1.Previous Concepts

详细总结

1. **软件架构的基本概念**
   - 软件架构是软件工程的子学科，主要与中大型系统开发相关，通过划分系统为大的部分来实现业务目标，同时要避免过度工程化。
   - 传统Web应用架构简单，便于开发、测试、部署和扩展，但对于大规模增长的业务，可能需要更复杂的系统架构。
   - 软件架构设计不仅受功能需求的引导，还受质量属性需求的影响。质量属性需求的示例包括：
质量属性	示例
性能	ATM应能在1秒内验证取款操作
可用性	销售服务器故障时，销售服务应在5分钟内恢复
可移植性	应用程序应能在Windows、Linux或MacOS上无需修改即可运行
2. **软件架构设计的案例分析**
   - **书店信息系统需求**：书店分为小书店和大书店，小书店每日售书量少，由一人经营；大书店每日售书量大。小书店期望低成本且易用的解决方案，大书店则要求高可用性，其销售服务中断不能超过5分钟。
   - **解决方案差异**：小书店使用配有网络浏览器的PC和运行销售服务及库存管理的全局服务器；大书店使用PC网络和本地服务器，本地服务器故障时可切换到全局服务器。
   - **代码复用策略**：相同的Web服务可部署在全局服务器或本地服务器上，必要时可通过更改URL重新定向调用。
3. **软件架构设计的要点**
   - **利益相关者影响**：利益相关者包括客户、终端用户、开发者、项目经理等，他们的利益诉求不同，有时甚至相互冲突。例如，终端用户希望程序易用且可靠，客户希望成本低且开发周期短，市场利益相关者希望产品有竞争力且上市时间短。
   - **质量属性冲突**：不同质量属性间存在冲突，如可用性与安全性（更多冗余资源会带来更多薄弱点）、性能与安全性（防火墙和安全协议会增加通信时间成本）、可修改性与性能（分层架构相比单体架构会增加时间成本）。

- **架构设计决策**：软件架构设计关注模块接口，而非算法和数据结构等实现细节，但对等系统除外。架构设计的早期决策非常重要，这些决策是业务目标和质量需求的直接结果，影响深远且后期更改成本高。架构设计能简化后续开发流程，包括明确系统边界和外部接口、识别与遗留子系统的集成需求、选择合适的技术以及定义最终实现的约束等。

关键问题

1. **软件架构设计中如何平衡不同利益相关者的需求？**
   - 答案：在软件架构设计中，需要深入了解各利益相关者的需求，通过沟通协商确定业务目标和质量属性需求。例如在书店信息系统中，针对大小书店不同的需求，设计不同的解决方案，同时尽量复用代码，以平衡各方利益。在设计过程中，要充分考虑质量属性间的冲突，权衡利弊做出决策。
2. **软件架构设计关注模块接口而非实现细节的原因是什么？**
   - 答案：软件架构的早期决策影响深远且后期更改成本高，关注模块接口能使架构具有更好的通用性和扩展性。不同的实现细节（如数据结构的选择）可以在后续设计和实现阶段确定，这样架构设计可以保持一定的灵活性，适应不同的业务场景和技术发展。同时，关注接口也便于各模块的独立开发，提高开发效率。
3. **软件架构的早期决策为什么对整个系统如此重要？**
   - 答案：软件架构的早期决策是业务目标和质量需求的直接结果，这些决策确定了系统的基本结构和框架。例如选择的技术、系统的分解方式等，会对系统的功能实现、性能表现、可维护性等方面产生深远影响。而且后期更改这些决策的成本很高，因为会涉及到系统多个部分的调整，所以早期决策至关重要。

2.Quality Attribute Scenarios

详细总结

1.**软件架构设计流程**：在软件开发过程中，架构设计需经历多个环节。首先要创建系统的业务案例，全面理解需求，包括业务目标、上市时间、成本、与现有系统的接口等，其中质量属性需求（QARs）通过质量属性场景来表达。接着是创建或选择架构，依据QARs应用相应策略、风格或模式，并对其进行优先级排序。之后要与所有利益相关者沟通架构，通过基于场景的分析评估架构，最后基于选定架构进行系统实现。架构设计本身也是一个迭代过程，直至所有利益相关者达成一致。

2.**质量属性分类**

质量属性	定义	相关说明
可用性	系统在需要时可使用的概率（计划内停机时间不计）。系统应对故障，在规定时间内恢复服务。	如手机即时通讯应用在异常关闭后 1 秒内恢复聊天对话。
性能	系统对事件响应的时间，反映在特定负载条件下的表现。	手机即时通讯应用发送消息时，在服务器负载不超 75%、网络带宽至少 1Mb 的情况下，消息需在 1 秒内发送完成。
易用性	产品能被特定用户有效、高效且满意地用于实现特定目标的程度。	手机即时通讯应用在接收方读取消息后 1 秒内，向发送方显示已读通知。
安全性	系统抵御未授权使用并持续提供服务的能力。	手机即时通讯应用登录模块在用户连续 3 次登录失败后，封锁该用户登录 30 分钟。
可测试性	准备测试（通常是测试程序）以引发故障的难易程度。	手机即时通讯应用的联系人管理模块提供 API 用于测试联系人的插入或删除操作，测试设置在 1 秒内完成。
可修改性（或可维护性）	对系统功能或满足某些 QAR 进行变更的成本。	手机即时通讯应用的 GUI 模块添加新语言配置文件后，2 小时内完成加载和审查。

3.**质量属性场景构成**：质量属性场景是描述系统QARs的操作手段，包含六个部分。**刺激源**是产生刺激的实体；**刺激**为内外部事件或利益相关者的请求；**环境**指刺激发生的条件；**制品**是被刺激的系统或其部分；**响应**是刺激到达后的活动；**响应度量**用于衡量响应，以检查需求是否满足。

4.**应用案例分析**：以手机即时通讯应用为例，不同质量属性场景有不同的表现和要求。在性能方面，如消息发送需在规定时间内完成；可用性方面，系统异常关闭后要快速恢复聊天对话。此外，虚拟灌溉系统为解决测试时间过长的问题，添加了改变模拟时间速度的功能，该功能对系统架构产生了较大影响，因为涉及多个使用定时器的功能模块。

5.**场景模板示例**：文档给出了不同质量属性场景的模板，明确了各部分的可能取值。例如，安全场景的刺激源可能是身份不明的内部或外部个体，刺激行为包括尝试显示、更

改或删除数据等；可用性场景的刺激可能是系统故障，响应包括检测事件、通知相关方等。这些模板为架构设计和评估提供了重要参考。

关键问题

1.质量属性场景如何帮助实现软件架构的质量属性需求？

- 质量属性场景通过明确刺激源、刺激、环境、制品、响应和响应度量六个方面，将抽象的质量属性需求转化为具体可操作的描述。这使得架构师能更清晰地理解需求，进而在架构设计阶段针对性地选择合适的策略、风格或模式，确保系统在实际运行中满足各项质量属性要求。例如，在设计手机即时通讯应用时，根据性能质量属性场景的要求，优化消息发送的流程和算法，以实现消息在1秒内发送的目标。

2.在软件架构设计中，如何平衡不同质量属性之间的关系？

- 在软件架构设计中，不同质量属性之间可能存在冲突，如性能与安全性。为平衡这些关系，架构师首先要明确业务目标和各利益相关者的需求，确定质量属性的优先级。例如，如果业务更注重安全性，可能会在一定程度上牺牲性能，采用更严格的安全措施。同时，通过合理的架构设计，如采用分层架构、缓存技术等，可以在一定程度上缓解冲突。在选择技术和设计方案时，要综合考虑对多个质量属性的影响，确保整体架构满足系统的功能和质量要求。

3.以虚拟灌溉系统为例，说明架构变更对系统的影响有哪些？

- 虚拟灌溉系统添加改变模拟时间速度的功能后，对架构产生了多方面影响。从功能实现角度，涉及多个使用定时器的模块，如灌溉触发、 pivot运动、天气控制等，需要对这些模块进行调整以适应新的时间速度。从性能方面，可能会影响定时器的精度和系统的稳定性，需要重新评估和优化相关算法。在可维护性方面，新增功能可能会使系统结构变得复杂，增加后续维护的难度。但从测试性角度，该功能大大缩短了测试时间，提高了测试效率，方便开发者在不同情况下测试系统行为。

3.Tactics

详细总结

1.**架构策略定义**：架构策略是一种设计决策，它对系统在质量场景中如何控制响应产生影响，是架构师长期使用的良好实践方法。针对不同的质量属性，有着不同的策略组。

2.**电商系统案例分析**：在电商系统中，通过不同的策略来满足各种质量属性场景需求，具体如下表所示：

代码	属性	刺激(环境)	响应	采用策略
SC1	安全性	入侵企图	阻止入侵并识别来源	入侵检测
SC2	可修改性	数据库管理系统（DBMS）变更	进行修改且不影响其他子系统	维护语义一致性、遵循定义的协议/接口（SQL）
SC3	可用性	Web服务器宕机	保持Web服务可用	主动冗余
SC4	性能	1秒内收到1000个op1的http请求(正常运行)	服务器在2秒内对所有请求做出响应	并发处理

3.**可修改性策略分类**

-**定位修改**：通过抽象公共服务，将多个模块的公共服务提取到一个新模块，减少重复代码，如定义抽象类包含公共服务；泛化模块，使模块更通用，降低修改可能性，如将具体类型的队列类泛化为通用队列类；维护语义一致性，根据设计时预期的变更，将受同一变更影响的职责放在一起，如采用状态 - 逻辑 - 显示三层分解，清晰划分关注点，减少变更范围，常用于业务应用、多人游戏、基于Web的应用等。

-**防止连锁反应**：由于模块间存在依赖会产生连锁反应，可通过限制通信路径，减少产生或消费数据的模块子集，防止数据格式变更引发的连锁反应，有利于线性架构；使用中介，作为阻止变更传播的屏障，可用于转换数据格式、适配服务、隐藏身份或位置等。

-**延迟绑定时间**：目标是避免重新编译代码，支持即插即用操作，如Eclipse、Firefox的插件，Chrome的扩展等；通过配置文件（可能通过图形用户界面修改）和遵循定义的协议/接口（如关系数据库的SQL）来实现。

4.**课堂练习**：课堂练习要求学生4人一组，每组收到装有策略描述的信封，将策略分类到可用性（监控系统运行、从故障中恢复）、性能（减少系统负载、提高吞吐量或防止饥饿）、安全性、可用性、可测试性等类别中，检查分类是否正确，每组选择两个策略，由一名成员进行简要讲解。

关键问题

1.在电商系统中，若要提高系统的安全性，除了入侵检测策略，还可以采用哪些架构策略?

- 除入侵检测策略外，还可采用访问控制策略，限制不同用户对系统资源的访问权限，确保只有授权用户能访问敏感数据和服务；加密策略，对系统中的敏感数据进行加密处理，防止数据在传输和存储过程中被窃取或篡改；安全审计策略，记录系统中的操作行为，便于在发生安全事件时进行追踪和分析。

2.在可修改性策略中，定位修改策略的抽象公共服务和泛化模块这两种方式有何区别?

- 抽象公共服务是将多个模块共有的服务提取出来，形成一个新的模块，让原模块依赖这个新模块，从而减少重复代码，提高代码的可维护性，它侧重于消除代码重复；而泛化模块是使模块更通用，能够处理多种类型的数据或场景，降低特定需求变化对模块的影响，侧重于提高模块的通用性和适应性。

3.在课堂练习中，如何确保学生准确理解不同策略并正确分类?

- 教师在练习前应详细讲解各类质量属性和对应策略的概念及特点，通过实际案例加深学生理解；在学生分组后，教师可在组内讨论过程中进行巡视指导，及时解答学生的疑问；每组检查策略分类时，教师可引导学生对每个策略进行深入分析，对比策略与各类别的定义和特征，确保分类准确。

4.Views

详细总结

1.**架构视图概述**：架构在利益相关者间如同战场，良好的文档化架构能助其早期理解需求影响，促进讨论和谈判。架构视图是代表架构设计相关决策的一组图表或文档，单一视图无法完整描述系统，需多个视图从不同角度展示，且各视图应保持一致。常用表示符号有UML或非正式图形符号，UML扩展如SysML、ArchiMate、C4模型更适用于业务建模和面向更广泛受众。

2.**模块视图**

-**构成元素与关系**：模块是具有职责或服务的集合，有公开接口和隐藏的实现细节。模块视图通过分解、使用和泛化三种关系展示模块间联系，可用UML类和包图表示。

-**示例**：以空中交通管制应用为例，分为位置显示、冲突避免等功能模块，从下至上有平台层、通信层、应用支持层和子系统层，各层模块协同工作，如平台层提供基础支持，子系统层实现核心业务功能。

3.组件和连接器视图

-**构成元素**：运行时组件是计算和存储的主要单元，如对象、进程；连接器是组件间的交互机制，像方法调用、管道、套接字、中间件和协议等，常见交互风格有客户端 - 服务器、发布 - 订阅。

-**作用与示例**：该视图可解答系统运行时的关键问题，如主要执行组件及其交互方式等。以银行系统为例，客户端柜员与账户服务器采用客户端 - 服务器风格交互，柜员间通过发布 - 订阅风格通信，账户服务器有备份以满足可用性需求。

4.分配视图

-**定义与作用**：分配视图展示软件元素与外部元素（硬件、团队、软件仓库等）的映射关系，可解答软件元素在何处执行、存储，以及与开发团队和硬件的功能分配问题。

-**示例**：在Web系统架构中，Web浏览器、服务器、数据库服务器等软件组件分别与特定硬件和网络环境关联，如Web浏览器在用户计算机上运行，通过HTTP协议与Web服务器交互，Web服务器与数据库服务器通过JDBC连接，展示了软件与硬件、网络的协同关系。

关键问题

1.不同架构视图在项目开发的不同阶段分别起到什么作用？

- 在项目规划阶段，模块视图可帮助架构师梳理系统功能模块，明确职责边界；组件和连接器视图用于分析系统运行时的交互机制，评估性能和可扩展性；分配视图可规划软件与硬件资源的匹配，估算成本。在开发阶段，模块视图指导代码模块的编写和组织；组件和连接器视图辅助调试和优化组件交互；分配视图用于部署和测试环境的搭建。在维护阶段，模块视图便于理解系统架构，定位修改点；组件和连接器视图帮助分析运行时问题；分配视图可评估系统升级对硬件和团队的影响。

2.在实际项目中，如何确保不同架构视图之间的一致性？

- 建立统一的架构规范和标准，明确各视图的表示方法、命名规则和设计原则。在设计过程中，架构师牵头进行多轮评审，确保各视图在功能、接口、数据流向等方面保持一致。使用工具进行辅助设计，如一些建模工具可关联不同视图，当一个视图发生变化时，自动提示相关视图的潜在变更点。开发团队成员需充分沟通，在需求变更或设计调整时，同步更新相关视图，保证一致性。

3.以电商系统为例，模块视图、组件和连接器视图、分配视图分别重点关注哪些方面？

- 电商系统的模块视图重点关注商品管理、订单管理、用户管理等功能模块的划分与关系，如商品模块为订单模块提供商品信息。组件和连接器视图重点关注购物车组件、支付组件等运行时组件的交互，以及数据库连接、消息队列等连接器的使用，如购物车组件与订单组件通过消息队列传递订单信息。分配视图重

点关注电商系统软件组件与服务器硬件的分配，如将用户管理模块部署在高性能服务器上，以及开发团队成员与功能模块的对应关系，如前端团队负责用户界面相关模块开发。

Language Certification Center需求分析

Language Certification Center（语言认证中心，简称CC）的核心业务是为学生颁发官方语言证书，为提升盈利，力求实现申请人评估流程的自动化。具体业务流程和需求如下：

1.**学生申请流程**：学生通过网站提交证书申请，若申请未通过，需等待1个月后再次申请。认证中心收到申请后，在其注册库中寻找合适的认证评估人员（教授/学院）。根据认证类型，可能需学生完成笔试或参加视频会议面试。认证中心会从试题库中选题生成笔试题目，若涉及视频会议，还会安排学生与评估人员的会议时间。

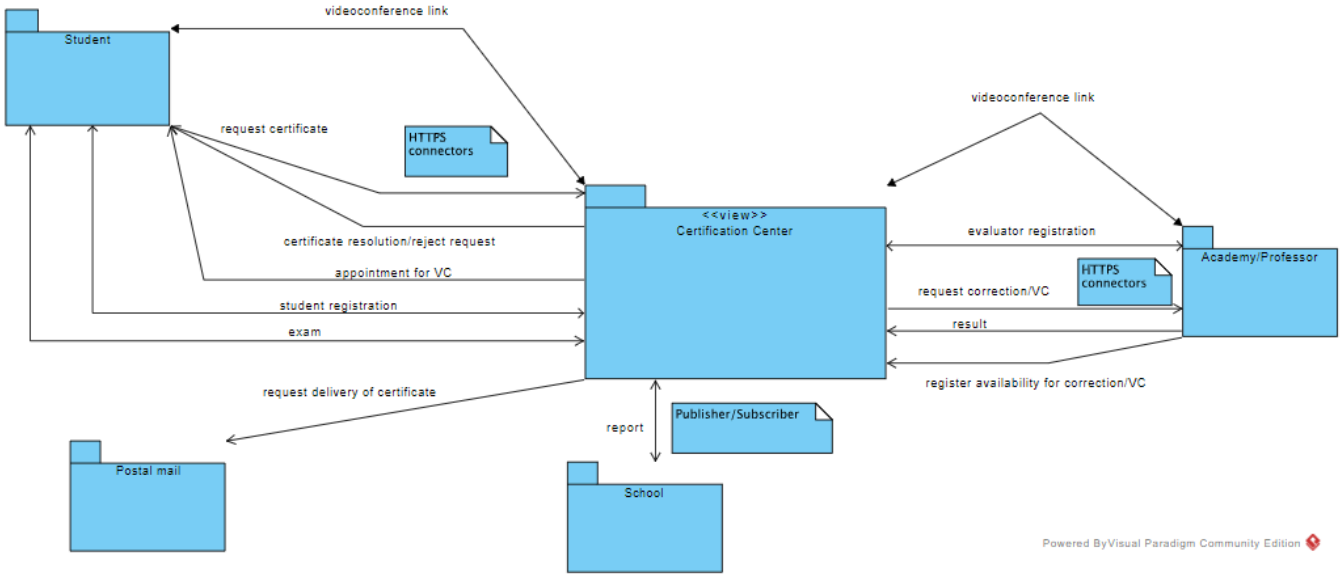
2.**评估与结果通知**：评估人员在一周内将考试成绩反馈给认证中心，认证中心再通知学生结果。若学生通过考试，认证中心会邮寄证书。

3.**数据交互与存储**：认证中心与学生、评估人员通过各自的网站进行交互，所有用户需先注册再登录。认证中心定期向订阅学校发送认证过程的统计报告。认证中心与学生、评估人员之间的通信（除视频会议数据外）需防止未经授权的访问，同时认证中心要实施备份策略，确保整个数据库的副本及时更新，以便在系统故障时能在半小时内于不同机器上恢复服务。

4.**技术组件与架构**：视频会议由认证中心的应用程序主持，预计未来视频会议技术发展，第三方视频会议组件（Videoconference Manager）会多次更换。笔试题目由依赖AI技术的第三方组件（Exam Factory）生成，鉴于AI技术发展迅速，该组件中期也可能被替换。系统的前端（网站和视频会议组件）和后端（业务逻辑和数据库）组件分别在不同机器上运行，必要时各组件可使用多台机器，故障时通过备份机器恢复，并利用其在正常运行时存储数据库副本。

基于结果图的解释

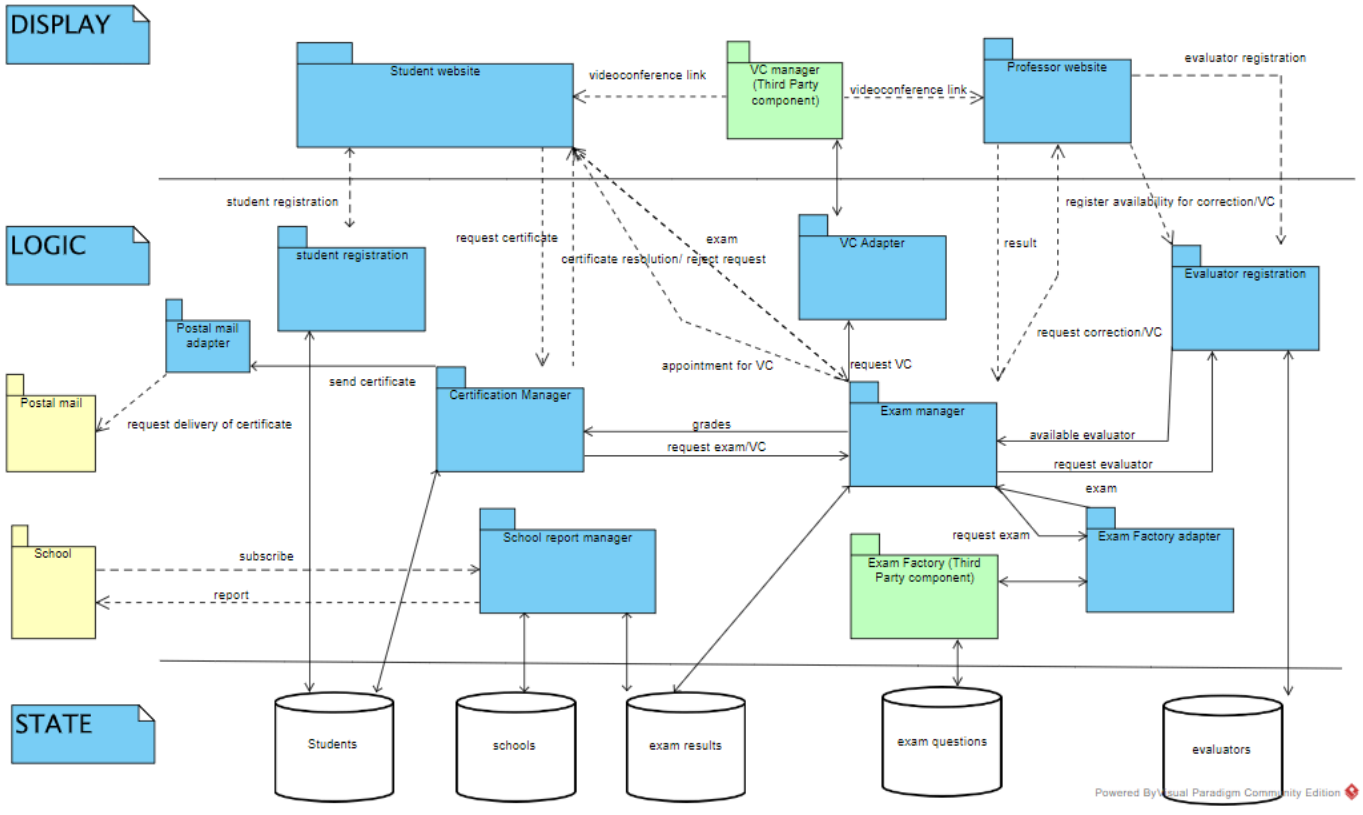
1.**组件和连接器视图（certification center component and connector view.pdf）**



-**组件**：包括学生、认证中心、学院/教授（评估人员）、学校。学生发起证书申请，认证中心负责协调评估流程，学院/教授进行评估，学校接收统计报告。

-**连接器与数据交换**：学生与认证中心通过HTTPS连接器进行通信，交换的数据有证书申请请求、考试结果等；认证中心与学院/教授之间同样通过HTTPS连接器通信，传递评估任务请求、考试成绩等数据。此外，还存在Publisher/Subscriber模式的连接器，用于更灵活的数据交互（可能用于通知相关方重要事件，如评估任务分配、成绩发布等）。视频会议则通过专门的视频会议链接进行连接，实现学生与评估人员之间的沟通。

2.模块视图（Certification Center module view.pdf）

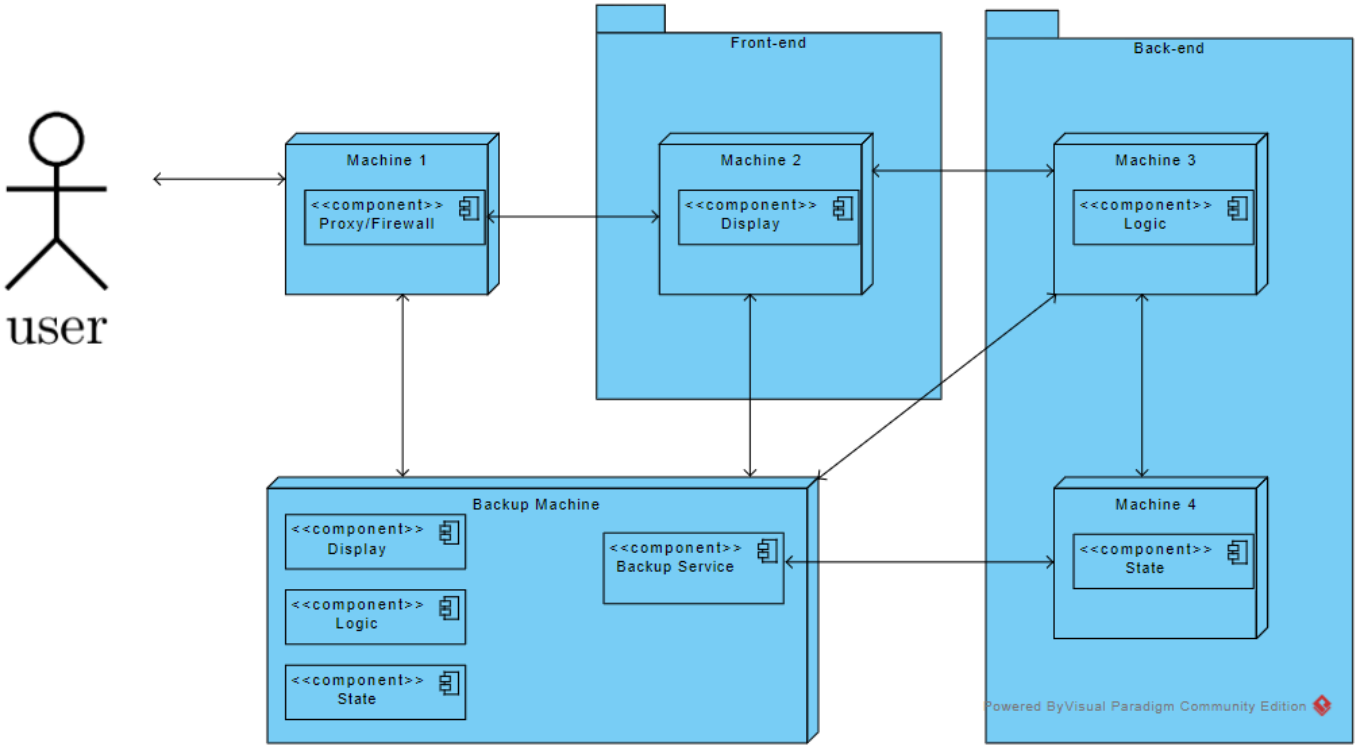


-**显示（DISPLAY）模块**：负责与用户（学生、评估人员）进行交互的界面展示，如学生网站和教授网站的页面呈现，以及视频会议链接的管理（可能涉及视频会议的发起、接入等功能）。

-**逻辑（LOGIC）模块**：包含多个子模块，如认证管理（Certification Manager）负责处理证书申请、决议和拒绝请求；考试管理（Exam manager）负责协调考试相关事宜，如向Exam Factory请求考试题目；视频会议适配器（VC Adapter）用于适配视频会议组件，处理与视频会议相关的逻辑；学校报告管理（School report manager）负责生成并向学校发送统计报告。各子模块之间通过数据交互协同工作，如认证管理模块向考试管理模块请求考试，考试管理模块向Exam Factory请求题目，再将题目传递给认证管理模块用于学生考试。

-**状态（STATE）模块**：存储系统运行所需的各种数据，如学生注册信息、考试题目、评估人员信息、学校订阅信息、考试结果等，为逻辑模块的业务处理提供数据支持。

3.分配视图（certification center allocation view.pdf）



-**前端与后端分离**：前端组件（Display）部署在Machine 1，后端的逻辑（Logic）和状态（State）组件分别部署在Machine 2和Machine 3。这种分离有助于提高系统的可维护性和扩展性，前端专注于用户界面展示和交互，后端负责业务逻辑处理和数据存储。

-**备份与安全机制**：设置Proxy/Firewall（代理/防火墙），用于保障系统通信的安全性，防止未经授权的访问。备份机器（Backup Machine）在正常运行时存储数据库副本，在前端或后端出现故障时，可用于恢复系统，确保服务的连续性，符合认证中心对数据备份和快速恢复的需求。

城市电力管理系统需求分析

格林镇（Greentown）是一座拥有8万居民的城市，日均耗电量600兆瓦时，高峰需求可达40兆瓦。为满足城市用电需求，有太阳能电站、风电场和水电站三座电站供电。由于城市电力需求全天变化，且太阳能电站和风电场受自然条件影响，加之电站可能出现故障，因此需要对电力供应进行有效管理。

中央控制中心（Central Control）负责决策各电站发电量，通过软件应用和通信基础设施每30分钟提前10分钟向电站发送发电指令，并根据多种因素提前2天制定初步发电计划，实时根据实际情况调整。同时，中央控制中心还需应对输电网故障，通过传感器检测故障并推荐修复程序。其软件应用需为操作员提供多种视图，且与多个外部系统通信。

该系统有诸多质量属性需求，如软件或系统故障时需记录状态、停机时间控制在1秒内、支持撤销发电请求、验证用户身份、保护数据安全、高效轮询传感器数据、降低通信协议变化影响等。部分关键模块开发将分包，且需复用一个遗留模块，并对其输入数据格式进行适配。

基于结果图和QA Classification的解释

1. **组件和连接器视图（image.png）**：该图展示了中央控制中心与外部子系统的交互。中央控制中心通过https连接器（用于Web服务）与电站、天气预报提供商、城市事件提供商等进行通信，交换发电指令、天气预报、重要事件等数据。与网络传感器和城市传感器通过安全套接字连接器传输传感器数据和当前电力需求数据。此外，还使用发布/订阅连接器，可能用于及时推送重要信息，如电站的最大可用功率、重要事件通知等。通过这些连接器和数据交互，实现了中央控制中心对电力生产、传输和使用情况的全面监控与管理。
2. **质量属性需求分类及策略（demand production electricity management QA classification.pdf）**
    - **可测试性**：需求是在软件组件或整个应用程序发生故障时，技术人员能查明应用程序当时的状态。采用的策略是在所有模块中内置监视器记录日志，以便故障发生时可通过日志分析故障原因和系统当时的状态。
    - **可用性**：要求整个系统或关键软件组件发生故障时，停机时间最多1秒。通过主动冗余策略实现，使用3台计算机，A和B运行中央控制软件，C监控主计算机A的运行情况，当A出现故障时，B能迅速接替工作，保证系统持续运行。
    - **可用性**：操作员发出发电请求后，需在5分钟内有机制撤销请求。在当前生产管理控制台增加支持取消或撤销生产请求的功能，方便操作员及时调整发电指令，提高操作的灵活性。
    - **安全性**：操作员下达发电指令时需验证用户身份，且只有中央控制中心能向电站发送指令，并对数据交换保密。通过使用密码/生物识别ID验证用户身份，

使用数字证书认证中央控制中心，采用安全协议保证数据机密性，防止未经授权的访问和数据泄露。

- **性能**：系统需每5分钟在1分钟内轮询所有传输网络传感器。采用提高计算效率（如使用快速通信协议）和并发处理的策略，确保系统能及时获取传感器数据，为电力管理决策提供实时依据。
- **可修改性**：与电站、网络传感器和城市传感器的通信协议易变，需降低其对中央控制软件的影响。通过关注点分离和使用中介的策略，设置专门与外部系统通信的模块，封装协议实现和硬件依赖，当通信协议变化时，只需修改这些专门模块，减少对整个系统的影响。

5.Architectural Styles

详细总结

1.**软件架构风格的定义与作用**：软件架构风格是在给定开发环境（包含问题以及技术/资源因素）下，一系列可应用的架构设计决策的集合。这些决策针对特定系统，能够激发每个最终系统产生有益的特性，是总结软件系统设计经验的主要方式，一个软件系统可能融合多种风格。

2.**常见软件架构风格分类及特点**

-**传统和受语言影响的风格**：包括主程序和子程序、面向对象、分层、客户端 - 服务器、虚拟机等风格。例如客户端 - 服务器风格，由客户端向服务器请求服务，服务器返回所需信息，采用同步调用（如REST API）的连接器，具有计算和数据集中化的特点，适用于服务器强大、客户端轻量且注重安全和可修改性的场景，但存在单点故障风险和网络带宽瓶颈问题 。其变体如增加多个服务器和负载均衡器可提升可用性和性能，使用容器则能提高性能和可维护性。

-**数据 - 流风格**：如批处理顺序、管道 - 过滤器、共享内存等风格。以管道 - 过滤器风格为例，通过管道连接过滤器处理数据流，过滤器相互独立，便于在不同场景复用，但不适用于需要随机访问数据或组件间频繁交互的情况，常见于UNIX过滤器、编译器等。

-**其他风格**：涵盖基于规则、解释器、移动代码、事件 - 驱动、对等网络等风格。比如移动代码风格，代码可移动到其他主机执行，能利用主机计算能力，根据实现方式分为代码按需获取、远程评估、网格计算、MapReduce范式、移动代理等类型；事件 - 驱动风格通过事件总线实现组件间解耦，具有高扩展性，但异步通信导致事件处理不确定，常用于嵌入式系统、金融市场、物联网等领域。

3.**典型风格的具体示例**

-**虚拟机风格**：以Android系统为例，它采用分层架构，包括Linux内核、服务层、Android运行时、库和应用框架等。各层分工明确，如Linux内核提供硬件抽象，应用框架为应用程序提供高层服务。这种架构在安全性、可修改性、可用性等方面有诸多设计考量，例如应用安装时需申请权限保证安全，应用可注册服务供其他应用使用提升可修改性。

-**基于规则的系统风格**：Prolog是典型例子，通过定义事实和规则进行推理。例如根据定义的家庭关系事实和规则，可以查询祖辈关系等。

-**对等网络风格**：BitTorrent是混合对等网络的代表，通过种子文件、追踪器等机制实现大文件分发。其设计决策如Choke算法、最稀有优先算法等提高了性能，多追踪器种子文件和无追踪器模式则提升了可用性；Kademlia是纯对等网络，基于分布式哈希表实现文件定位和存储，每个节点存储其他节点文件位置信息，通过特定算法查找文件 。

风格名称	组件	连接器	拓扑结构	适用场景	优势	局限
客户端 - 服务器风格	客户端、服务器	同步调用（如 REST API）	两层结构（客户端、服务器）	服务器强大、客户端轻量，注重安全和可修改性场景	计算和数据集中化	单点故障风险，网络带宽可能成瓶颈
虚拟机风格	包含多个程序的层	同步调用	线性或有向无环图	操作系统、网络协议等	良好的可修改性，高层不受低层实现变化影响	多层可能导致效率低下，多层共享数据时不适用
管道 - 过滤器风格	过滤器	数据流传的管道	线性	数据流可并发处理场景	过滤器独立性便于复用	需要随机访问数据、组件间需频繁交互时不适用
基于规则的系统风格	UI、推理引擎、规则库、事实库	直接内存访问或同步调用	三层系统	推理问题、快速原型开发和迭代开发	通过增减规则/事实改变行为	规则数量大且对性能要求高时，维护困难
移动代码风格	代码/数据提供者、执行坞	网络协议	网络	提升服务器可用性/性能、处理大数据	利用主机计算能	安全性受影响，依赖可用主机

风格名称	组件	连接器	拓扑结构	适用场景	优势	局限
				集、动态定制本地处理节点	力，动态适应性强	
事件-驱动风格	独立的并发事件生成器和消费者	事件总线	组件共享事件总线	嵌入式系统、金融市场、物联网系统	强解耦、高扩展性、易演进	异步通信，事件处理不确定
对等网络风格	对等节点（兼具客户端和服务器功能）	专用网络协议	动态定义的网络	信息和操作分布式、自组织网络场景	高可用性和扩展性	信息检索有时效性要求时存在问题，存在信任风险

关键问题

1.在选择软件架构风格时，如何平衡系统的性能和可维护性？

- 答案：不同软件架构风格对性能和可维护性的影响各异。例如客户端 - 服务器风格，增加服务器数量和使用容器能提升性能，但可能增加维护复杂度；虚拟机风格的多层结构可提高可维护性，但过多层次会降低性能。选择时需综合考虑系统需求，若系统对性能要求极高，可优先考虑能提升性能的风格变体；若可维护性是关键，则选择结构清晰、易于修改的风格，如虚拟机风格或采用良好分层设计的架构。同时，在设计过程中可以通过优化实现细节，如合理设置缓存、优化算法等，在一定程度上平衡两者关系。

2.移动代码风格在实际应用中如何解决安全问题？

- 答案：移动代码风格因代码在不同主机间移动，存在安全隐患。可以采用加密技术对移动代码和数据进行加密，防止传输过程中被窃取或篡改；实施严格的代码签名机制，确保代码来源可靠，只有经过授权的代码才能在目标主机上执行；对执行环境进行安全配置，限制移动代码的权限，如限制其对系统资源的访问范围，避免恶意代码破坏系统。

3.对等网络风格中，纯对等网络和混合对等网络在资源发现机制上有何不同？

- 答案：纯对等网络（如Kademlia）中，所有节点地位平等，资源发现通过在整个网络中传播查询实现。每个节点不仅存储数据，还记录其他节点数据的位置信息，根据节点和文件的全局ID计算距离，通过特定算法查找文件。而混合对等网络（如BitTorrent）存在特殊节点或服务器作为目录，帮助定位资

源。种子文件提供追踪器URL，追踪器协助客户端找到拥有文件片段的节点，相比纯对等网络，资源发现更高效，但特殊节点或服务器可能成为性能瓶颈，影响可用性。

6.Architectural Patterns

详细总结

1.《architectural patterns1.pdf》

-**软件模式的定义**：软件模式是对特定设计场景中反复出现问题的成熟通用解决方案，如MVC模式解决交互式应用中用户界面易变和数据展示多样化的问题。

-**软件模式的特点**：是经过验证的构建模块，体现经验丰富从业者的设计知识，有助于构建具备期望特性的软件，还提供了简洁描述设计的词汇。

-**与其他概念的关系**：架构模式提供预定义子系统及其职责和关系；设计模式细化子系统或组件关系；架构风格侧重定义组件和连接器词汇及聚合规则，而软件模式更聚焦于解决设计问题。

-**POSA模式分类**：涵盖从整体任务分解（如分层、管道 - 过滤器、黑板模式）、人机交互（MVC、Presentation - Abstraction - Control）、分布式应用（Broker）到系统扩展适应（Reflection、Microkernel）等几类。

2.《architectural patterns2.pdf》

-**MVC模式**：1979年由Trygve Reenskaug提出，如今在许多GUI和Web框架中广泛应用。通过分离模型、视图和控制器，解决交互式应用中数据展示和界面变化的问题，存在多种变体，如分层 - MVC、基于Web的MVC等。

-**Presentation - Abstraction - Control（PAC）模式**：适用于多用户或分布式应用，将交互应用中的代理分为抽象、控制和表示三层，各层协作，以书店系统为例，不同层级的代理协同处理业务逻辑。

-**Broker模式**：支持分布式系统中异构客户端和服务器在运行时集成。通过客户端和服务器代理、Broker组件及远程服务目录，实现位置透明的服务调用，在股票市场、CORBA、Android隐式意图等场景有应用，但存在性能较低、容错性差和测试困难等缺点 。

3.《architectural patterns3.pdf》

-**Microkernel模式**：适用于易扩展和适应不同使用场景的应用开发。将应用分为最小功能核心和扩展功能插件组件，如Eclipse、Chrome等应用以及保险公司业务逻辑处理都采用了该模式，但存在可扩展性瓶颈和设计实现复杂的问题。

-**Reflection模式**：一种编程机制，允许程序在运行时检查自身结构、修改代码和创建未知类型对象。在支持插件扩展的应用、持久化框架、调试测试等场景应用广泛。Reflection架构模式将软件分为元级别和基础级别，通过元对象协议（MOP）实现动态修改，但存在效率较低和元级别修改易出错的问题。

-**Microservice模式**：适用于服务器端企业应用，将应用拆分为松耦合服务，每个服务可独立部署和扩展，通过REST或消息传递进行通信。以电商系统为例，不同服务负责不同业务功能，但该模式存在操作复杂、协作耗时和安全风险高等缺点。

模式名称	适用场景	核心组件	解决的主要问题	优势	局限
MVC 模式	交互式应用	模型、视图、控制器	用户界面易变和数据多方式展示问题	提高 GUI 可修改性，便于维护	大型项目中代码复杂度增加，协调成本高
PAC 模式	多用户或分布式应用	抽象层、控制层、表示层（可选）	多用户应用中各代理状态维护、UI 管理及交互问题	层次清晰，便于管理和扩展	设计相对复杂，不适用于简单应用
Broker 模式	分布式异构系统	客户端代理、服务器代理、Broker 组件	分布式系统中组件间位置透明的服务调用、服务动态管理问题	支持异构组件集成，方便服务添加和删除	性能较低，容错性差，测试困难
Microkernel 模式	易扩展和适应不同场景的应用	微内核、插件组件	应用功能扩展和核心功能分离问题	便于应用扩展，功能模块化	微内核可能成为瓶颈，设计实现复杂
Reflection 模式	需在运行时自我修改的应用	元级别（元对象）、基础级别	简化系统变化的代码实现，应对不同场景下的变化	实现软件系统动态修改，提高灵活性	效率较低，元级别修改易出错

模式名称	适用场景	核心组件	解决的主要问题	优势	局限
Microservice模式	服务器端企业应用	多个松耦合服务	高扩展性、容错性以及持续部署和快速演进需求	可独立部署和扩展，提高可用性	操作复杂，协作耗时，安全风险高

关键问题

1.MVC模式和PAC模式在处理用户交互方面有何不同?

> - MVC模式通过控制器处理用户交互，将用户操作转化为对模型的修改，进而更新视图；而PAC模式中每个代理都有自己的控制层来处理与其他代理的通信以及用户交互，并且将交互相关的逻辑与数据模型和业务逻辑分离，更适用于多用户或分布式应用中复杂的交互场景。

2.Broker模式与Microservice模式在分布式系统中的应用有何区别?

> - Broker模式主要解决分布式系统中异构组件的集成问题，通过代理和Broker组件实现位置透明的服务调用，更侧重于服务的统一管理和调用；Microservice模式则将应用拆分为多个松耦合的服务，每个服务独立部署和扩展，更注重系统的可扩展性、容错性以及持续部署和快速演进，适用于大型复杂的服务器端企业应用。

3.Reflection模式和Microkernel模式在实现应用扩展方面有什么不同的思路?

> - Reflection模式通过在运行时修改软件结构和行为来实现扩展，将软件分为元级别和基础级别，利用元对象描述软件行为，基础级别根据元对象保持独立于易变部分；Microkernel模式则是将核心功能与扩展功能分离，通过插件组件添加新功能，核心系统提供基本服务，插件组件负责扩展功能，更侧重于功能的模块化和插件化扩展。

例子：

这3篇文档围绕议会选举选票重新计票场景展开，探讨不同软件架构风格在该场景下的应用及质量属性差异。《recounting of votes.pdf》提出基于客户端 - 服务器风格且采用MVC模式的架构设计需求，以及移动代码风格和对等网络风格架构的设计要求，并需对比三种方案的质量属性。《recounting of votes SOL.pdf》给出了客户端 - 服务器风格且采用MVC模式的架构设计结果，包括组件和连接器视图以及政府子系统的模块化视图。《Quality attribute comparison.pdf》对比了客户端 - 服务器（单服务器）、客户端 - 服

务器 + 移动代码（单服务器）和对等网络这三种架构风格在可用性、性能、安全性、可扩展性和可测试性等质量属性方面的表现。

1.客户端 - 服务器风格且采用MVC模式的架构分析

-**组件和连接器视图**：在《recounting of votes SOL.pdf》中，学校、公民、政府和管理员为主要组件，通过https连接器交互。学校向政府发送计票结果和学校状态；公民向政府请求选举结果图表；管理员通过政府系统查看学校状态和计票结果。这种架构利用https保证通信安全，符合《recounting of votes.pdf》中对通信保密性的要求。

-**政府子系统模块化视图（MVC模式）**：公民监听器和学校监听器作为控制器，分别处理公民图表请求和学校结果及状态通知。模型负责存储和管理选举相关数据，如学校计票结果、政府更新的初步选举结果等。视图通过图表构建器生成图表展示数据，管理员视图和公民获取的图表都依赖于此。例如，公民请求图表时，公民监听器接收请求，从模型获取数据，经图表构建器处理后以HTML图表形式呈现给公民。

2.**移动代码风格架构的可能修改**：结合《recounting of votes.pdf》和《recounting of votes SOL.pdf》分析，在原客户端 - 服务器架构基础上应用移动代码风格，可将部分数据处理和展示逻辑以移动代码形式部署到客户端设备上。比如公民获取选举结果图表时，可在客户端执行图表渲染代码，减轻服务器压力，提升响应速度。但这可能带来安全风险，如代码在客户端执行时可能遭受攻击。同时，移动代码风格对网络依赖性强，网络不稳定时可能影响性能。

3.**对等网络风格架构分析**：若采用对等网络风格，如《recounting of votes.pdf》所述，政府不再负责收集所有计票结果和公布初步结果。各学校和公民节点直接交互，形成分布式网络。每个节点既是数据提供者，也是数据请求者。这种架构可避免政府服务器成为性能瓶颈，提高系统可用性和可扩展性。然而，对等网络中节点间信任关系难以建立，数据安全性较低，如可能存在恶意节点篡改计票结果。而且，分布式环境下数据一致性维护困难，会增加测试难度。

4.质量属性对比结果解释（参考《Quality attribute comparison.pdf》）

-**可用性**：对等网络风格（P2P）表现最佳（++），因其分布式结构，不存在单点故障，一个节点故障不影响其他节点工作。客户端 - 服务器 + 移动代码（C/S + Mobile Code）次之（+），移动代码可在一定程度上分担服务器压力，但仍依赖服务器；客户端 - 服务器（单服务器）最差（-），服务器故障会导致服务中断。

-**性能**：对等网络风格（P2P）表现最佳（++），分布式处理可提高整体性能；客户端 - 服务器 + 移动代码（C/S + Mobile Code）次之（+），移动代码减轻服务器负载提升性能；客户端 - 服务器（单服务器）最差（-），服务器易成为性能瓶颈。

-**安全性**：客户端 - 服务器（单服务器）表现最佳（++），集中式管理便于统一安全策略；客户端 - 服务器 + 移动代码（C/S + Mobile Code）次之（+），移动代码增加安全风险；对等网络风格（P2P）最差（-），节点间信任管理困难，数据易受攻击。

-**可扩展性**：对等网络风格（P2P）表现最佳（++），易于添加新节点扩展网络；客户端 - 服务器 + 移动代码（C/S + Mobile Code）和客户端 - 服务器（单服务器）为中级（+），但单服务器架构扩展时可能需大量改造。

-**可测试性**：客户端 - 服务器（单服务器）表现最佳（++），集中式架构便于测试；客户端 - 服务器 + 移动代码（C/S + Mobile Code）次之（+），移动代码增加测试复杂性；对等网络风格（P2P）最差（-），分布式环境下数据一致性和节点交互复杂，测试难度大。