第 12 章 不写注释的四个借口

代码内的文档在软件设计中起着至关重要的作用。注释对于帮助开发人员理解系统和高效工作至关重要，但是注释的作用不止于此。文档在抽象中也起着重要作用。没有注释，您就无法隐藏复杂性。最后，**编写注释的过程（如果正确完成）实际上会改善系统的设计。**相反，如果没有很好的文档记录，那么好的软件设计会失去很多价值。

不幸的是，这种观点并未得到普遍认同。大部分的生产代码基本上不包含任何注释。许多开发人员认为注释是浪费时间。其他人则看到了注释中的价值，但不知何故从不动手编写它们。幸运的是，许多开发团队认识到了文档的价值，并且感觉这样的团队越来越普及了。但是，即使在鼓励文档的团队中，注释也经常被视为繁琐的工作，而且许多开发人员也不了解如何编写注释，因此生成的文档通常是平庸的。文档不足会给软件开发带来巨大且不必要的拖累。

在本章中，我将讨论开发人员用来避免写注释的借口，以及注释真正重要的原因。然后，第 13 章将描述如何编写好的注释，其后的几章将讨论相关问题，例如如何选择变量名以及如何使用文档来改进系统的设计。我希望这些章节能使您相信三件事：好的注释可以对软件的整体质量产生很大的影响；写好注释并不难；并且（可能很难相信）写注释实际上很有趣。

当开发人员不写注释时，他们通常会以以下一种或多种借口为自己的行为辩护：

- “好的代码是自解释的。”
- “我没有时间写注释。”
- “注释很容易过时，并会产生误导。”
- “我所看到的注释都是毫无价值的，何必呢？”

在后续的章节中，我将依次讨论这些借口。

12.1 好的代码是自解释的

有人认为，如果代码编写得当，则代码逻辑是清晰的，不需要注释。这是一个“诱人的谬误”，就像谣言说冰淇淋对您的健康有益：我们真的很想相信！不幸的是，事实并非如此。可以肯定的是，在编写代码时可以做一些事情来减少对注释的需求，例如选择好的变量名（请参阅第 14 章）。尽管如此，仍有大量设计信息无法用代码表示。例如，只能在代码中对类接口的一小部分进行形式化的指定，例如其方法的签名。接口的非形式化部分，例如对每个方法的作用或其结果的含义的高层级描述，只能在注释中进行描述。还有许多代码中无法描述的东西，比如特定设计决策背后的考量，或者调用特定方法的前提条件。

一些开发人员认为，如果其他人想知道某个方法的作用，那么他们应该只需要阅读该方法的代码：这将比任何注释都更准确。读者确实可能会通过阅读其代码来推断该方法的抽象接口，但这既费时又痛苦。另外，如果你在编写代码时期望用户会阅读方法的实现，那你将尝试使每个方法尽可能短，以便于阅读。如果该方法做的事情不简单，你会将其分解为几个较小的方法。这将导致大量的浅方法。此外，这并没有真正使代码更易于阅读：为了

理解顶层方法的行为，读者可能需要了解其内嵌的方法的行为。对于大型系统，让用户通过阅读代码来了解其行为是不切实际的。

此外，注释是抽象的基础。回顾第 4 章，抽象的目的是隐藏复杂性：抽象是实体的简化视图，它保留了必要的信息，但省略了可以安全忽略的细节。**如果用户必须阅读方法的代码才能使用它，那就没有任何抽象可言**：方法的所有复杂性都将暴露出来。没有注释，方法的唯一抽象就是其声明，该声明指定其名称以及其参数和返回结果的名称和类型。该声明缺少太多基本信息，无法单独提供有用的抽象。例如，提取子字符串的方法可能有两个参数，起始和结束，表示要提取的字符范围。仅凭声明，无法确定提取的子字符串是否将包含结束位置所指向的字符，或者如果起始位置在结束位置的后面时会发生什么。注释使我们能够得到调用者所需的额外信息，从而在隐藏实现细节的同时得到简化的视图。用人类语言（例如英语）写注释也很重要，虽然这会使它们不如代码精确，但也提供了更好的表达能力，因此我们可以创建简单直观的描述。如果要使用抽象来隐藏复杂性，则注释必不可少。

12.2 我没有时间写注释

与其他开发任务相比，将注释的优先级降低是很诱人的。如果要在添加新功能和为现有功能写注释之间做出选择的话，选择新功能似乎合乎逻辑。但是，软件项目几乎总是处于时间压力之下，并且总会有比编写注释优先级更高的事情。因此，如果您允许取消文档的优先级，则最终将没有文档。

反驳该借口的是第 3.2 节讨论过的投资思维。如果您想要一个整洁的软件结构，以允许你长期有效地工作，那么您必须花一些额外的时间才能创建该结构。好的注释对软件的可维护性有很大的影响，因此花费在它们上面的精力将很快收回成本。此外，编写注释不需要花费很多时间。问问您自己，假设您不需要写任何注释，那么您花费了多少开发时间来写代码（与设计、编译、测试等相比）。我怀疑答案是否超过 10%。现在假设您花在写注释上的时间与写代码所花费的时间一样多，这应该是一个安全的上限。基于这些假设，编写好的注释最多也只会增加您约 10% 的开发时间。拥有良好文档的好处将迅速抵消这一成本。

此外，许多最重要的注释是与抽象有关的，例如类和方法的顶层文档。第 15 章认为，这些注释应该是设计过程的一部分，并且写文档的行为是用来改进整体设计的一个重要工具。这些注释很快就会物有所值。

12.3 注释很容易过时，并会产生误导

注释有时确实会过时，但这在实践中并不是主要问题。使文档保持最新状态并不需要付出巨大的努力。仅当对代码进行了较大的更改时才需要对文档进行大的更改，并且代码更改将比文档的更改花费更多的时间。第 16 章讨论了如何组织文档，以便在修改代码后尽可能容易地对其进行更新（关键的思想是避免重复的文档，并保持文档靠近相应的代码）。代码审查提供了一种检测和修复陈旧注释的有效机制。

12.4 我所看到的所有注释都是毫无价值的

在这四个借口中，这可能是最有价值的借口。每个软件开发人员都看到过没有提供有用信息的注释，并且大多数现有文档充其量都是这样。幸运的是，这个问题是可以解决的。一旦你知道怎么做，写出有效的文档并不难。接下来的几章将为如何编写良好的文档并持续进行维护提供一个框架。

12.5 良好注释的好处

既然我已经讨论了（并希望揭穿了这些）反对编写注释的论点，让我们考虑一下从良好注释中将获得的好处。**注释背后的总体思想是捕获设计者所想的但不能在代码中表示的信息。** 这些信息包括从低层级的详细信息（例如，导致了复杂代码的硬件奇葩行为）到高层级的概念（例如，类的基本原理）的所有内容。当其他开发人员后续对代码进行修改时，这些注释将使他们能够更快、更准确地工作。没有文档，未来的开发人员将不得不重新研究或猜测开发人员的原始想法，这将花费额外的时间，并且如果新开发者误解了原始设计者的意图，则存在导致代码缺陷的风险。即使是原作者在修改代码时注释也是有价值的：如果距离你最后一次在一段代码中工作已经有几个星期了，你会忘记许多最初的设计细节。

第 2 章介绍了复杂性在软件系统中表现出来的三种方式：

- **变更放大**：看似简单的变更需要在许多地方进行代码修改。
- **认知负荷**：为了进行更改，开发人员必须累积大量信息。
- **未知的未知**：尚不清楚需要修改哪些代码，或必须考虑哪些信息才能进行这些修改。

好的文档可以帮助解决后两个问题。通过为开发人员提供他们进行更改所需的信息，并使开发人员可以忽略不相关的信息，文档可以减轻认知负荷。没有足够的文档，开发人员可能必须阅读大量代码才能重新构建出设计人员的想法。文档还可以通过阐明系统的结构来减少“未知的未知”，从而可以清楚地了解与任何给定的变更相关的信息和代码。

第 2 章指出，导致复杂性的主要原因是依赖性和模糊性。好的文档可以阐明依赖关系，并且可以填补空白以消除模糊性。

接下来的几章将向您展示如何编写好的文档。还将讨论如何将文档编写集成到设计过程中，从而改善软件设计。

12.6 不同的观点：注释是一种失败

Robert Martin 在其所著的《代码整洁之道》一书中对注释持一种更加消极的观点：

> ... 注释最多也就是一种必须的恶。若编程语言有足够的表达力，或者我们长于用这些语言来表达意图，就不那么需要注释——也许根本不需要。

> 注释的恰当用法是弥补我们在代码中未能表达清楚的内容... 注释总是代表着失败，我们总有不用注释便很难表达代码意图的时候，所以总要有注释，这并不值得庆贺。

我同意好的软件设计可以减少对注释的需求（特别是方法体中的注释）。但注释并不代表失败。它们提供的信息与代码提供的信息完全不同，而且当前还无法用代码来表示这些信息。代码和注释各自擅长表达它们所代表的信息，它们各自也提供了重要的好处。即使注释中的信息可以在代码中得到某种程度的体现，这是否是一个改进也并不明晰。

注释的目的之一是可以避免不必要的代码阅读：例如，与其阅读方法体中的整个代码块，开发人员只需要阅读一个简短的接口注释来获得调用该方法所需的所有信息。Martin 采取了相反的方法：他提倡用代码替换注释。与其通过写注释来解释方法代码块中发生了什么，Martin 建议将该代码块提取到单独的方法中（不加注释）并使用方法名来代替注释。这会导致很长的名称，例如 `isLeastRelevantMultipleOfNextLargerPrimeFactor`。即使有了所有这些单词，这种名称也晦涩难懂，提供的信息还不如一个好的注释。而且，通过这种方式，开发人员每次调用该方法时都相当于在重新输入方法的文档！

我担心 Martin 的设计哲学会鼓励程序员有一种不好的态度，他们会避免写注释，以免看起来像是一种失败。这甚至可能导致好的设计师受到错误的批评：“你这些还需要注释的代码是有什么问题吗？”

良好的注释不是失败。它们增加了代码的价值，并发挥着定义抽象和管理系统复杂性的基本作用。