Assignment 1

学号：121250163　　姓名：邬文怀

Task1: Quality Attribute Scenarios

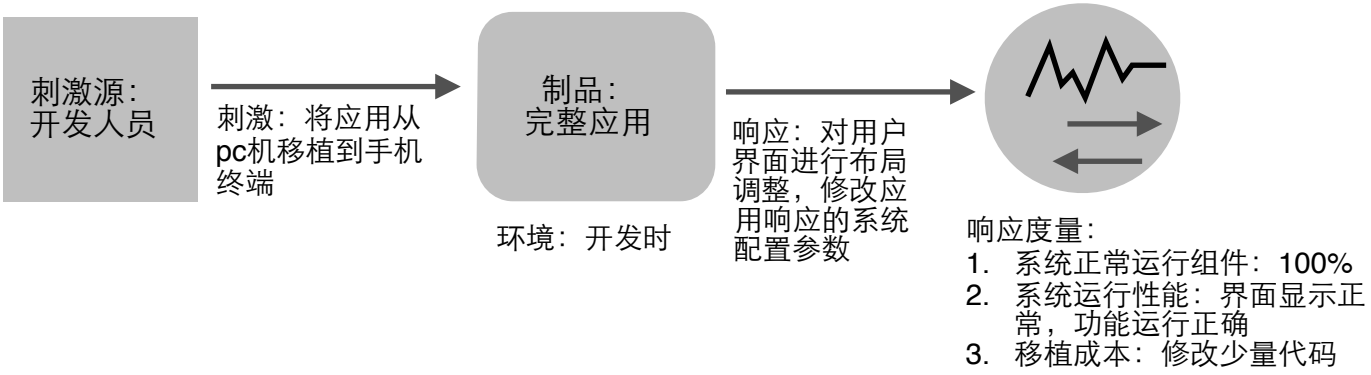
1. 可移植性的一般场景和具体场景

1.1 一般场景

场景的组成部分	可能的值
源	开发人员，系统用户。
刺激	将系统移植到不同的终端中运行。从windows系统移植到linux系统，从pc机移植到移动终端，从
制品	通常是完整的应用，也有可能系统组件。
环境	开发时，测试时，运行时
响应	系统应能够适应不同终端的显示环境，UI做出响应式调整。 系统在不同的环境应能够正常的运行全部功能。 系统在不同的环境中对异常和中断的处理应区别对待，适应不同环境的特性。
响应度量	系统移植的成本（包括时间，人力，物力，财力）， 系统移植后可以正常运行的组件百分比， 系统移植后的出错率，奔溃率， 系统移植后的运行性能（包括响应时间，处理速度，显示效果等）。

表1.1 可移植性的一般场景

1.2 具体场景



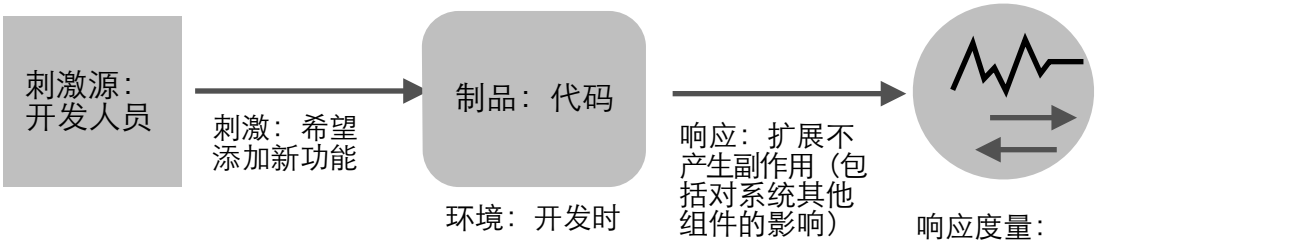
2. 可扩展性的一般场景和具体场景

2.1 一般场景

场景的组成部分	可能的值
源	开发人员，系统管理员，最终用户。
刺激	在原有系统的基础上，希望增加新功能，添加新的插件，增加新的质量属性，扩大容量。
制品	系统用户界面，源代码，接口，组件等
环境	运行时，构建时，设计时
响应	查找系统中需要扩展的位置，进行扩展的同时不影响其他功能，对所作的扩展进行测试，部署所做的扩展。确保扩展的可用性。
响应度量	扩展所需要的成本，资金，复杂度。 对系统原有组件的影响程度，与原油组件的兼容性。 扩展给系统带来的新的不确定因素和风险。

表2.1 可扩展性的一般场景

2.2 具体场景



Task2: 复杂度和成本是否是质量属性?

对于复杂度：

通常质量属性可以按两种方式进行划分：一是按运行时可识别或不可识别来划分，二是按对用户很重要的可识别特性或对开发者和维护者很重要的可识别特性来划分。按第一种方式划分，复杂度在运行时是不可识别的，用户无法观测到系统内部的运行逻辑是有多复杂，另一方面，界面上的操作复杂度应属于可用性的范畴，将复杂度独立开来成为一种质量属性的意义不是很大。按第二种方式划分，复杂度主要体现在用户的使用复杂度，以及开发人员的开发复杂度和维护人员的维护复杂度上，这三种复杂度在可用性，可修改性，可测试性，可扩展性，可重用性方面都有涉及，所以没有必要把复杂度看作是一个独立的质量属性。

另一方面，从一般场景来分析复杂度，复杂度的概念过于笼统，它涉及到系统的方方面面，而质量属性应当是系统明确的质量目标。

综上所述，复杂度不是一种质量属性。

对于成本：

除了与系统直接相关的质量属性外，还有很多商业质量目标也会对系统的构架产生较大的影响。成本属于商业质量属性的范畴，在实际的系统构建过程中，往往需要考虑成本，然后做出符合实际条件的架构设计。

另一方面，成本是一个明确的概念，与系统其他质量属性一样，是一个可以明确定义，并且可以做出有效响应的质量属性。

综上所述，成本是一种质量属性。

Task3: 设计模式的优劣势（从质量属性的角度分析）

设计模式	优势	劣势
分层模式	• 更好的可复用性和可修改性 理由：只要遵守其交互协议，不同的层次部件就能够互相替换,具有很好的可复用性。 在不影响交互协议的情况下，每个层次可以自由安排其内部实现机制，内部可修改性。	• 额外的层会增加系统的成本和复杂度 • 性能损失 • 交互协议难以修改
经纪人模式	• 更好的可用性，互操作性 理由：broker作为中介，连接客户端和服务器，使得客户端和服务器可以双向操作。	• broker层增了了通信延迟 • broker是一个单点失效 • broker难以测试 • broker容易被攻击 • broker增加了复杂度
MVC模式	• 视图和控制的可修改性 • 适宜与网络应用的开发 理由：模型是相对独立的,所以对视图实现和控制实现的修改不会影响到模型实现。 对业务逻辑、表现和控制的分离使得一个模型可以同时建立并保持多个视图,非常适用于网络系 统开发。	• 对于简单的用户接口，MVC过于复杂 • MVC抽象可能不适合某些用户接口工具 • 模型修改困难
管道过滤器模式	• 更好的可复用性 理由：任何两个过滤器互不直接影响。只要输入和输出的数据流内容符合要求，就可以通过建立相应的管道将一个过滤器融入一个软件系统。 • 更好的内部可修改性 理由：只要不修改输入和输出的数据流，过滤器可以自由安排对内部实现的修改 • 更好的可扩展性 理由：过滤器的正确执行不依赖于前面与后面的过滤器，也不依赖于整个过滤器网络的执行顺序。	• 交互性较差 • 空间效率，不适合复杂运算 • 性能浪费 • 错误处理能力较弱
客户端服务器模式	• 易开发 理由：客户端/服务器风格限定了 Server 的标识，这使得网络通信功能的设计和开发变得容易。 • 客户端的动态性 理由：客户端 / 服务器风格中的客户端可以随时增减，能够实现客户端的动态性。	• 服务器可能成为性能瓶颈 • 服务器是一个单点失效 • 难以调整 • 不易更新
点对点模式	• 系统负载压力小 理由：每个参与者都既是客户端，又是服务器，是一种分布式系统风格。	• 管理复杂度，数据持久化，备份和恢复都更复杂 • 小型点对点系统可能不能长久地保证性能和可用性
SOA模式	• 更好的可重用性 理由：一个服务创建后能用于多个应用和业务流程。	• 服务器可能成为性能瓶颈 • 服务器性能无法保证 • 构建复杂
发布订阅模式	• 更好的安全性 理由：消息的可读性和可修改性都有发布者决定。	时间延迟 消息传送的可预测性较差 消息传送的成功率无法保证

设计模式	优势	劣势
分享数据模式	• 更好的空间效率 理由：所有的知识源交互都需要经过共享数据来传递 • 知识源的可修改性 理由：各个知识源是相互独立的，只需要依赖于共享数据。 • 更好的容错性和健壮性 理由：全局的数据都存储在一起，使得它可以集中精力保障系统的容错性和健壮性，例如建立共享数据的备份、控制共享数据的安全、控制共享数据的并发等。 • 更好的性能 理由：允许多进程并发	• 共享数据存储可能成为性能瓶颈 • 共享数据存储是一个单点失效 • 偶合度较高
多层模式	• 易开发 理由：客户端/服务器风格限定了Server 的标识，这使得网络通信功能的设计和开发变得容易。 • 客户端的动态性 理由：客户端 / 服务器风格中的客户端可以随时增减，能够实现客户端的动态性。	• 先期成本和复杂度较高
映射归约模式	• 性能出色；（可以并行计算）； • 数据共享； • 可靠性高；（ 对数据集的大规模操作分发给网络上的每个节点实现可靠性）；	• 一般只适用于大数据计算； • 如果不能把数据划分为相同大小的子集，并行计算的优点不会体现； • 计算复杂；