Chapter 1. What is Software Architecture?

理解：

软件体系结构（软件架构）的定义

系统的软件体系结构是建立一个对系统来说所需要的结构，包括软件元素，它们之间的关系，以及两者的性质。

架构模式的概念。

架构模式，也叫架构风格，一个架构模式描述软件系统里的基本的结构组织或纲要。架构模式提供一些呈先定义好的子系统，指定它们的责任，并给出把它们组织在一起的法则和指南。

掌握：

软件系统有哪几类结构？

模块、组件和连接器、配置。

模块分配具体的职责，是工作的基础；

我们调用运行时结构组件和连接器结构，在我们的使用中，组件始终是运行时实体；

分配结构描述从软件结构映射到系统的环境，组织、发展、安装、执行；

在每类结构里，元素及其之间的关系是什么？

元素是一类模块（类、层或功能的划分等），模块与其他模块相关联通过概括化或专业化的关系；

元素运行组件如服务、同行、客户、服务器、过滤器等，连接器是组件间的通信工具；

每类结构各有哪些常见的结构？其特点是什么？

模块：

分解结构：分解结构决定了系统的可修改性，以确保可能的变化是局部的；

使用结构：使用关系是一种专门的依赖关系，用来拓展或缩小系统，创建子系统，进行增量开发；

层结构：通过接口提供一个有凝聚力的服务集合；

类结构：允许重载和增量增加的功能；

数据模型：描述了数据实体及其关系的静态信息结构；

组件和连接器：

服务结构：单位服务与其他服务的协调机制；

并发结构：确定机会的并行性和资源争夺可能发生的位置；

配置：

部署结构：显示软件如何被分配给硬件处理和通信元素，特别感兴趣分布式和并行系统；

实施结构：显示了软件元素如何在系统的开发、集成和配置控制环境中映射到文件结构。分配责任，实施和整合模块给将要执行它的团队；

了解：结构与视图是什么关系？好的结构的一些经验法则。

Chapter 2. Why is Software Architecture Important?

理解：13个理由。

1、影响质量属性。

2、当系统的发展有助于改变和管理变化。

3、一个系统的质量的早期预测。

4、加强利益相关者之间的沟通。

5、捕捉最早的，因此最根本的，最难改变的设计决策。

6、定义了一组在随后的执行上的约束。

7、决定一个组织的结构，反之亦然。

8、为进化的原型设计提供依据。

9、让建筑师和项目经理了解成本和进度。

10、作为一种可重用的可重用模型，该模型是产品线的核心。

11、架构为基础的发展重点关注的组件组件，而不是简单地在创作。

12、减少设计和系统复杂度。

13、为培训新的团队成员奠定基础

Chapter 3. The Many Contexts of Software Architecture

理解：技术环境、项目生命周期、商业环境、架构师职业环境中的软件体系结构。架构与环境的相互影响。

最重要的技术背景是一系列的质量属性，架构可以帮助实现，架构的当前技术环境也是一个重要因素；

软件开发过程是软件系统开发的标准，强加给软件工程师团队，告诉他们下一步要做什么，四个主要的软件开发过程：瀑布、迭代、敏捷、模型驱动开发；

结构与系统构造不轻浮，服务于某些商业用途，目的可能会随时间而改变；

系统被创建以满足一个或多个组织的业务目标，架构师需要了解客户是谁，他们的目标是什么，这些目标将对许多架构产生影响；

每一个质量属性都源于一些附加值，有些业务目标在需求的形式中不会出现，还有一些商业目标对架构没有影响；

了解：涉众。

商业、技术、项目影响利益相关者，利益相关者和专业背景影响架构师，从而影响架构和系统；

Chapter 4. Understanding Quality Attributes

了解：系统的功能需求。功能需求与系统架构的关系。功能需求与质量需求的关系。系统约束。

功能是系统做工作的能力，这是它的目的（什么与怎样）；

功能不确定体系结构，给定一组所需的功能，可以不受约束的创建满足该功能的架构；

质量需求保证功能需求；

理解：系统的质量需求。战术的概念。

质量需求是指对产品需要的表述或将需要转化为一组针对实体特性的定量或定性的规定要求，以使其实现并进行考核。

掌握：质量属性场景的概念和举例。质量设计的7种决策。

质量属性的场景：

刺激：是一个条件，当它到达一个系统时需要一个响应；

刺激源：是一些实体（人、计算机系统、其他机构）产生的刺激；

响应：刺激到来之后的活动；

响应措施：当响应发生时，它应该是可以以某种方式测量的，因此需求是可测试的；

环境：刺激发生在一定的条件下；

人造物品：一些人造物品也是刺激；

刺激源——刺激——人造物品——响应——相应措施；

责任分配、协调模型、数据模型、资源管理、架构元素映射、绑定时间决定、技术选择；

Chapter 5. Availability

理解：可用性概念。

可用性是在某个考察时间，系统能够正常运行的概率或时间占有率期望值。

可用性建立在可靠性上增加恢复的概念（修复）。

从根本上讲，可用性是通过减少故障来减少服务中断时间的。

了解：可用性公式。可用性一般场景。

（总时间-服务中断的时间）/总时间；

掌握：

可用性战术

故障检测：信号/响应、监测器、心

跳、时间戳（用于检测不正确的序列）、正常检查（检查序列的有效性和合理性）、状态检测、投票（赋值结果是否相同）、异常检测、自测试；

从故障中恢复：主动冗余（保护所有结点）、被动冗余（保护活动成员）、备用、异常处理、回滚、软件升级、重试、忽略、退化、重构、阴影、状态再同步、升级重启、非同步转发；

预防故障：从服务器中移除、事物、预测模型、异常预防、提高能力集；

可用性设计清单

采用合适的架构形式和手法，建立可用性模型、设计专门的可用性视图；

确定需要高度使用的系统的责任，确保分配额外的责任来检测故障；

确保这些责任（记录故障、通知适当的实体、事件资源不可用导致的故障、暂不可使用、修复或掩盖故障、在退化模式中操作）；

Chapter 6. Interoperability

理解：互操作性概念

互操作性是在一定程度上两个或两个以上的系统可以有效地交换有意义的信息。

了解：互操作性一般场景。

掌握：

互操作性战术

定位：发现服务（定位一个服务通过搜索已知的目录服务（程序））；

接口管理：业务流程（使用控制机制来协调、管理和排序服务的调用，系统以复杂的方式相互完成复杂的任务），接口调整（对一个接口添加或删除功能）；

互操作性设计清单

采用合适的架构形式和手法，设计专用的互操作性视图；

确定哪些职责需要与其它系统进行交互操作，确保已分配的责任（接受/拒绝/记录请求，交换信息，通知适当的实体）；

确保协调机制符合关键质量属性的要求；

确定可能在互操作系统之间交换的主要数据抽象的语法和语义，确保这些主要的数据抽象与互操作系统的数据的一致性；

Chapter 7. Modifiability

理解：可修改性概念

可修改性是有关改变和我们的有关成本和风险变化的兴趣；

了解：可修改性一般场景。

掌握：

可修改性战术

减小模块的规模：拆分模块（把具有大量能力的模块细分为几个较小的模块）；

增强凝聚力：增加语义的连贯性（如果一个模块中的职责A和B不是服务相同的目的，他们应该被放在不同的模块中）；

减少耦合：封装，使用中介（减少依赖性），限制依赖关系，重构（两个模块被相同的变化影响），抽象共同的服务；

延迟绑定；

可修改性设计清单

采用合适的架构形式和手法，设计专用的可修改性视图；

确定那个变化或类别可能会发生改变；

确定哪些功能或质量属性可以在运行时改变以及如何影响协调，确定哪些用于协调的设备、协议和通信路径有可能改变，确保变化的影响被限制在一小部分模块；

关注一些元素的可修改

性（使用协调模型减少耦合、延迟绑定）；

Chapter 8. Performance

理解：性能概念。

性能是有关时间和软件系统满足时间需求的能力；

了解：性能公式。性能一般场景。

掌握：

性能战术

控制资源需求：管理采样速率（减少采样频率，需求可能会减少），限制事件响应（设定处理事件的最高速率来确保实际处理时更可预测），按重要性排列事件；

减少开销（中介增加了资源消耗，删除中介提高了延迟），绑定执行时间（限制响应事件的最大时间），提高资源效率；

管理资源：增加资源（减少延迟），增加并发性（减少阻塞），保持多路计算，维护多分数据（保存数据的副本），绑定队列大小（控制进入队列的最大数目），安排资源（优先级）；

性能设计清单

采用合适的架构形式和手法，建立性能模型；

确定涉及重载、有时间临界响应要求、被大量使用或者影响系统关键事件发生的系统的责任（处理需求并判断是否造成瓶颈，附加责任认识和处理需求，职责交叉过程或处理器边界，

管理线程，调度共享资源或管理性能相关的队列）；

确定那些必须相互协调的系统的要素，选择沟通协调机制（并发事件优化，所需的性能响应可以被传递，具有通讯机制）；

确定那部分涉及重载、有时间临界响应要求、被大量使用或者影响系统关键事件发生的部分的数据模型（维护多份关键数据，数据分区，减少处理需求/增加资源）；

确定系统中那些资源对性能至关重要，确保这些资源在系统运行时被监控和管理（需要注意和管理的系统元素：时间和其它性能关键资源，进程/线程模型，

资源的优先顺序和资源的获取，调度和锁定策略，部署而外资源）；

Chapter 9. Security

理解：安全性概念。

安全性是衡量系统从越权访问保护数据和信息的能力，同时还提供访问给那些被授权的人或系统；

了解：安全一般场景。

掌握：

安全性战术

检测攻击：检测入侵，检测服务拒绝，验证消息完整性，检测消息延迟；

抵抗攻击：识别系统外部输入源，验证参与者，授权参与者，限制访问，最少数量的接入点，加密数据，单独的实体，更改默认设置，对敏感资源限制访问，锁定计算机（如果有

重复失败的访问，限制访问权限），通知参与者；

从攻击中恢复：审计（记录用户和系统的行为及其影响，跟踪行为，确定攻击者）；

安全性设计清单

采用合适的架构形式和手法，建立安全性模型；

确定哪些系统需求需要保护，分配额外的责任（识别/验证/授权参与者，授予/拒绝数据访问，记录尝试修改数据和服务的记录，加密

数据，识别攻击，从攻击中恢复，验证校验和哈希值）；

Chapter 10. Testability

理解：可测试性概念。

软件的可测试性是指通过测试来证明其故障的容易程度。

了解：可测试性一般场景。

掌握：

可测试性战术

控制和观察系统状态：专用接口（控制/获取组件的变量值），记录/回放（获取信息作为进一步测试的输入），局部化状态存储，

抽象数据资源，沙盒（隔离系统），可执行断言；

限制复杂性：限制结构的复杂性，限制非决定论；

可测试性设计清单

采用合适的架构形式和手法，建立可测试性模型；

确定哪些系统的责任是最关键的，因此需要最彻底的测试；

确保已分配的其他系统的责任：执行测试并获取导致故障的原因，控制并且观察系统的有关状态；

Chapter 11. Usability

理解：易用性概念。

易用性是关心用户如何用系统所提供的支持容易的完成所要完成的任务。

了解：易用性一般场景。

掌握：

易用性战术

支持用户的主动性：取消，暂停/恢复，撤销，合计（聚合）；

支持系统的主动性：维护任务模型，维护系统模型，维护用户模型；

易用性设计清单

采用合适的架构形式和手法，采用多种易用性原型；

确保已分配的其它系统的责任，如需要在以下方面协助用户：学习如何使用系统，有效的完成手中的任务，调整和配置系统，从用户和系统的错误中恢复；

Chapter 12. Other Quality Attributes

了解：其它软件质量属性如

可变性：系统和他的构建支持不同变量生成的能力；

可移植性：从一个平台移植到另一个平台的难易程度；

开发可分布性：支持分布式软件开发的软件设计的质量；

伸缩性：水平可伸缩性（将更多的资源添加到逻辑单元，如服务器），垂直可伸缩性（将更多的资源添加到物理单元，如添加内存）；

可部署性：关注的是一个执行怎样到达主机平台以及它是怎样调用的；

移动性：处理一个平台可以流动的问题；

可监控性：在系统执行时对操作人员的监控能力；

生命财产安全性：避免系统受到损害，造成损失的能力；

其它类别的质量属性如架构质量属性、商业属性、系统质量属性。ISO/IEC FCD 25010 产品质量标准。

理解：如何处理未知的质量属性

对质量属性建模，为质量属性设计一组战术，构建设计清单；

Chapter 13. Patterns and Tactics

了解：架构模式（架构风格）的概念。

一个架构模式确定语境、问题和解决方案之间的关系。

掌握：

层次模式

语境：关系分离；

问题：软件在某些情况下需要分离，模块可以开发和发展分别与各部分间的相互作用，支持可移植性，

可修改性和再利用；

解决方案：将软件分为单元层，每一层都是一组模块，这些模块提供一个有凝聚力的服务集合，使用必须是单向的；

概述：层次模式定义层和使用关系；元素：层；关系：允许使用；

约束：每一个软件模块被分配给一个层，至少有两层，较低的层不能使用它上面的层；

缺点：增加了系统的成本和复杂性，不利于性能；

代理模式

语境：分布式服务之间的交互操作；

问题：如何构造分布式软件，使用户不需要知道供应商的性质和位置，使其易于动态的改变用户和供应商之间的捆绑；

解决方案：通过插入一个中介服务（服务器）将用户服务（客户端）分开；

概述：代理模式调解一些客户机和服务器之间的通信；

元素：客户端/请求服务，服务器/提供服务，代理/中介，客户端代理，服务器端代理（辅助性中介，管理与代理实际实际通信）；

关系：客户端和服务器与代理的依赖关系；

约束：客户端/服务器只能附加到一个代理；

缺点：代理添加一个间接层，可能是一个通信瓶颈，该代理可能是一个单一的故障点，代理增加了复杂性，代理可能是安全性攻击的目标，代理可能很难测试；

MVC模式

语境：从模型中分离视图；

问题：如何将用户界面的功能与应用程序的功能分开，但是仍可以响应用户输入，或者改变底层应用程序数据。当底层应用数据改变时，如何创建、维护和协调用户界面的多视图；

解决方案：MVC模式把应用功能分成三种成分；

概述：MVC模式将系统功能分为模型、视图和调解模型与视图关系的控制器；

元素：模型、视图、控制器；

关系：通知关系连接模型、视图、控制器的实例，通知元素相关状态的变化；

约束：模型组件不能直接与视图交互；

缺点：它的复杂性对于简单的用户界面来说可能是不值得的，模型、视图、控制器的抽象可能不适用于一些用户界面工具箱；

管道-过滤器模式

语境：处理数据流；

问题：一些系统可能被分解为可重用的、松散耦合的组件，具有简单通用的交互机制；

解决方案：管道和过滤器模式的相互作用模式是以连续变换的数据流为特征；

概述：数据通过管道连接的过滤器进行转换；

元素：过滤器，管道；

关系：过滤器的输出与管道的输入的依赖关系，反之亦然；

约束：管道连接过滤器的输出端口到输入端口，连接过滤器必须同意在连接管道上传递的数据类型；

CS模式

语境：大量的分布式客户希望访问共享的资源和服务；

问题：我们希望通过集中控制资源和服务，同时在多台物理服务器上分配资源，来提高可扩展性和可用性；

解决方

案：客户通过请求提供一组服务的服务器的服务进行交互；

概述：客户端与服务器进行交互，从服务器中按需要调用服务，并等待请求结果；

元素：客户端，服务器，请求/应答连接器；

关系：客户与服务器的连接关系；

约束：客户端通过请求应答连接器连接到服务器；

缺点：服务器可能成为性能瓶颈，服务器可能是单一的故障点，关于定位功能的决定通常是复杂的，系统建立后在改变会很昂贵；

P2P模式

语境：分布式计算实体被认为是平等的；

问题：如何将一组平等的分布式计算实体通过一个共同的协议连接到彼此，以便他们可以有高可用性和可扩展性的组织和共享他们的服务；

解决方案：在P2P模式中，组件直接作为对等点进行交互，所有的对等点都是平等的，没有对等点对系统的健康来说是至关重要的；

概述：通过对等点的合作实现计算；

元素：对等点，请求/应答连接器；

关系：对等点与他们的连接者的关系，附件可能在运行时更改；

约束：对任何给定对等点的附件的数目，用于搜索对等点的跳数，那些点知道其它的点，一些P2P网络用星型拓扑结构组织，对等点只能连接到超级节点；

缺点：管理安全性、数据一致性、数据服务可用性、备份与恢复都比较复杂，小的对等系统可能无法始终实现质量目标（性能/可用性）；

SOA（面向服务架构）模式

语境：一些没有任何详细知识的服务的互操作性的实现；

问题：如何在不同的平台上运行分布式组件的互操作性，并且用不同的语言实施，提供不同的组织并发送到互联网；

解决方案：SOA模式描述了一个提供或消费服务的分布式组件的集合；

概述：计算是通过网上的一组合作服务实现的；

元素：服务供应商/服务消费者，ESB（中介元素），服务注册处，业务流程服务器，连接器（SOAP/REST/HTTP连接器）；

关系：可以使不同种类的组件的附件连接到各自的连接器；

约束：服务消费者与服务供应商连接；

缺点：建立系统很复杂，有一个关于中间件的性能开销，服务可能是性能瓶颈，不提供性能保证；

发布订阅模式

语境：数据生产者和消费者的精确数量和性质不是预定的或固定的，他们也不共享数据；

问题：我们如何创建集成机制，支持在生产者和消费者之间传递信息的能力，是他们不知道彼此的身份，甚至是潜在的存在；

解决方案：在发布订阅模式中，组件通过发布消息或时间进行交互；

概述：组件发布和订阅事件，一个事件由组件发布，通过连接器发送给所有注册用户；

元素：发布者，订阅者，发布/订阅连接器；

关系：发布/订阅连接器与组件的关

系，哪些组件发布事件，哪些组件接收事件；

约束：发布端口被连接到发布角色，订阅端口被连接到接听角色；

缺点：增加延迟，对消息的可扩展性和可预测性有负面影响，没有消息顺序控制，不能确保消息传递；

共享数据模式

语境：各种组件计算需要处理和共享大量的数据；

问题：如何系统存储和持久操作数据，并由多个独立的组件访问；

解决方案：在数据共享模式中，互动通过在多个数据存取器和至少一个共享数据存储之间持续的数据交换；

概述：数据存取器之间的通信通过共享数据存储做媒介；

元素：共享数据存储，数据存取器组件，数据读写连接器；

关系：数据存取器和连接数据存储的依赖关系；

约束：数据存取器只能与数据存储交互；

缺点：共享数据存储可能是性能瓶颈，共享数据存储可能是单一故障点，生产者和消费者的数据可能是高耦合；

Map-Reduce（映射规约）模式

语境：需要快速分析大量数据；

问题：有效的执行一个分布式和并行的大数据集，为程序员指定的分析提供一个简单的方法；

解决方案：按需要分配数据的专业的基础设施，过滤数据检索项目的映射，组合映射结果的规约；

概述：映射规约模式提供了一个分析大型分布式数据的框架，映射执行提取和分析变换的部分，规约执行结果的加载；

元素：映射、规约、基础设施框架；

关系：部署，在基础设施和映射与规约的实例间实例化、监视和控制他们的关系；

约束：被分析的数据必须作为一组文件存在，映射功能是无状态的，不能彼此间通信；

缺点：间接费用，如果数据不能分解为更小的子集，并行性优点丢失，多个规约操作是很复杂的；

多级模式

语境：将一个系统的基础设施分配到不同的子集；

问题：如何将系统划分为多个独立计算的执行结构（由一些通信媒体连接的软硬件）；

解决方案：许多系统的执行结构被组织为一组逻辑组件，每个组被称为一级；

概述：系统的执行结构被组织为一组逻辑组件；

元素：层（级）；

关系：是一部分，与……通信，被分配给……；

约束：一个软件组件属于一个层；

缺点：大量的前期成本和复杂性；

理解：模式与战术的关系

模式建立在战术上，战术增强模式；

Chapter 14. Quality Attribute Modeling and Analysis

了解：模型。常见质量属性模型的成熟度

可用性（硬件（适度成熟度）/软件（不太成熟）），互操作性（低成熟度），可修改性（大量研究，仍需更多经验），性能（高成熟度），安全性（低成熟度），易用性；

了解：

思想实验：思想实验是通过特定的场景在思想上或口

头上进行工作。

粗略分析：分析不需要特别详细或精确，对可疑领域或重要领域进行深入分析。

原型、模拟仿真、实验：

许多工具可以帮助执行实验，来确定一个设计的行为；取决于具有部分或原型的实现；基于事件的模拟器可以模拟系统在不同负载下的行为；

必须创建模拟，必须有不同的负载和检查响应；

Chapter 15. Architectures in Agile Projects

了解：敏捷开发思想与准则。

思想：敏捷开发对一个项目来说（对他们的利益相关者更敏感，更快的开发用户关心的功能，在项目的生命周期中显示更多更早的进展，通过文件编写减少负担）；

这些需求与架构不冲突，我们应该做多少架构/架构文档；

准则：通过早期与持续交付满足客户需要，欢迎不断地变化需求，在较短的时间内频繁的提交工作软件，商业人士与开发工作者一起工作，围绕动机个体构建工程，

面对面交谈，工作软件是进步的主要手段，可持续发展，关注先进的技术和好的设计，最大化的发挥团队的工作量，团队效率；

理解：敏捷开发的甜蜜点。

甜蜜点：向左/快速的变化，向右/高保证；

了解：

敏捷开发与架构编档

为读者（有可能是项目维护者或不在项目中的人）而写，如果读者不需要，就不必写了；

敏捷开发与架构演化

架构权衡分析法ATMT（确定利益相关者最关心的问题）；

Chapter 16. Architecture and Requirements

理解：

ASR：重要的架构需求，会对架构产生深远影响的需求；

ASR的几种获取方法：需求文档，采访利益相关者，理解商业目标，效应数；

QAW：质量属性车间，是一种促进，利益相关者集中注意的方法，来生产、排序、并细化质量属性情况，在软件架构设计完成之前；

QAW介绍、业务介绍、架构计划、识别架构工程师、情景头脑风暴、情景整合、情景优先级、情景细化；

了解：商业目标场景

目标来源、目标主体、目标对象、环境、目标、目标测量、目标的波动性和价值；

理解：PALM方法（寻找商业目标的方法）

由那些可以讲述商业目标的架构师和利益相关者参加的会议；

PALM概要介绍、商业驱动介绍、架构驱动介绍、引出业务目标、从业务目标识别潜在的质量属性、分配现有的质量属性驱动、做出总结；

掌握：效用树

在一个地方记录所有ASR的方法；

按照对架构的影响，业务或者任务价值的优先级建立ASR；

ASR是被捕获的场景，树根是占位符节点被称为效用，二级树包含广泛的质量保证类别，三级树细化这些类别；

效应——质量属性——细化质量属性——ASR

一个QA没有任何ASR不一定是错误或者遗漏，注意寻找没有记

录ASR的地方，ASR效率的评级是最值得关注的；

Chapter 17. Designing an Architecture

理解：

Generate and Test（架构设计的假设检验法）

认为当前的设计是一个假设，询问当前的设计是否满足需求，如果不满足需求，就生成一个新的假设；

当前假设：现有系统、框架，模式与策略，区域分解，设计清单；

测试：使用已覆盖的分析技术、根据质量属性设计战术、ASR；

下一步假设：添加缺失的需求，运用战术调整假设的质量属性；

初始化、迭代、终结（所有的ASR都已经满足、用光了预算）；

掌握：ADD方法（属性驱动设计）

包中的许多技术已经讨论过，迭代法（选择系统的一部分进行设计，整理重要的架构需求），ADD不产生一个完整的设计（需求容器、交互的信息流），不生产API或容器的标志；

输入：需求（功能、质量、约束），输出：容器（责任、相互作用、信息流）；

步骤：选择系统的一个元素来设计，为所选元素确定ASR，为所选元素生成设计方案，库存剩余需求和为下一次迭代选择输入，重复1~4步直至所有的API被满足；

整个系统——效用树

某个元素——从元素的需求生成一个效用树

Chapter 18. Documenting Software Architectures

理解：

架构编档的重要性:如果人们不知道它是什么，那么最好的架构也是没有用的，设计团队所有的努力都将白费；

架构文档的用途：

教育：把人介绍给系统；

利益相关者之间沟通的主要工具：分析师与开发商，分析师与未来分析师；

系统分析和构建的基础；

读者：新员工、开发商、分析师；

了解：如何选择视图进行编档。

构建一个利益相关者/视图表；

结合视图来减少他们的数量；

优先级和阶段：早期的分解视图发布、不需要满足所有的信息需求、再启动另一个视图之前不必要完成前一个视图；

掌握：

如何对视图进行编档：

主要介绍：显示视图的元素和关系、在该视图的词汇中包含你想传达的系统信息、主要表现为图形、偶尔会有文字；

元素目录：元素及其性质、关系及其性质、元素接口、元素行为；

上下文图：如何在这个系统中涉及到环境（人、其他系统、物理对象）；

变异指南：指出变异点；

原理：所选模式的架构问题以及选择它的理由；

视图文档模版。架构文档摘要信息。

如何对系统行为进行编档：

行为文档通过描述视图中的架构元素如何相互作用来补充视图；

行为文档推理：系统死锁的可能性，系统在规定的时间完成任务的能力，最大内存消耗；

行为在视图模版元素目录中有他自己的部分；

如何对质量属性进行编档：

质量属性应该展示在文档的哪一部

分：原理、提供服务的架构元素经常有质量属性提供的范围、架构文档经常包含一个映射需求展示如何满足需求、路线图告诉利益相关者在哪寻找关注点；

Chapter 19. Architecture, Implementation, and Testing

理解：实现与架构的一致性。

4种技术帮助代码和架构保持一致：

在代码中嵌入设计、框架、代码模版、保持架构和代码的一致性；

掌握：

将架构嵌入代码：

架构行为作为实施的蓝图，这意味着：

实施者知道他们正在实施什么架构；

他们可以在代码中作为注释记录这些结构；

在所有级别使用注释；

在项目中说明，工具可以自动实现代码与架构相关联；

框架方法：

一个框架是一些可重用的类围绕特定主题组织起来的；程序员使用框架提供的服务；

基类、组件交互、控制器、配置文件；

代码模版方法：

代码模版是代码的集合，代码的程序员提供应用程序特定部分；

使用每个关键组件的代码模版必须有一个热备份；

将应用程序的特定代码放置在模版内的固定位置；

防止架构侵蚀：

使用工具来强制约束架构；

当侵蚀发生时，标记过期文件，将给予剩余部分更多的信任；

进度文档/代码同步时间；

了解：架构师在测试中的角色。

Chapter 20. Architecture Reconstruction and Conformance

了解：架构重构的背景和目的。

背景：系统已经存在，但是不知道它的架构（没有记录、文件丢失、文件不同步）；如何维持系统；如何管理发展；

目的：记录一个没有文档或文档已经过时的体系结构；确保架构与架构设计的一致性；在架构改造中，架构是从现有的构件中逆向设计的；

理解：架构重构的阶段。每个阶段的方法。

原始视图提取：从各种来源（源代码）提取原始信息；

数据库建设：将原始信息转换为标准形式来建设数据库；

视图融合与操作：推断静态视图的运行时行为；

架构分析：对生成的假设进行测试；

Chapter 21. Architecture Evaluation

了解：架构评审的3种形式及其特点。

设计人员在设计过程中进行评价；

同行在设计过程中进行评价；

外部人员在架构设计过程中进行分析；

轻量级架构评审。

掌握：ATAM（架构权衡分析）方法：目的、参与人员、步骤、采用的方法、结论。

目的：确定利益相关者最关心的问题、揭示架构满足质量目标的情况、质量目标之间的联系，如何权衡众多的质量目标；

参与人员：评估小组、项目决策者、架构涉众；

步骤：介绍ATMT，介绍业务驱动，介绍架构，确定架构方法，创建效应树，分析架构方法，头脑风暴和优先方案，分析架构方法，介绍结果；

Chapter 26. Architectures for the Cloud

了解：云计算的特点、主要机制、技术。

掌握：云计算下的质量属性。

安全性：无意的数据共享；

性能：负载增加时，自动缩放；

可用性：失败后检测和修正；

考题类型

选择题（30%）

简答题（30%）

分析设计题（40%）