# 《恰如其分的软件架构》读书报告

《恰如其分的软件架构》一书，初读之下，颇有学术气息，内容略显晦涩。然而，书中提倡的风险驱动架构设计理念，实为值得借鉴。简而言之，这是一种“问题驱动法”。面对业务中的难点和问题，我们应当一一列举，进而选择适宜的技术与架构来逐一解决。

## 一、什么是软件架构？

软件架构就是系统设计，以及它对诸如性能、安全和可修改性等系统所产生的影响。  
软件架构抉择很重要，架构是系统骨骼，直接影响质量属性，并约束整个系统。

从某个角度来说，架构与功能可以互相组合，只是不同组合有不同表现效果而已。

利用模型和抽象概念去构建和解释系统架构。

软件设计可分为软件架构和详细设计。

软件架构的定义，来自卡内基.梅隆大学软件工程研究所（MEI）定义：

计算系统的软件架构是解释给系统所需的结构体集合，其中包括：软件元素、元素之间的相互关系，以及二者各自的属性。

该定义的重点：元素、元素之间的关系以及元素的属性。

架构是设计的宏观部分。

架构是必须在项目早期做出的一组设计决策。

软件架构，简而言之，就是系统设计及其对性能、安全和可修改性等属性的影响。架构的抉择至关重要，它如同系统的骨骼，直接影响着系统的质量属性，并约束着整个系统。架构与功能在某种程度上是正交的，不同的组合会产生不同的效果。软件设计可以分为软件架构和详细设计两个层面，架构关注的是设计的宏观部分，是项目早期必须做出的一系列设计决策。

## 二、软件架构为什么重要

软件架构重要性在于它会影响整个软件的系统。只有审慎的选择架构，才能降低风险，避免失败。

1. 架构扮演者系统骨架的角色
2. 架构影响质量属性
3. 架构与功能（基本上）正交的
4. 架构还是对系统的约束

系统架构不仅要支持所需的功能，同时还能促进或者抑制诸如安全或性能等系统质量。  
作者举例：  
人和马的身体骨架都能够运输苹果到市场的功能，但是运输效率和数量上相差甚远。

选择一种架构使得系统能够工作并不是难事，但满足质量属性方面，有的选择是事半功倍，有的选择则会事倍功半。

软件架构的重要性不言而喻，它关系到整个软件系统的走向。审慎选择架构能够降低风险，避免失败。架构不仅是系统的骨架，影响质量属性，还对系统构成约束。系统架构不仅要支持所需功能，还可能促进或抑制系统的质量，如安全或性能。

## 三、架构何时变得重要

**有以下几个方面：**

解空间小：问题复杂，很难设计出好的解决方案，那么架构就显得尤为重要。

高的失败风险：任何时候，失败的风险越高，就越需要保证正确架构

难以实现的质量属性：架构会影响质量属性的能力。开发一个邮件系统并不难，然而，一般要求高性能支持百万级用户，就变得异常困难

全新领域：面对全新领域，或者对于设计者而言是全新领域，就需要对架构给予更多关注。

产品线：一些产品线会共享某一通用架构。

**企业架构师和应用架构师：**

企业架构师：企业架构师是负责多个应用系统的开发者。他们并不会控制任何一个应用系统的功能。相反，他们会设计一个生态系统，位于其中的每个应用系统都为整个企业做出自己的贡献。

应用架构师：应用架构师是单个系统的开发者。他们设计系统功能，而非架构。当然，他们也可以专注架构设计或提升架构的设计运用到应用系统中。

**架构设计的方式有哪些？**

1、不做预先设计：开发者一上来就写代码。这也会发生设计，只不过是与编码起头并进  
2、用一些时间比例来做设计：比如，10%用于架构设计，50%编码，20%测试，10%集成  
3、详细的设计：一开始就做详细架构设计，并形成详细的架构文档  
4、随机应变方式：根据项目需求随机做出决定，是否需要架构设计？多少时间用来做架构设计？

## 四、风险驱动模型

风险驱动模型的核心在于三个步骤：识别风险并排定优先级、选择并运用技术降低风险、评估风险降低的程度。项目领域的典型风险可以通过分析来识别，并据此做出风险决策。

### 4.1 什么是风险驱动模型：

我的风险是什么？ 用于降低这些风险的技术是什么？风险是否化解，可以开始编码了吗？

**风险驱动模型归纳三个步骤：**  
1、识别风险，并排定优先级  
2、选择并运用一组技术降低风险  
3、评估风险降低的程度

### 4.2 项目领域典型的风险：



### 4.3 识别风险以及风险决策：

**识别风险**

经验丰富的开发者很容易识别风险，如果开发者缺乏经验，或者对该领域不熟，怎么办？最容易办法从需求开始，去寻找那些似乎难以实现的内容，不完整或容易引起误解的质量属性需求，是最为常见的风险。  
捕获风险并提供一份失败场景的优先级列表。  
必须认识到，及时竭尽全力，项目仍会面临一些未识别的风险，潜在的未知风险。

**典型风险**

在某一领域工作一段时间后，你就会注意到一些典型风险，他们对于该领域大多数项目而言都是很常见的。  
所以，我们需要找到它。

**决策风险优先级**由于风险有大有小，因此需要对他们进行优先级排序。

**技术：**  
一旦识别了风险，就运用期望的技术去降低风险。

比如，访问量大了之后，数据库可能抗不住，你会想到增加一层缓存来降低对数据库的直接访问。

**软件工程以及其他领域中，工程避险技术的若干示例：**

|  |  |
| --- | --- |
| 软件工程 | 其他工程 |
| 运用设计模式或架构模式 | 应力计算 |
| 领域建模 | 断点测试 |
| 吞吐量预估 | 热分析 |
| 安全性评估 | 可靠性测试 |
| 原型测试 | 原型测试 |

风险驱动模型是以分析为目的的相关技术，他们都是过程性的，而且是独立的问题域。这些技术包括：使用模型-层级图、组件模型以及部署模型；对性能、安全和可靠性进行分析的技术；利用各种架构风格，如client-server，pipe-and-filter（管道-过滤器）去实现某个紧迫的质量属性需求。

设计是一个神秘的过程，只有“大师”才能实现从推理问题到解决方案的跨越。我们要从“大师”中学得这门技艺。

做出技术合理决策公式：如果你面临<某种风险>，可以考虑使用<某种技术>降低它。

利用你掌握的若干知识，把风险与技术之间做一个若干映射。

任何特定技术都擅长降低某些风险，而对于其他风险却未必。

有些风险可以通过多种技术去缓解，而另外一些风险甚至需要发明一些技术才能解决。

这些技术往往是大公司发明的，因为他们的业务体量，需要新的技术才能解决。比如google发表的那3篇大数据论文。就改变了大数据处理领域。

放眼望去，总是很难判断运用哪些技术才是合适的。每种技术都有其价值，但却未必是项目最需要的价值。

这就是项目技术选型难点之一。技术选择多，会“眼花缭乱”。

## 五、技术选型指导原则

上面介绍了风险驱动模型，并主张根据所面对的风险去挑选技术。那如何才能做出合适的选型决策？

**重要原则：**  
1、首先，当面临一个要解决的问题，而在其他情况下又遇到需要证明的问题，技术决策应该与具体需求相匹配  
2、其次，某些问题可以通过类比模型解决，而其他问题，借助分析模型解决，此时需要分辨不同模型之间的差异  
3、再次，采用特定类型的模型，才能有效分析特定问题  
4、最后，某些技术之间存在着密切的关系，分析他们之间关系也很重要

要求解的问题和要证明的问题：要求解的问题，找到答案就可以了；要证明的问题，需要找到所有情况为真，这比求解问题难的多。

举例：“数据库能否保存多大一百个字符的名称？”它属于要求解的问题，为此问题编写测试用例就很容易验证出来。

“该系统是否一直符合该框架的应用程序编程接口？” 这就是需要证明的问题。尽管可以做多种测试，但，仍然会有某些情况被忽略了。

上面说的选型问题？ 做技术选型还是要考虑人和技术社区等因数，公司技术人是否会这种技术，该技术社区是否成熟，成熟的标准就是遇到了问题，能有人热心提供帮助。其次，就是该技术提供的解决方案是否与遇到的问题相匹配。最后，该技术后续升级发展，与公司业务发展周期是否相匹配。不要遇到那种纯kpi项目，突然就停止开发。但公司业务还在发展，遇到问题怎么办？这也是选型的风险。

## 六、如何把握设计与架构的度

架构设计是一种平衡，需要在当前利益间做出取舍。我们不需要识别出所有现在和未来的风险才开始架构设计，只需确保架构能够满足当前业务发展，并能应对未来1到2年的发展即可。

**6.1 演进式设计**

从历史过往来看，演进式架构饱受争议，因为局部而又不协调的设计决策会导致混乱，从而创造出一个大杂烩，既难维护，又难以进一步演进。然而，随着敏捷实践中的重构、测试驱动设计以及持续集成可以对付各种混乱问题。  
**重构**（对代码进改进）清除了不协调的局部设计，  
**测试驱动设计**确保对系统更改不会导致系统丢失或破坏现有功能，  
**持续集成**则为整个团队提供了统一代码库。

因为有上述种种实践，一些人认为可以不用做计划式设计。  
其中，重构是克服演进式设计中大杂烩问题的主力。但，重构缺陷是，它并没有为架构规模的转换提供指导。

**6.2 计划式设计**

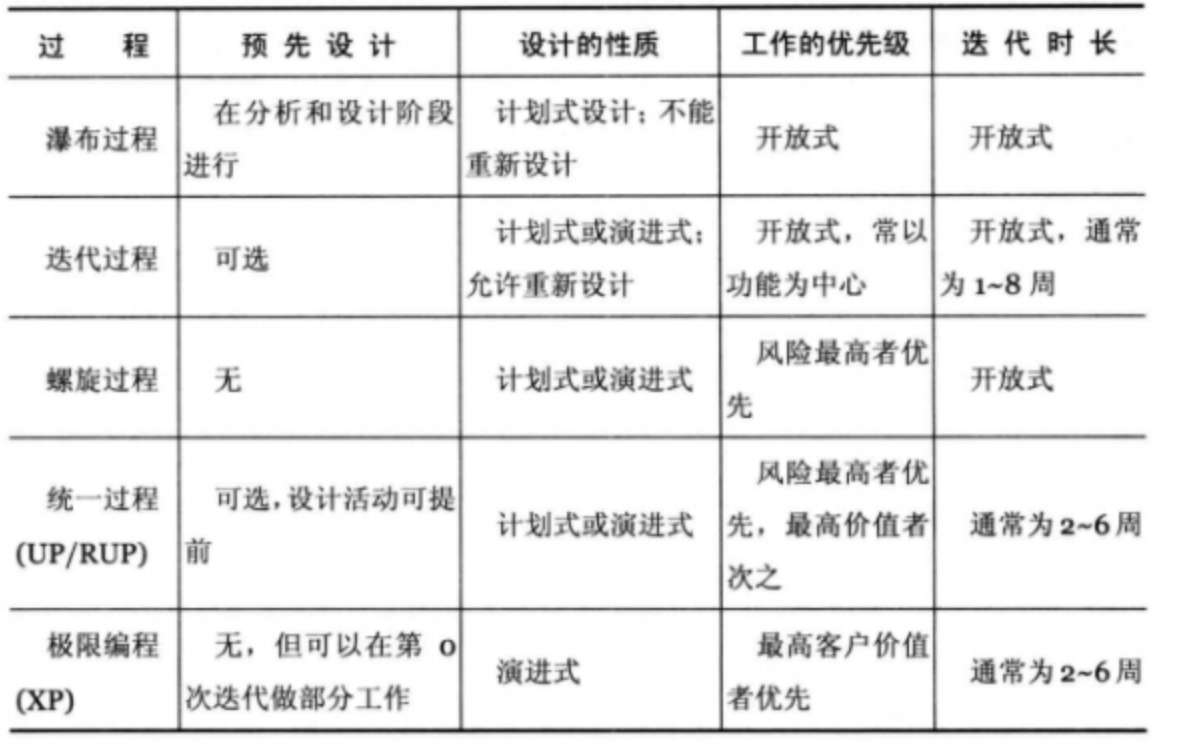
计划式设计则是在项目开始前就制定详细的计划和设计文档，以指导整个项目的进展。

**6.3 最小计划式设计**

最小计划式设计 （little design up front，Martin，2009）。这是一种介于演进式设计和计划式设计之间的一种设计方法。

在计划式设计与演进式设计之间取得平衡是可能的。  
一种方法是做一些初步的计划式设计，确保架构可以处理一些最大风险。在初始计划式设计完成后，未来的需求变化往往通过局部设计去处理，或采用演技式设计，前提是重构、测试驱动设计和持续集成等实践在项目中已经开展起来。

### **6.4理解过程变化**



### **6.5 风险驱动模型缺点**

它能帮助我们决定何时可以停止架构设计，以及应到我们开展各种适当的架构活动。  
但是，它并不擅长预测在设计上到底花多长的时间。

## 七、建模

工程师的目标是将现实世界的问题转换为现实世界的解决方案。若是简单问题，则无需抽象即可解决。然而，对于复杂的问题，则需要通过抽象来建模解决。  
现实世界的问题在抽象模型中体现出来，在建模领域中解决，再将该解决方案映射到现实世界的解决方案。架构模型也是如此。

### **7.1 规模与复杂度**

规模越大，复杂度越高，越需要抽象进行建模。

模型为解决问题提供了洞察力和解决手段。它是解决复杂问题的一种方式。

建立架构模型是一种理解和解决棘手问题的好方法，因为它可以去掉无关的细节，使得你能够将注意力放在主要部分以及相关关系上，做出预测，评估候选方案。

模型忽略了细节。

站在巨人肩膀上，以前的“巨人”不仅为我们提供了可见的编译器和数据库，还提供了一套抽象的编程思维理论。一部分抽象概念已经植根于编程语言中--函数、类、模块等。其余则包括组件、端口和连接器。

### **7.2 领域模型、设计模型和代码模型**

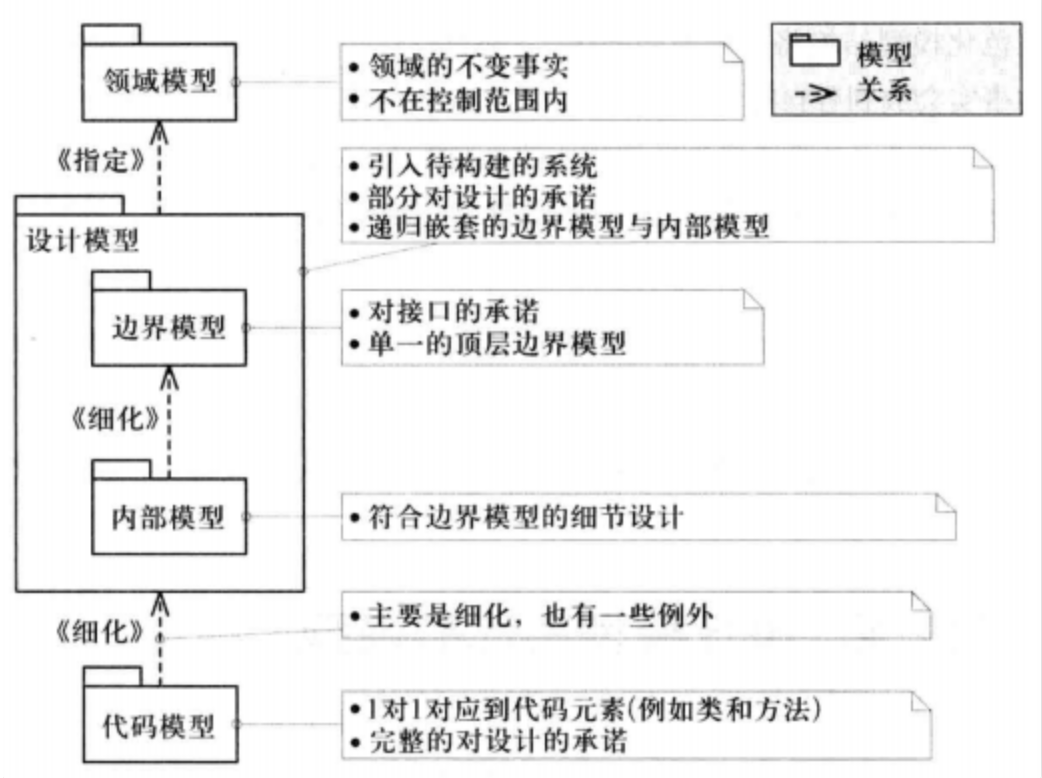
规范化模型结构顶部是抽象层次最高的模型（领域），底部模型则代表具体（代码）。指定关系和细化关系能确保模型一致性，又使得他们区分不同的抽象层次。

领域模型：领域模型表达了与系统相关的现实世界的不变事实。

设计模型：需要构建的系统不仅会显示在领域模型中，还会在设计模型中体现。

设计模式由递归嵌套的边界模型和内部模型组成。边界模型涉及公共接口，内部模型介绍了内部设计。

代码模型：系统的源代码实现。

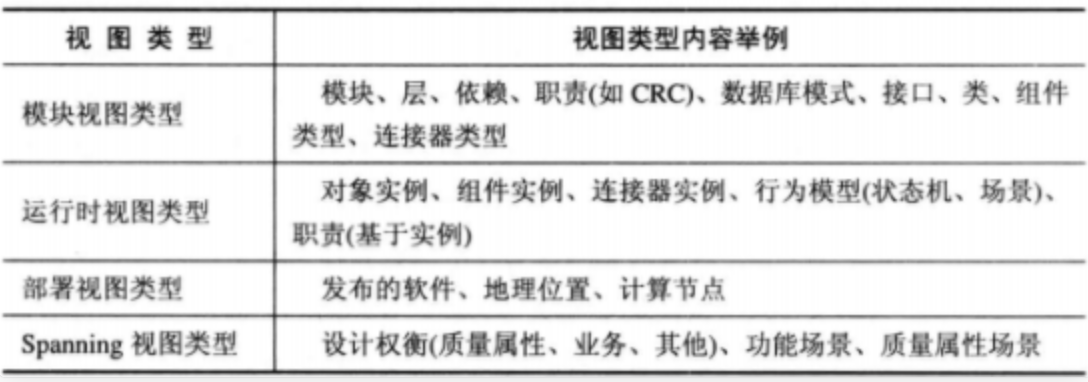


不同作者提出的模型如何与本书提到的业务模型、领域模型、设计模型（边界模型和内部模型）和代码模型的对应：



## 八、设计模型

视图类型以及视图类型内容举例：



视图类型和质量属性归纳，也就是设计时关注的一些指标，比如延迟，效率，可伸缩性，安全性，等等：

## 

## 九、模型关系

设计模型、实现模型和领域模型（分析模型）之间的关系是构建软件架构时需要考虑的重要方面。



## 十、架构风格

架构风格涉及元素、关系、约束以及指标，这些是评估和选择架构风格时的关键因素。

