### 第一部分：风险驱动的软件架构

#### 软件架构的核心要点

**定义与内涵**：软件架构是系统设计及其对质量属性影响的体现，涵盖软件元素、关系和属性，设计决策影响系统优劣，架构与详细设计界限模糊，关注宏观部分，如模块连接方式，同时某些低层设计细节也可能属于架构范畴，其核心是把握设计意图，判断细节是否影响整体质量。

**重要性体现**：架构是系统骨架，影响质量属性，虽与功能基本正交，但二者组合效果各异，好的架构能降低风险、避免失败，满足功能同时促进质量属性实现，约束系统以体现设计意图，促进概念完整性、降低复杂度、帮助理解运行时行为。

**关键时机判断**：在系统规模大、复杂度高、解空间小、失败风险高、质量属性难实现、涉及全新领域或产品线架构等情况下，架构决策至关重要，架构风险与系统风险成正比，应根据风险程度决定架构投入。

#### 风险驱动模型的运用

**模型的本质与步骤**：风险驱动模型以风险为核心，指导开发者选择最小架构技术集合降低紧迫风险，包括识别风险并排定优先级、选择运用技术、评估风险降低程度三个步骤，通过合理选择技术，避免过度或不足设计，提高设计效率。

**风险的识别与分类**：风险定义为觉察到的失败概率乘以影响，包括工程风险（如系统分析、设计、实现相关）和项目管理风险（如进度、团队等），可通过多种方式识别，如从需求找难以实现内容、关注质量属性需求、参考典型风险检查表等，同时要注意不同领域有典型风险，且需根据利益相关者和开发者感知对风险分类及排优先级。

**技术的选择与应用**：技术选择应多样化，根据风险类型选择合适技术，不同技术擅长降低不同风险，应挑选能降低优先级高风险的最佳技术，避免浪费时间和资源，同时考虑技术间关系，某些技术专为特定风险而生，选择时要确保技术与风险匹配，以有效降低风险。

#### 不同设计方式的选择

**架构无关设计**：开发者常忽视架构，可能照搬前项目架构、用推定架构或企业标准，虽能构建系统，但架构非审慎挑选，可能导致系统问题，如复杂度增加、分析困难等，不过在低风险项目或采用推定架构时可能成功。

**专注架构设计**：开发者了解并审慎挑选架构，以满足软件目标，包括功能和质量属性，涉及架构层面抽象概念和视图，需留意影响架构决策的需求，关注架构对需求实现的影响，可与任何软件开发过程兼容。

**提升架构设计**：以保证系统目标或属性为目的设计架构，将目标提升至架构中，使架构承担部分系统负荷，如应用服务器提升并发性、伸缩性和持久性，提升架构虽有约束，但可让开发者专注功能，适用于特定需求场景，需权衡利弊。

#### 计划式与演进式设计的权衡

**计划式设计**：在项目构建前详细制订计划，如桥梁设计施工，对大型复杂项目或多团队并行开发项目，有助于确保全局属性、协调团队，但可能因决策过早而犯错，且需在编码前完善设计，常难以完全做到。

**演进式设计**：系统设计随实现增长，如敏捷实践中的重构、测试驱动设计和持续集成可应对局部不协调设计问题，但重构在架构规模转换上有局限，遗留代码缺乏测试用例也影响重构，适用于需求变化快的项目。

**最小计划式设计**：介于两者之间，平衡二者优缺点，先做部分初始计划式设计处理最大风险，后续需求变化通过局部设计或演进式设计处理，开发者可根据项目需求和自身能力在不同设计风格间选择。

#### 风险驱动模型在实例中的应用

**家庭媒体播放器案例分析**：以家庭媒体播放器开发为例，展示风险驱动方法的应用。

**团队沟通风险应对**：通过创建模块模型、描述质量属性与设计决策、绘制运行时模型等方式，降低新增开发者对系统设计和架构理解不足导致的风险，确保团队高效协作，防止架构侵蚀。

**COTS 组件集成风险处理**：针对新组件集成，研究组件特性、创建边界模型、设计集成方案，通过功能场景和原型设计验证，解决集成、可靠性、屏幕显示和延迟性等风险，确保系统能在多平台运行且性能良好。

**元数据一致性风险解决**：采用领域建模，分析现有模型与互联网模型差异，设计新模型，提高元数据表现力，降低第三方插件开发风险，确保系统扩展性。

**案例总结与启示**：该案例体现了风险驱动方法在软件架构中的应用，通过识别风险、选择技术降低风险、评估剩余风险，实现恰如其分的架构设计，避免过度或不足设计，同时也展示了架构建模与敏捷过程的匹配性，为软件架构设计提供了实践指导。

### 第二部分：架构建模

#### 工程师使用模型的意义与方法

**模型解决问题的原理**：工程师使用模型解决复杂问题，将现实世界问题映射到抽象模型，在模型内解决后再映射回现实世界，抽象模型提供洞察力和解决手段，如代数模型解决列车相遇问题，架构模型帮助分析系统质量，去除无关细节，聚焦关键部分及其关系，作出预测和评估。

**模型的抽象与细节处理**：面对大型复杂问题，抽象至关重要，合适的抽象能让程序更易理解，模型应忽略无关细节，避免混乱，影响分析，但简化模型可能导致错误或非最优答案，需在完整与可用模型间权衡，确保模型包含正确细节，提高分析能力。

**模型与推理能力提升**：建模技能包括理解模型、编写模型和增强推理三个层次，软件开发者应使用模型增强推理能力，通过创建模型减轻记忆负担，检测错误，预测系统行为，协调细节，避免犯错，如房屋设计者利用模型设计复杂房屋，软件设计师也应如此。

#### 软件架构概念模型的构建

**规范化模型结构**：包括领域模型、设计模型和代码模型，从抽象到具体排列，通过指定和细化关系确保一致性并区分抽象层次，视图用于展示模型细节子集，确保各视图与主模型一致，模型范围广泛，需筛选细节便于分析，如 Yinzer 系统案例所示。

**领域模型**：表达领域不变事实，与设计实现无关，可帮助理解领域、与主题专家交流、避免设计误解，应关注与架构相关问题，避免分析瘫痪，通过信息模型、快照、导航、不变量和场景等描述，如手机联系人列表和用户授权案例，体现其在解决实际问题中的作用。

**设计模型**：是对软件系统设计的建模，包含所有设计细节，通过视图、封装和嵌套展示相关细节，与领域模型和代码模型有关联，边界模型展现系统外观和接口承诺，内部模型显示边界模型未考虑的细节，二者通过细化关系关联，如 Yinzer 系统设计展示了其在系统设计中的应用。

#### 设计模型的深入解析

**设计模型概述**：设计模型是包含所有设计细节的主模型，实际中通过视图、封装和嵌套构建部分细节模型并关联到主模型，与领域模型是 “指向” 关系，与代码模型关联复杂类似细化关系，使用视图、封装和嵌套可高效思考设计。

**边界模型与内部模型**：边界模型展现系统外观、行为、数据交互和质量属性，是接口承诺，有顶层边界模型描述系统与领域交互，内部模型显示边界模型未涉及的细节，二者描述相同内容但内部模型更详细，包含多种元素且需保持一致性，如 Yinzer 系统的边界和内部模型展示了系统不同层面的设计。

**质量属性与设计权衡**：质量属性描述系统外部特性，架构专家更关注，其与功能性正交且通常自然产生，需设定优先级，不同系统因需求不同而有不同优先级，设计中要在质量属性间权衡，如 Yinzer 系统设计考虑了伸缩性、可修改性、安全性和可用性等属性及其权衡。

#### Yinzer 系统设计实例剖析

**用例和功能场景**：用例图展示系统功能及参与者交互，功能场景描述系统特定行为路径，二者结合可明确系统功能及用例调用次序，用例图简洁概览，功能场景详细描述步骤，如 Yinzer 系统用例图和功能场景展示了用户与系统的交互及系统功能实现过程。

**系统上下文与组件**：系统上下文图展示系统与外部系统交互及连接器，明确通信通道和端口，鼓励详细描述交互，组件是系统主要计算和存储元素，通过端口和连接器通信，组件实例图应完整显示通信路径，组件类型和实例关系与类和对象类似，如 Yinzer 系统上下文图和组件实例体现了系统运行时连接和组件情况。

**端口、连接器与设计决策**：端口用于系统通信，有提供和要求服务之分，连接器是组件交互通道，系统上下文图将用例细化到端口，设计决策影响系统设计，重要决策应文档化，如 Yinzer 系统端口和连接器展示了系统通信方式，其设计决策体现了架构选择。

**模块与部署**：模块组织源代码，与架构抽象结合，端口可视为模块，模块间有依赖关系，部署视图展示系统在硬件上的部署，包括组件实例、环境元素和通信通道，功能场景可描述部署相关行为，如 Yinzer 系统模块和部署视图展示了系统结构和部署情况。

**质量属性场景与架构驱动**：通过对质量属性排优先级和编写质量属性场景来表达质量属性需求，场景包含源、触发、环境、响应和响应测量，评级系统用于评估场景重要性和实现难度，架构驱动元素影响架构决策，如 Yinzer 系统的质量属性场景和架构驱动展示了如何根据质量属性需求设计系统架构。

#### 代码模型

**模型 - 代码差异**：架构模型与源代码在词汇、抽象、设计承诺、内涵 - 外延语句等方面存在差异，如组件概念在模型中存在但在源代码中难以直接表达，这导致了模型 - 代码差异，给开发人员带来挑战。

**一致性管理**：模型和代码会随时间产生分歧，团队可采用忽略分歧、临时建模、概要模型、在里程碑或危急关头同步、定期同步等策略管理，但需综合考虑工具使用、模型详细级别和项目对偏离的容忍度等因素。

**架构明显的编码风格**：将架构线索嵌入源代码，遵循模型嵌入代码原理，有助于理解和演化，减少设计意图丢失，通过有意提示、按合约设计等方式表达意图，利用硬机制和软机制传递模型。

#### 封装和分割

**多层级故事与主分解**：大型系统应如多层级故事般易于理解，层级嵌套有效，但并非所有系统都适合层级分解。主分解应选择单一关切组织系统，如功能、典型类型等，不同分解策略各有优劣，需根据质量属性要求选择。

**有效封装与接口创建**：封装降低复杂性，但需区分有效与低效封装，如工作时间表系统接口设计不佳。帕纳斯模块可隐藏设计秘密，实现有效封装需预测使用和考虑可替换性，创建封装接口可基于抽象数据类型思想，描述操作签名、前后置条件和类型模型。

#### 模型元素

**部署相关元素**：软件运行于硬件，部署图展示部署情况，环境元素和通信通道用于描述部署相关内容，可部署元素包括用户界面、数据库等代码及配置文件，组件类型和实例在部署中有不同表现，模块概念与组件相关但有区别。

**组件与连接器**：组件是重要计算和存储元素，通过端口通信，连接器实现组件交互，其类型多样且在架构中有重要地位，可被替换，其选择影响系统质量，还具有属性、角色等特性，可通过细化展示内部设计。

**其他元素**：设计决策指导设计方向，功能场景描述系统行为，不变量对系统进行限定，模块用于组织代码，端口规定组件通信，质量属性影响架构设计，职责分配与架构意图链相关，权衡在系统设计中普遍存在。

#### 模型关系

**投影（视图）关系**：投影是模型细节子集或转换，如房屋建筑蓝图和成本表视图，可帮助应对复杂性和规模问题，但需维护视图一致性，视图可解释为需求、主模型投影或现实世界投影。

**分割、组合与分类关系**：分割将模型分为无相交部分，组合则相反，分类关系用于确定元素类别，允许一个实例有多种类型，需区分类和类型概念。

**其他关系**：泛化描述类型间关系，指定在不同域间建立桥梁，细化关联同元素高细节和低细节表现形式，绑定将源模型概念与目标模型元素关联，依赖表示模型间变化影响关系，这些关系可用于系统建模。

#### 架构风格

**风格概述**：架构风格是架构层面模式，由元素和约束组成，提供预制约束集，具有一致性、可理解性等优势，可促进系统演化、方便沟通、支持设计重用、确保质量属性和便于分析，但实际系统可能不完全遵循柏拉图式风格。

**常见风格**：包括分层风格（可提高可修改性等）、大泥球风格（虽有稳定性但可维护性差）、管道 - 过滤器风格（支持增量式处理和再配置）、批量顺序处理风格（阶段独立，吞吐量较大）、以模型为中心的风格（组件与中心模型交互）、分发 - 订阅风格（解耦事件生产者和消费者）、客户端 - 服务器风格（不对称关系，可维护性高）、对等风格（节点对等通信，提高可用性）、map - reduce 风格（适合处理大数据集，可伸缩性强）、镜像等部署风格（影响系统性能和可用性）。

#### 使用架构模型

**理想模型特性**：好模型应具备充分精确、准确、预言、促进理解、细节一致、单主题视图、经济等特性，应根据问题选择合适精度和细节的模型，以确保模型有用且能回答相关问题。

**视图运用与问题处理**：使用视图可遵循分而治之策略，但可能面临特定视图焦点窄、问题落在视图之间、视图不一致等问题，可通过功能场景串联视图、让场景动起来、编写活动规范等技术改善，还可通过在图上加图例、不画连接器箭头提高图质量。

**模型分析与其他注意事项**：分析架构模型可采用非正式（如质量属性讨论会等）和正式技术（需转换模型），需警惕架构不匹配问题，构建系统接口模型时要选择合适抽象级别，规划用户界面时应与后端同步设计，明确指定性和描述性模型区别，对现有系统建模需明确目的并选择合适模型。

#### 总结

本书主要围绕软件架构的风险驱动模型展开。首先介绍了软件架构的核心要点，包括定义、重要性及关键时机判断。接着阐述风险驱动模型的运用，涵盖模型本质、风险识别与分类以及技术选择。随后对比不同设计方式，强调架构无关设计、专注架构设计和提升架构设计的特点。还通过各种案例，展示风险驱动模型在实际项目中的应用，包括团队沟通、组件集成和元数据处理等方面。整体通过风险驱动模型，为软件架构设计提供了清晰的思路和方法。