第 16 章 修改现有的代码

第 1 章介绍了软件开发是如何迭代和增量的。大型软件系统是通过一系列演化阶段开发的，其中每个阶段都添加了新功能并修改了现有模块。这意味着系统的设计在不断演进。不可能从一开始就为系统构思出正确的设计。一个成熟系统的设计更多地取决于系统演化过程中所做的更改，而不是初始的概念。前面的章节描述了如何在初始设计和实现过程中降低复杂性。本章讨论如何防止复杂性随着系统的演进而蔓延。

16.1 保持战略式的思考

第 3 章介绍了战术式编程和战略式编程之间的区别：在战术式编程中，主要目标是使某些事物快速工作，即使这会导致额外的复杂性；而在战略式编程中，最重要的目标是进行出色的系统设计。战术式的方法很快会导致系统设计混乱。如果您想要一个易于维护和扩展的系统，那么“能工作的”并不是一个足够高的标准。您必须优先考虑设计并从战略角度进行思考。当您修改现有的代码时，此想法也是适用的。

不幸的是，当开发人员对现有代码进行更改（例如缺陷修复或加入新功能）时，他们通常不会从战略角度进行思考。一种典型的心态是“实现该功能，我能做出的最小改变是什么？”有时开发人员认为这是合理的，因为他们对修改的代码不放心。他们担心较大的更改会带来更大的风险，会引入新的缺陷。然而，这导致了战术式的编程。每一个最小的变化都会引入一些特殊情况、依赖性或其他形式的复杂性。结果，系统设计变得更糟了一点，并且问题随着系统演进的每一步而累积。

如果要保持系统的简洁设计，则在修改现有代码时必须采取战略式的方法。**理想情况下，当您完成每次更改时，系统的结构将像是在最开始的设计中就考虑了这个更改。** 为了实现此目标，您必须抵制快速解决问题的诱惑。相反，请根据所需的更改来考虑当前的系统设计是否仍然是最佳的。如果不是，请重构系统，以便最终获得最佳设计。通过这种方法，每次修改都会持续改善系统设计。

这也是第 3.2 节介绍的投资思维的一个示例：如果您花费一些额外的时间来重构和改善系统设计，您将得到一个更整洁的系统。这将加快开发速度，您将收回在重构方面投入的精力。即使您的特定更改不需要重构，您仍然应该注意在代码中可以修复的设计缺陷。每当您修改任何代码时，都尝试在该过程中至少找到一些改进系统设计的地方。**如果您没有使设计变得更好，则您有可能会使它变得更糟。**

如第 3 章所述，投资思维有时与商业软件开发的现实相冲突。如果以“正确的方式”重构系统需要三个月，而快速且不整洁的修复仅需两个小时，则您可能必须采取快速而不整洁的方法，尤其是当您被要求在紧张的期限内完成工作时。或者，如果重构系统会造成不兼容，从而影响许多其他的人员和团队，则这个重构可能有些不切实际。

尽管如此，您应尽可能抵制这些妥协。问问自己：“考虑到我目前的限制，这是否是我能做的最好的工作来创建一个整洁的系统设计？”也许有一种替代方法几乎可以像 3 个月的重构一样整洁，但是可以在几天内完成？或者，如果您现在没有能力做大规模的重构，请让您的老板为您分配时间，让您在当前的截止日期之后再来做。每个开发组织都应计划将其全部工作的一小部分用于清理和重构；从长远来看，这项工作一定是物有所值的。

16.2 维护注释：将注释保留在代码附近

当您更改现有代码时，更改很有可能会使某些现有的注释失效。修改代码时，也很容易忘记更新注释，从而导致注释不再准确。陈旧的注释使读者感到沮丧，如果有很多这样的注释，读者就会开始不信任所有注释。幸运的是，只要有一点纪律和一些指导规则，就可以在不需要大投入的情况下使注释保持更新。本节及随后的部分提出了一些具体的技巧。

确保注释更新的最佳方法是将注释放置在它们所描述的代码附近，以便开发人员在更改代码时可以看到它们。注释离其关联的代码越远，被正确更新的可能性就越小。例如，方法的接口注释的最佳位置是在代码文件中，紧靠该方法主体的位置。对方法的任何更改都将涉及此代码，因此开发人员很可能会看到接口注释，并在需要时进行更新。

对于 C 和 C++ 等具有单独的代码和头文件的语言，一种替代方法是将接口注释放在 `.h` 文件中方法声明的旁边。但是，这距离代码还有很长的路要走。开发人员在修改方法的主体时将看不到这些注释，因此需要打开其他文件并查找接口注释来更新它们，这需要额外的工作。有人可能会争辩说接口注释应该放在头文件中，以便用户在不查看代码的情况下就能了解如何使用这个抽象层。然而，用户应该不需要阅读代码和头文件；他们应该从由 Doxygen 或 Javadoc 等工具编译的文档中获取信息。此外，许多 IDE 都会提取文档并将其呈现给用户，例如在键入方法名称时显示方法的文档。鉴于已经有这样的工具，文档应位于对开发人员进行代码开发最方便的位置。

在编写实现注释时，不要将整个方法的所有注释放在方法的顶部。把他们分解开来，将每个注释向下写到最合适的范围，即包括该注释所引用的所有代码的范围。例如，如果一种方法具有三个主要阶段，则不要在方法的顶部写一个详细描述所有阶段的注释。而是为每个阶段编写一个单独的注释，并将该注释放置在相应阶段的第一行代码的正上方。另一方面，在方法实现的顶部添加注释描述总体的策略也可能会有所帮助，例如：

```
//  We proceed in three phases:
//  Phase 1: Find feasible candidates
//  Phase 2: Assign each candidate a score
//  Phase 3: Choose the best, and remove it
```

更多的细节可以在各个阶段代码的正上方记录。

通常，离描述的代码越远，注释应该越抽象（这减少了注释因代码更改而无效的可能性）。

16.3 注释属于代码，而不是提交日志

修改代码时，常见的错误是将有关更改的详细信息放入源代码存储库的提交消息中，而不是将其记录在代码中。尽管将来可以通过扫描存储库的日志来浏览提交消息，但是需要该信息的开发人员不太可能知道要查看存储库的日志。即使他们确实查看了日志，找到正确日志的过程也会很乏味。

在编写提交消息时，请问问自己：未来的开发人员是否需要使用该信息？如果是，则应该在代码中记录此信息。以一个描述了微妙问题导致代码变更的提交消息为例，如果代码中未对此进行记录，那么开发人员可能会稍后撤消这个更改，而没有意识到他们已经重新引入了一个缺陷。如果您也想在提交消息中包含此信息的副本，那也可以，但是最重要的事情是把它放在代码中。这说明了将文档放置在开发人员最有可能看到它的地方的原则，而提交日志可不是这样的地方。

16.4 维护注释：避免重复

确保注释更新的第二种技术是避免重复。如果文档重复，那么开发人员将很难找到并更新所有相关副本。因此尽量将每个设计决策精确的记录一次。如果代码中有多个地方受某个特定决策的影响，请不要在所有这些地方重复注释。而应该找到放置注释最明显的位置。例如，假设存在与某个变量相关的棘手行为，这会影响使用变量的几个不同地方，您可以在变量声明旁边的注释中记录该行为。如果开发人员在理解使用该变量的代码时遇到麻烦，他们自然会在这里进行检查。

如果没有一个“明显的”地方来将特定的文档放在开发人员可以找到的地方，那么可以创建一个 `designNotes` 文件，如第 13.7 节所述；或者在现有的地方中选择一个最好的地方，并把文档放在那里。此外，可以在其它地方添加简短的注释以指向中心位置，比如：“查看 `xyz` 中的注释以理解下面的代码。”如果引用因为主注释被移动或删除而变得过时，这种不一致性将是很明显的，因为开发人员将无法在指定的位置找到注释，他们可以使用版本控制历史记录来确认注释发生了什么事情，并相应地更新引用。相反，如果文档是重复的，而一些副本没有得到更新，那么开发人员就不会知道他们使用的是陈旧的信息。

不要将一个模块的设计决策记录在另一个模块中。例如，不要在方法调用前添加注释，以解释被调用方法中发生的事情。如果读者想知道，他们应该查看该方法的接口注释。好的开发工具通常会自动提供此信息，例如，如果您选择了方法的名称或将鼠标悬停在该方法的名称上，则将显示该方法的接口注释。尽量让开发人员容易找到合适的文档，但是不要通过重复文档来做到这一点。

如果信息已经在程序之外的某个地方记录了，不要在程序内部重复记录，只需引用外部文档。 例如，如果您编写一个实现 HTTP 协议的类，那么就不需要在代码中描述 HTTP 协议。在网上已经有很多关于这个文档的来源，只需在您的代码中添加一个简短的注释，并附上其中一个来源的 URL 即可。另一个例子是已经在用户手册中记录的特性。假设您正在编写一个实现命令集合的程序，其中每个命令都有一个负责实现的方法。如果有描述这些命令的用户手册，就不需要在代码中重复这些信息。相反，在每个命令方法的接口注释中包含如下简短说明即可：

```
// Implements the Foo command; see the user manual for details.
```

让读者能轻松找到理解代码所需的所有文档是很重要的，但这并不意味着您必须编写所有这些文档。

16.5 维护注释：检查待提交的变更

确保文档保持最新状态还有一个好方法，在将变更提交到版本控制系统之前，花费几分钟以检查该提交的所有变更，并确保文档正确反映了每个变更。这些提交前的检查还有可能检测到其他问题，例如意外地将调试代码留在系统中，或者尚未完成的 TODO 项目。

16.6 更高层级的注释更易于维护

关于文档维护的最后一个想法：如果注释比代码更高层级和更抽象，则注释更易于维护。这些注释不会反映代码的详细信息，因此它们不会受到次要的代码更改的影响，只有整体行为的变化才会影响这些注释。当然，正如第 13 章所讨论的那样，某些注释的确需要详细和精确。但总的来说，最有用的注释（它们不是简单地重复代码）也最容易维护。