互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

第 2 章

两个系统的故事：现代软件神话

Pete Goodliffe

架构是一种很浪费空间的艺术。

—— Philip Johnson

软件系统就像一座由建筑和后面的路构成的城市 —— 由公路和旅馆构成的错综复杂的网络。在繁忙的城市里发生着许多事情，控制流不断产生，它们的生命在城市中交织在一起，然后死亡。丰富的数据积聚在一起、存储起来，然后销毁。有各式各样的建筑：有的高大美丽，有的低矮实用，还有的坍塌破损。随着数据围绕着它们流动，形成了交通堵塞和追尾、高峰时段和道路维护。软件之城的品质直接与其中包含多少城市规划有关。

某些软件系统很幸运，创建时由有经验的架构师进行了深思熟虑的设计，在构建时体现出了优雅和平衡，有很好的地图，便于导航。另一些软件系统就没有这么幸运，基本上是一些通过偶然聚集的代码渐渐形成的，交通基础设施不足，建筑单调而平凡，置身于其中时会完全迷失，找不着路。

你的代码愿意待在怎样的“城市”中？你愿意构建哪一种城市？

在本章中，我将讲述这样两个软件城市的故事。这是真实的故事，就像所有好的故事一样，这个故事最终是有教育意义的。人们说经验是伟大的老师，但最好是别人的经验，如果你能从这些项目的错误和成功之中学习，你（和你的软件）可能会避免很多的痛苦。

本章中的这两个系统特别有趣，因为它们有很大不同，尽管从表面上看非常相似：

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

- 它们具有相似的规模（大约500 000行代码）。
- 它们都是“嵌入式”消费音频设备。
- 每种软件的生态系统都是成熟的，已经经历了许多的产品版本。
- 两种解决方案都是基于Linux的。
- 编码采用C++语言。
- 它们都是由“有经验的”程序员开发的（在某些情况下，他们本应知道得更多）。
- 程序员本身就是架构师。

在这个故事中，人名都已改变，目的是保护那些无辜的人（和有罪的人）。

2.1 混乱大都市

> 你们修筑、修筑，预备道路，将绊脚石从我百姓的路中除掉。
>
> ——《以赛亚书》第57章14节

我们要看的第一个软件系统名为“混乱大都市”。它是我喜欢回顾的一个系统——既不是因为它很好，也不是因为它让参与开发的人感到舒服，而是因为当我第一次参与它的开发时，它教给了我有价值的软件开发经验。

我第一次接触“混乱大都市”，是在我加入了创建它的公司时。初看上去这是一份有前途的工作。我将加入一个团队，参与基于Linux的、“现代”的C++代码集开发，已有的代码集已经开发几年了。如果你像我一样拥有特殊的技术崇拜，就会觉得很兴奋。

工作起初并不顺利，但是你不能指望在加入一个新团队、面对新的代码集时会觉得很轻松。然而，日复一日（周复一周），情况却没有任何好转。这些代码要花极长的时间来学习，没有显而易见的进入系统中的路径。这是个警告信号。从微观的层面来说，也就是从每行程序、每个方法、每个组件来看，代码都是混乱而粗糙地垒在一起的。不存在一致性、不存在风格、也没有统一的概念能够将不同的部分组织在一起。这是另一个警告信号。系统中的控制流让人觉得不舒服，无法预测。这又是一个警告信号。系统中有太多的“坏味道”（Fowler 1999），整个代码集散发着腐烂的气味，是在大热天里散发着刺激性气体的一个垃圾堆。这是一个清晰的警告信号。数据很少放在使用它的地方。经常引入额外的巴罗克式缓存层，目的是试图让数据停留在更方便的地方。这又是一个警告信号。

当我试图在大脑中建立“大都市”的全图时，没有人能解释它的结构；没有人知道它的所有层、它的藤蔓，以及那些黑暗、隔离的角落。实际上，没有人知道它究竟有多少部分是真正能工作的（它实际上靠的是运气和英雄式的维护程序员）。人们知道他们面对

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

的那一小部分区域，但没人了解整个系统。很自然，没有任何文档。这也是一个警告信号。我需要的是一份地图。

这是一个悲伤的故事，我曾是其中的一部分：“大都市”是城市规划的恶梦。在你开始整治混乱之前，先要理解混乱，所以我们花了很大的精力和毅力，得到了一份“架构图”。我们标出了每一条公路、每一条主干道、每一条很少人了解的小路、所有灯光昏暗的辅路，并将它们画在一张主图上。我们第一次看到了这个软件的样子，并不令人赏心悦目。它是一些混乱的区块和线条。为了让它更好理解一些，我们用颜色标出了控制路径，突出了它们的类型。然后我们后退一步看着它。

它令人吃惊。它令人目眩神迷。它就像一只喝醉了的蜘蛛，跌进了一些海报颜料罐里，然后在一张纸上织成了一张彩色的网。它看起来就像图2-1那样（这是一个简化后的版本，细节已经修改了，为了保护那些有罪的人）。事情变得很清楚了。我们画出了伦敦地铁图。它甚至有环线。

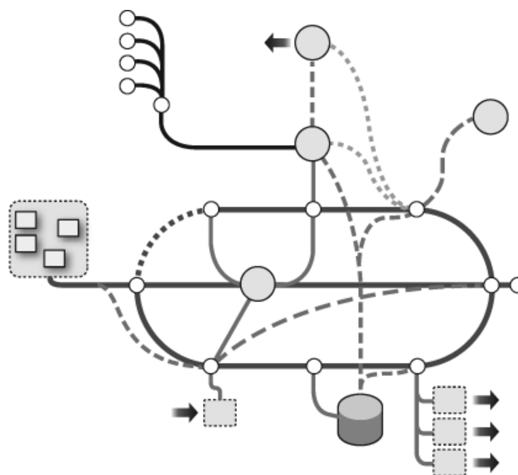

图2-1：“混乱大都市”的“架构”

这就是那种让跑遍各地的销售员恼怒的系统。实际上，它与伦敦地铁的相似性让人印象深刻：从系统的一端到另一端有很多条路线，哪条路最好通常是不明显的。地理位置很近的目的地常常很难到达，你希望能在两点之间再挖掘一条隧道。有时候，走出地铁换乘巴士实际上是更好的选择。或者干脆步行。

无论从哪个角度来看，这都不是一个“好的”架构。“大都市”的问题超出了设计的范畴，它涉及开发过程和企业文化。这些问题实际上导致了许多架构腐烂。代码经过几年的“有机”生长，没有人进行过任何架构设计，而且各个部分是随着时间推移，没有经过太多思考就栓在一起的。我们这么说真的算是客气的了。没有人停下来为代码建立一

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

个理智的结构。它增长、膨胀，成为绝对没有任何架构设计的系统的一个典型。代码集从来不会没有架构。这个系统只是拥有一个很糟糕的架构。

如果我们回顾创建“大都市”的公司的历史，它所处的状态是可以理解的（但是不可宽恕）：这是一个初创的公司，快速提供许多新版本的压力很大。延期是不可容忍的——这会带来财务灾难。软件工程师被迫尽其极限，快速交付。所以代码是以一系列疯狂冲刺的方式垒在一起的。

注意：不好的公司结构和不健康的开发过程将在糟糕的软件架构中得到反映。

2.1.1 后果

“大都市”缺少城市规划，这带来了许多后果，我们将在这里进行分析。这些后果的影响是很严重的，远远超出了你对不良设计的天真想象。地铁变成了云宵飞车，飞速地朝下猛冲。

不可理解

正如你已经看到的，“大都市”的架构以及缺乏强制的结构，导致了一个很难理解的软件系统，实际上几乎不可能修改。新加入项目的团队成员（譬如我）会被复杂性惊呆，不能够搞清楚状况。

坏的设计确实会招致在它上面叠加坏的设计（实际上它简直就是迫使你那样做），因为没有一种明智的方式可以扩展该设计。在所有能解决手上工作的方法之中，阻力最小的总会被采用，没有明显的办法来修复这些结构问题，所以只要能减少麻烦，就会扔进去新的功能。

注意：重要的是要保持软件设计的品质。坏的架构设计会招致更坏的架构设计。

缺乏内聚

系统的组件完全没有内聚性。每个组件本来都应该有一个定义良好的角色，但是它们却包含了一堆杂乱的、不一定相关的功能。这使我们很难确定组件存在的原因，也很难弄明白系统中已经实现了哪项具体的功能。

很自然，这让缺陷修复成为了一场噩梦，严重地影响了软件的品质和可靠性。

功能和数据都放在了系统中错误的地方。许多你认为是“核心服务”的部分却没有在系统的核心部分实现，而是由边远的模块来模拟实现（非常痛苦并且代价很大）。

进一步的软件历史考察揭示了原因：原来的团队中存在个人斗争，所以一些关键程序员开始创建他们自己的软件小帝国。他们把自己认为酷的功能放到他们的模块中，即使它不应该属于那里。为了做到这一点，他们于是又实现了更为巴罗克式的通信机制，把控

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

制连回到正确的地方。

注意：开发团队中健康的工作关系将直接有益于软件设计。不健康的关系和个性膨胀会导致不健康的软件。

内聚和耦合

软件设计的关键品质是内聚和耦合。这不是什么新奇的“面向对象”概念；自从20世纪70年代出现结构化设计开始，开发者对这一概念已经谈论了许多年。我们的目标是通过设计使系统的组件具备下列品质：

- 高内聚（Strong cohesion）

内聚是一个测量指标，说明相关的功能如何聚集在一起，模块内的各部分作为一个整体工作得如何。内聚性是将模块粘成一个整体的胶水。

弱内聚的模块是不良分解的信号。每个模块都必须具有清晰定义的角色，而不只是一堆不相关的功能。

- 低耦合（Low coupling）

耦合是模块之间独立性的测量指标——它们之间进出“电线”的数量。在最简单的设计中，模块几乎没有什么耦合，所以彼此间的依赖关系较少。显然，模块不能够完全解耦，否则它们将根本不能够一起工作！

模块之间的联系有多种方式，有的是直接的，有的是间接的。模块可以调用其他模块中的函数，或被其他模块所调用。它可能使用其他模块提供的Web服务或设施，可能使用其他模块的数据类型，或提供某些数据让其他模块使用（可能是变量或文件）。

好的软件设计会限制通信的线路，只提供那些绝对需要的。这种通信线路是确定架构的一部分因素。

不必要的耦合

“大都市”没有清晰的分层。模块之间的依赖关系不是单向的，耦合常常是双向的。组件A会到达组件B的内部，目的是完成它的一项任务。在其他的地方，组件B又通过硬编码调用了组件A。系统没有最底层，也没有控制中心。它是整体式的一大块软件。

这意味着系统的各部分之间耦合非常紧密，你想启动系统骨架就不得不创建所有的组件。单个组件的任何改变都会波及其他组件，需要修改许多依赖它的组件。孤立地看代码组件没有任何意义。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

这使得低层次的测试不能够进行。不仅是代码层次的测试不可能进行，而且组件层次的集成测试也不能够创建，因为每个组件都依赖于几乎所有其他组件。当然，在公司中，测试从来也不具有很高的优先级（我们根本没有时间来做这种测试），所以这“不成为问题”。不必说，这个软件不太可靠。

注意：好的设计考虑到内部组件连接的连接机制和连接数（以及连接性质）。系统的单个部分应该能够独立存在。紧耦合将导致不可测试的代码。

代码问题

不良的顶层设计所带来的问题也影响到了代码层面。问题会引起其他问题（参见Hunt和Davis[1999]中关于破窗理论的讨论）。因为没有通用的设计，也没有整体项目“风格”，所以也没有人关心共同的编码标准、使用共同的库，或采用共同的惯例。组件、类和文件都没有命名惯例。甚至都没有共同的构建系统。胶带、Shell脚本、Perl胶水与makefile和Visual Studio项目文件混在一起。编译这个怪物被视为一场复杂的成人仪式！

“大都市”最微妙而又最严重的问题是重复。由于没有清晰的设计，也不清楚功能应该处于的位置，所以轮子在整个代码集中不断重新发明。一些简单的东西，如通用算法和数据结构，在许多模块中重复出现，每种实现都带有自己的一些未知的缺陷和怪异的行为特征。更大范围的关注点，如外部通信和数据缓存，也实现了许多次。

更多的软件历史考察揭示了原因：“大都市”开始是从一系列独立的原型拼起来的，这些原型本该抛弃。“大都市”实际上是偶然形成的城市群。当代码组件缝合在一起时，组件之间匹配得不好。随着时间的推移，这种随意的缝合开始破裂，所以组件互相拉扯，导致代码集破碎，而不是和谐地协作。

注意：松弛而模糊的架构将导致每个代码组件编写得不好，并且相互之间匹配得不好。它也会导致重复的代码和工作。

代码以外的问题

“大都市”内部的问题已经超越了代码集，在公司中其他的地方导致了混乱。不仅开发团队中有问题，而且架构的腐烂也影响到了支持和使用该产品的人。

开发团队

项目的新成员（例如我）被复杂性惊呆了，不能够搞清楚状况。这很好地解释了为什么很少有新人能在公司里待下来——员工流失率非常高。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

那些留下来的人非常努力地工作，项目的压力非常大。规划新的功能会导致极大的恐惧。

缓慢的开发周期

由于维护“大都市”是一项恐怖的任务，所以即使是最简单的变更或“很小的”缺陷修复都不知道要花多少时间。管理软件开发周期非常难。客户只好等着实现重要的特征，管理层对开发团队不能满足业务目标感到越来越沮丧。

支持工程师

在支持这个极不寻常的产品时，产品支持工程师度过了可怕的时光，他们要设法弄明白很小的软件版本差异之间错综复杂的行为差异。

第三方支持

项目开发了一个外部支持协议，支持其他设备远程控制“大都市”。由于它只是软件内部结构上面薄薄的一层，所以它反映了“大都市”的架构，这意味着它也是巴罗克式的、难以理解的、容易偶尔出错的、不可能使用的。第三方工程师的生活也被“大都市”的可怕结构搞得一团糟。

公司内部政治

开发问题导致了公司内部不同“种族”的分裂。开发团队与营销销售团队之间关系紧张，每次新版本要推出时，制造部门总是要承受巨大的压力。经理们已经绝望了。

注意：不良架构的影响不仅限于代码。它会进一步影响到人、团队、过程和时间表。

清晰的需求

软件历史考察凸显了“混乱大都市”之所以混乱的一个重要原因：在项目开始之初，团队并不知道要构建的是什么。

本来的初创公司知道它要占领哪个市场，但不知道哪种产品能占领这个市场。所以他们两面下注，要求一个可以做许多事情的软件平台。噢，我们昨天就想得到它了。所以程序员们急急忙忙创建了一个毫无希望的总体基础设施，它具有做任何事情的潜力（但做得不好），而不是创建一个把一件事情做好的架构，并能够在将来进行扩展，做更多的事情。

注意：重要的是要在开始设计系统之前知道你打算设计什么。如果你不知道它是什么，也不知道它将做什么，暂时不要开始设计它。只设计你知道需要的东西。

在规划“大都市”的早期阶段，有太多的架构师。面对糊涂的需求，他们都拿着一块拼

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

不起来的拼图，试图独自工作。他们在工作时没有考虑到整个项目，所以当他们试图将这些拼图拼在一起时，就拼不起来了。没有时间进一步思考架构，软件设计的各个部分有一些重叠，于是开始了“大都市”的城市规划灾难。

2.1.2 现状

“大都市”的设计几乎完全是无可救药的——相信我，随着时间的推移，我们也尝试过修复它。修复工作需要返工、重构、修改代码结构中的问题，这些已经成为不可能的任务。重写也不是省事的方案，因为支持老的、巴罗克式的控制协议是需求的一部分。

你可以看到，“大都市”的“设计”产生的后果是残酷的，并且会无情地变得更糟。很难加入新的特性，所以人们只是忙着添加更多不完善的功能、救急补丁和编造的谎言。没有人在面对代码时感到愉快，项目正盘旋着向下栽。缺乏设计导致了不良的代码，从而又导致了不良的团队精神和不断变长的开发周期。这最终导致了公司严重的财务问题。

最后，管理层宣布“混乱大都市”已经不盈利了，它被抛弃了。对于任何组织机构来说，这都是勇敢的一步，特别是这个公司一直都眼高手低，同时又试图避免沉沦。带着团队从以前版本中得到的所有C++和Linux经验，他们在Windows上用C#重写了系统。猜猜看会怎么样。

2.1.3 来自“大都市”的名信片

那么我们学到了什么？不良的架构会产生深远的影响和严重的反弹。在“混乱大都市”中缺少预见性和架构设计，导致了下面的问题：

- 低品质的软件和漫长的版本发布周期。
- 系统没有弹性，不能够适应变更或添加新的功能。
- 无处不在的代码问题。
- 员工问题（压力大、士气低、跳槽等）。
- 大量混乱的公司内部政治。
- 公司不能成功。
- 许多痛苦和面对代码深夜加班。

2.2 设计之城

> 形式永远服从功能。
>
> ——Louis Henry Sullivan

“设计之城”软件项目表面上与“混乱大都市”非常相似。它也是用C++写的消费音频

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

产品，运行在Linux操作系统上。但是，它的构建方式有很大不同，所以内部结构也非常不同。

我从一开始就参加了“设计之城”项目。我们用有能力的开发者组成了一个全新的团队，从头开始构建这个产品。团队很小（开始有4名程序员），像“大都市”项目一样，团队的结构是扁平的。幸运的是，没有出现“大都市”项目中的个人对抗，在团队中也没有出现任何争权夺利的事。在此之前，团队成员之间并不非常熟悉，不知道我们在一起可以配合得多好，但我们对这个项目都很热心，喜欢这项挑战。

这样很好。

Linux和C++是项目早期的决定，这项决定确定了团队成员的组成。从一开始，项目就有清晰定义的目标：具体的首个产品和将来功能的路线图，代码集必须能够支持这些功能。这将是一个通用目标的代码集，可以适用于多种产品配置。

开发过程采用了极限编程（XP）（Beck和Andres 2004），很多人相信这种开发过程避开了设计：直接开始编码，不要想太远。实际上，一些旁观者对我们的选择感到震惊，并预言项目将以泪收场，就像“大都市”一样。但这是一种常见的误解。XP没有贬低设计，它贬低的是不必要的工作（即YAGNI原则，You Aren't Going To Need It）。但是，如果需要前端设计，XP就要求你进行设计。它也鼓励使用快速原型（所谓的“spike”），快速展现并验证设计的有效性。这些都非常有用，对最终的软件设计产生了极大的影响。

2.2.1 设计之城的第一步

在设计过程的早期，我们确定了主要的功能领域（这包括核心的音频通道、内容管理和用户控制/界面）。我们考虑了它们如何在系统中适配，推出了初步的架构，包括了实现性能需求所必需的核心线程模型。

系统中各独立部分的相对位置关系体现为传统的分层结构，图2-2展示了简化后的结果。请注意，这并不是庞大的前端设计。它是有意为之的“设计之城”的简单概念模型：图中只有一些大块，这是一个基本的系统设计，可以随着功能模块的添加而轻松地增长。虽然很基本，但这个初始架构为增长提供了坚实的基础。“大都市”没有总体规划，在“方便”的地方嫁接（或修补）功能。

我们在系统的核心上花了额外的设计时间：音频通道。它实际上是整个系统的一个内部子架构。为了确定它，我们考虑了穿过一系列组件的数据流，最后得到了一个“过滤器和管道”音频架构，如图2-3所示。根据产品的不同物理配置，它包含了这样一些管道。同样，开始时这些管道只是一个概念，即图中的一些方块。我们当时还没有决定如何将所有模块拼装在一起。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

图2-2 :“ 设计之城 ” 的初始架构

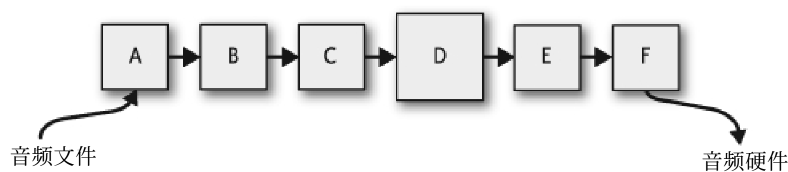

图2-3 :“ 设计之城 ” 的音频管道

我们在早期也选择了项目将采用的支持库（例如，可以从*http://www.boost.org*获得的Boost C++库和一组支持数据库的库）。关于一些基本关注点的决定是这时候做出的，目的是确保代码能够容易而一致地增长，这些决定包括：

- 顶层文件结构。
- 我们如何对事物命名。
- “ 内部 ” 展示的风格。
- 共用的编码惯例。
- 选择单元测试框架。
- 支持基础设施（例如版本控制、适合的构建系统和持续集成）。

这些 “ 细节 ” 完美的因素非常重要：它们与软件架构密切相当，影响到后来的许多决定。

2.2.2 故事展开

在团队完成了初始设计之后，“ 设计之城 ” 项目按照XP过程推进。设计和编码要么以结对的方式完成，要么经过仔细的复审，确保工作的正确性。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

随着时间的推移，设计和代码不断发展和成熟；随着“设计之城”的故事逐渐展开，产生了下面的结果。

定位功能

由于从一开始我们就有系统结构的清晰总体视图，所以新的功能单元可以一致地添加到代码集的正确功能区域。代码应该属于哪一块从来就不是问题。在扩展功能或修复问题时，我们总是很容易找到已有功能的实现代码。

现在，把新的代码放到“正确”的位置有时候比简单“嫁接”到方便而不妥的地方而更难一些。所以，架构规划的存在有时候让开发者的工作变得更难一些。这些额外工作的回报就是今后的生活要容易很多，当我们维护或扩展系统时，不愉快的事情会很少。

注意：架构有助于定位功能：添加功能、修改功能或修复缺陷。它为你提供了一个模板，让你将工作纳入到一张系统导航图中。

一致性

整个系统是一致的。各个层次的所有决定都是在整个设计的背景下做出的。开发者从一开始就有意为之，这样得到的所有代码都完全符合系统设计，并与编写的所有其他代码相匹配。

在项目的历史中，尽管有许多变更，涉及代码集的各处（从单行代码到系统结构），但这些变更都符合最初的设计模板。

注意：清晰的架构设计将导致一致的系统。所有决定都应该在架构设计的背景下做出。

顶层设计的好风格和优雅很自然会为较低的层带来好处。即使在最低层，代码也是统一而整洁的。清晰定义的软件设计确保了没有重复，熟悉的设计模式到处使用，熟悉的接口惯例普遍采用，没有特殊的对象生命周期或奇怪的资源管理问题。代码是在城市规划的背景之中写成的。

注意：清晰的架构有助于减少功能重复。

架构的增长

有一些全新的功能领域出现在了设计“全图”中，例如存储管理和外部控制功能。在“大都市”项目中，这是致命的一击，难度超乎想象。但在“设计之城”项目中，事情就不一样了。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

系统设计就像代码一样，被认为是可扩展、可重构的。开发团队的一项核心原则就是保持敏捷，没有什么是一成不变的，所以在需要时架构也可以修改。这促使我们让设计保持简单并易于修改。这样一来，代码可以快速地增长，同时又保持好的内部结构。添加新的功能块不是问题。

注意：软件架构不是一成不变的。需要时就改变它。要想做到可以修改，架构就必须保持简单。牺牲简单性的修改要抵制。

延迟设计决定

有一项XP原则确实提高了“设计之城”的架构品质，这就是YAGNI（如果你不是马上需要，就不要去做）。这促使我们在早期只设计了重要的部分，将所有余下的决定推迟，直到我们对实际的需求有了更清晰的理解并知道如何放到系统中最好时，再做出这些决定。这是一种非常强大的设计方法，在很大的程度上解放了我们的思想。

- 当你还不理解问题时就开始设计，这是一件糟糕的事。YAGNI迫使你等待，直到你知道真正的问题是什么，它应该怎样由设计来体现为止。这消除了猜测的工作，确保设计是正确的。
- 当你开始创建软件设计时就加入所有可能需要的东西（包括厨房水槽）是危险的。你的大部分设计工作会变成无用功，得到的只是额外的负担，你不得不在软件的整个变更生命周期中支持这些设计。它一开始就增加了成本，而且在项目的生命周期中不断增加成本。

注意：延迟设计决定，直到你必须做出这些决定为止。不要在你还不知道需求的时候就做出架构决定。不要猜测。

保持品质

从一开始，“设计之城”就准备好了一些品质控制过程：

- 结对编程。
- 对没有结对编程的工作进行代码/设计复审。
- 对每一段代码进行单元测试。

这些过程确保了系统中从未加入不正确的、不合适的变更。所有不符合软件设计的内容都被拒之门外。这可能听起来有点过于严厉，但这些是开发者们坚信的过程。

这种信念凸显了一个重要的态度：开发者们相信设计，认为设计对项目相当重要。他们拥有设计，对设计负责。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

注意：必须保持架构品质。只有当开发者们相信它并对它负责时，才能做到这一点。

管理技术债务

除了这些品质管理方法之外，“设计之城”的开发是相当注重实效的。随着最后期限的临近，一些不太重要的功能被砍掉，让产品能够准时推出。小的代码“瑕疵”或设计问题允许存在于代码集中，要么是为了让功能快一点实现，要么是为了在接近发布时避免高风险的改动。

但是，与“混乱大都市”项目不同的是，这些逃避职责的地方被标记为技术债务，并安排在后续的版本发布中修正。这些问题很清楚，开发者对它们不满意，直到将它们处理掉为止。同样，我们看到了开发者对设计的品质负责。

单元测试打造了设计

关于代码集的一项核心决定就是所有代码都要有单元测试（这也是在XP开发中强制要求做到的）。单元测试带来了许多好处，其中一点就是能够修改软件的一些部分，而不必担心在修改的过程中破坏其他的东西。我们对“设计之城”内部结构的某些部分进行了相当激进的返工，单元测试给了我们信心，让我们相信系统的其他部分没有被破坏。例如，线程模型和音频管道的内部连接接口都进行了彻底的改变。这是在子系统开发较晚的时候发生的严重设计变更，但与音频通道接口的其他代码仍然执行得很好。单元测试让我们能够改变设计。

随着“设计之城”的逐渐成熟，这种类型的“主要”设计变更越来越少了。在经过一些设计返工之后，情况稳定下来，此后只有一些不重要的设计变更。系统开发得很快：以迭代的方式进行，每一次迭代都改进了设计，直到它达到了相对稳定的状态。

注意：你的系统应该有一组不错的自动化测试，它们让你在进行根本的架构变更时风险最小。这为你提供了工作的空间。

单元测试的另一个主要好处在于，它们在很大程度上定型了代码设计：它们实际上迫使我们实现好的结构。每个小的代码组件都被定型成定义良好的实体，可以独立存在，因为它必须能够在单元测试中构造出来，不需要围绕它构造系统的其他部分。编写单元测试确保了每个代码模块的内聚性，也确保了与系统其他部分之间的松耦合。单元测试迫使我们仔细考虑每个单元的接口，确保该单元的API是有意义的，内部是一致的。

注意：对你的代码进行单元测试将带来更好的软件设计，所以设计时要考虑可测试性。

设计时间

“设计之城”成功的另一个因素是分配的开发时间段，它既不长也不短（就像金发歌蒂的粥，既不热也不冷，刚刚好）。项目需要一个有利的环境才能获得成功。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

如果时间太多，程序员常常会想创建他们的巨作（那种总是快要好了，但永远不会实现的东西）。有一点压力是好事，紧迫感有助于完成事情。但是，如果时间太少，就不可能得到任何有价值的设计，你只会得到半生不熟的解决方案，就像“大都市”那样。

注意：好的项目计划将带来优质的设计。分配足够的时间来创建架构杰作，它们不会立即出现。

与设计同行

尽管代码集很大，但它是一致而易于理解的。新的程序员可以比较容易地拿起代码并开始工作。不需要去理解不必要的复杂内部关系，也不需要面对奇怪的遗留代码。

由于代码中产生的问题比较少，工作起来有乐趣，所以团队人员的流失率很低。这是因为开发者们负责设计，并不断希望改进它。

看着开发团队动态地遵守架构设计是一件有趣的事情。“设计之城”的项目原则规定没有人“拥有”哪一部分设计，这意味着任何开发者都可以改动系统的所有地方。每个人都应该写出高品质的代码。“大都市”是许多不协作的、互相争斗的程序创造的一团混乱，而“设计之城”则是由密切合作的同事创建的一组干净、一致、密切合作的软件组件。在很大程度上，Conway法则（注）反过来也生效，团队的组织方式就像软件的组织方式一样。

注意：团队的组织方式必然对它产生的代码有影响。随着时间的推移，架构也会影响到团队协作的好坏。当团队瓦解时，代码的交互就很糟糕。当团队协作时，架构就集成得很好。

2.2.3 现状

在一段时间之后，“设计之城”的架构如图2-4所示。也就是说，它与最初的设计非常相似，同时也包含了一些值得注意的变更。此外，它还包含了大量的经验，证明这个设计是正确的。健康的开发过程，小的、更善于思考的开发团队，适当注意确保一致性，带来了极为简单、清晰、一致的设计。这种简单性为“设计之城”带来了好处，得到了可扩展的代码和快速开发的产品。

在编写本书时，“设计之城”项目已走过了3年。代码集仍在使用，而且扩展出了一些成功的产品。它还在开发、成长、扩展，还在每天发生变化。下一个月它的设计可能与这个很不同，但也可能没有不同。

我要澄清一点：这些代码并不完美。有些地方存在着技术争论，但是它们在整洁的背景下显得特别突出，会在将来得到解决。没有什么是一成不变的，由于适应性强的架构和灵活的代码结构，这些问题都可以解决。几乎所有东西都各就各位，因为架构很好。

注： Conway法则指出，代码结构符合团队的结构。简而言之，“如果你让4个小组开发一个编译器，就会得到一个4阶段编译器。”

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

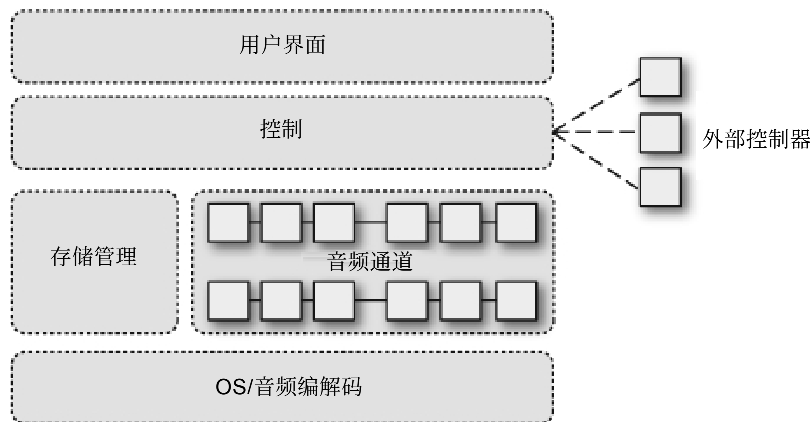

图2-4 :“ 设计之城 ” 的最终架构

2.3　说明什么问题

等那完全的来到，这有限的必归于无有了。

——《哥林多前书》第13章10节

这个关于两个软件系统的简单故事当然不是软件架构的全面介绍，但我已展示了架构如何对软件项目产生深远的影响。架构几乎会影响所有与之相关的人和事，它决定了代码集的健康，也决定了相关领域的健康。就像一个繁荣的城市会为当地带来成功和声望，好的软件架构将帮助项目获得发展，为依赖于它的人带来成功。

好的架构是很多因素的结果，包括以下方面（但不限于此）：

- 确实进行有意为之的前端设计。（许多项目甚至还没开始，就因为这一点而失败了。）
- 设计者的素质和经验。（以前犯过一些错误是有帮助的，这能在下一次为你指出正确方向！“ 大都市 ” 项目肯定教会了我一些东西。）
- 在开发过程中，保持清晰的设计观点。
- 授权团队负责软件的整体设计，而团队也承担起这一责任。
- 不要害怕改变设计：没有什么是一成不变的。
- 让合适的人加入到团队中，包括设计者、程序员和经理，确保开发团队的规模合适。确保他们具有健康的工作关系，因为这些关系将不可避免地影响代码的结构。
- 在合适的时候做出设计决定，当你知道所有必要信息时再做出决定。延迟那些暂时不能做出的决定。
- 好的项目管理，以及合适的最后期限。

互动出版网 China-Pub.com 专业图书网上第一专营店 www.china-pub.com 独家提供样章

2.4　轮到你了

绝不要失去神圣的好奇心。

—— 阿尔伯特 · 爱因斯坦

你正在读这本书是因为你对软件架构感兴趣，而且你对改进自己的软件感兴趣。所以这里就有一个极好的机会。对于你目前的软件经验，请考虑以下简单的问题：

1. 什么是你看到过的最好的系统架构？
    - 你怎么知道它是好的？
    - 这个架构在代码集之内和之外带来了什么结果？
    - 你从中学到了什么？
2. 什么是你看到过的最差的系统架构？
    - 你怎么知道它是差的？
    - 这个架构在代码集之内和之外带来了什么结果？
    - 你从中学到了什么？

参考文献

Beck, Kent, with Cynthia Andres. 2004. *Extreme Programming Explained*, Second Edition. Boston, MA: Addison-Wesley Professional.

Fowler, Martin. 1999. *Refactoring: Improving the Design of Existing Code*. Boston, MA: Addison-Wesley Professional.

Hunt, Andrew, and David Thomas. 1999. *The Pragmatic Programmer*. Boston, MA: Addison-Wesley Professional.