编号_____

南京大学

课 程 作 业

题　目　**Software Architecture Assignment 2**

学生姓名	王一辉
学　　号	191250142
学　　院	软件学院
专　　业	软件工程
班　　级	二班

2022 年 5 月

Task 1. Architecture Pattern Analysis

I. Layered Pattern 【分层模式】

- Availability 可用性
  - 可能产生的影响: 因为分层使得系统的内聚性更高, 各层次负责的功能更单一, 所以更容易针对性地发现故障并修复, 使得可用性提高。
  - 举例: 对于一个基于分层模式实现的管理系统, 如果产生了一个在数据持久化层面上的漏洞, 在页面发现错误和故障之后, 可以立刻根据故障特征确定漏洞的相应层次。从而更快修复相应的故障, 尽快重新启动整个系统。
- Interoperability 互操作性
  - 可能产生的影响: 各个层次之间可以屏蔽其他层次, 在不同的系统或平台之间交互信息、互相操作;
  - 举例: 基于 OSI 模型实现的网络结构就采用了分层模式。每一个网络节点之间的交互都是在分层的基础之上的。每一层比如网络层、传输层都只需要关心自己层的业务, 而不需要关心其他层的细节, 跨系统之间的数据传输和交互统一、方便, 使得互操作性有了很好的改善。
- Modifiability 可修改性
  - 可能产生的影响: 接口划分好的情况下, 分层使得系统易于维护, 更加“高内聚、松耦合”, 修改可以只针对相应的层次, 提高可修改性。接口划分不好的情况下, 底层的产生的修改可能会导致依赖它的上层需要修改, 最终导致“级联修改”, 使得可修改性变差。
  - 举例①: 编写网络程序时不需要关系其他层次的细节, 只需要关注当前层次的代码和逻辑, 不需要担心当前层次对其他层次的影响, 系统的可修改性提高了。
  - 举例②: 假设一个 JAVA 开发项目, 底层的数据持久化模块的数据格式发生改动, 那么上层依赖这份数据的层次都要被更改, 因此产生的级联修改使得整个系统

的可修改性变差了。

- 候选策略：拆分模块和增加语义一致性。不同层次中的各模块的内聚性应该更好，层次内部不应该彼此依赖而产生循环依赖。描述同一个功能和模块的语义应当尽量一致；尽可能不修改基础的数据结构和格式。
- 候选策略的适用情况：同层次中模块相互依赖的情况；设计系统的数据结构和数据格式的情况；划分系统模块和接口的情况；

- Performance 性能
  - 可能产生的影响：层次嵌套会降低系统的性能。
  - 举例：比如 OSI 模型，从顶层到下层在传输信息的过程中，需要解析上层的内容，并且在不同层次之间传递信息的过程中要产生额外的校验并且调用方法也会产生额外的开销。
  - 候选策略：引入并发。对一个相对庞大的系统，层次中的模块彼此耦合较低，在这种情况下，可以引入并发，使得系统同时处理多个用户与请求，并且不会发生功能之间的相互干扰。
  - 候选策略的适用情况：请求密集的层次模式系统；具有良好并发性的系统；
- Security 安全性
  - 可能产生的影响：不同层次由不同的用户访问以及专门的管理员进行管理，提高系统安全性。
  - 举例：Linux 系统中，分为用户层和内核层。用户层需要认证才能进入内核层。而风险操作需要在内核层才能进行，所以分层设计使得系统安全性更好。
- Testability 可测试性
  - 可能产生的影响：系统的不同层次可以分别测试，并且可以引入插装、集成测试等测试方式，系统整体的可测试性变好。
  - 举例：分层设计的系统更易于去设计分层次的测试用例。分层系统内聚性更好，

更容易对某一层次的功能做测试，使得测试性提高。

- Usability 易用性
  - 可能产生的影响：分层使得底层实现细节被屏蔽，用户只需要使用上层接口和内容，易用性增加。
  - 比如分层设计的 web 项目，用户只需要关心和考虑表示层中的内容和逻辑，而不需要关心底层的实现逻辑和算法。在这种情况下，用户可以更简单方便地使用这个系统。

II. Broker Pattern【代理人模式】

- Availability 可用性
  - 可能产生的影响：一个代理人可能会和大量服务器和大量的客户产生依赖，在这样的情况下，代理人出现故障就会出现单点失效问题，使得系统中大量的用户和服务器无法正常工作，很大程度上延长故障时间。
  - 举例：代理人模式的系统架构中很容易出现“集线器”形式，代理和多个服务器之间彼此有依赖。另外，代理可能面对更大数量的客户请求。因此，在架构出现上述的星型结构后：中心一点失效就会造成整个系统失效，即单点失效问题。
  - 候选策略：消除单点故障。为代理人准备相应的备用资源与重启方案。在当前的代理人故障时，启用备用代理人和方案，在这种情况下可以较快重启系统。
  - 适用情况：有充足资源的系统；构成集线器形式的代理人模式系统架构；
- Interoperability 互操作性
  - 可能产生的影响：使用了代理人之后，用户可以更加灵活地动态关联到某一个服务器，从而更加方便地请求服务。
  - 举例：比如一个服务中，新加入了某个服务器，系统并不需要为每个用户直接建立系统到用户的关联，只需修改用户关联的代理，即可请求或操作新的服务器。
- Modifiability 可修改性

- 可能产生的影响：绑定时机靠后，系统就更灵活，修改系统的成本就越低，可修改性就越强。
- 举例：延迟绑定，可以使系统在运行时绑定服务器和用户，而不需要在一开始的开发阶段就确定绑定关系。代理人的存在使得绑定修改更加简单。

➢ Performance 性能

- 可能产生的影响：单点情况可能会造成性能瓶颈。
- 举例：对于一个多用户多服务器都关联一个代理人的架构，某一个单点的性能下降，会造成整个系统中大量服务的处理请求性能下降。
- 候选策略：增加资源。
- 适用情况：为单点备用资源，限制单点请求量。负载均衡。

➢ Security 安全性

- 可能产生的影响：单点情况中，单点失效会造成整个系统的失效。
- 举例：恶意攻击者攻击单点机构，就很容易造成整个系统的宕机。此时，单点就会成为整个系统最高风险的安全漏洞。
- 候选策略：通过流量或签名发现入侵。
- 适用情况：流量较大的单点结构；有权限认证功能的系统；

➢ Testability 可测试性

- 可能产生的影响：增加系统的不确定性，测试成本提高。
- 举例：一个用户在不同时刻可以绑定不同服务器，用户和服务器的组合关系就会变得非常复杂，系统的不确定性增加，测试难度和工作量提高，可测试性降低。
- 候选策略：限制结构复杂性，不要让系统中某个单点关联过多的用户和服务器。
- 适用情况：出现单点结构的系统。

➢ Usability 易用性

- 无明显影响。

Task 2. Architecture Pattern vs. Quality Attribute & Tactics

模式	Availability 可用性								
	Detect Faults			Recover from Faults			Prevent Faults		
	Monitor	Timestamp	Exception Detection	Active Redundancy	Rollback	Software Upgrade	Transactions	Predictive Model	Exception Prevention
MVC	X	X		X	X	X	X		
Pipe-filter	X	X	X	X	X	X	X	X	X
client-server	X	X	X						
Microservice	X	X	X			X	X		

- MVC 模式：MVC 是分层模式的一类变种。它使得整个系统的内聚性更高、耦合性更低。因此在错误探查阶段，相比一般的模式，MVC 模式的系统在发现错误的成本更低。同时，MVC 模式的系统生命周期成本低易于进行管理，可以快速部署。
- Pipe-Filter 模式：管道-过滤模式在错误发现时可以通过在相应的管道或者过滤器设置错误探测器比如 Monitor 来尽快发现可能的错误；并且管道和过滤模式还可以在找到的出错点尽可能靠近的位置重启，对于错误的恢复也有优势。在避开错误的过程中，和错误探测相似，可以在相应的管道和过滤器进行预测。
- Client-Server 模式：和代理人模式相似，容易出现单点失效问题，并且单点出现问题较难修复。可以应用错误探测相关手段尽可能避免损失。
- Microservice 模式：将相应的请求分配给对应的服务，能更迅速地发现对应的错误和故障。并且可以分别重启服务、更新等。

模式	Interoperability 互操作性		
	Locate	Manage Interfaces	
	Discover Service	Orchestrate	Tailor Interface
MVC	X	X	X
Pipe-filter	X	X	X
client-server	X	X	X
Microservice	X		X

- MVC 模式：MVC 在服务发现上很有优势，它主要是在 Model 模型中进行对请求的响应，并将请求分配给对应的 Controller 控制器，再由控制器根据请求发现对应的服务，返回相应的 View 视图。MVC 的模型在返回视图的过程中，由相应的服务进行处理。在服务中，接口可以被精心安排和剪裁。
- Pipe-Filter 模式：可以利用 Filter 来进行请求和服务的对应进行服务发现。在管理接口的时候也可以使用精心编排接口和接口裁剪来进行相互的分配。
- Client-Server 模式：与 Pipe-Filter 类似，可以在开发期间绑定对应的客户端和服务端。但是相比代理模式，客户端服务端模式是静态的，并没有那么灵活。
- Microservice 模式：微服务模式是通过将对应的请求分配给相应的云服务。因此服务发现是该模式进行交互的一个重要策略。另外，每个服务运行的时候都需要关注本身的功能和负载来管理接口。

模式	Performance 性能								
	Control Resource Demand				Manage Resources				
	Limit Event Response	Prioritize Events	Bound Execution Times	Increase Resource Efficiency	Increase Resources	Introduce Concurrency	Maintain Multiple Copies of Computations	Bound Queue Sizes	Schedule Resources
MVC	X	X	X	X	X	X		X	X
Pipe-filter	X	X	X	X	X	X	X	X	X
client-server	X	X	X	X	X	X	X	X	X
Microservice	X	X	X	X	X	X	X	X	X

- MVC 模式：因为 MVC 模式是分层模式的一种，所以在各层次传递数据和交互的过程中都会造成相应的性能损耗。因此可以通过控制资源需求和资源的来保证整个系统的性能。
- Pipe-Filter 模式：过滤器在系统中起到数据管理、计算的作用。那么，过多的过滤器就可能会导致大量的计算开销。另外如果多个模块在系统中依赖某一个或某几个过滤器，也可能会产生性能瓶颈问题，需要相应的策略来控制请求、管理资源。
- Client-Server 模式：和代理人模式类似，在客户端-服务端模式上，很容易出现星型结构，在这种情况下就可能会出现单点问题，从而出现性能瓶颈，因此也需要策略来缓解相应的负面影响。
- Microservice 模式：对于微服务模式，不同的服务负责不同的功能，但如果对某一个服务的请求过多，或远远高于其他服务就也可能出现性能瓶颈问题。在这个时候就需要进行负载均衡、控制请求、管理资源等等策略。

模式	Security 安全性								
	Detect Attacks		Resist Attacks			React to Attacks		Recover from Attacks	
	Verify Message Integrity	Detect Message Delay	Authenticate Actors	Limit Access	Encrypt Data	Revoke Access	Inform Actors	Maintain Audit Trail	See Availability
MVC	X	X	X	X	X	X	X		X
Pipe-filter	X	X	X	X	X	X	X	X	X
client-server	X	X	X	X	X	X	X	X	X
Microservice	X	X	X	X	X	X	X		X

- MVC 模式：MVC 模式在模型中可以进行用户验证与请求控制，相应的策略可以在这里实现来避免攻击和抵抗攻击。而在应对攻击时，可以屏蔽掉相应的请求或者通知管理员等方式来进行反制。
- Pipe-Filter 模式：过滤器可以负责控制、过滤、管理一些危险请求，来避免相应的安全问题。一旦出现攻击也可以过滤掉相应的有害信息。
- Client-Server 模式：与代理人模式类似，一旦出现多个服务器或者多个客服端依赖同一个客户端或者服务器时，就可能会产生大面积的安全问题。这个时候就需要从发现攻击、抵抗攻击、恢复等多个方面考虑策略来提高系统安全性。比如进行加密认证等等安全操作；
- Microservice 模式：和 MVC 模式类似，在服务发现和分配的过程中可以应用对应的发现危险和规避危险手段。在遭遇安全风险时也可以采取同样的手段反制和恢复。

模式	Testability 可测试性						
	Control and Observe System State					Prevent Faults	
	Specialized Interfaces	Record/ Playback	Localize State Storage	Abstract Data Sources	Executable Assertions	Limit Structural Complexity	Limit Nondeterminism
MVC	X	X	X	X	X		
Pipe-filter	X	X	X	X	X	X	X
client-server	X	X	X	X	X	X	X
Microservice	X	X	X		X		X

- MVC 模式：虽然 MVC 模式相对比较复杂，而且在开发的过程中相对繁琐，但是它可以和分层模式一样进行分层的测试，比如对接口专门化等控制系统状态的策略就能够很好地起到作用。
- Pipe-Filter 模式：管道-过滤器模式中，管道和过滤器可能会导致整个系统的组件和连接器的组合复杂，对于测试用例的需求量较大，并且全部情况较难考虑全，所以可测试性会受到影响。因此需要尽可能地限制系统的复杂度和不确定性来节约测试成本，并且控制系统的相关状态来减轻负面影响。
- Client-Server 模式：与代理模式和管道-过滤器模式类似，系统整体的复杂度偏高和组合数量较多，测试成本较大。需要相关的策略进行成本降低的处理。
- Microservice 模式：与 MVC 模式类似，可以对相应的服务分别进行测试，并且控制功能的复杂度，降低了测试成本，整体系统的测试成本降低。

模式	Usability 可用性						
	Support User Initiative				Support System Initiative		
	Cancel	Undo	Pause/Resume	Aggregate	Maintain Task Model	Maintain User Model	Maintain System Model
MVC					X	X	X
Pipe-filter	X		X	X	X		
client-server					X		
Microservice					X		

- MVC：可用性提高，专门开发展示给用户的部分——视图。
- Pipe-Filter/ Client-Server/Microservic 模式：影响不大。

模式	Portability 可移植性							
	Reduce Dependencies		CI/CD		Formalize Requirement		Open Source	
	Use cross-platform API	User Internal API	Auto-Deploy	Auto-Testing	Design by Users	Consider more before coding	Community Version	Embrace Contribution
MVC			X	X				
Pipe-filter	X		X	X	X	X	X	X
client-server	X	X	X	X	X	X	X	X
Microservice	X	X	X	X			X	

- MVC/ Pipe-Filter/ Client-Server 模式：影响不大但可以通过相应策略改善。
- Microservice 模式：不同的微服务可以在不同平台部署，可移植性提高。

模式	**Flexibility 灵活性**							
	Reduce Dependencies		CI/CD		Formalize Requirement		Open Source	
	Use cross-platform API	User Internal API	Auto-Deploy	Auto-Testing	Design by Users	Consider more before coding	Community Version	Embrace Contribution
MVC			X	X				
Pipe-filter	X		X	X	X	X	X	X
client-server	X	X	X	X	X	X	X	X
Microservice	X	X	X	X			X	

- MVC 模式：MVC 模式高内聚低耦合，整体系统的灵活性和可维护性提高。在应用相应的策略之后，可以进一步完善系统的灵活性，系统的扩张能力变强，易于修改和整合新的功能。
- Pipe-Filter 模式：无明显影响。
- Client-Server 模式：客户端与服务端的绑定确定比较早，并且容易出现单一节点，使得整个系统都依赖于某一个节点，在这种情况下，系统整体的灵活性会变差。
- Microservice 模式：微服务模式中，不同功能依赖于不同服务，整个系统应用哪些服务和功能是十分灵活的。另外，在微服务模式的系统中，添加新功能也十分灵活，这提高了整个系统的灵活性。