

**《软件项目管理》课程读书笔记**



**《恰如其分的软件架构》读书笔记**

|  |  |
| --- | --- |
| 学 院： | 软件学院 |
| 专 业： | 软件工程 |
| 学生姓名： | 黄愫 |
| 学 号： | 2022141461212 |
| 指导老师： | 毌攀良 |
| 评阅意见： |  |
|  |  |
|  |  |
|  |  |

二零二四 年 十二 月 二十 日

目录

[一、 概述 3](#_Toc17626)

[二、 风险驱动的软件架构部分 3](#_Toc15176)

[2.1 本书对软件架构的定义 3](#_Toc24108)

[2.2 风险驱动模型 4](#_Toc4902)

[2.2.1识别风险 4](#_Toc19128)

[2.2.2将风险驱动模型应用到敏捷项目 5](#_Toc21042)

[2.2.3建模建议 5](#_Toc23594)

[三、 架构建模部分 6](#_Toc7231)

[3.提问在前建模在后 6](#_Toc17526)

[3.2软件架构的概念模型 6](#_Toc19228)

[3.2.1规范化模型结构 6](#_Toc20276)

[3.2.2领域模型 7](#_Toc30975)

[3.2.3代码模型 8](#_Toc7894)

1. 概述

这篇读书笔记主要提炼了《恰如其分的软件架构：风险驱动的设计方法》这本书的重点内容，并穿插了我的相关思考。

全书分为两大部分，第一大部分介绍了软件架构与风险驱动方法，第二大部分聚焦架构建模，帮读者建立软件架构的概念模型，讲解抽象的细节。

全书描述了一种恰如其分的软件架构设计方法。作者建议根据项目面临的风险来调整架构设计的成本，并从多个视角阐述了软件架构的建模过程和方法，包括用例模型、概念模型、域模型、设计模型和代码模型等。本书不仅介绍方法，而且还对方法和概念进行了归类和阐述，将软件架构设计融入开发实践中，与 敏捷开发方法有机地结合在一起。根据作者的前言，本书在软件架构的介绍上与其他书籍的不同主要体现在以下几个方面：风险驱动的架构设计、促进架构设计的民主化、积累陈述性知识、强调工程实践、提供实践指导。

1. 风险驱动的软件架构部分

## **2.1 本书对软件架构的定义**

本书对软件架构的定义沿用了卡内基·梅隆大学软件工程研究所(SEI)(Clementsetal.，2010):

计算系统的软件架构是解释该系统所需的结构体的集合，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。

作者这样划分架构与详细设计之间的界线：“架构是设计的宏观部分，例如模块及模块之间的连接方式，至于详细设计，则涵盖了设计的其他方面。”但同时软件架构不仅只包含系统的宏观部分，并引出概念：意图，即当无法区分架构与设计，或者架构师的高层意图传达不明晰时，从诸多细节中分辨出属于架构的部分。

书中的核心理念是“恰如其分”，即在软件架构设计中，我们应当根据项目的实际需求和潜在风险来决定架构的复杂度。这种思想挑战了传统的“越大越好”的架构设计观念，强调了在满足需求的同时，尽可能地减少不必要的复杂性，以降低项目风险。

## **2.2 风险驱动模型**

这章的本质思想在于:你设计软件架构所付出的精力应与你在项目中面对的风险成正比。

风险驱动模型可以指导开发者运用最小的架构技术集合去降低最紧迫的风险，以求事半功倍。

风险驱动模型归纳为三个步骤:

(1)识别风险，并排定优先级;

(2)选择并运用一组技术;

(3)评估风险降低的程度。

本章特别强调了风险识别的重要性。在实际工作中，往往容易忽视这一点，导致在项目后期不得不面对一系列预料之外的问题。通过提前识别风险，我们可以制定出更加合理的架构策略，从而减少项目失败的可能性。

2.2.1识别风险

从需求开始，寻找那些似乎难以实现的内容。可以召开质量属性研讨会，分发基于分类的调查问卷(taxonomy-basedquestionnaire，Carr etal.,1993)，或者采用其他相似方法，从而捕获风险并提供一份失败场景优先级列表。

典型风险 典型风险可能已被编制成一些检查表，用于描述确定存在的问题域，或许就产生于架构评审。

优先级分为两个维度:一个维度是对利益相关者而言的优先级，另一维度是由开发者察觉到的难度。必须认识到，利益相关者无法轻易评估某些技术风险，例如，平台的选择。



不同项目领域典型风险列表

2.2.2将风险驱动模型应用到敏捷项目

以下是对于如何将风险驱动模型应用于敏捷项目的总结（注：本章节并没有给读者可使用的技术清单）：

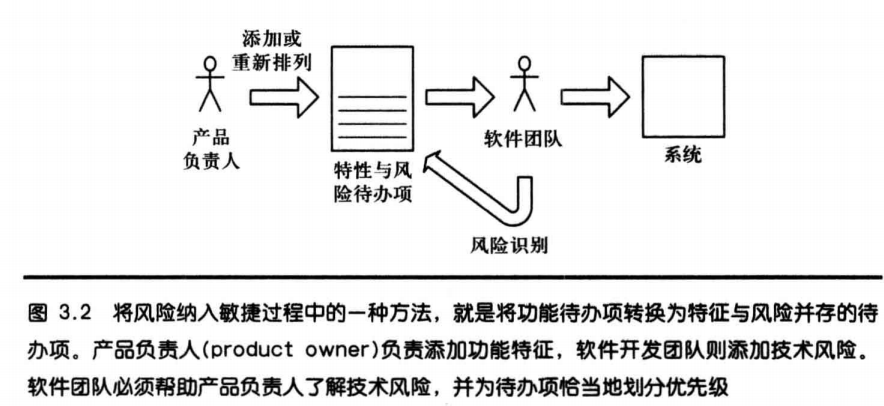
（1）风险驱动模型的应用：在敏捷项目中，尤其是采用两周迭代的开发模式时，需将软件架构风险融入到迭代过程中。这包括识别风险、优先级排序、缓解和评估风险。主要挑战在于如何处理初始工程风险以及如何将后续发现的风险纳入待办工作列表。

（2）识别和降低风险：项目开始时，可以识别出一些风险，如架构风格、框架和商业组件的选择。在第0次迭代中，可以通过建立开发环境和召开会议来降低这些风险，例如进行原型测试以评估关键组件的性能。

（3）风险待办项：在每次迭代结束时，需要评估活动是否有效降低了风险。未能降低的风险应记录为风险待办项，形成可测试的项目。小风险可以在迭代中解决，而较大风险则需进行规划，确保它们被纳入风险待办项中。

（4）风险与特征的优先级排序：敏捷项目通常将角色分为产品负责人和开发者。将特征与风险放入同一待办项中可以提高管理效率，但也增加了复杂性。产品负责人需同时为特征和架构风险划分优先级，以确保安全性等关键特征得到及时处理。

（5）处理架构风险的策略：敏捷过程可以通过以下方式处理架构风险：在第0次迭代中处理已知风险；在迭代中及时解决小风险；将大风险与功能特征同等对待，纳入待办项中。



2.2.3建模建议

避免预先大量设计

避免自顶向下设计：高层设计并不适合低层的模式。同样，为了重用而去挖掘现有的COTS组件和模块会非常困难，只有到了最后才能发现它们。

自顶向下的设计可以通过Conway法则来固定组织结构。由于团队结构很难改变，一开始的分解就可能成为最终的分解。

1. 架构建模部分

本书第2部分将介绍一组适合软件系统架构建模的抽象。

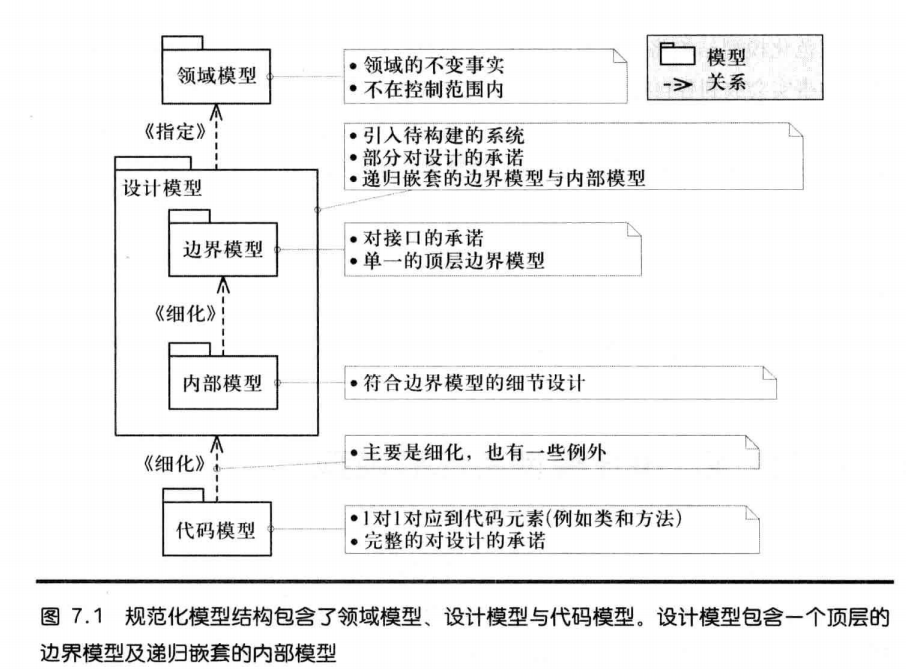
## **3.提问在前建模在后**

在建立模型之前，要明确自己需要模型解决哪些问题。如此才能更容易选择它的抽象层次，以及要包含的细节。

## **3.2软件架构的概念模型**

3.2.1规范化模型结构

规范化模型结构将不同类型的事实分配到相应的模型中，包括领域模型、设计模型和代码模型。通过将不同类型的事实分开处理，规范化模型结构能够缩小每个问题的规模。这意味着在分析领域问题时，开发者不会被代码细节所干扰，从而使得分析过程更加清晰和高效。在深入理解领域模型、设计模型和代码模型的定义和功能后，可以进一步探讨它们之间的关系。这种关系有助于在软件开发过程中更好地管理和协调不同层面的信息。其中领域模型表达了与系统相关的现实世界的不变事实。设计模型主要是在设计者控制下。 代码模型既是系统的源代码实现，又相当于一个模型。



3.2.2领域模型

领域模型专注于领域知识，帮助团队建立统一的词汇和共同理解，从而减少设计和编码中的问题。领域模型是对现实领域的简化，需明确包含和排除的内容，决定模型的广度和深度，以便有效回答领域中的关键问题，如联系网络中的联系数量或员工的工作经历。最终，模型应能解决与项目风险相关的问题，确保团队在开发过程中对领域的理解一致，以避免误解导致的失败。

本段内容探讨了领域模型中类型定义的重要性及其与信息模型的关系，主要包括以下几个要点：

类型定义的价值：领域模型中最基本且有价值的部分是类型列表和定义，这些类型描述了招聘广告和业务网络中的概念及其关系。清晰的类型定义有助于澄清领域概念，即使对于领域专家来说，这也是一项挑战。

信息模型的图形化：信息模型可以通过图形化方式展示，清晰表达类型之间的关联。图形化模型中包含的不变量（如约束）可以通过多重性和文字描述来表达，确保模型的准确性和一致性。

导航与关联：模型中的关联允许通过导航来遍历类型之间的关系。例如，个人类型与联系类型之间的关联可以通过角色来描述，确保在模型中遵循不变量的定义。

快照与实例图：快照展示了领域模型在某一时刻的实例状态，而信息模型则表达了所有可能的快照。通过绘制快照，可以识别模型中的潜在错误，并确保不变量的有效性。

功能场景的作用：功能场景描述了一系列事件如何导致领域模型状态的变化。每个场景都有初始状态，并通过场景中的每一步改变领域模型的状态。关注场景中的变化有助于确保模型的准确性和一致性。

真实世界的映射：领域模型中的实例（如Owen和Bradley）代表真实的人和关系，强调了模型与现实世界之间的联系。

总体而言，领域模型通过清晰的类型定义、信息模型的图形化表示、快照的使用以及功能场景的描述，帮助团队更好地理解和管理复杂的领域问题。

3.2.3代码模型

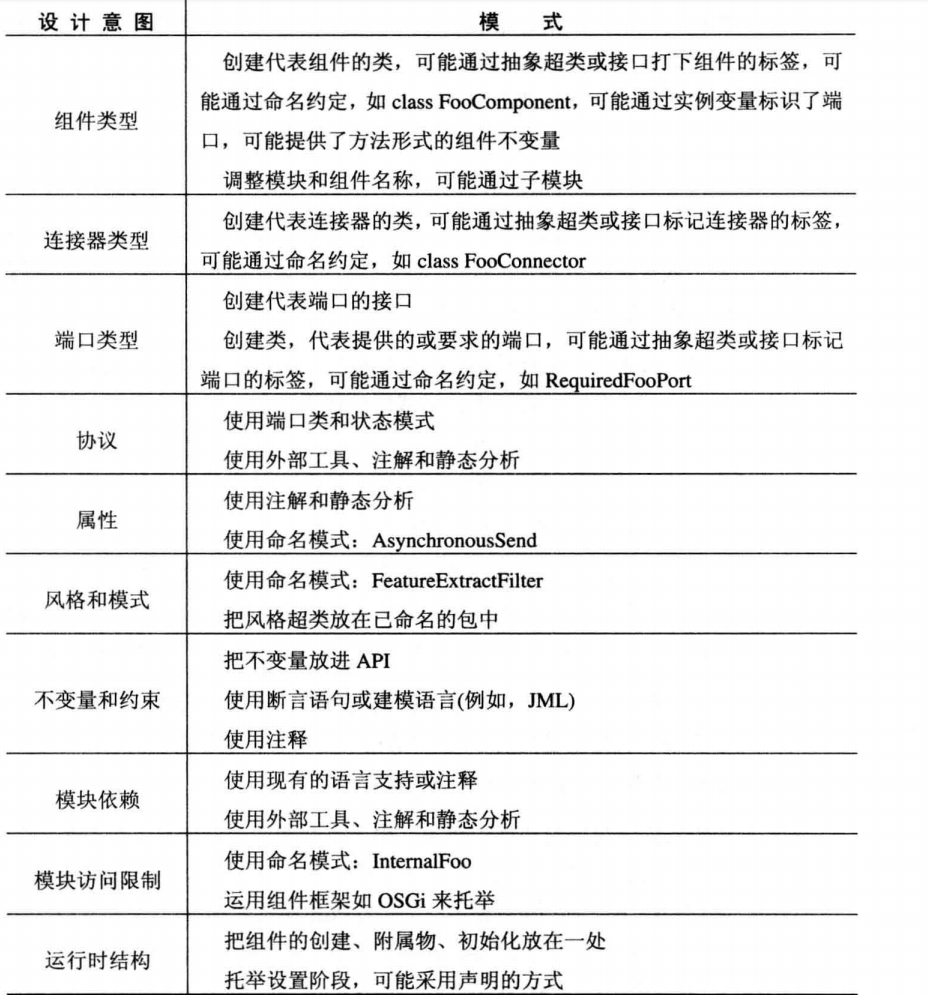
首先提出在代码上表示设计意图的概念。

总体而言，按合约设计通过明确的条件和目标简化了代码的理解，而软机制和硬机制则提供了不同层次的代码可读性和可验证性，帮助开发人员更好地维护和理解代码。

按合约设计：由Bertrand Meyer推广的按合约设计概念强调在方法中定义前置条件和后置条件，使得方法和对象可以被视为黑盒，客户无需关注内部实现。遵循这一设计原则的代码通常具有清晰的目标和明确的方法名称，避免模糊不清的命名（如“doSomeStuf”），从而降低了代码的复杂性。

软机制：软机制依赖于人为解释，例如类和方法的命名以及代码注释。如果命名不清晰或使用不熟悉的语言，阅读代码的难度会增加，类似于反编译的代码。

硬机制：硬机制是指可以通过机器进行检查的机制，能够分析代码的正确性和自洽性。虽然没有绝对的正确或错误，但可以检查源代码是否遵循设计意图，例如，是否允许X和Y之间的通信。



在代码中表达设计意图的模式一览

以下通过一个处理电子邮件的系统示例，探讨了如何在代码中使架构模型可见，主要包括七个要点：

电子邮件处理系统概述：该系统的功能是读取电子邮件并自动回复，如果无法理解请求，则交由人工处理。此类系统特别适用于处理大量重复邮件的场景，如航运跟踪查询。

处理阶段：电子邮件处理分为几个阶段：

清理阶段：移除HTML和其他标记，生成纯文本消息。

标记阶段：对消息文本进行标记，识别主题、发送者、段落、句子、单词、账号、姓名和跟踪单号。

特性分析：多个特性分析器对消息进行密集计算，提取特征。

分类阶段：分析器的结果汇聚后送入分类器，分类器判断是否理解消息。

架构模型的可视化：系统可以通过流程图或面向对象的编码风格实现，展示架构的明显特征。通过包结构的组织，可以清晰地显示模块结果，使组件和连接器类型可视化，帮助理解系统的运行时结构。

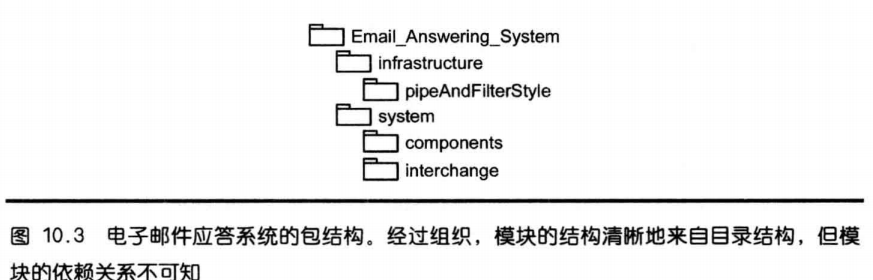
包结构的组织：源代码的模块视图可以通过包和模块的结构反映架构信息。电子邮件系统的包结构展示了与管道-过滤器风格相关的共享基础设施及系统特定部分。良好的包组织使得查找组件和数据流动变得容易。

包结构的局限性：尽管包的组织有助于展现架构，但也存在局限性：

组件代码可能无法集中在一个包中，导致依赖于外部库（如Java标准库）。

包的层次结构无法显示依赖关系，开发人员需要阅读源文件才能找到依赖。

跨包的约束无法表达，可能导致意外的依赖关系增加。



以下代码段--Filter类的源代码为例。因为出现了Filter类，读到这段代码的人就会知道你正在使用管道-过滤器风格。另一种做法是简单地把“flter”加到其他类的名称中，例如，TaggimgFilter，但是，显式的Filter类还有其他的好处。现在大多数集成开发环境(IDES)都可以显示类的层次结构，使用显式的Filter类，就可以显示 Filter 类的所有子类，这样，一眼就可以看到所有已定义的过滤器了。Filter类作为Component类的子类，不仅提供了具体的实现，还通过标准化并发处理和增量式输入处理增强了系统的可维护性。利用集成开发环境（IDE），开发者可以轻松查看类的层次结构，识别所有过滤器及其子类。尽管在代码中强制增量式处理和通过管道通信的约束较难表达，但可以通过注释进行描述。此外，Java的单继承限制在某些系统中可能会影响Filter类的使用。总体而言，Filter类通过明确的结构和功能提升了代码的可读性和架构的清晰度。

