**Chapter 1: What is SoftWare Architecture?**

1. **软件体系架构（软件架构）的定义**

The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

一个系统的软件体系架构是构成系统所需的一系列结构，包括软件元素，软件元素之间的关系以及两者的属性。

1. **软件系统有哪几类结构？**

Module; component and connector; allocation

1. **Module Structures 模块结构**

Structures partition systems into implementation units are called modules. Modules are assigned specific responsibilities, and are the basis of work.

把系统分成若干个实现单元的结构，称为模块。每个模块被分配到具体的职责，它们是工作的基础。

元素是某种类型的模块（类、层、功能分区等）。模块通过泛化或特定的关系与其他模块联系。模块结构允许我们回答这样的问题：分配给每个模块的主要功能职责是什么？它使用和依赖什么其他软件？

常见的模块结构：Decomposition structure，Uses structure，Layer structure，Class (or generalization) structure，Data model. 即分解结构、使用结构、层次结构、类结构（泛化结构）、数据模型。

1. **Component-and-connector Structures 组件与连接件结构**

Structures focus on the way the elements interact with each other at runtime to carry out the system’s functions are called C&C structures. A component is always a runtime entity.

关注运行时元素之间如何彼此交互以执行系统的功能的结构，称为C&C（组件与连接件）。一个组件通常是一个运行时实体。

元素是运行时组件，例如服务、对等体、客户端、服务器、过滤器等。连接件是组件之间的通信工具，例如call-return,进程同步操作符、管道等。组件与连接件视图允许我们回答这样的问题：主要的共享数据存储是什么？系统的哪些部分可以并行运行？组件与连接件视图对于提出有关系统运行时属性的问题至关重要。

常见的C&C结构：Service structure，concurrency structure. 即服务结构、并发结构。

1. **Allocation Structures 分配结构**

Allocation structures describe the mapping from software structures to the system’s environments.

分配结构描述了从软件结构到系统环境的映射。

分配视图可帮助我们回答如下问题：每个软件元素执行什么处理器？在开发，测试和系统构建期间每个元素存储在什么目录或文件中？每个软件元素对开发团队的分配是什么？

常见的分配结构：deployment structure，implementation structure，work assignment structure. 即部署结构、实现结构、工作分配结构。

1. **结构与视图是什么关系？**

A view is a representation of a structure. Architects design structures. They document views of those structures.

视图是结构的表示。架构师设计结构，然后把这些结构的视图记录成文档。

1. **好的结构的一些经验法则**

架构应该是单个架构师或一组架构师的产品，这个（些）架构师都应该是认证过的技术领导者。

架构师（或者架构团队）应该将架构建立在明确制定的质量属性要求的优先级列表上。

应该使用视图来记录架构。

应该评估架构提供系统的重要质量属性的能力。（这个应该在生命周期的早期做到，并适当重复）

架构应该适用于增量实现。

架构应该具有定义良好的模块，其功能职责的分配建立在信息隐藏和关注分离的原则上。

应该使用每个属性特有的知名架构模式和战术来实现质量属性。

架构不应该依赖于特定版本的商业产品或工具。

产生数据的模块应该与使用数据的模块分开。

每个进程都应该被编写，以便可以很容易地改变对特定处理器地分配，甚至是在运行时。

架构应该具有少量用于组件交互的方式。

架构应该包含特定的和小的资源争用集合。

**Chapter 2: Why is Software Architecture Important?**

1. **Thirteen Reasons**

(1) Influence quality attributes. 影响质量属性

(2) Help reason about and manage change as the system evolves. 随着系统的发展，帮助理解和管理变化

(3) Early prediction of a system’s qualities. 早期预测系统的质量

(4) Enhances communication among stakeholders. 促进涉众之间的沟通

(5) Capture the earliest and hence most fundamental, hardest-to-change design decisions. 捕获最早的，也是最根本的，最难以改变的设计决策

(6) Defines a set of constraints on subsequent implementation. 定义后续实现的一系列约束

(7) Dictates the structure of an organization,or vice versa. 描述组织的结构，反之亦然

(8) Provide the basis for evolutionary prototyping. 提供进化原型的基础

(9) Allows the architect and project manager to reason about cost and schedule. 允许架构师和项目经理推断成本和进度

(10) As a transferable, reusable model that from the heart of a product line. 作为一个来自产品线心脏的可转移的、可重复使用的模型

(11) Architecture-based development focuses attention on the assembly of components, rather than simply on their creation. 基于架构的开发重点关注组件的组装，而不是简单地创建它们

(12) Reducing design and system complexity. 减少设计和系统复杂性

(13) Be the foundation for training a new team member. 作为培训团队新成员的基础

**Chapter 3: The Many Contexts of Software Architecture**

1. **Contexts of Software Architecture 架构的上下文环境**

Technical, project life cycle, business, professional. 技术环境、项目生命周期、商业环境、职业环境

1. **Technical context**

最重要的技术上下文因素：一系列质量属性。架构的当前技术环境也是一个重要的因素，包括行业标准、专业社区流行的软件工程技术等。

1. **Project Life-cycle Context**

软件开发过程是开发软件系统的标准方法，它们告诉团队成员下一步做什么。最主要的四种软件开发过程是：瀑布模型，迭代开发，敏捷开发，模型驱动开发。

1. **Business context**

架构设计和系统设计不是漫无目的的，它们为一些商业目的而服务，这些商业需求可能随时间而变化。

1. **Professional context**

除了直接生产一个架构之外，架构师需要更多除了技术技能之外的技能，比如外交、谈判、清晰沟通思路；架构师还需要了解最新的知识，了解业务注意事项等。

1. **Stakeholder 涉众**

A stakeholder is anyone who has a stake in the success of the system. 涉众是和系统成功有利害关系的任何人。

Developing organization's management stakeholder 开发组织管理涉众

Marketing stakeholder 营销涉众

End user stakeholder 终端用户涉众

Maintenance organization stakeholder 维护组织涉众

Customer stakeholder 客户涉众

1. **How is architecture influenced? 架构如何被影响**

需求影响架构。一个软件架构是受商业、社会、技术的影响。一个架构的存在反过来影响未来架构的商业、社会、技术等因素。特别的，每个架构的上下文环境在影响架构师和架构中起着作用。

archinf.tiff

1. **What Do Architectures Influence? 架构影响什么**
2. 技术环境。架构可以影响下一个系统的涉众需求。它给客户提供一个比从头开始更可靠、更及时、更经济的方式去接收一个系统，客户可能会乐意放松他们的一些需求去获得这些利益。
3. 项目环境。架构影响开发组织的结构。团队围绕着每个单独的软件单元而组成，开发、测试、集成活动都围绕着单元。团队嵌入组织结构。
4. 商业环境。架构可以影响开发组织的业务目标。架构构建成功的系统可以使公司在特定的市场细分中建立立足点。架构可以为类似系统的高效生产和部署提供机会，并且组织可以调整其目标以利用其新发现的专业知识来探测市场。
5. 职业环境。系统建设的过程将影响架构师的后续经验。围绕特定技术方法构建成功的系统将使架构师更倾向于在未来使用相同的方法构建系统。失败的架构不太可能被选择用于未来的项目。

AIC.tiff

**Chapter 4: Understanding Quality Attributes**

1. **Functional requirements 功能需求**

Functionality is the ability of the system to do the work for which it was intended. 功能是系统完成预期工作的能力。

功能和架构的关系：功能不决定架构，给定一系列要求的功能，你能创造的以满足这些功能的架构是没有尽头的。

1. **Quality attribute scenarios 质量属性场景**
2. Stimulus 刺激：到达时要求系统做出响应的一个条件
3. Stimulus source 刺激源：产生刺激的一些实体（人，计算机系统等）
4. Response 响应：系统在刺激到达时采取的活动
5. Response measure 响应度量：当响应发生时，应该以某种方式测量，以便可以测试该需求
6. Environment 环境：刺激发生的某些特定条件
7. Artifact 制品：受到刺激产生的制品
8. **Tactics 战术**

A collection of primitive design techniques that an architect can use to achieve a quality attribute response.

战术：架构师能用来实现质量属性相应的原始设计技术的集合。

1. **Quality design decisions 质量设计决策（7种）**
2. Allocation of responsibilities 职责分配
3. Coordination model 协调模型
4. Data model 数据模型
5. Management of resources 资源的管理
6. Mapping among architectural elements 架构元素之间的映射
7. Binding time decisions 绑定时间决策
8. Choice of technology 技术选择

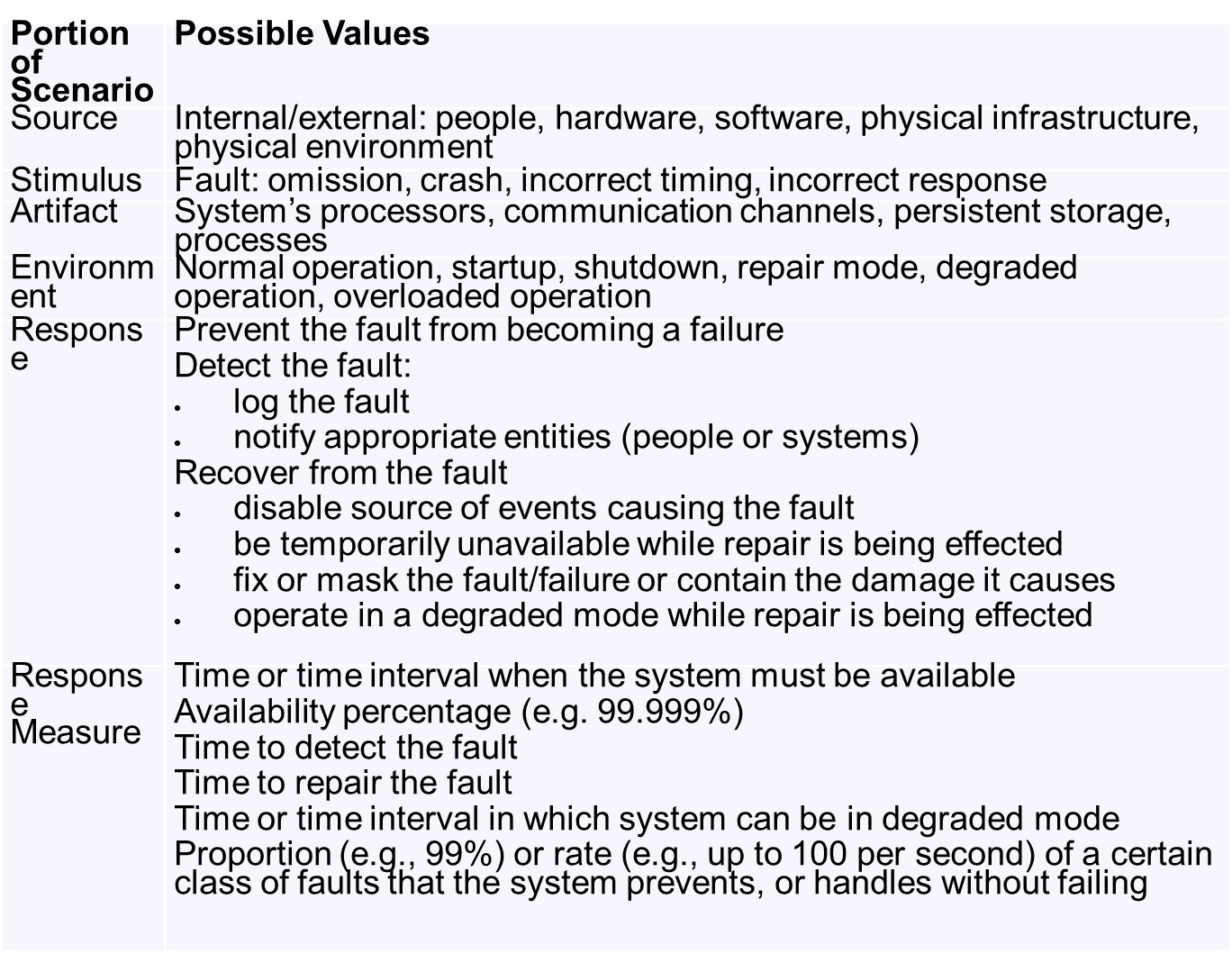
**Chapter 5: Availability 可用性**

1. **可用性概念**

Availability refers to a property of software that it is there and ready to carry out its task when you need it to be.

可用性是软件的一种属性，它存在那里并且时刻准备着执行他的任务。当你需要它时。

1. **可用性的一般场景**



1. **可用性公式**

MTBF/(MTBF+MTTR)

MTBF：平均故障间隔时间

MTTR：平均修复时间

1. **可用性战术**

目标：使系统能够承受故障，从而使服务保持符合其规范。战术使系统能够承受故障，或者至少束缚故障的影响，使修复称为可能。



1. **可用性设计清单：**
2. 采用合适的架构模式和战术
3. 建立可用性模型/公式
4. 如有必要，建立专用可用性视图
5. 确定高度可用的系统职责，确保分配了额外的职责来检测故障
6. 确保有以下职责：记录故障；通知适当的实体（人或系统）；禁用导致故障的事件源；暂时不可用；修复或屏蔽故障/失败；在降级模式下操作

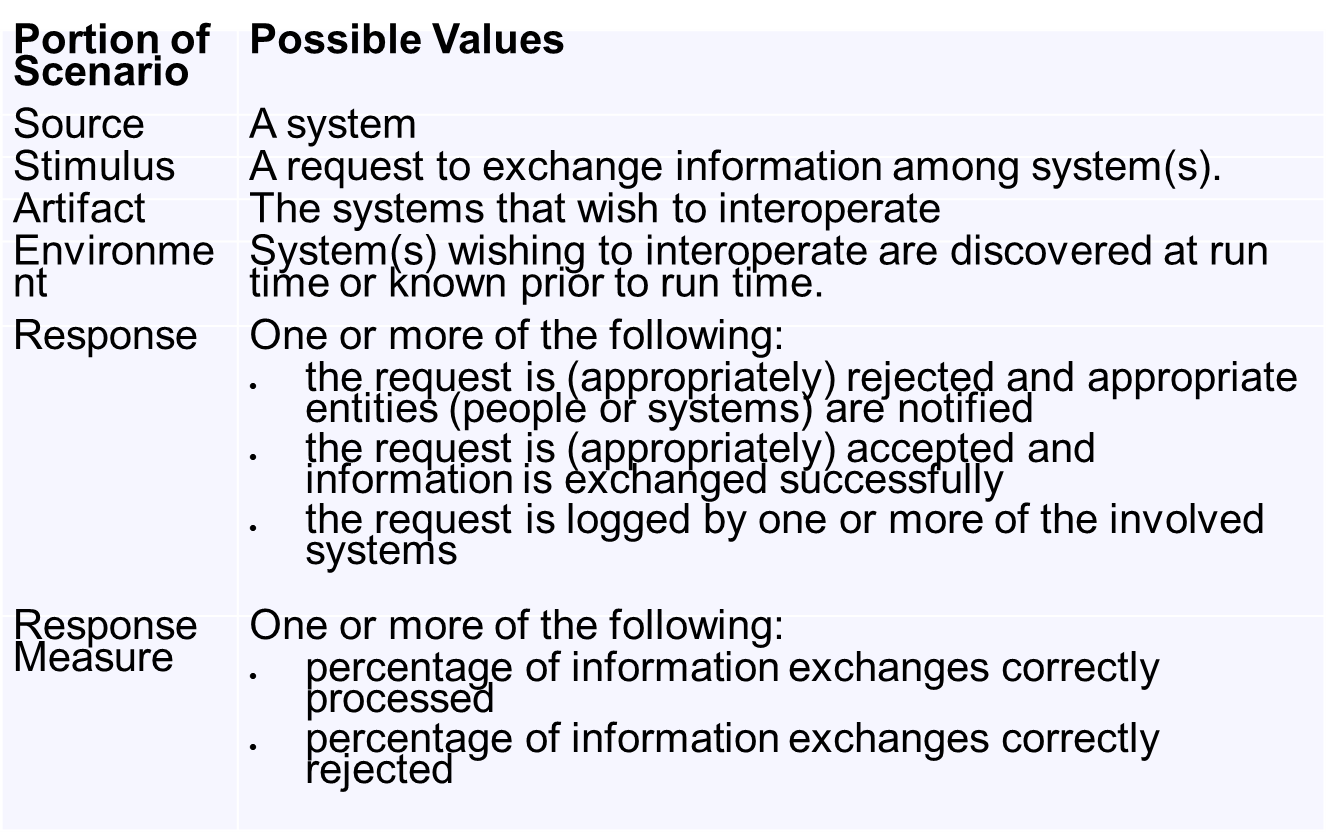
**Chapter 6: Interoperability 互操作性**

1. **互操作性概念**

Interoperability is about the degree to which two or more systems can usefully exchange meaningful information.

互操作性是指两个或多个系统可以有效交换有意义信息的程度。

1. **互操作性一般场景**



1. **互操作性战术**



1. **互操作性设计清单**
2. 采用合适的架构模式和战术
3. 如有必要，设计专用的互操作性视图
4. 确定哪些职责需要与其他系统互操作
5. 确保已分配以下职责：交换请求；交换信息；拒绝请求；通知适当的实体（人或系统）；记录请求（用于审计跟踪）；确保协调机制能够满足关键的质量属性要求；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，并确保这些主要数据抽象与互操作系统的数据一致；

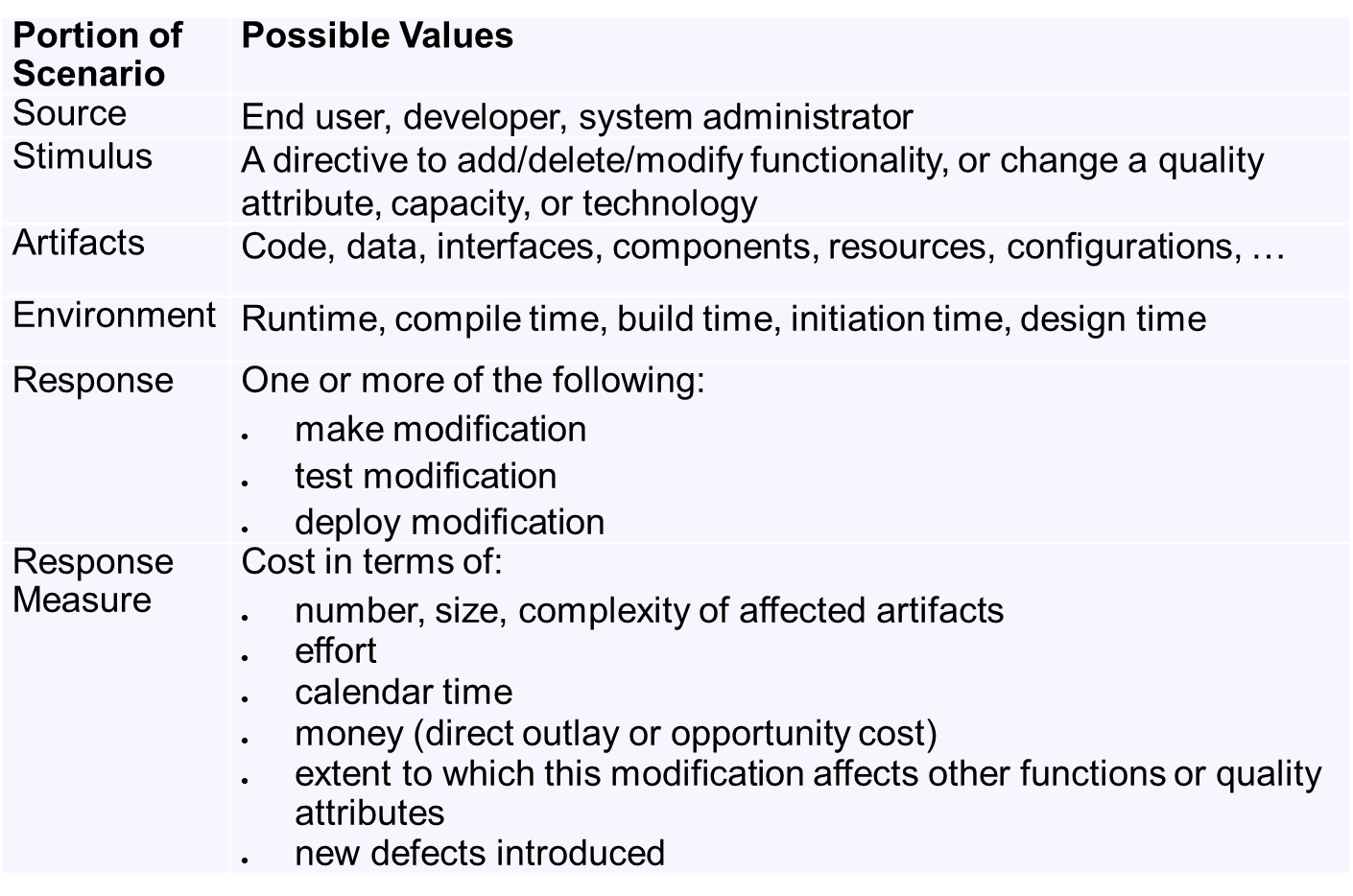
**Chapter 7: Modifiability 可修改性**

1. **可修改性概念**

Modifiability is about change and our interest in it is in the cost and risk of making changes.

可修改性是关于变化的，我们对它的兴趣在于变化的变化的成本和风险。

1. **可修改性一般场景**



1. **可修改性战术**



1. **可修改性设计清单**
2. 采用合适的构架模式和战术
3. 如有必要，设计专用的可修改性视图
4. 确定可能发生的更改或类别的更改。对于每个潜在的变化或者类别的变化：确定需要添加、修改或删除以进行更改的职责；确定其他职责受到变化的影响；分配将在统一模块中一起更改的职责，以及将在不同时间在不同模块中更改的职责。
5. 确定哪些功能或质量属性可以在运行时更改，以及这会如何影响协调。
6. 确定可能会更改的用于协调的设备、协议和通信路径。
7. 对于那些需要考虑可修改性的元素，请使用减少耦合的协调模型。

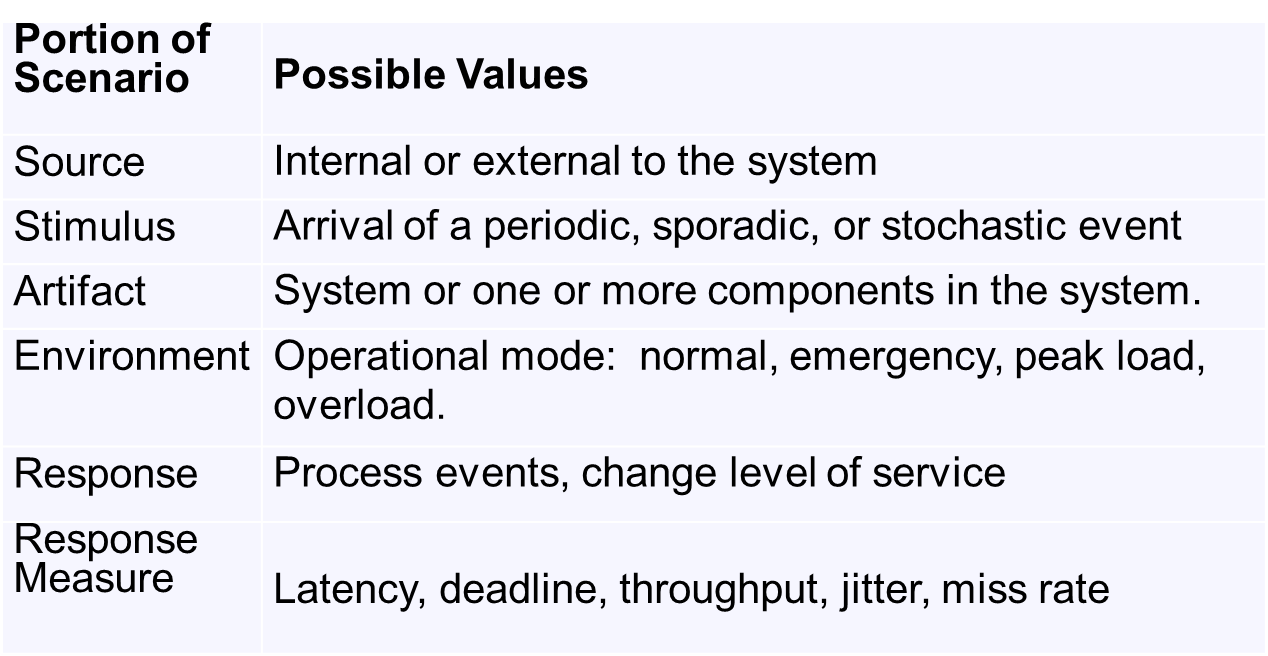
**Chapter 8: Performance 性能**

1. **性能概念**

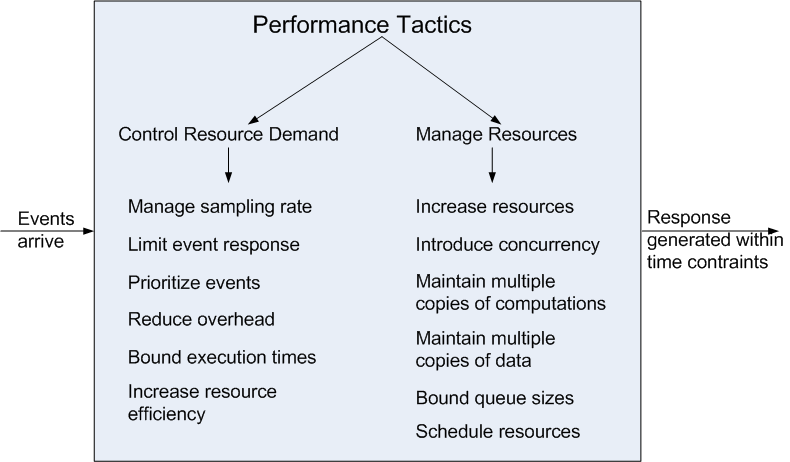
Performance is about time and the software system’s ability to meet timing requirements.

性能是关于时间的，是软件系统满足时序要求的能力。

1. **性能一般场景**



1. **性能战术**



1. **性能设计清单**
2. 采用合适的架构模式和战术
3. 建立性能模型/公式
4. 确定系统的职责，这将涉及重负载、具有时间关键的的相应需求、被大量使用的组件，或者影响系统中发生重负载或时间关键事件的部分。对于这些职责，确定：处理要求，并确定是否可能导致瓶颈；额外的责任来承认和处理请求，包括：处理跨越过程或处理器边界的职责，管理线程的职责、调度共享资源或管理管理与性能相关的队列、缓冲区和高速缓存的职责
5. 确定必须与每个协调的系统的元素，选择以下的通信和协调机制：支持任何引入的并发，事件优先化或调度策略；确保可以传递所需的性能响应；具有通信机制的适当属性；
6. 确定数据模型中将受到严重负载、具有事件响应请求、被大量使用的那些部分，或影响系统中发生重负载或时间关键事件的部分。对于这些数据抽象，确定：是否维护关键数据的多个副本将有益于性能、数据分区将有益于性能、是否可以减少处理要求/添加资源以减少枚举数据抽象的创建、初始化、持久性、操作、转换或销毁的瓶颈；
7. 确定系统中哪些资源对性能至关重要。对于这些资源，确保它们将在正常和过载的系统操作下进行监视和管理。

**Chapter 9: Security 安全性**

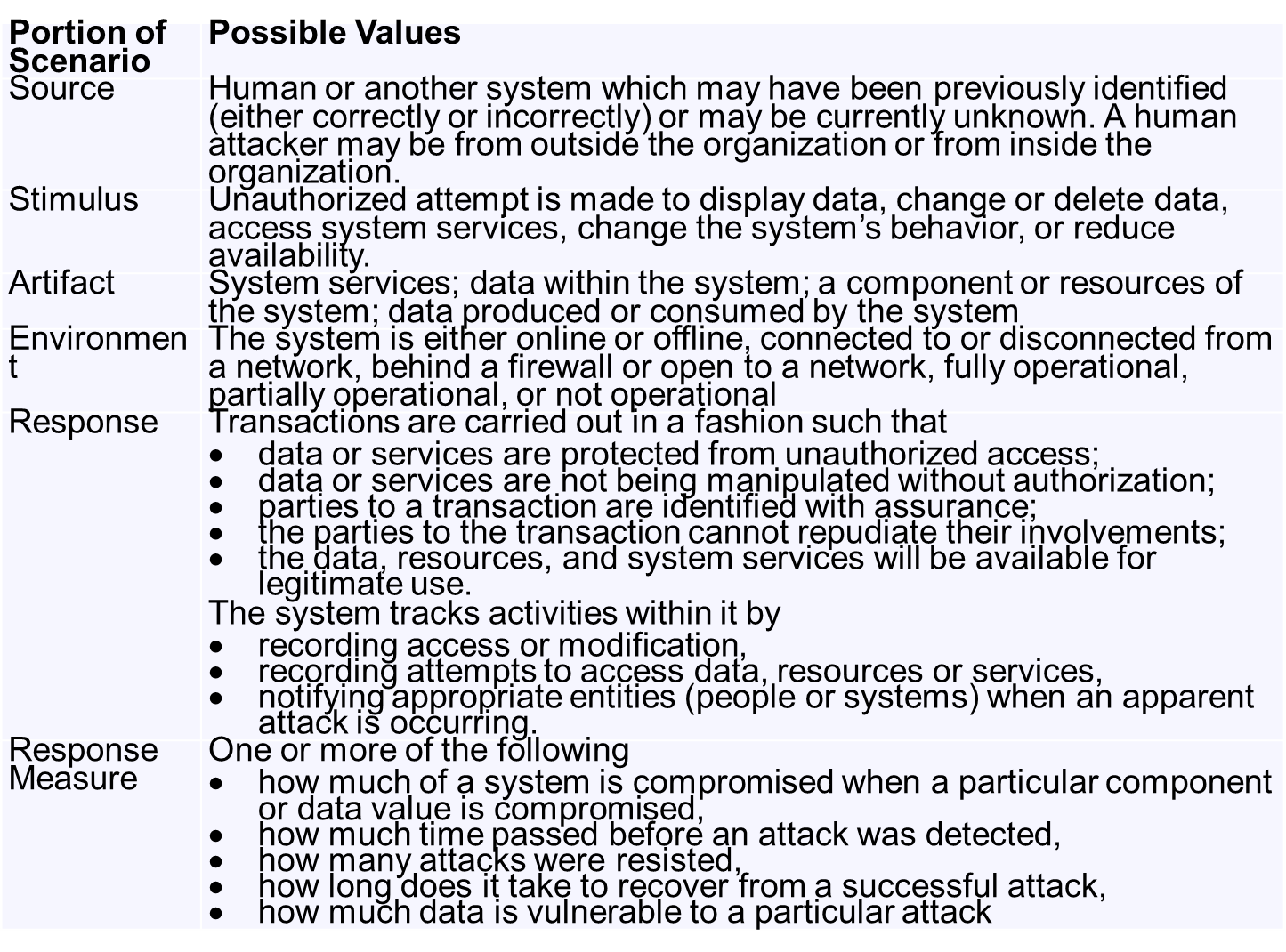
1. **安全性概念**

Security is a measure of the system’s ability to protect data and information from unauthorized access while still providing access to people and systems that are authorized.

安全性是系统保护数据和信息免受未授权访问的能力的度量，同时仍然提供被授权的人和系统的访问。

安全性有三个主要特征，称为CIA：confidentiality（私密性）、intergrity（完整性）、availability（可用性）。其他支持CIA的特征有：authentication（验证）、nonrepudiation（不可否认）、authorization（授权）。

1. **安全性一般场景**



1. **安全性战术**



1. **安全性设计清单**
2. 采用合适的架构模式和战术
3. 建立安全性模型/公式
4. 确定哪些系统职责需要安全性。确保已分配额外的职责来：表示、验证和授权参与者、授予或拒绝访问数据或服务、记录访问或修改数据或服务的尝试、加密数据、识别DOS攻击、从攻击中恢复、验证校验和和散列值

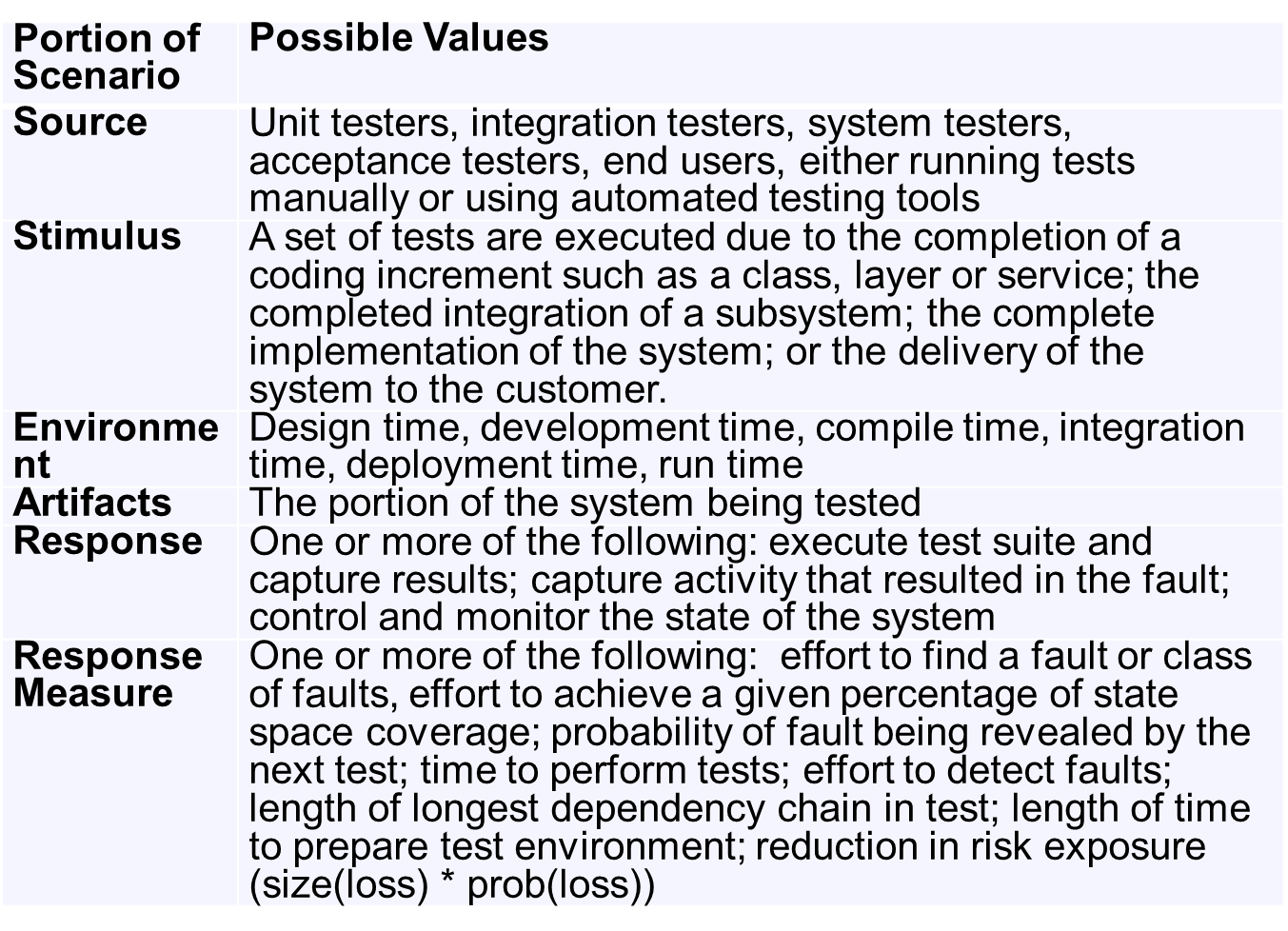
**Chapter 10: Testability 可测试性**

1. **可测试性概念**

Software testability refers to the ease with which software can be made to demonstrate its faults through testing. Specifically, testability refers to the probability, assuming that the software has at least one fault, that it will fail on its next test execution.

软件可测试性是指通过测试可以使软件易于演示其故障。具体来说，可测试性指的是假设软件具有至少一个故障，在下一次测试执行时失败的概率。

1. **可测试性一般场景**



1. **可测试性战术**



1. **可测试性设计清单**
2. 采用合适的架构模式和战术
3. 建立可测试性模型/公式
4. 确定哪些系统职责是最关键的，因此需要进行最彻底的测试
5. 确保分配了以下额外的系统职责：

执行测试套件并捕获导致故障的活动的结果和记录；控制和观察相关系统状态进行测试

**Chapter 11: Usability 易用性**

1. **易用性概念**

Usability is concerned with how easy it is for the user to accomplish a desired task and the kind of user support the system provides.

Usability comprises the following areas:

Learning system features.

Using a system efficiently.

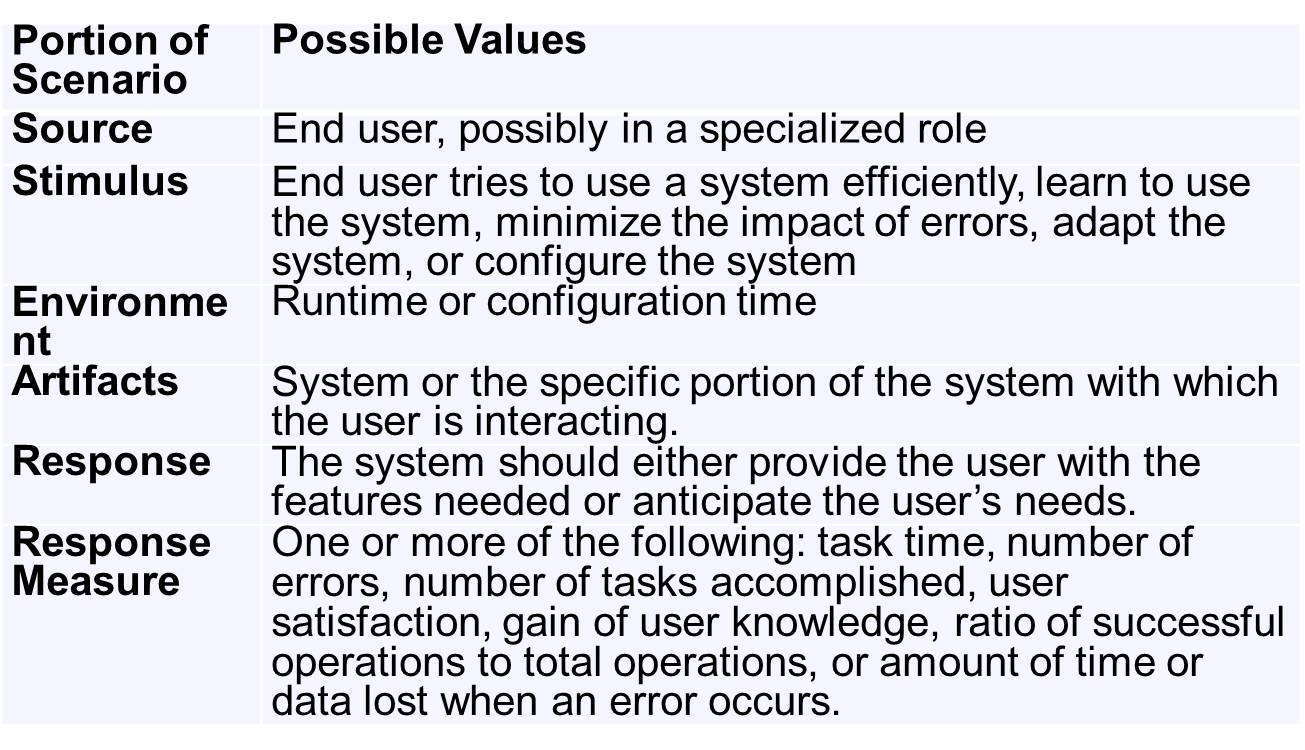
Minimizing the impact of errors.

Adapting the system to user needs.

Increasing confidence and satisfaction.

易用性涉及用户完成期望的任务有多容易，还有系统提供的用户支持的种类。可用性包括以下几个方面：学习系统特性。有效使用系统。最小化错误的影响。使系统适应用户需求。增加信心和满意度。

1. **易用性一般场景**



1. **易用性战术**



1. **易用性设计清单**
2. 确保根据需要分配了其他系统职责，以帮助用户：学习如何使用系统、有效地实现手头的任务、适应和配置系统、从用户和系统错误中恢复

**Chapter 12: Other Quality Attributes**

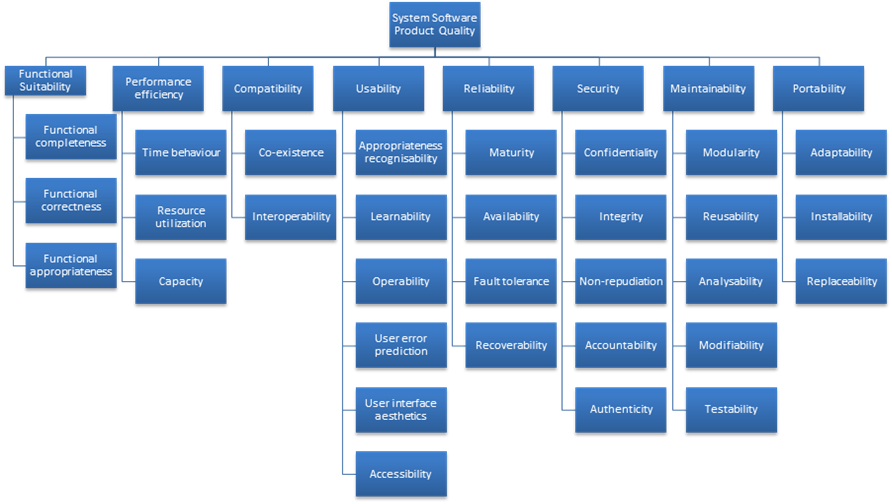
1. **其他软件质量属性**

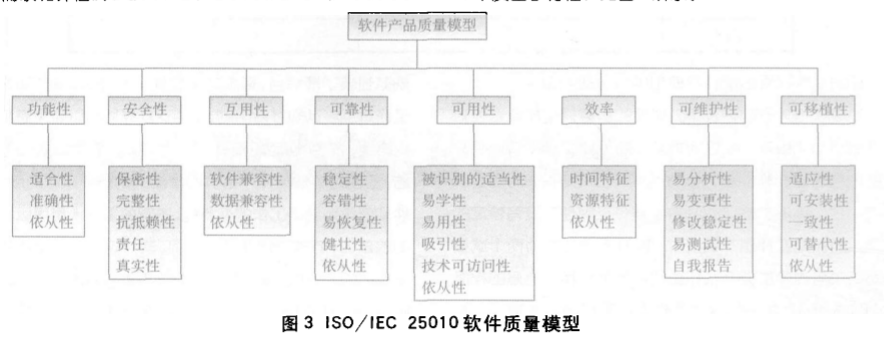
Variability（变异性）、portability（可移植性）、development distributability（开发可分布性）、scalability（可伸缩性）、deployability（可部署性）、mobility（可移动性）、monitorability（可监控性）、safety（安全）

1. **其他类别的质量属性**

Conceptual integrity（概念完整性）、marketability（市场性）、quality in use（使用质量）

1. **ISO/IEC FCD 25010 产品质量标准**





1. **如何处理未知的质量属性**
2. Model the quality attribute 对质量属性建模
3. Assemble a set of tactics for the quality attribute 为质量属性组装一套战术
4. Construct design checklists 构造设计清单

**Chapter 13: Patterns and Tactics**

1. **架构模式的概念**

An architectural pattern establishes a relationship between: A context，A problem and A solution.

架构模式建立了一个上下文、一个问题和一个解决方案它们之间的联系。

1. **Layer pattern 层次模式**

上下文：关注点的分离。

问题：需要对软件进行分段，使得模块可以单独开发和发展，部件之间几乎没有交互，支持可移植性，可修改性和重用。

解决方案：将软件以层为单位划分。 每个层是一组模块，提供一组连贯的服务。 使用必须是单向的。

1. **Broker Pattern 代理模式**

上下文：分布式服务彼此之间互操作。

问题：我们如何构建分布式软件以便服务用户不需要知道服务提供商的性质和位置就可以容易地动态更改用户和提供商之间的绑定？

解决方案：代理模式通过插入一个称为代理的中介，将服务的用户（客户端）与服务的提供者（服务器）分离。

1. **Model-View-Controller Pattern MVC模式（模型-视图-控制器模式）**

上下文：从模型中分离视图。

问题：如何将用户界面功能与应用程序功能分开，但仍然响应用户输入或底层应用程序数据的更改？ 当底层应用程序数据发生变化时，如何创建、维护和协调用户界面的多个视图？

解决方案：模型 - 视图 - 控制器（MVC）模式将应用程序功能分为三种类型的组件。

1. **Pipe and Filter Pattern 管道-过滤器模式**

上下文：处理数据流。

问题：这样的系统需要被分成可重复使用的松散耦合的组件，这些组件具有简单的通用的交互机制。

解决方案：管道-过滤器模式中的交互模式的特征在于数据流的连续变换。

1. **Client-Server Pattern 客户端-服务器模式**

上下文：有大量分布式客户端希望访问的共享资源和服务。

问题：我们希望通过集中控制这些资源和服务来提高可伸缩性和可用性，同时将资源本身分布在多个物理服务器上。

解决方案：客户端通过请求服务器的服务进行交互，服务器提供一系列服务。

1. **Peer-to-Peer Pattern P2P模式**

上下文：分布式计算实体被认为是对等的。

问题：一组“相等”的分布式计算实体如何通过公共协议相互连接，以便它们能够以高可用性和可扩展性组织和共享其服务？

解决方案：在P2P模式中，组件直接作为对等体进行交互。所有对等体是“相等的”，并且没有对等体或对等体组对于系统的健康是至关重要的。

1. **Service Oriented Architecture Pattern 面向服务的架构模式（SOA）**

上下文：许多服务互操作，但是对它们的实现没有任何详细的了解。

问题：我们如何支持在不同平台上运行并以不同实现语言编写的、由不同组织提供并分布在Internet上的分布式组件的互操作性？

解决方案：面向服务的体系结构（SOA）模式描述了提供和/或使用服务的分布式组件的集合。

1. **Publish-Subscribe Pattern 发布-订阅模式**

上下文：数据生产者和消费者的确切数量和性质不是预先确定的或固定的，也不是它们共享的数据。

问题：我们如何创建支持在生产者和消费者之间传输消息的能力的集成机制，使他们不知道对方的身份甚至存在？

解决方案：在发布-订阅模式中，组件通过已发布的消息或事件进行交互。

1. **Shared-Data Pattern 共享数据模式**

上下文：各种计算组件需要共享和操作大量的数据。

问题：系统如何存储和操作由多个独立组件访问的持久数据？  
解决方案：在共享数据模式中，交互主要由多个数据访问器和至少一个共享数据存储之间的持久数据交换所主导。

1. **Map-Reduce Pattern 映射-减少模式**

上下文：需要快速分析大量数据。

问题：有效地执行大型数据集的分布式并行排序，并为程序员指定要执行的分析提供了一种简单的方法。

解决方案：map-reduce模式需要三个部分：一个负责根据需要分配数据的专用基础设施。一个用于过滤数据以检索项目的map。一个结合了映射结果的reduce。

1. **Multi-Tier Pattern 多级模式**

上下文：将系统的基础结构分布到不同的子集中。

问题：我们如何将系统分成许多计算独立的执行结构：由一些通信介质连接的软件组和硬件组？

解决方案：许多系统的执行结构被组织为一系列组件的逻辑分组。 每个分组被称为级。

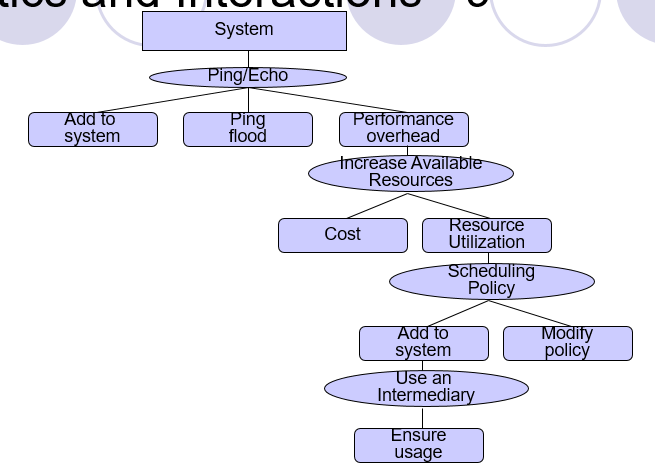
1. **战术与模式之间的关系**

Patterns are built from tactics; if a pattern is a molecule, a tactic is an atom. Tactics Augment Patterns.

模式是从战术建立起来的。如果模式是一个分子，那么战术就是一个原子。战术增强了模式。模式解决了一个特定的问题，但它是中立的或者存在其他质量问题。使用战术可以缓解这些问题，减轻副作用。

每次使用战术都会引入新的关注。每一个新的关注都会引起新的战术。

最后，每个战术的副作用变得足够小到可以忽略。

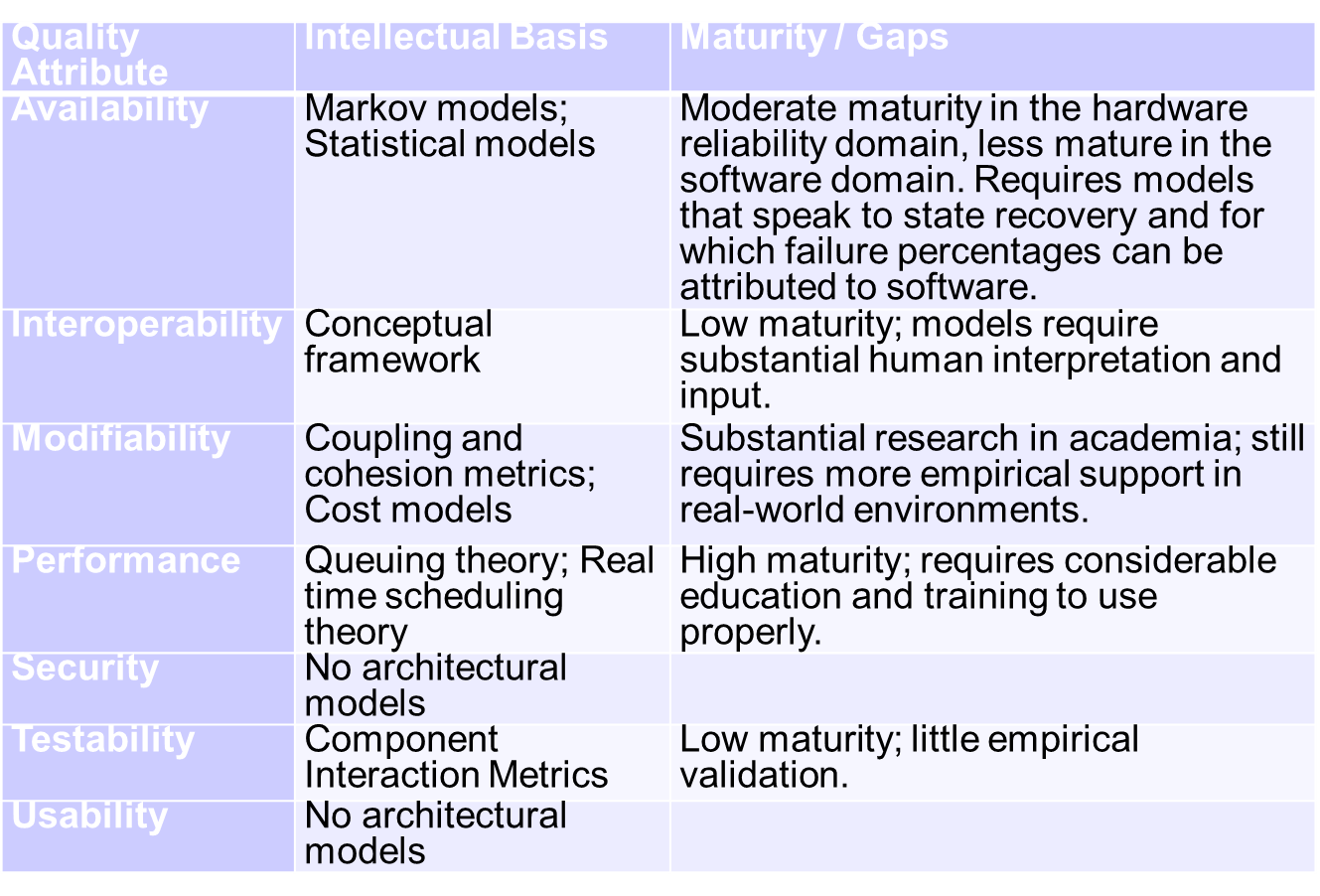


**Chapter 14: Quality Attribute Modeling and Analysis**

1. **Quality Attribute Modeling 什么是对质量属性建模？**

一些质量属性具有可被用于辅助分析的、已被充分理解的、经时间测试的分析模型。有分析模型，意味着支持定量分析**。**主要是确定参数。

1. **Maturity of Quality Attribute Models 质量属性模型的成熟度**



1. **Thought Experiment Steps 思想实验的步骤**
2. 枚举用例的步骤
3. 在每一步，问自己和架构师：正在实施什么机制来支持实现哪些特定的质量要求？这个机制是否阻碍了其他属性的实现？
4. 记录问题，以供后续深入分析或原型构建
5. **在生命周期的不同阶段进行不同类型的分析**
6. 需求阶段：分析模型和背景分析可以帮助规划容量；检查表可以帮助确保捕获正确的要求
7. 设计阶段：原型可以帮助探索设计选项
8. 实现阶段：实验和合成负载测试可以在实现过程中或字段化之后使用
9. 字段化阶段：监视器可以在字段化之后使用，以确定实际行为并发现瓶颈

**Chapter 15: Architectures in Agile Projects**

1. **敏捷过程的思想**

敏捷过程是为以下这些项目服务的：对其涉众更加敏感；更快地开发用户关注的功能；在项目生命周期中显示更多和更早的进展；通过文档记录减轻负担；

敏捷项目关注问题：“我应该做多少架构？”、“我应该记录多少架构？”

1. **Twelve Agile Principles 12条敏捷原则**
2. Highest priority is to satisfy the customer through early and continuous delivery. 最高优先级是通过早期交付和持续交付来满足客户的
3. Welcome changing requirements, even late in development. 欢迎不断变化的需求，甚至是在开发晚期
4. Deliver working software frequently with a preference to the shorter timescale. 经常提交工作中的软件，时间范围越小优先级越高
5. Business people and developers work together daily throughout the project. 业务人员和开发人员每天在整个项目中工作
6. Build projects around motivated individuals. 围绕积极的个人构建项目
7. Face-to-face conversation. 面对面交谈
8. Working software is the primary measure of progress. 工作中的软件是进度的主要衡量标准
9. Agile processes promote sustainable development. 敏捷过程促进可持续发展
10. Continuous attention to technical excellence and good design. 持续关注卓越的技术和良好的设计
11. Simplicity—the art of maximizing the amount of work not done—is essential. 简单是至关重要的---最大化未完成工作量的艺术
12. Self-organizing teams. 自组织的团队
13. At regular intervals, the team reflects on how to become more effective. 团队定期反思如何变得更高效
14. **Agility and Documentation 敏捷开发与架构编档**

敏捷项目中的文档是为读者写的，如果读者不需要它，不要写它。但记住，读者可能是一个维护者或者还没在项目里的新成员。

1. **Agility and Architecture Evaluation 敏捷开发与架构评估**

架构评估是敏捷过程的一部分。满足涉众重点关注是敏捷哲学的基石。架构评估方法由架构权衡分析方法（ATAM）来说明。ATAM确定了涉众一组最重要的关注点。

**Chapter 16: Architecture and Requirements**

1. **ASR**

An architecturally significant requirement (ASR) is a requirement that will have a profound effect on the architecture.

ASR是对架构有着深远影响的需求

1. **ASR的几种获取方法**
2. Gathering ASRs from Requirements Documents 从需求文档中获取
3. Gathering ASRs by Interviewing stakeholders 通过采访涉众

涉众访谈的结果应该包括：架构驱动程序列表，一组涉众优先考虑的QA场景

1. Gathering ASRs by Understanding the Business Goals 通过理解业务目标
2. Capturing ASRs in a Utility Tree 在效用树中捕获
3. **Quality Attribute Workshop（QAW）质量属性研讨会**

QAW是一种便利的，以涉众为中心的方法，用于在软件体系结构完成之前生成、优先排序和改进质量属性场景。

QAW专注于系统级的关注，特别是软件在系统中的作用。

QAW的步骤：

(1)QAW演示和介绍; (2)业务/任务介绍; (3)架构计划演示; (4)架构驱动程序的识别; (5)场景头脑风暴; (6)场景合并; (7)场景优先级; (8)场景优化

**4. PALM**

PALM: A Method for Eliciting Business Goals 一种引导业务目标的方法

通常在研讨会里进行一天半，由架构师和涉众参与，说明相关的业务目标

PALM的七个步骤：

(1)PALM overview presentation PALM概述介绍

(2)Business drivers presentation. 业务目标介绍

(3)Architecture drivers presentation 架构驱动介绍

(4)Business goals elicitation 业务目标引出

(5)Identification of potential quality attributes from business goals. 从业务目标中

(6)Assignment of pedigree to existing quality attribute drivers. 将谱系分配给现有的质量属性驱动

(7)Exercise conclusion 应用结论

**5. Utility Tree 效用树**

效用树是一种记录ASR的方法，确定每个ASR的优先级。ASR作为场景被捕获。树的根是占位符节点称为“实用程序”。第二级树包含广泛的质量检查类别。第三级树精化这些类别。

**Chapter 17: Designing an Architecture**

1. **Generate and Test 架构设计中的假设-检验法**
2. 把当前的设计当作一个假设
3. 询问当前的设计是否已经满足需求（检验）
4. 如果不是，产生一个新的假设，迭代
5. **初始的假设从哪里来？**

可靠的来源：已经存在的系统；框架

不太可靠的来源：模式和战术；域分解；设计清单

1. **如何测试假设？**

已经涵盖的分析技术；质量属性讨论的分析策略；ASR

测试的输出是什么？ 当前设计未满足的需求列表

1. **如何产生下一次假设？**

添加缺少的职责；使用战术来调整假设的质量属性行为

1. **什么时候完成假设检验法？**

所有的ASR都被满足。或者，设计活动的预算用完。

1. **The Attribute-Driven Design Method（ADD）属性驱动设计方法**

**包装了许多已经讨论过的技术；是一个迭代的方法，每次迭代选择系统的一部分进行设计，整顿该部分的ASR；ADD不会产生完整的设计，而是产生带有职责的容器以及容器之间的交互和信息流；不为容器生产API或签名发布；**



1. **ADD的步骤**
2. Choose an element of the system to design. 选择要设计的系统元素
3. Identify the ASRs for the chosen element. 标识所选元素的ASR
4. Generate a design solution for the chosen element. 为选定的元素生产设计解决方案
5. Inventory remaining requirements and select the input for the next iteration. 清点剩下的需求并选择下一次迭代的输入
6. 重复步骤1-4，直到满足所有ASR

**Chapter 18: Documenting Software Architectures**

1. **架构文档的重要性**

文档是交流的基础，生命周期远长于代码。文档帮助涉众理解整个软件架构的设计和实现。架构文档既是规范性的又是描述性的。

1. **架构文档的读者**

新的雇员，开发者，分析师

1. **架构文档的用途**
2. 教育：把系统介绍给别人
3. 涉众之间的主要沟通手段：特别是架构师和开发人员之间、架构师和未来的架构师之间
4. 系统分析和构建的基础
5. **View 视图**

视图将软件架构分为系统的多个可管理的标识。不同的视图支持不同的目标和用途。你应该记录的视图取决于你对文档的期望使用。每个视图都有成本和收益，你应该确保维护视图的好处超过其成本。

1. **Module Views 模型视图**

元素：module

关系：is part of, depends on, is a

约束：拓扑约束、行为约束

用法：构建代码的蓝图、受变化影响的分析、规划增量开发、需求跟踪分析

工作分配，时间表和预算信息

如果没有至少一个模块视图，任何软件体系结构的文档都不可能是完整的。

1. **C&C views C&C视图**

元素：components, connectors

关系：attachments, interface delegation

约束：组件只能连接到连接器，连接器只能连接到组件，只能在兼容端口和角色之间进行附件、接口委托只能在两个兼容端口（或两个兼容角色）之间定义，连接器不能孤立出现，连接器必须连接到组件

用法：显示系统如何交互。帮助推理运行时系统的质量

1. **Allocation Views 分配视图**

元素：软件元素、环境元素

关系：分配到

约束：随视图而变化

用法：推理性能、可用性、安全性和安全；推理分布式开发和团队的工作分配；推理软件版本的并发访问；推理系统安装的形式和机制

1. **Method for Choosing the Views 选择视图的方法**
2. Build a stakeholder/view table. 建立一个涉众/视图表格

行为涉众，列为应用于系统的视图；

1. Combine views to reduce their number 结合视图，减少视图数量

有一些经常被自然合并的：各种C&C视图、无论是SOA还是通信过程视图的部署视图、分解视图、分层视图

1. Prioritize and stage. 优先级和阶段

分解视图尽早发布、不必满足所有信息需求、开始另一个新视图之前不必一定要完成当前的一个视图

1. **Documenting a View 对视图进行编档**
2. The Primary Presentation. 主要演示

显示视图的元素和元素之间的关系。主要演示最常见的是图形。

1. The Element Catalog 元素目录

元素目录至少详细描述了主演示文稿中描述的那些元素。

1. Context Diagram. 上下文图

该视图中描述的系统如何与环境相关

1. Variability Guide. 变异性指南

指出那些地方容易发生变异

1. Rationale. 理由

选择该视图的理由。以及选择架构中的战术的问题和理由。

1. **Documenting Information Beyond Views 对视图之外的信息编档**
2. Documentation Roadmap. 文档路线图

什么信息在哪里、哪里可以找到它。范围和摘要。如何组织文档。查看概述。涉众如何使用文档。

1. How a View Is Documented. 如何记录视图

说明你用于记录视图的标准组织。

1. System Overview. 系统概述

系统功能、用户和重要的背景或约束的简短描述、为读者提供一致的系统心理模型和目的。

1. Mapping Between Views. 视图之间的映射

帮助读者深刻洞察架构是如何作用一个统一的概念整体工作的。架构中跨视图的元素之间的关联通常是多对多的。视图到视图的关联可以列成表格。

1. Rationale 理由

记录适用于多个视图的体系架构决策

1. Directory. 目录

一套参考资料、术语索引、词汇表、首字母缩略词

1. **Documenting Behavior 对行为编档**

行为文档通过描述每个视图中的架构元素如何相互交互来补充每个视图。行为文档允许推理：一个系统的潜在死锁、系统在期望的时间量内完成任务的能力、最大内存消耗等等。

对行为编档可以用面向轨迹的语言。轨迹描述了结构元件之间的特定的活动或相互作用序列。比如：用例图、顺序图、通信图、活动图、消息序列表、时序图、业务过程执行语言。

对行为编档可以用综合语言、综合模型，这类文档可以推理从起始状态到结束状态的所有可能路径，比如：状态机。

1. **Documenting Quality Attributes 对质量属性编档**

质量属性在文档中显示在哪里？

理由。提供服务的架构元素通常具有分配给它们的质量属性边界。架构文档通常包含映射到需求，显示如何满足需求。路线图告诉涉众在文档中查找问题的位置。

**Chapter 19: Architecture, Implementation, and Testing**

1. **帮助保持代码和架构一致的4种技术**
2. Embedding the design in the code 将设计嵌入代码中
3. Frameworks 框架
4. Code templates 代码模板
5. Keeping code and architecture consistent (i.e. avoiding “architecture erosion”) 保持代码和架构一致（即 避免“架构侵蚀”）
6. **Embedding the design in the code 将设计嵌入代码中**

架构作为实现的蓝图。意味着：实现者要知道他们正在实现什么样的架构；他们可以把代码中的架构结构记录为注释，然后任何人拿起代码就能知道一些约束；在所有级别里都使用注释；项目中要有关于如何做到这一点的约定，然后工具可以自动关联代码和架构。

1. **Framework 框架**

框架是围绕特定主题组织的一组可重用的类。程序员使用框架提供的服务。

1. **Code template 代码模板**

代码模板是代码的集合，是程序员提供的应用的特定部分。

代码模板的优点：具有相似性质的组件以相似的形式表现；模板只需要调试一次；复杂的部分可以由熟练的人员完成并且交付给不太熟练的人员。

1. **Keeping code and architecture consistent (i.e. avoiding “architecture erosion”)** 保持代码和架构一致（即 避免“架构侵蚀”）

使用工具强制架构约束：可以添加在构建或检入期间强制执行的体系架构规则；将侵蚀发生时的文件标记为过期，给予剩余部分更多的信任；调度文档/代码同步时间。

1. **架构师在测试中的角色**

架构师应该积极参与：

1. 测试计划。因为架构师知道系统的敏感区域。
2. 测试开发。测试驱动开发是一种技术，系统的下一个增量被开发以满足预定测试。
3. 测试解释。架构师知道各种检测值应该是什么，并且最适合解释测试结果。
4. 测试工具构建。测试工具必须与系统密切交互，这需要架构知识。

**Chapter 20: Architecture Reconstruction and Conformance**

1. **架构重构的背景**
2. 系统已经存在但是你不知道它的架构：没有记录/文档记录丢失/文档记录和系统发展不同步
3. 你如何维护这样一个系统？
4. 你如何管理它的发展以维持其架构提供的质量属性？
5. **架构重构的目的**
6. 记录一个文档从未存在或文档已经过时的架构
7. 确保完成时的架构和设计的架构之间的一致性
8. 在架构重构中，构建的架构是从现有系统组件反向设计过来的
9. **架构重构的阶段**
10. Raw view extraction. 原始视图提取
11. Database construction. 数据库构建
12. View fusion and manipulation. 视图融合和操纵
13. Architecture analysis. 架构分析

**Chapter 21: Architecture Evaluation 架构评估**

1. **架构评估的三种形式及其特点**
2. Evaluation by the designer within the design process. 在设计过程中由设计师进行评估

每当设计师做出关键设计决策时，都要对所选择的和竞争的替代品进行评估。设计师的评估是架构设计的“生成-测试”方法中的“测试”部分。要进行多少分析决定于决定的重要性。

1. Evaluation by peers within the design process. 在设计过程中由同行进行评估

可以在设计过程中的任何时间点执行同行评估。分配至少几个小时甚至半天。

1. Analysis by outsiders once the architecture has been designed. 架构设计完成后，外部人员进行分析

外部评估者眼光客观。“外”是相对的。选择外部人是因为他们拥有专门的知识或经验、或成功评估架构的长期经验。管理者往往倾向于倾听外部团队发现的问题。

1. **The Architecture Tradeoff Analysis Method (ATAM) 架构权衡分析法**

ATAM的设计使得评估者不需要熟悉架构或其业务目标，系统不需要构建，并且可能有大量的涉众。

1. **ATAM的参与者**
2. The evaluation team. 评估小组

三到五人。从项目外部对其架构进行评估。

1. Project decision makers. 项目决策者

通常包括PM、客户、架构师

1. Architecture stakeholders. 架构涉众

包括开发人员、测试人员、集成人员、维护人员、性能工程师、用户、建设者等等

1. **ATAM的输出**
2. 简洁的架构演示
3. 业务目标的说明
4. 表示为质量属性场景的优先级质量属性
5. 一组风险和非风险
6. 一组风险主题
7. 将架构决策映射到质量要求
8. 一组确定的灵敏度和权衡点
9. **ATAM的步骤**
10. Present the ATAM 呈现ATAM

评估领导向项目代表介绍ATAM。使用标准演示文稿，简要介绍ATAM步骤以及评估结果。

1. Present Business Drivers 提供业务驱动因素

参与评估的每个人了解系统背景以及促进其发展的主要业务驱动因素。项目决策者从业务角度介绍系统概述。

1. Present the Architecture 呈现架构

首席架构师（或架构团队）做一个描述架构的演示。描述用于满足要求的架构方法。提供有助于理解系统最重要的质量属性的观点。

1. Identify Architectural Approaches 确定架构方法

评估小组对已确定的模式和战术进行编目。

1. Generate Utility Tree 生成效用树

正在考虑的架构重要质量属性在步骤2中被命名。在此步骤中。评估小组与项目决策者合作，确定、优先排序和完善系统最重要的质量属性目标。将重要质量属性表示为场景，填充效用树的叶节点。

1. Analyze Architectural Approaches 分析架构方法

评估小组每隔一段时间检查最高排名的场景，要求架构师解释架构如何支持每个场景。评估小组探查架构师用于执行场景的架构方法。

1. Brainstorm and Prioritize Scenarios 头脑风暴和优先化场景

涉众集体讨论了在涉众的个人观点中具有操作意义的场景。收集场景后，通过投票确定优先级。

1. Analyze Architectural Approaches 分析架构方法

在此步骤中，评估小组使用最高排名新生成的方案执行与步骤6相同的活动。

1. Present Results 呈现结果

评估小组把风险分组为风险主题。对于每个风险主题，确定在步骤2中列出的业务驱动因素是否受到影响。从评估中收集的信息被汇总并提交给涉众。

提出以下输出：

架构方法记录、一组场景以及其各自在头脑风暴中的优先级、效用树、发现的风险、非文件记录、找到的灵敏度点和平衡点、风险主题和业务驱动因素受到的威胁

1. **Lightweight Architectural Evaluation 轻量级架构评审**

ATAM是一项重大任务，需要大约20-30个 人-天 的努力。

给予ATAM的轻量级架构评估方法、用于更小、更低风险的项目。

会议可以一天甚至半天内完成。可以完全由组织内部的成员执行。轻量级架构评估的步骤和阶段可以更快地执行。典型的会议例程是4-6个小时。

**Chapter 26: Architectures for the Cloud 云架构**

1. **云的基础定义**

On-demand self-service. 按需自助服务

Ubiquitous network access. 无处不在的网络接入

Resource pooling. 资源池

Location independence. 位置独立性

Rapid elasticity. 快速弹性

Measured service. 测量服务

Multi-tenancy. 多租户

1. **云的基本服务模型**
2. Software as a Service (SaaS).
3. Platform as a Service (PaaS).
4. Infrastructure as a Service (IaaS).
5. **云的部署模型**

Private cloud. 私有云；Public cloud. 公有云；Community cloud. 社区云；Hybrid cloud. 混合云

1. **云的主要机制**

Hypervisor 管理程序；Virtual Machine 虚拟机；File system 文件系统；Network 网络

1. **云的主要技术**

Virtual Memory Page Table 虚拟内存页表；Hypervisor Manages Virtualization 管理程序管理虚拟化；HDFS Hadoop分布式文件系统；IaaS; PaaS; DataBases

1. **IaaS (Infrastructure as a Service)**

An arrangement of servers that manages the base technologies.

管理基础技术的服务器的布置

服务器以集群形式排列；一些服务器作为IaaS的基础结构；每个服务器都有一个管理程序作为其基础。

IaaS的组件：

Cluster Manager：负责管理每个集群；

Persistent Object Manager：管理持久性

Virtual Resource Manager：管理其他资源，充当信息的网关

The File System Manager：与HDFS类似，管理网络文件系统

IaaS提供的服务：

在底层虚拟机实例故障的情况下自动重新分配IP地址。

自动缩放。 根据负载创建或删除新虚拟机。

1. **PaaS(Platform as a Service)**

为开发人员提供集成堆栈；PaaS管理对堆栈底层的分配。

1. **Database**

关系型数据库不适用，引入新的数据模型：键值对，以文档为中心

Key Value – HBase

一列指定为键，其他全都值。没有模式，所以数据可以有键+其他所有值。

Hbase不支持事务。时间戳用于在事实之后检测碰撞。

Document Centric – MongoDB

存储对象而不是数据。通过包含对象访问数据。对象还可以包含指向其他对象的链接。没有主要索引或次要索引的概念。

1. **云架构的质量属性**
2. Security 安全性

-多租户引起了对非云环境的额外关注：无意中的信息共享、虚拟机“逃离”、侧信道攻击、拒绝服务攻击

-当决定在云中托管什么应用程序时，组织需要考虑风险

1. Performance 性能

自动缩放在负载增长时提供额外的性能。新资源的响应时间可能不适合，架构师需要了解应用程序的资源需求。

1. Availability 可用性

故障是云系统的常见事件。云提供商必须在出现一些异常时确保云本身仍然可用。应用程序开发人员必须假设实例会失败，并在失败的情况下构建检测和纠正体制。