第 2 章 复杂性的本质

这本书是关于如何设计软件系统以最小化其复杂性。第一步是了解敌人。究竟什么是“复杂性”？您如何判断系统是否过于复杂？是什么导致系统变得复杂？本章将在较高的层级上解决这些问题。后续章节将向您展示如何从较低的层级上根据特定的结构特征来识别复杂性。

识别复杂性的能力是至关重要的设计技能。它使您可以先找出问题，然后再付出大量努力，并可以在不同的选择中做出正确的选择。判断一个设计是否简单比创建一个简单的设计要容易得多，但是一旦您能认识到一个系统过于复杂，就可以使用该能力指导您的设计哲学走向简单。如果设计看起来很复杂，请尝试其他方法，看看是否更简单。随着时间的流逝，您会注意到某些技术往往会导致设计更简单，而其他技术则与复杂性相关。这将使您更快地产出更简单的设计。

本章还列出了一些基本假设，这些基本假设为本书的其余部分奠定了基础。后面的章节将采用本章的内容，并用其论证各种改进和结论。

2.1 复杂性的定义

出于本书的目的，我以实用的方式定义“复杂性”。**复杂性是指那些与软件系统相关的而且让系统难以理解和修改的任何事物。** 复杂性可以采取多种形式。例如，可能很难理解一段代码是如何工作的，可能需要花费很多精力才能实现较小的改进，或者可能不清楚必须修改系统的哪些部分才能进行改进，也可能是在不引入额外问题的情况下很难修复一个代码缺陷。如果一个软件系统难以理解和修改，那它就是复杂的。如果很容易理解和修改，那它就是简单的。

您还可以从成本和收益的角度来评估复杂性。在复杂的系统中，即使实施很小的改进都需要大量的工作。而在一个简单的系统中，可以用更少的精力实现更大的改进。

复杂性是开发人员在尝试实现特定目标时在特定时间点所经历的。它不一定与系统的整体大小或功能有关。人们通常使用“复杂”一词来描述具有复杂功能的大型系统，但是如果这样的系统易于使用，那么就本书而言，它并不复杂。当然，实际上几乎所有大型复杂的软件系统都很难使用，因此它们也符合我对复杂性的定义，但这不一定是事实。小型的功能不复杂的系统也可能非常复杂。

复杂性取决于最常见的活动。如果系统中有一些非常复杂的部分，但是几乎不需要触摸这些部分，那么它们对系统的整体复杂性不会有太大影响。为了用粗略的数学方法来表征：

$$C = \sum_{p} c_p t_p$$

系统的总体复杂性（C）由每个部分的复杂性（cp）乘以开发人员在该部分上花费的时间（tp）加权。将复杂性隔离在一个永远不会被看到的地方几乎和完全消除复杂性一样好。

读者比作者更容易理解复杂性。如果您编写了一段代码，对您来说似乎很简单，但是其他人认为它很复杂，那么它就是复杂的。当您遇到这种情况时，有必要对其他开发人员进行调查，以找出为什么这段代码对他们而言似乎很复杂；从您的观点与他们的观点之间的脱节中可能可以学到一些有趣的教训。作为开发人员，您的工作不仅是创建您自己可以轻松使用的代码，而且还要创建其他人也可以轻松使用的代码。

2.2 复杂性的症状

复杂性通过以下段落中描述的三种一般方式表现出来。这些表现形式中的每一种都使执行开发任务变得更加困难。

变更放大：复杂性的第一个征兆是，看似简单的变更需要在许多不同地方进行代码修改。例如，考虑一个包含几个页面的网站，每个页面都显示一个带有背景色的横幅。在许多早期的网站中，颜色是在每个页面上明确指定的，如图 2.1（a）所示。为了更改此类网站的背景，开发人员可能必须手动修改每个现有页面；对于拥有数千个页面的大型网站而言，这几乎是不可能的。幸运的是，现代网站使用的方法类似于图 2.1（b），其中横幅颜色一次在中心位置指定，并且所有各个页面均引用该共享值。使用这种方法，可以通过一次修改来更改整个网站的标题颜色。

认知负荷：复杂性的第二个症状是认知负荷，这是指开发人员需要多少知识才能完成一项任务。较高的认知负荷意味着开发人员必须花更多的时间来学习所需的信息，并且由于错过了重要的东西而导致错误的风险也更大。例如，假设 C 语言中的一个函数分配了内存，返回了指向该内存的指针，并假定调用者将释放该内存。这增加了使用该功能的开发人员的认知负荷。如果开发人员无法释放内存，则会发生内存泄漏。如果可以对系统进行重组，以使调用者不必担心释放内存（分配内存的同一模块也负责释放内存），它将减少认知负荷。（认知负荷出现在很多方面，例如有很多方法的API、全局变量、不一致和模块间的依赖）

系统设计人员有时会假设可以通过代码行来衡量复杂性。他们认为，如果一个实现比另一个实现短，那么它必须更简单；如果只需要几行代码就可以进行更改，那么更改必须很容易。但是，这种观点忽略了与认知负荷相关的成本。我已经看到了只需要几行代码就能编写应用程序的框架，但是要弄清楚这些行是什么极其困难。**有时，需要更多代码行的方法实际上更简单，因为它减少了认知负荷。**

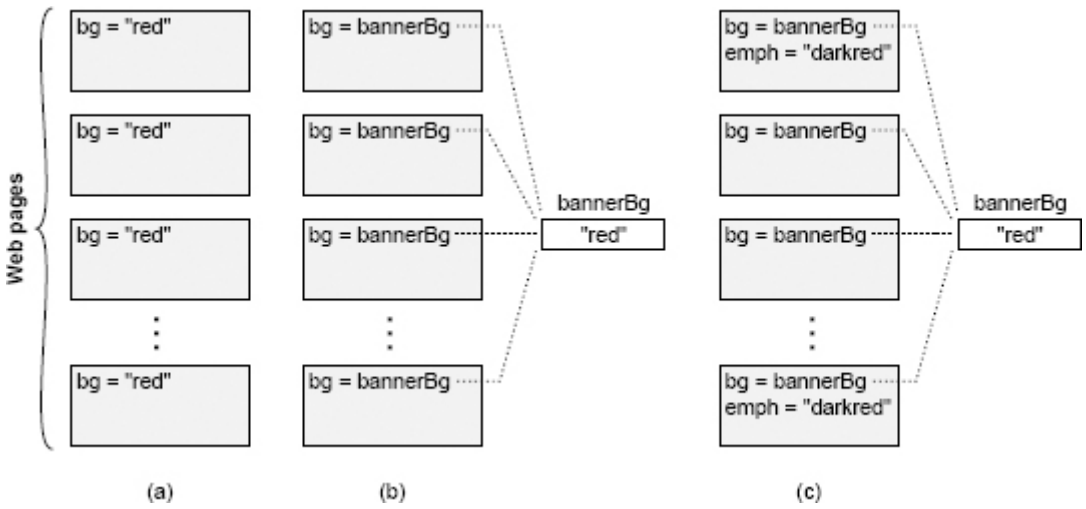

图 2.1：网站中的每个页面都显示一个彩色横幅。在（a）中，横幅的背景色在每个页面中都明确指定。在（b）中，共享变量保存背景色，并且每个页面都引用该变量。在（c）中，某些页面会显示其他用于强调的颜色，即横幅背景颜色的暗色；如果背景颜色改变，则强调颜色也必须改变。

未知的未知: 复杂性的第三个症状是，必须修改哪些代码才能完成任务，或者开发人员必须获得哪些信息才能成功地执行任务，这些都是不明显的。图 2.1(c)说明了这个问题。网站使用一个中心变量来确定横幅的背景颜色，所以它看起来很容易改变。但是，一些网页使用较暗的背景色来强调，并且在各个页面中明确指定了较暗的颜色。如果背景颜色改变，那么强调的颜色必须改变以匹配。不幸的是，开发人员不太可能意识到这一点，所以他们可能会更改中心变量 `bannerBg` 而不更新强调颜色。即使开发人员意识到这个问题，也不清楚哪些页面使用了强调色，因此开发人员可能必须搜索网站中的每个页面。

在复杂性的三种表现形式中，未知的未知是最糟糕的。未知的未知意味着你需要知道一些事情，但是你没有办法找到它是什么，甚至不知道是否存在问题。直到你的修改导致了代码缺陷之前，你都不会发现它。变更放大是令人恼火的，但是只要清楚哪些代码需要修改，一旦更改完成，系统就会工作。同样，高的认知负荷会增加变更的成本，但如果明确要阅读哪些信息，变更仍然可能是正确的。对于未知的未知，不清楚该做什么，或者提出的解决方案是否有效。唯一确定的方法是读取系统中的每一行代码，这对于任何大小的系统都是不可能的。这甚至可能还不够，因为更改还可能依赖于一个从未记录的细微设计决策。

良好设计的最重要目标之一就是使系统的更改能够被预见，这与高认知负荷和未知的未知相反。在这样的系统中，开发人员可以快速了解现有代码的工作方式以及进行更改所需了解的内容，并可以在不费力思考的情况下快速猜测要做什么，同时又可以确信该猜测是正确的。第 18 章 讨论了使代码更改所带来的影响更容易理解的技术。

2.3 复杂性的原因

既然您已经了解了复杂性在较高层级的症状以及为什么复杂性会使软件开发变得困难，那么下一步就是了解导致复杂性的原因，以便我们能设计系统来避免这些问题。复杂性是由两件事引起的：依赖性和模糊性。本节从较高层级讨论这些因素。随后的章节将讨论它们与较低层级的设计决策之间的关系。

就本书而言，当无法孤立地理解和修改给定的一段代码时，便存在依赖关系。该代码以某种方式与其他代码相关，如果更改了给定代码，则必须考虑和/或修改其他代码。在图 2.1（a）的网站示例中，背景色在所有页面之间创建了依赖关系。所有页面都必须具有相同的背景，因此，如果更改一页的背景，则必须更改所有背景。依赖关系的另一个示例发生在网络协议中。通常，协议的发送方和接收方有单独的代码，但是它们必须分别符合协议。更改发送方的代码几乎总是需要在接收方进行相应的更改，反之亦然。方法的签名创建了方法实现方和方法调用方之间的依赖关系：如果向方法添加了一个新参数，则必须修改调用该方法的代码以指定该参数。

依赖关系是软件的基本组成部分，不能完全消除。实际上，我们在软件设计过程中有意引入了依赖性。每次编写新类时，都会围绕该类的 API 创建依赖关系。但是，软件设计的目标之一是减少依赖关系的数量，并使依赖关系保持尽可能简单和明显。

考虑网站示例。在每个页面分别指定背景的旧网站中，所有网页都是相互依赖的。新的网站通过在中心位置指定背景色并提供一个 API，供各个页面在呈现它们时检索该颜色，从而解决了该问题。新的网站消除了页面之间的依赖关系，但是它围绕 API 创建了一个新的依赖关系以检索背景色。幸运的是，新的依赖性更加明显：很显然，每个单独的网页都取决于 `bannerBg` 颜色，并且开发人员可以通过搜索其名称轻松找到使用该变量的所有位置。此外，编译器还有助于管理 API 依赖性：如果共享变量的名称发生变化，任何仍使用旧名称的代码都将发生编译错误。新的网站用一种更简单、更明显的方式代替了一种不明显且难以管理的依赖性。

复杂性的第二个原因是模糊性。当重要的信息不明显时，就会产生模糊性。一个简单的例子是一个变量名，它是如此的通用，以至于它没有携带太多有用的信息(例如，时间)。或者，一个变量的文档可能没有指定它的单位，所以找到它的惟一方法是扫描代码，查找使用该变量的位置。模糊性常常与依赖项相关联，在这种情况下，依赖项的存在并不明显。例如，如果向系统添加了一个新的错误状态，可能需要向一个包含每个状态的字符串消息的表添加一个条目，但是对于查看状态声明的程序员来说，消息表的存在可能并不明显。不一致性也是造成模糊性的一个主要原因：如果同一个变量名用于两个不同的目的，那么开发人员就无法清楚地知道某个特定变量的目的是什么。

在许多情况下，模糊性来源于文档的不足。第 13 章讨论了这个主题。但是，模糊性也是设计问题。如果系统设计简洁明了，则所需的文档将更少。对大量文档的需求通常是表明设计不正确的危险信号。减少模糊性的最佳方法是简化系统设计。

依赖性和模糊性共同构成了第 2.2 节中描述的三种复杂性表现。依赖性导致变更放大和高认知负荷。模糊性会产生未知的未知，还会增加认知负荷。如果我们找到最小化依赖性和模糊性的设计技术，那么我们就可以降低软件的复杂性。

2.4 复杂性是增量产生的

复杂性不是由单个灾难性错误引起的；它堆积自许多小块。单个依赖项或模糊项本身不太可能显著影响软件系统的可维护性。之所以会出现复杂性，是因为随着时间的流逝，成千上万的小依赖项和模糊项逐渐形成。最终，这些小问题太多了，以至于对系统的每次更改都会受到其中几个问题的影响。

复杂性的增量本质使其难以控制。可以很容易地说服自己，当前更改所带来的一点点复杂性没什么大不了的。但是，如果每个开发人员对每种更改都采用这种方法，那么复杂性就会迅速累积。一旦积累了复杂性，就很难消除它，因为修复单个依赖项或模糊项本身不会产生很大的变化。为了减缓复杂性的增长，您必须采用第 3 章中讨论的“零容忍”理念。

2.5 结论

复杂性来自于依赖性和模糊性的积累。随着复杂性的增加，它会导致变更放大、高认知负荷和未知的未知。结果，需要更多的代码修改才能实现每个新功能。此外，开发人员花费更多时间获取足够的信息以安全地进行更改，在最坏的情况下，他们甚至找不到所需的所有信息。最重要的是，复杂性使得修改现有代码库变得困难且危险。