

软件架构师研修讲座

胡协刚

软件架构师 UML/RUP专家

内容提要

- 小故事：七人分粥
- 当前软件团队的开发现状和面临的问题
- 软件项目的特点
- 解决之道：从瀑布模型到迭代模型
- 解决项目风险的关键——构架基线
- 构架基线原理：为大量的构造提供坚实的框架基础
- 不是所有的产品都容易得到构架基线
- 参考构架模型
- 以构架为中心的开发，对团队能力提出了严峻挑战
- 自己摸索还是培训、咨询和教练
- 架构师网和东软培训中心联合推出高端培训服务

小故事： 七人分粥

小故事：七人分粥

➤如何对权力制约的制度问题一直是人类头疼的难题。请看下边的这个小故事——

有7个人组成了一个团体共同生活，其中每个人都是平凡而平等的，没有什么凶险祸害之心，但不免自私自利。他们想用非暴力的方式，通过制定制度来解决每天的吃饭问题——要分食一锅粥，但并没有称量用具和有刻度的容器。

委托一人分粥

➤大家试验了不同的方法，发挥了聪明才智、多次博弈形成了日益完善的制度。大体说来主要有以下几种：

方法一：拟定一个人负责分粥事宜。很快大家就发现，这个人為自己分的粥最多，于是又换了一个人，总是主持分粥的人碗里的粥最多最好。阿克顿勋爵作的结论是：权力导致腐败，绝对的权力绝对腐败。

还是轮流分粥吧

方法二：大家轮流主持分粥，每人一天。这样等于承认了个人有为自己多分粥的权力，同时给予了每个人为自己多分的机会。虽然看起来平等了，但是每个人在一周中只有一天吃得饱而且有剩余，其余6天都饥饿难挨。认为这种方式导致了资源浪费。

德高望重的人或者分粥委员会

方法三：大家选举一个信得过的人主持分粥。开始这品德尚属上乘的人还能基本公平，但不久他就开始为自己和溜须拍马的人多分。不能放任其堕落和风气败坏，还得寻找新思路。

方法四：选举一个分粥委员会和一个监督委员会，形成监督和制约。公平基本上做到了，可是由于监督委员会常提出多种议案，分粥委员会又据理力争，等分粥完毕时，粥早就凉了。

其实最好的办法往往很简单

方法五：每个人轮流值日分粥，但是分粥的那个人要最后一个领粥。令人惊奇的是，在这个制度下，7只碗里的粥每次都是一样多，就像用科学仪器量过一样。每个主持分粥的人都认识到，如果7只碗里的粥不相同，他确定无疑将享有那份最少的。

对软件开发过程启示？

➤分粥制度的原理：

分粥的过程可以细分为：分粥和领粥两个步骤；（如果不能分步骤，估计谁也没招了）

通过对领粥步骤来设计某种规则（分粥的那个人要最后一个领粥），促使分粥步骤能够做得更公平

➤启示：

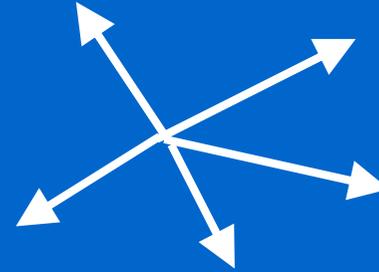
我们通过设计一套合适的软件过程（方法和技术），来解决软件开发中遇到的问题

当前软件团队的开发 现状和面临的问题

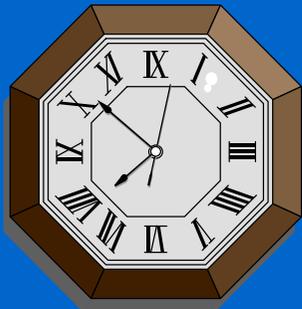
软件开发的现状



世界经济对软件的依赖性越来越强



软件应用的规模、复杂度和分布程度越来越高



现实业务对软件开发的生产率和质量提出了更高的要求



缺乏足够的称职的开发人员

软件开发已经成为一种团队工作

挑战Challenges

- 更大规模的团队
Larger teams
- 更加专业化
Specialization
- 更为分散Distribution
- 技术变化发展更为迅速Rapid technology change



Analyst



Performance Engineer



Project Manager



Developer



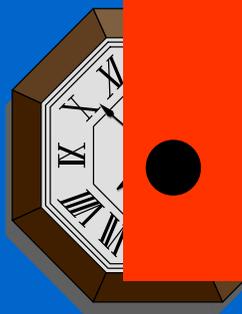
Tester



Release Engineer

我们做得怎样？

- 有不少成功
- 更多的是失败



Tester



Release
Engineer

软件开发常见问题的症状Symptoms

- 错误地理解最终用户的需要
- 面对变化中的需求无计可施
- 开发出一些无法整合为一体的软件模块
- 难以维护和扩展的软件系统
- 在后期才发现项目中所存在的严重缺陷
- 低下的软件质量
- 不能接受的软件性能
- 项目成员各自为政，搞不清楚谁在何时、何处、为了什么而变更了哪些内容

软件项目的特点

复杂性complexity

- ✓ 《人月神话》根据软件复杂性的内在性质分为两类：
 - 附加复杂性accidental complexities——并非软件本身固有的、由其它外在因素所附加的额外复杂性，理论上可以被降低甚至完全被消除；
 - 固有复杂性essential complexities——软件本身固有的本源特性，理论上不可能被消除，但可以通过相关技术降低其负面影

一致性conformity

- ✓ 《人月神话》认为软件很难获得和保持其一致性
- 复杂性不是软件才有，一个大型的建筑工程同样极其复杂，然而建筑的基本原则是简单和一致的；
- 软件不存在一种途径或方法，可以同时解决所有的问题；因为软件的每个范畴都是独特的（例如一个OO系统中，为了满足性能要求，仍然要保留部分存储过程），找不到统

可变性changeability

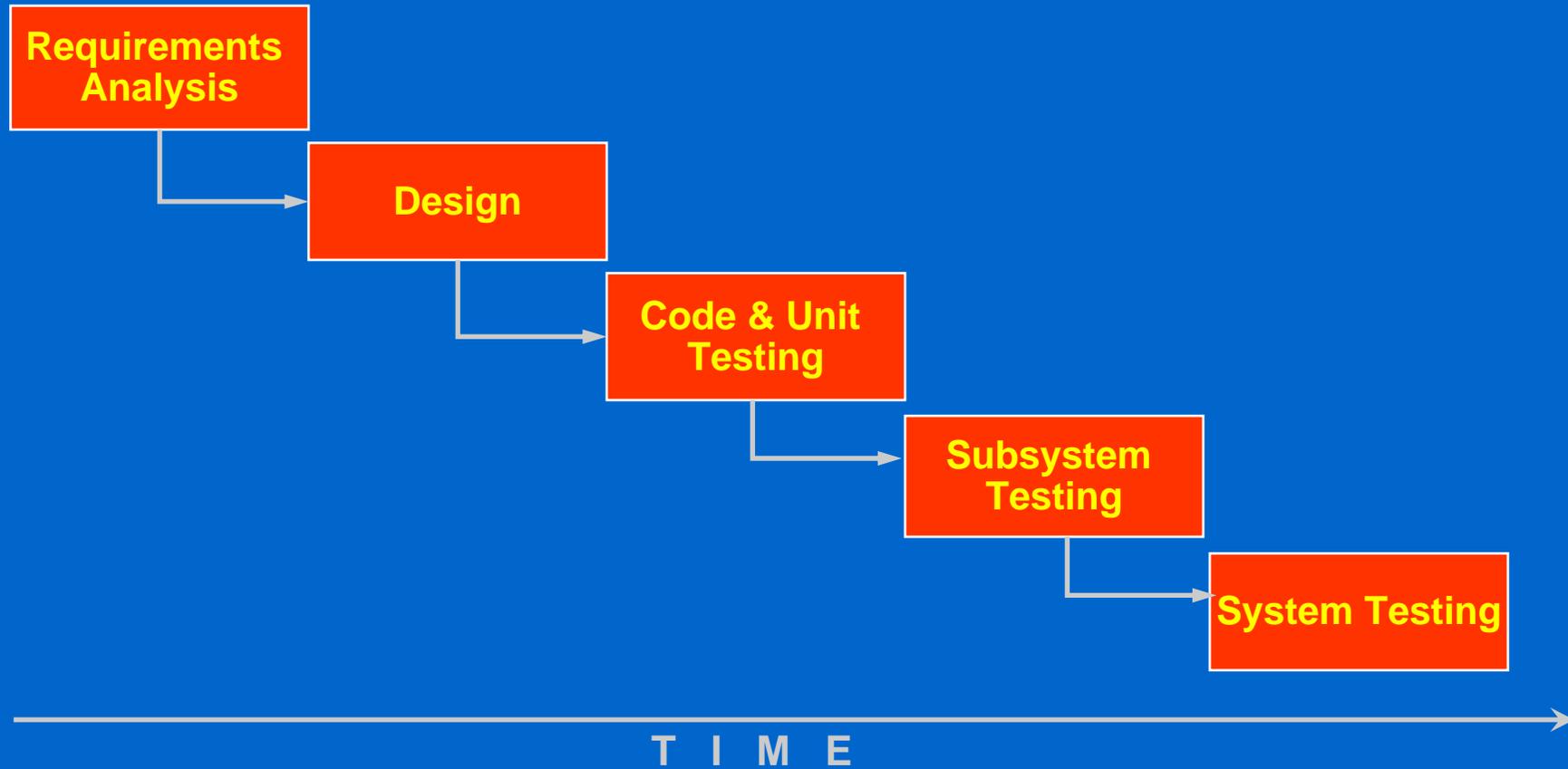
- ✓ 《人月神话》认为软件的可变性是独一无二的
- 建筑等其它产品同样面临变化的需求压力，但远不如软件来得那么突出
- 软件是“软”的，理论上可以支持各种变化，更容易让客户有不切实际的变更冲动
- 软件的环境易变：可能被部署到新的平台下；支撑业务在不断发展
- 可变性的背后还有不确定性

不可视性invisibility

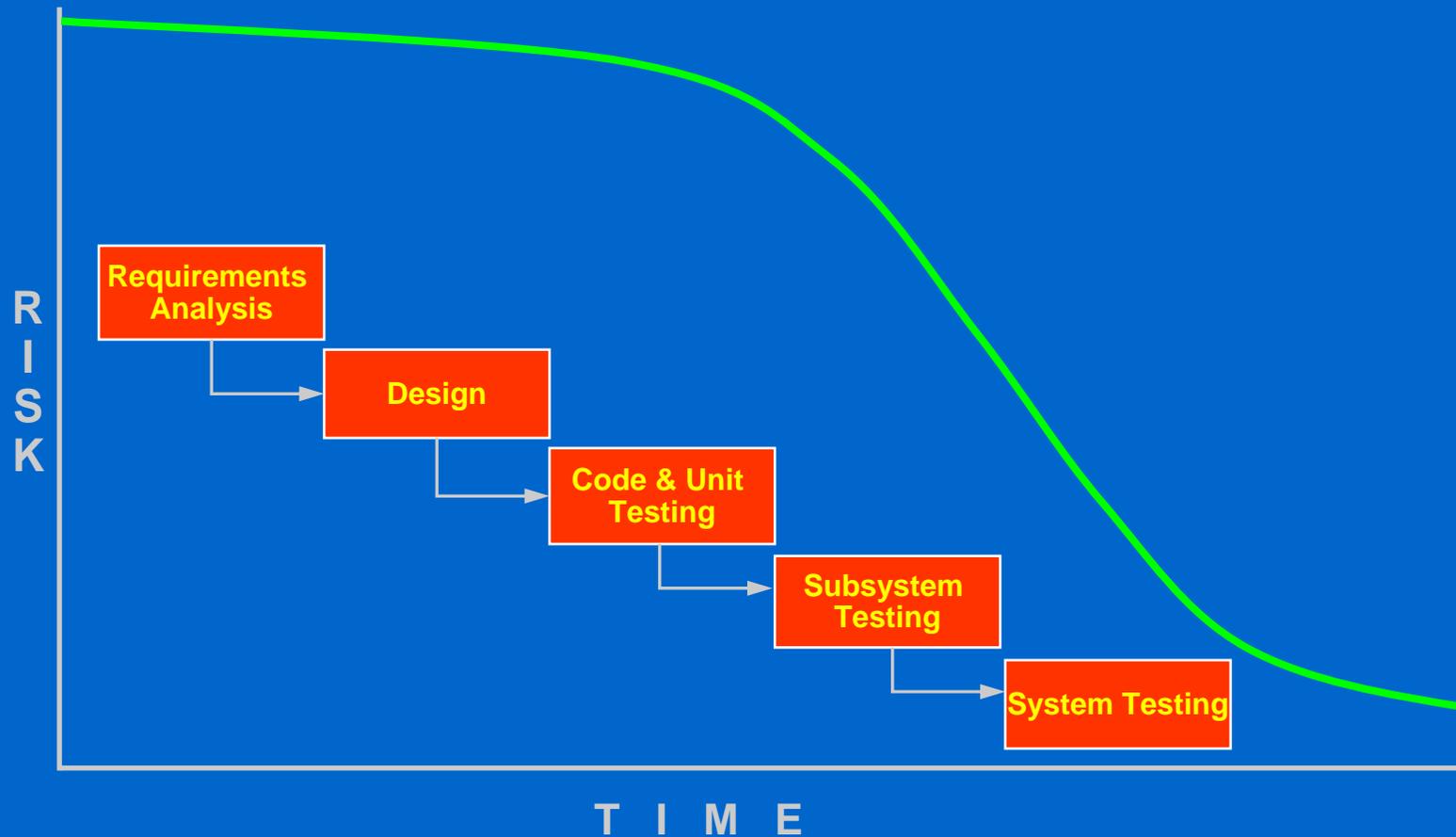
- ✓ 《人月神话》认为软件不可见
- 软件在开发出来之前都不存在物理的形态
- 软件只有在使用过程中，通过交互表现出其行为形态
- 解决不可视性的最可靠途径是：构建软件的交付，并进行测试

解决之道：从瀑布模型到迭代模型

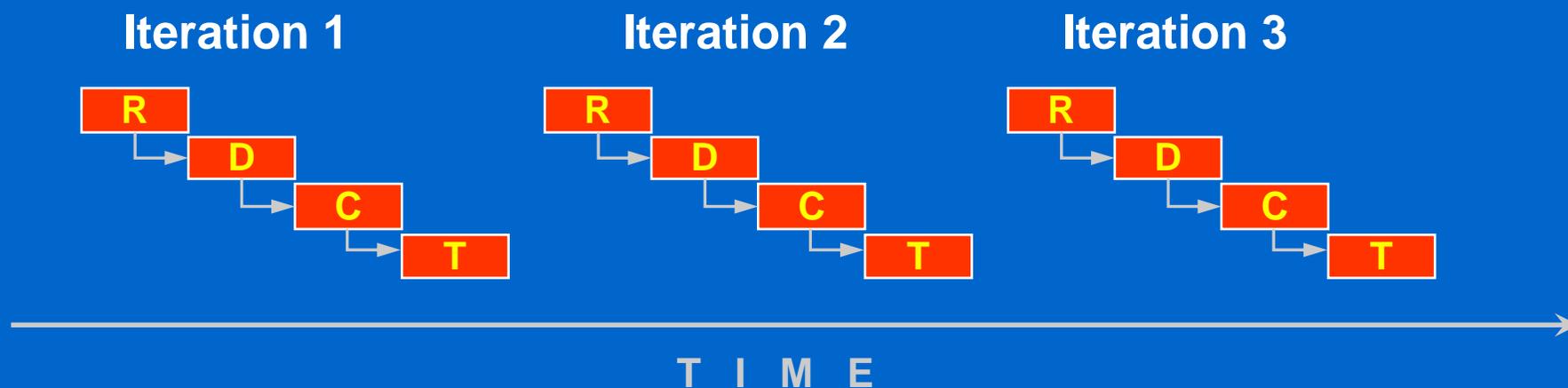
传统的瀑布式开发



瀑布式开发推迟了风险的规避

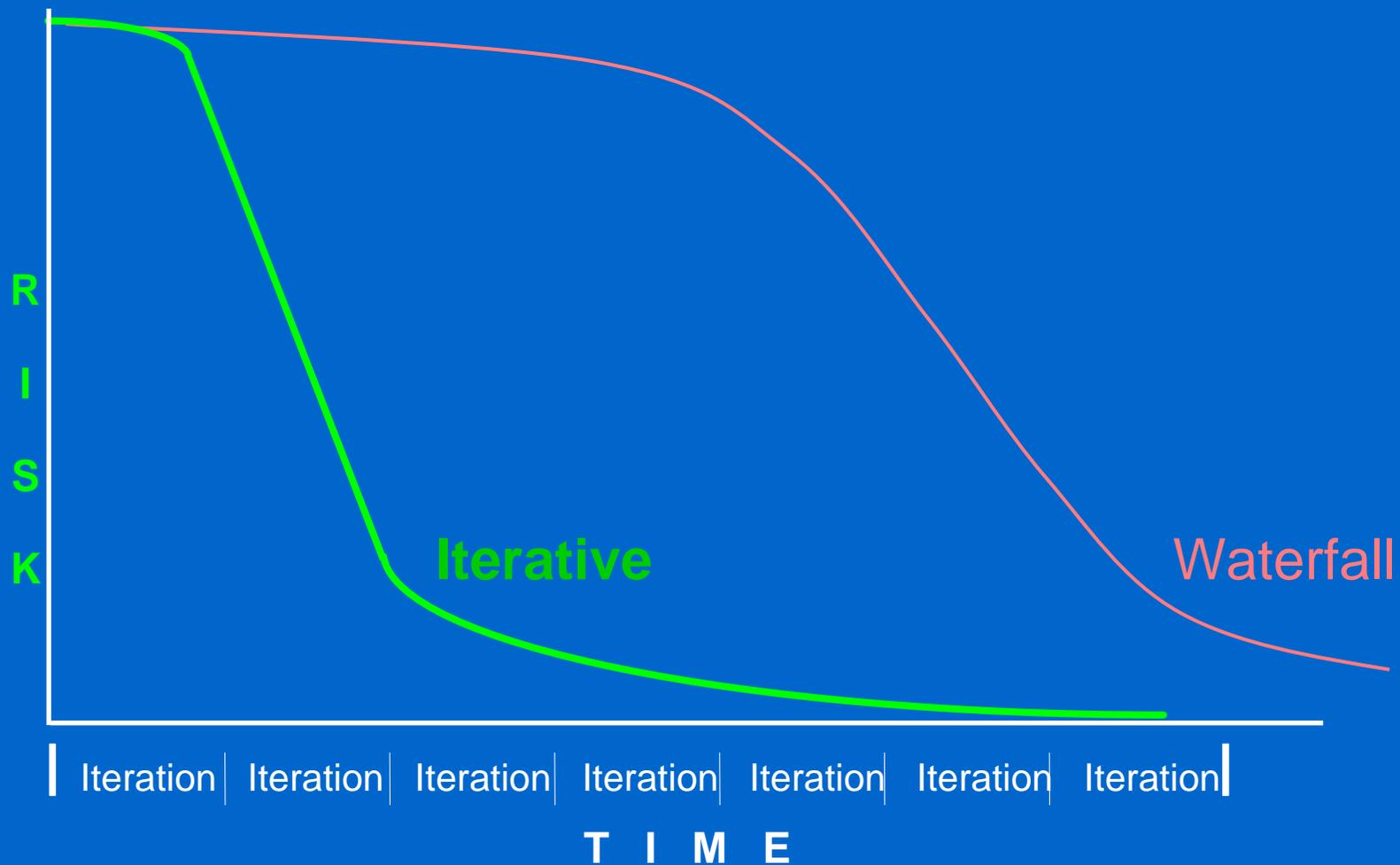


将瀑布开发迭代地应用于系统的增量



- 最早的迭代触及最重大的风险（例如需求或项目可行性风险）
- 每次迭代产出一个可执行（可以通过测试等较客观的途径加以验证）的交付，是系统的一个增量

迭代式开发促进风险规避



迭代式开发的特点

- 严重的风险在（项目）大规模投入之前被解决
- 初期的迭代能获取更早的用户反馈
- 测试与集成是连续的（增量式）
- 客观（可验证）的里程碑提供了短期的焦点
- 进度的度量直接依靠对实施成果的评估（而非主观的估计）
- 部分的实现可以被先行部署

风险驱动的开发模型

- 在RUP先启阶段的迭代中，项目组必须解决开发目标与范围、以及技术和商业可行性的根本风险——值不值得做，能不能做
- 而到了精化阶段的迭代，项目组关注的焦点则转到构架风险上——可否大量投入去做
- 进入项目中成本最高的构建阶段后，控制成本、进度和开发质量的风险将成为所有成员的责任——准备好交付给用户了？
- 最后到了迁移阶段，项目组将面临从客户和用户方面引入的各种风险（日程安排、需求亦而等）
空之日可世立

解决项目风险的关键—— 构架基线

思考：为什么需要软件构架

- 最终开发出的目标系统总是由多个组成部分所构成，这种结构如果没有预先定义，很难保证系统的构建过程能自发创建一个一致而满足需求的交付
- 当前的软件规模已大到需要采用团队开发的模式，多个开发人员的分工协作，必然依赖于一种对开发内容的合适划分，以减少相互干扰、缩短工期的关键路径，从而提高开发效率、加快项目进度——软件构架无疑是其中最关键的一类划分，它将被用来计划、管

思考：为什么需要软件构架

- 目标系统总是要面临各种变数，项目组期望系统在发生变更、部署到新环境中时，仍然保持既有的稳定、可靠和性能——目标系统应具备一种健壮性
- 系统的构建要经历一个不断增添新功能、加入新行为的过程，项目组期望做得比较容易、开销较低，且在此过程中不存在重大的风险——目标系统应具备一种可扩展性
- 而这些质量属性归根结底要落实到软件构架之上

思考：健壮性与可扩展性

- 要实现健壮性与可扩展性等质量特性，主要有两个途径——尽可能降低系统的冗余程度，同时隔离不同的关注面（实质是高内聚、低耦合，例如：将稳定部分与可变部分隔离，将用户交互与业务、数据等功能域分离，将功能和非功能的实施代码分离）
- 隔离关注面，使得扩展或变更时，对系统的修改局部化，对其它部分造成的影响被限制在较小范围内，避免出现那种牵一发而动全身的情形；高内聚的结构也利于聚焦于各部分的设计适应性上
- 低冗余，使得即使要变更，变更所触及的部分也尽可能地少；系统被改动的地方越少当然就越健壮，

构架基线原理：为大量的构造提供坚实的 框架基础

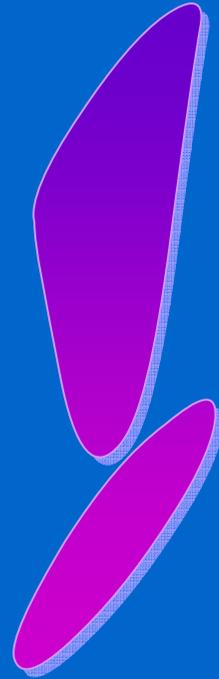
软件构架的定义

- IEEE1471-2000的定义:

一个系统的基础组织，它具体体现在系统的组成构件，构件相互之间、构件和环境之间的关系，以及指导其设计和演化的原则上

the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution

思考：仅仅是系统的分层结构吗



指南：如何理解软件构架

- （软件）系统进行分解的顶层结构，包括其组成元素，元素之间、元素与外部的关系

关注构架的静态方面，即系统大粒度（宏观）的总体结构（例如分层、子系统的划分等）

- 系统中解决各类关键的重复问题的通用解决方案

关注构架的动态方面，侧重于系统内部关键行为的共同特征（已经包含了微观细节，例如构架机制）

- 系统设计中影响深远（构架敏感）的各项最重要决定

这些决定严重影响系统的实施，一旦作出并被贯彻，其变更的代价将及其高昂（例如构架的样式、复用

思考：软件构架的意义

- 软件构架的静态方面，其着眼点在于——保持目标系统的最终交付在结构上的一致性；为分工协作提供划分依据，并避免结构上的重叠和冗余
- 软件构架的动态方面，其着眼点在于——保持目标系统在关键行为实现上的一致性，突出系统的既有风格；同时通过为各类关键重复问题提供通用解决方案来提高复用度，避免实施代码的冗余
- 上述两个方面，共同提供了构造目标系统过程中的健壮性与可扩展性——大量的功能实现将在这个构架基础上被不断添加，而同时系统整体上仍然保持既有的一致和完整

思考：软件构架基线的作用

- 顺利的话，占据整个设计过程中部分工作量的构架设计活动，能够为其它开发任务（绝大部分工作量）奠定一个坚实基础，使得开发组不需要再进行太多的开创性（高风险）工作——RUP中构架基线里程碑的提出，其理论基础便源于此
- 在项目的较早时期（精化阶段），便解决大部分的难题和风险，是开发组永远的追求目标
- 但是，只要是缺少整体的结构规划，或是通用问题还没有得到充分的解决，都无法做到让开发组在一定时间过后，就不再需要进行太多的开创性（高风险）工作

参考构架模型

实例：RM-ODP的元模型体系

- ISO/ITU-T RM-ODP定义的抽象层次（视角）：
 - 企业视点（Enterprise Viewpoint）
purpose, scope and policies
 - 信息视点（Information Viewpoint）
semantics of information and information processing
 - 计算视点（Computational Viewpoint）
functional decomposition
 - 工程视点（Engineering Viewpoint）
infrastructure required to support distribution
 - 技术视点（Technology Viewpoint）

实例：MDA的构架体系

- OMG组织定义的MDA架构视角包括：
 - 计算无关视点 *Computation Independent Viewpoint*
关注目标系统的应用环境，描述业务流程、信息等用以引申需求的上下文
 - 平台无关视点 *Platform Independent Viewpoint*
关注目标系统的运作，包括外部行为（功能需求）和内部协作（抽象的设计），而忽略其支撑平台的特定细节
 - 平台特定视点 *Platform Specific Viewpoint*
结合平台无关视点，增添特定平台细节的内容

指南: MDA vs. RM-ODP

MDA

计算无关视点
*Computation
Independent Viewpoint*

平台无关视点
*Platform Independent
Viewpoint*

平台特定视点
*Platform Specific
Viewpoint*

实施
Implementation

RM-ODP

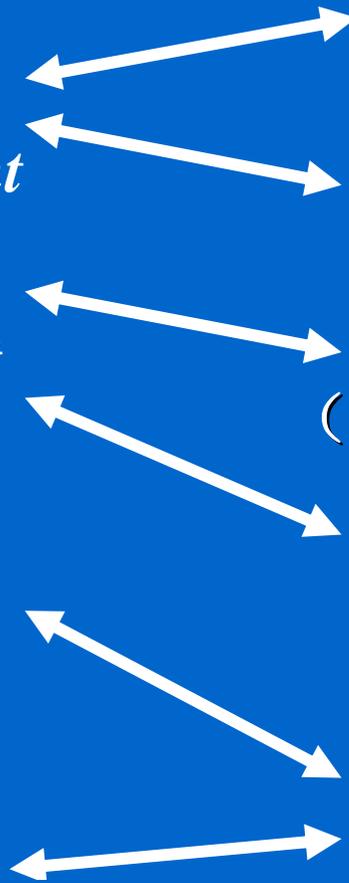
企业视点
(Enterprise Viewpoint)

信息视点
(Information Viewpoint)

计算视点
(Computational Viewpoint)

工程视点
(Engineering Viewpoint)

技术视点
(Technology Viewpoint)



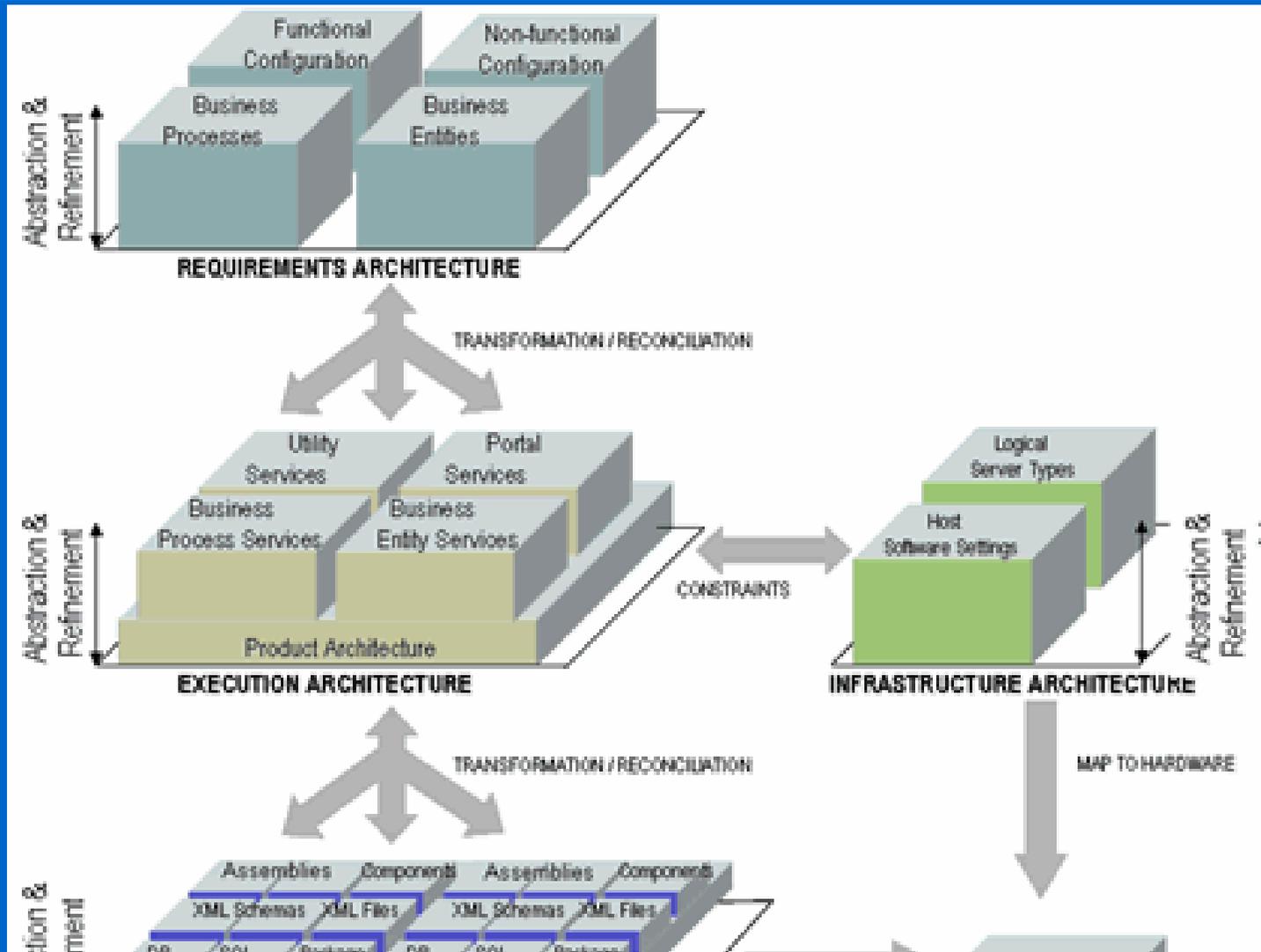
实例：Microsoft企业IT构架体系

- 微软定义的企业IT架构视角包括：
 - 业务 (Business Perspective)
 - 应用 (Application Perspective)
满足功能需求 *functional requirements*
 - 信息 (Information Perspective)
 - 技术 (Technology Perspective)
满足运作需求 (*operational requirements*) —
— 实质就是对性能等非功能性需求的抽取

实例：Microsoft的EAI参考模型

	<i>Business</i>	<i>Information</i>	<i>Application</i>	<i>Technology</i>
<i>Conceptual</i>	<ul style="list-style-type: none">▪ Use Cases And Scenarios▪ Business Goals And Objectives	<ul style="list-style-type: none">▪ Business Entities And Relationships	<ul style="list-style-type: none">▪ Business Processes▪ Service Factoring	<ul style="list-style-type: none">▪ Service Distribution▪ Quality Of Service Strategy
<i>Logical</i>	<ul style="list-style-type: none">▪ Workflow Models▪ Role Definitions	<ul style="list-style-type: none">▪ Message Schemas And Document Specifications	<ul style="list-style-type: none">▪ Service Interactions▪ Service Definitions▪ Object Models	<ul style="list-style-type: none">▪ Logical Server Types▪ Service Mappings
<i>Physical</i>	<ul style="list-style-type: none">▪ Process Specification	<ul style="list-style-type: none">▪ Database Schemas▪ Data Access Strategy	<ul style="list-style-type: none">▪ Detailed Design▪ Technology Dependent Design	<ul style="list-style-type: none">▪ Physical Servers▪ Software Installed▪ Network Layout

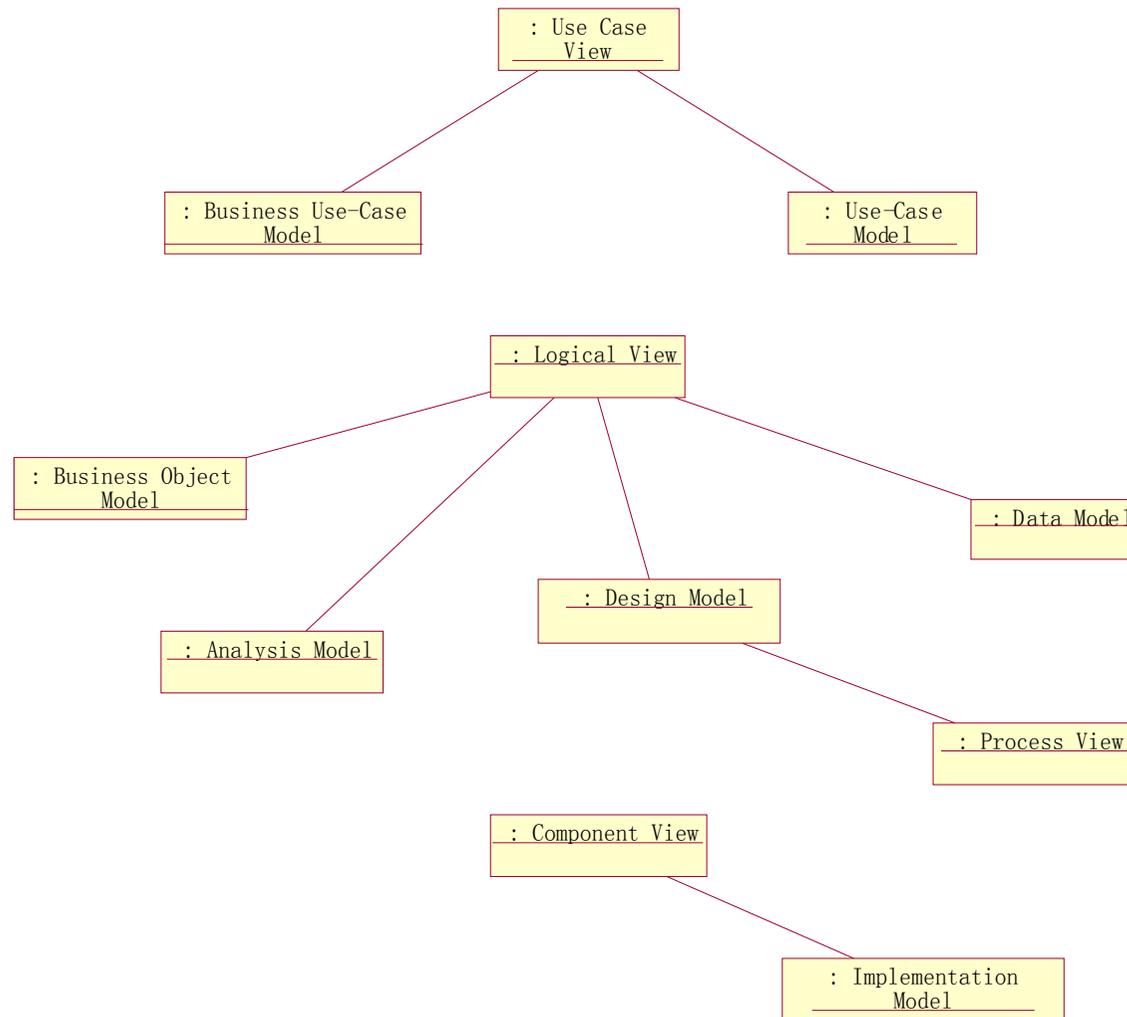
实例：Microsoft的不同构架视角



实例: Zachman Framework

Focus Perspective	What DATA	How FUNCTION	Where NETWORK	Who PEOPLE	When TIME	Why MOTIVATION
Objective/ Scope Contextual Role: Planner	List of Things Important in the Business	List of Core Business Processes	List of Business Locations	List of Important Organizations	List of Events	List of Business Goals/ Strategies
Enterprise Model Conceptual Role: Owner	Conceptual Data /Object Model	Business Process Model	Business Logistics System	Work Flow Model	Master Schedule	Business Plan
System Model Logical Role: Designer	Logical Data Model	System Architecture Model	Distributed Systems Architecture	Human Interface Architecture	Processing Structure	Business Role Model
Technology Model Physical Role: Builder	Physical Data /Class Model	Technology Design Model	Technology Architecture	Presentation Architecture	Control Structure	Rule Design
Detailed Representations Out of Context Role: Programmer	Data Definitions	Program	Network Architecture	Security Architecture	Timing Definition	Rule Specification
Functioning		Working	Usable	Functioning	Implemented	Working

实例：RUP4+1视图



不是所有的产品都容易得到构架基线

思考：不同类型系统的构架内涵

- 软件系统的种类繁多，其内在特性、对应的需求等差异很大；而其所拥有的构架，在内涵上将存在重大差别
- 这种差别将主要体现在软件构架中各类关键的重复问题上

有的系统（特别是系统级软件，例如IDE开发环境），其各功能或特性的实现要求差异极大，很难以一组所谓的通用解决方案来涵盖开发中将遇到的大部分问题——因而得到其构架基线也是非常困难的，在此重构技术将大有用武之地

企业的业务应用系统，其功能或特性实现的类似性

以构架为中心的开发，对团队能力提出了严峻挑战

架构师的职责



主导系统全局分析设计和实施、负责软件构架和关键技术决策的角色——

- 领导与协调整个项目中的技术活动（分析、设计和实施等）
- 推动主要的技术决策，并最终表达为软件构架描述
- 确定和文档化系统中对构架而言意义重大的方面，包括系统的需求、设计、实施和部署等“视图”
- 确定设计元素的划分以及这些主要分组之间的接口
- 为技术决策提供规则，平衡各类涉众的不同关注点，化解技术风险，并保证相关决定被有效传达和贯彻
- 理解、评价并接收系统需求

架构师的技能



- 技术全面、成熟练达、洞察力强、经验丰富，具备在缺乏完整信息、众多问题交织一团、模糊和矛盾的情况下，迅速抓住问题要害，并做出合理的关键决定的能力

完成项目的条件从来就是不理想的，总是有工期压力，总是存在各种矛盾，客户提供的信息总是不完整，而且总是在变化中，架构师必须面对这些状况

- 具备战略性和前瞻性思维能力，善于把握全局，能够在更高抽象级别上进行思考；

- 对项目开发涉及的所有问题领域都有经验，包括彻底地理解项目需求，开展分析设计之类软件工程活动

架构师的技能



- 拥有优秀的沟通能力，用以进行说服、鼓励和指导等活动，并赢得项目成员的信任；
架构师的设计思想要在项目中得以贯彻，依靠行政手段往往是适得其反的，架构师必须是一个有个人魅力的鼓动能手
- 以目标导向和主动的方式来不带任何感情色彩地关注项目结果，构架师应当是项目背后的技术推动力，而非构想者或梦想家（信仰折衷，而不追求完美）
- 精通构架设计的理论、实践和工具，并掌握多种参考构架、主要的可重用构架机制和模式（例如基于J2EE的参考架构等）；

架构师的活动与工件

●参与的活动——

确定用例或需求的优先级、进行构架分析、创建构架的概念验证原型、评估构架的概念验证原型的可行性、组织系统实施模型、描述系统分布结构、描述运行时刻构架、确定构架机制、确定设计元素、合并已有设计元素

●负责的工件——

软件构架文档、参考构架、分析模型、设计模型、实施模型、部署模型、构架概念验证原型

设计员的职责



在项目需求、构架和开发流程限制之下负责系统局部的分析设计的角色——

- 理解、评价并接收系统需求细节
- 理解、评价并接收相关软件架构的结构与机制等
- 依据需求规格分析系统的内部行为，在分析层面识别与定义各系统组成元素的职责、操作
- 识别与定义各设计元素的职责、操作、属性及其相互关系

但江甘说让控人始件始加 说只够详细到可

设计员的技能



- 掌握需求工程概念和技巧，以准确无误地理解相关系统需求；
- 熟悉软件架构模式、概念和技巧，以准确无误地理解相关系统构架；
- 精通软件设计理论、实践和工具，包括面向对象的分析、设计技术和统一建模语言等
- 掌握将用于实现系统的相关技术，例如组件开发（J2EE、EJB）、通讯机制、多线程与实时技术等；

设计员的技能



- 掌握将用于实现系统的程序设计语言（例如：Java、C++、C、HTML、CSS、XML、JavaScript、汇编语言）；
- 对目标设计元素的相关问题有深入的了解；
- 熟悉项目的设计指南，明了设计与实施的关系，包括在实施之前设计应当达到的详细程度；
- 具备实施员的所有技能，但程度更深、抽象级别更高；
- 通常兼任实施员的角色

设计员的活动与工件

- 参与的活动——

执行用例分析、设计用例实现、子系统设计、设计类、设计测试包与类库

- 负责的工件——

用例实现、分析类、设计子系统、设计包、设计类、测试类

Q&A

谢谢!

