设计：如何从零开始设计一个软件？

精选留言 (5)

写留言



再来二两杜康酒

2020-07-27

大道至简，知易行难，所以还是要多学习借鉴，多动手，还要有一定的悟性才行~

作者回复: 软件设计必须多多思考。



1



阳仔

2020-07-27

设计原则：
KISS原则
YAGNI原则
DRY原则
简单设计原则：...
展开 ▾

作者回复: 这个总结很到位。

💬 1



monalisali
2020-07-28

很多公司的代码都是没有设计的，这种项目中所谓的“架构师”就是用最简单的3层结构，然后选一个框架实现IoC而已，项目基本没有自动化测试可言。如果项目已经开展好几年，也没人敢改了，一改就是大手术，改出问题还要担责任，只能越陷越深。。。。。

展开 ▾

作者回复: 唉，祖传代码一言难尽啊！

💬 1



捞鱼的搬砖奇
2020-07-27

还记得老师在10X 里说过，“默认所有需求都不做，直到弄清楚为什么要做。用简单的设计，直到设计变得复杂”

展开 ▾

作者回复: 哈哈，是这样的。

💬 1



Jxin
2020-07-27

模块化、轻量级
展开 ▾

💬

👍

