# What is a Pattern? （了解）

An architectural pattern establishes a relationship between:

* **A context.** A recurring situation giving rise to a problem. （一种反复出现问题的情况。）
* **A problem**
* **A solution.**A successful architectural resolution to the problem, appropriately abstracted, determined and described by:
  + A set of element types
  + A set of interaction mechanisms or connectors（连接器）
  + A topological layout（拓扑布局） of the components
  + A set of semantic constraints covering topology, element behavior, and interaction mechanisms

# Pattern Catalogue（目录）（掌握）

* **Module patterns（模块）**
  + **Layer Pattern（掌握）**
  + **OOP Pattern**
* **Component and Connector Patterns**
  + **Broker Pattern（代理人模式）（掌握）**
  + **MVC Pattern（掌握）**
  + **Pipe and Filter Pattern（管道-过滤器模式）（掌握）**
  + **Client-Server Pattern（掌握）**
  + **Peer-to-Peer Pattern（掌握）**
  + **SOA Pattern （掌握）**
  + **Publish-Subscribe Pattern（发布-订阅模式）（掌握）**
  + **Shared-Data Pattern （掌握）**
* **Allocation Patterns**
  + **Map-Reduce Pattern （****掌握）**
  + **Multi-Tier Pattern （多级模氏）（掌握）**
* **Patterns are mainly determined by connectors**

## Layer Pattern （掌握）

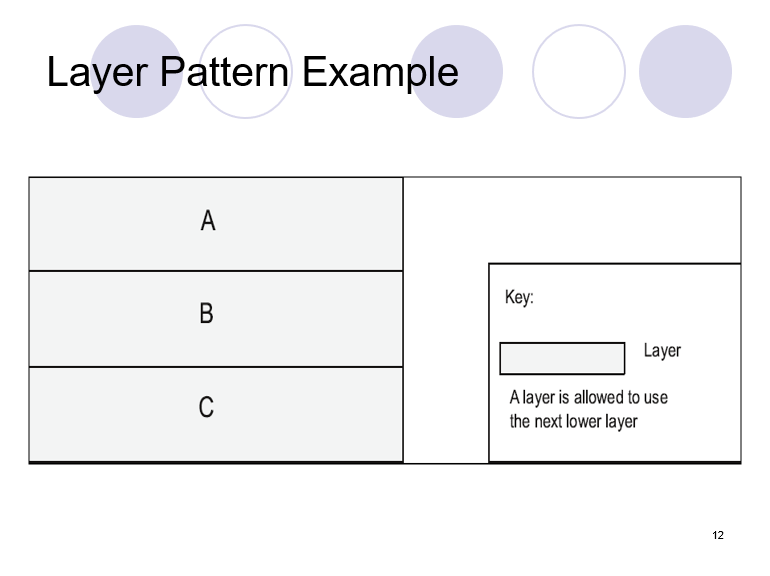
* **Context:** Separation（分离） of concerns.
* **Problem:** The software needs to be segmented in such a way that the modules can be developed（开发） and evolved（演化） separately with little interaction among the parts, supporting portability（可移植性）, modifiability（可修改性）, and reuse.（重用）
* **Solution:** Divides the software into units called layers. Each layer is a grouping of modules that offers a cohesive（紧密） set of services. The usage must be unidirectional.（单向）

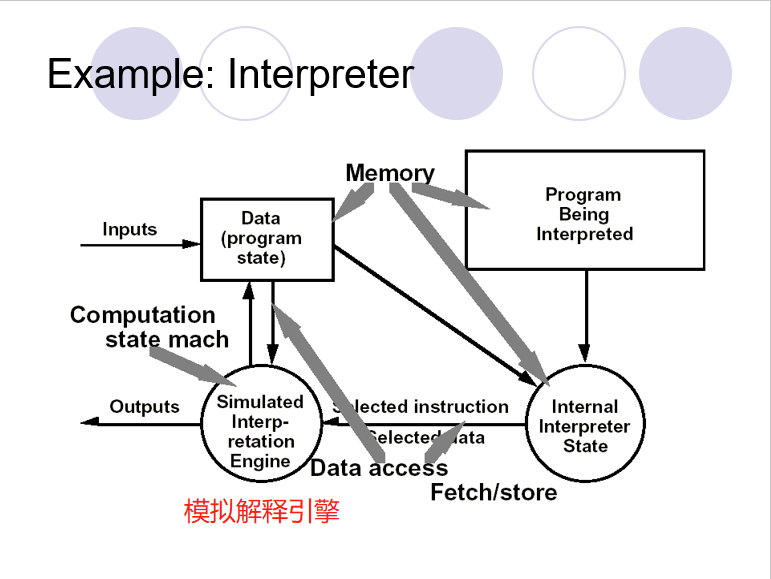
**Layer Pattern Solution**

* **Overview:** The layered pattern defines layers and a use relation.
* **Elements:** Layer.（图层？）
* **Relations:** Allowed to use.
* **Constraints:** 
  + Every piece of software is allocated to exactly one layer.
  + There are at least two layers (but usually there are three or more).
  + The allowed-to-userelations should not be circular（循环） (i.e., a lower layer cannot use a layer above).
* **Weaknesses:** 
  + The addition of layers adds up-front cost and complexity to a system.

Layers contribute a performance penalty（下降）.

**Example**



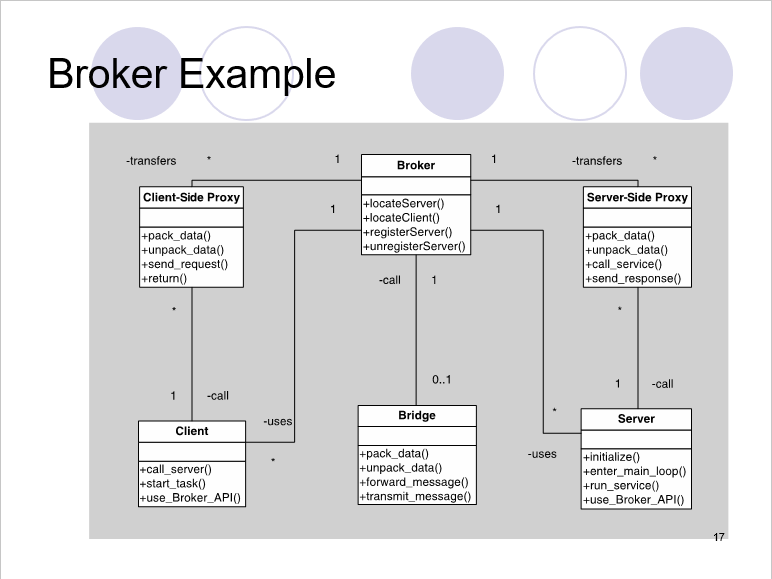


## **Broker Pattern 代理模式**（掌握）

* **Context**: distributed（分布式） services interoperate（互操作） with each other.
* **Problem**: How do we structure distributed software so that service users do not need to know the nature and location of service providers, making it easy to dynamically change the bindings between users and providers?
* **Solution**: The broker pattern separates users of services (clients) from providers of services (servers) by inserting an intermediary, called a broker.

**Broker Solution**

* **Overview**: 代理调解（mediates）多个客户端和服务器之间的通信。
* Elements:
  + 客户端，服务请求者
  + 服务器，服务提供者
  + 代理，中间人
  + 客户端代理，一个帮助中介，管理与代理的实际通信
  + 服务器端代理，一个帮助中介，管理与代理的实际通信
* Relations: （attachment）附件关系将客户端和服务器与代理关联。
* Constraints: 客户端只能附加到代理。 服务器只能附加到代理。
* Weaknesses:
  + 代理增加了一层间接，可能是沟通的瓶颈。
  + 代理可能是单点故障。
  + 代理增加了前期的复杂性。
  + 代理可能是安全攻击的目标。
  + 代理可能难以测试。

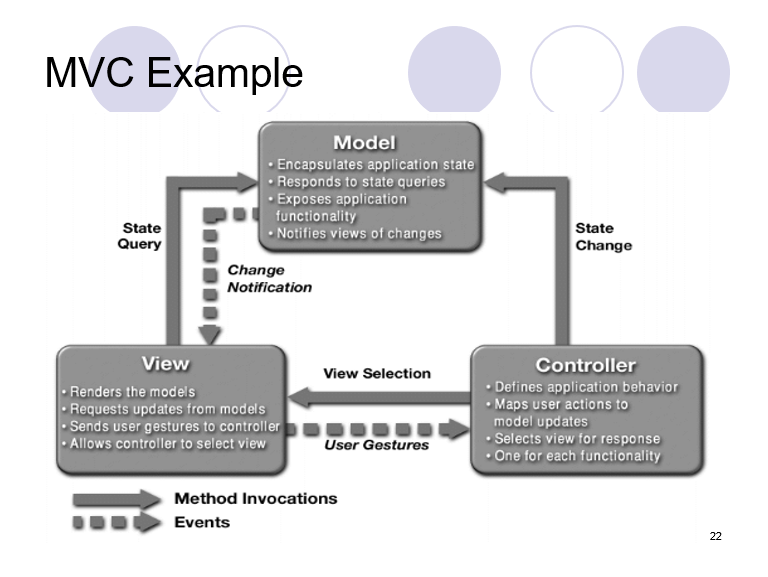


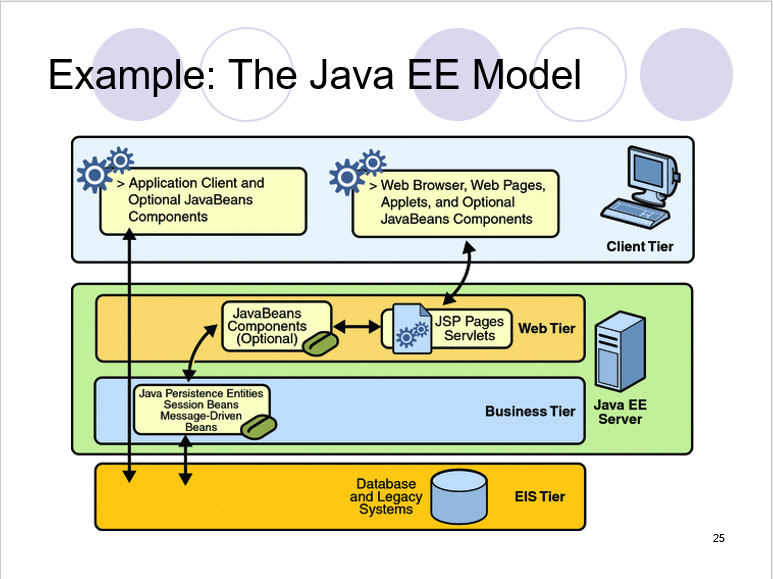
## Model-View-Controller Pattern（掌握）

* **Context**: 从模型中分离视图。
* **Problem**: 用户界面功能如何与应用程序功能分开，但仍然能够响应用户输入或底层应用程序数据的变化？ 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？
* **Solution**: 模型 - 视图 - 控制器（MVC）模式将应用程序功能分为三种组件。

**MVC Solution**

* **Overview:** MVC模式将系统功能分解为模型，视图和控制器，它们在模型和视图之间进行调解。
* **Elements:** 
  + The model.
  + The view.
  + The controller.
* **Relations:** 通知（notifies）关系连接模型，视图和控制器的实例，通知相关状态更改的元素。
* **Constraints:** 
  + 模型组件不应直接与视图交互。
* **Weaknesses:**
  + 对于简单的用户界面（interfaces），复杂性可能不值得。
  + 某些用户界面工具包可能不适合模型，视图和控制器抽象。



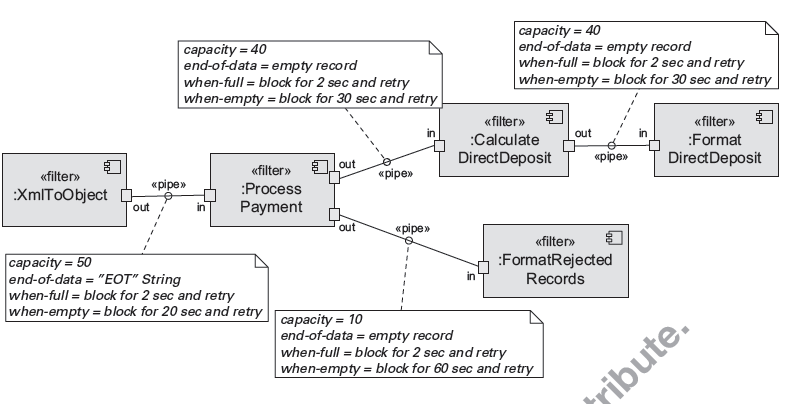


## Pipe and Filter Pattern （掌握）

* **Context:** 处理数据流。
* **Problem:** 这些系统需要通过简单，通用的交互机制分为可重用（reusable），松散耦合（loosely coupled）的组件。
* **Solution:** 管道和过滤器模式中的交互模式通过数据流的连续转换来表征。

**Pipe and Filter Solution**

* **Overview:** 数据由管道连接的过滤器转换。
* **Elements:** 
  + ***Filter*.**
  + ***Pipe*.**
* **Relations:** 附件（attachment）关系将过滤器的输出与管道输入相关联，反之亦然。
* **Constraints:**
  + 管道连接过滤器输出端口以过滤输入端口。
  + 连接的过滤器必须与连接管道传递的数据类型一致。

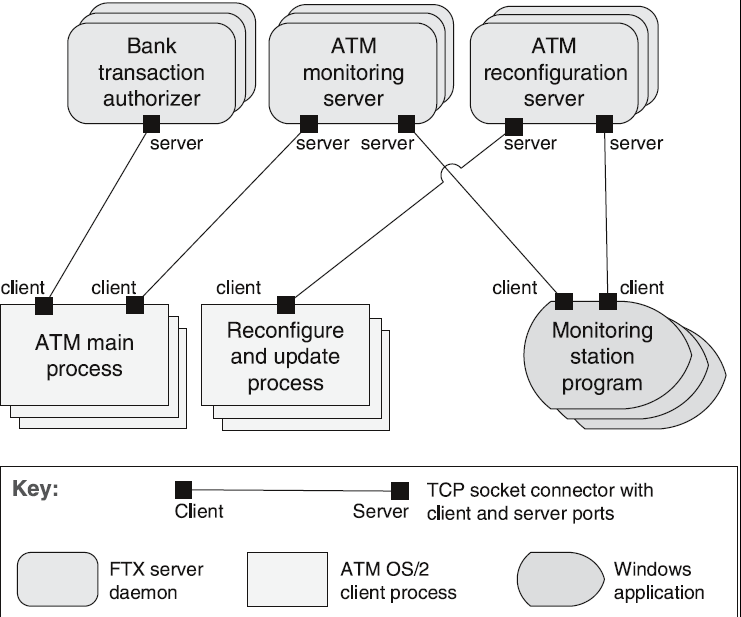


## Client-Server Pattern （掌握）

* **Context:** 大量分布式客户端希望访问共享资源和服务。
* **Problem:** 我们希望通过集中控制这些资源和服务来提高可扩展性和可用性，同时在多个物理服务器上分配资源本身。
* **Solution:** 客户端通过请求服务器的服务进行交互，服务器提供一组服务。

**Client-Server Solution**

* **Overview:** 客户端启动与服务器的交互，根据需要从这些服务器调用服务并等待这些请求的结果。
* **Elements:** 
  + ***Client*.**
  + ***Server*.**
* ***Request/reply connector:*** 采用请求/回复协议的数据连接器，由客户端用于调用服务器上的服务。
* **Relations:** 附件（attachment）关系将客户端与服务器相关联。
* **Constraints:** 
  + 客户端通过请求/回复连接器连接到服务器。
* **Weaknesses:** 
  + 服务器可能是性能瓶颈（performance bottleneck）.
  + 服务器可能是单点故障（a single point of failure）.
  + 在构建系统之后，关于在何处定位功能的决定通常是复杂且成本高昂的。

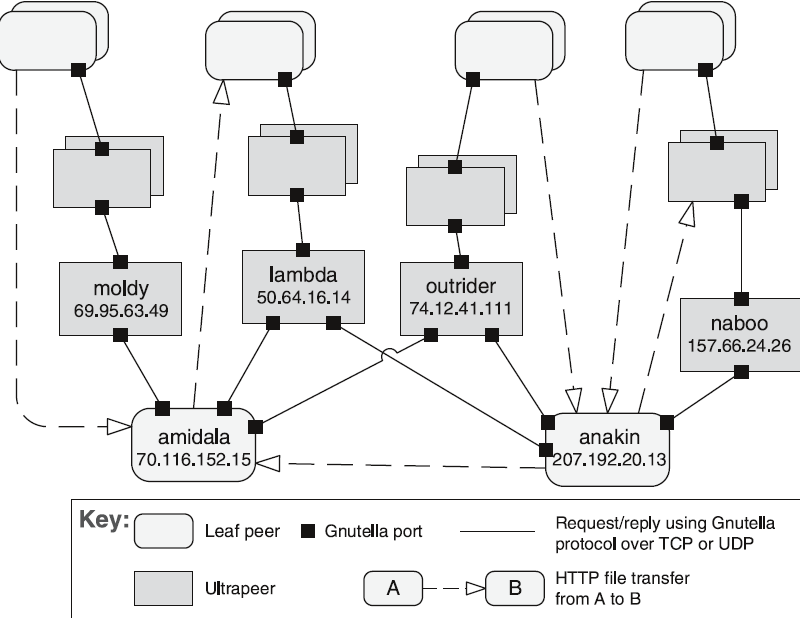


## Peer-to-Peer Pattern （掌握）

* **Context:** 分布式计算实体（entities）被认为是相等的.
* **Problem:** 一组“相等”的分布式计算实体如何通过通用协议相互连接，以便它们能够以高可用性和可扩展性组织和共享其服务？
* **Solution:** 在对等（P2P）模式中，组件直接作为对等体进行交互。 所有对等体都是“相等的”，并且没有对等体或对等体组对系统的健康状况至关重要。

**Peer-to-Peer Solution**

* **Overview:** 通过合作伙伴实现计算。
* **Elements:** 
  + Peer.
  + Request/reply connector.
* **Relations:** 关系将对等关系与其连接关联。 附件（attachment）可能会在运行时更改。
* **Constraints:** 可能会对以下内容施加限制（Restrictions）：
  + 任何给定对等方的允许附件数
  + 用于搜索对等体的跃点（hops）数
  + Which peers know about which other peers
  + 一些P2P网络采用星形拓扑（topologies）结构组织，其中对等体仅连接到超级节点（supernodes）。
* **Weaknesses:** 
  + 管理安全性，数据一致性，数据/服务可用性，备份和恢复都比较复杂。
  + 小型点对点系统可能无法始终如一地实现性能和可用性等质量目标。

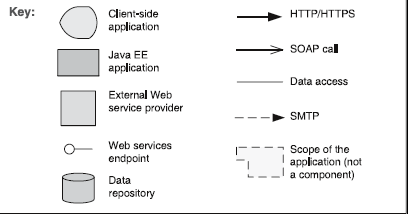
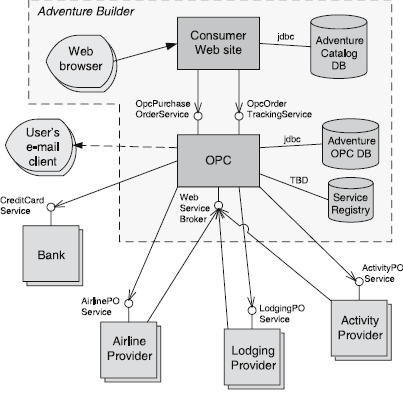


## Service Oriented Architecture Pattern （掌握）

* **Context**: 许多服务在没有任何详细的实现知识的情况下进行互操作（interoperates）。
* **Problem**: 我们如何支持在不同平台上运行并使用不同组织提供的不同实现语言编 写并分布在Internet上的分布式组件的互操作性？
* **Solution**: 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。

**Service Oriented Architecture Solution**

* **Overview:** 计算是通过网络上的一组合作服务（cooperating services）实现的。
* **Elements:** 
  + **Components:**
    - 服务提供者 Service providers.
    - 服务消费者Service consumers.
  + **ESB，一种可以在服务提供者和消费者之间路由和转换消息的中间元素。**
  + 服务登记处Registry of services.
  + 业务流程服务器（Orchestration server），它根据业务流程和工作流的语言协调服务使用者和提供者之间的交互。
  + **Connectors:**
    - SOAP连接器，它使用SOAP协议在Web服务之间进行同步通信，通常通过HTTP进行。
    - REST连接器，它依赖于HTTP协议的基本请求/回复操作。
    - 异步消息传递连接器，它使用消息传递系统提供点对点或发布 - 订阅异步消息交换。
* **Relations:** 各个连接器可用的不同类型组件的附件
* **Constraints:** 服务使用者与服务提供者连接。
* **Weaknesses:** 
  + 基于SOA的系统通常很复杂。
  + 存在与中间件相关联的性能开销，并且服务可能是性能瓶颈，并且通常不提供性能保证。

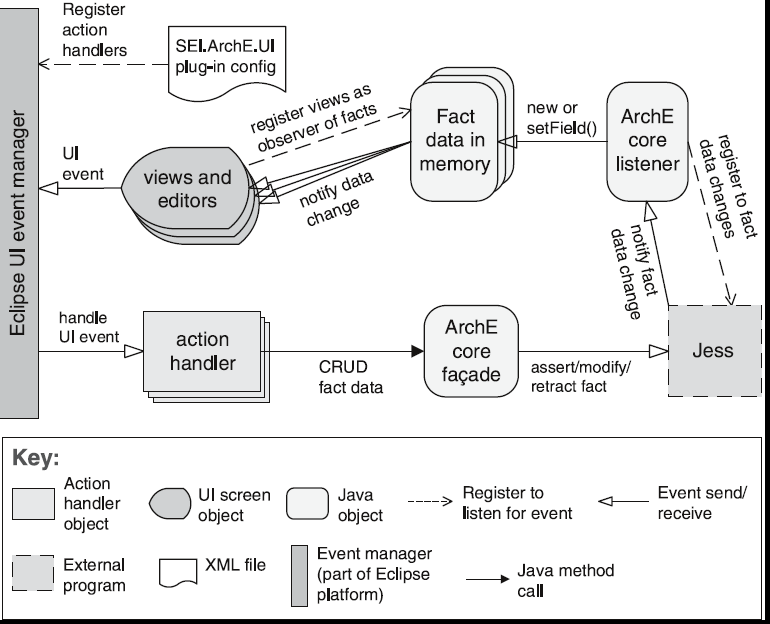


## Publish-Subscribe Pattern （掌握）

* **Context:** 数据生产者和消费者的确切数量和性质不是预先确定或固定的，他们共享的数据也不是。
* **Problem:** 我们如何创建支持在生产者和消费者之间传递消息的能力的集成机制，以便他们不知道彼此的身份，甚至可能不知道他们的存在？
* **Solution:** 在发布 - 订阅模式中，组件通过已宣布的消息或事件进行交互（interact）。

**Publish-Subscribe Solution**

* **Overview:** 组件发布和订阅事件。 当组件宣布事件时，连接器基础结构（connector infrastructure）会将事件分派（dispatches）给所有已注册的订户。
* **Elements:** 
  + 发布者，订阅者。Publisher, subscriber.
  + 发布 - 订阅连接器。The publish-subscribe connector.
* **Relations:** 附件关系通过规定哪些组件宣布事件以及哪些组件被注册以接收事件来将组件与发布 - 订阅连接器相关联。
* **Constraints:** 发布端口附加到通告角色，订阅端口附加到侦听角色。
* **Weaknesses:** 
  + 通常会增加延迟并对消息传递时间的可伸缩性（scalability）和可预测性（predictability）产生负面影响。
  + 不能保证对消息排序和消息传递的控制较少。

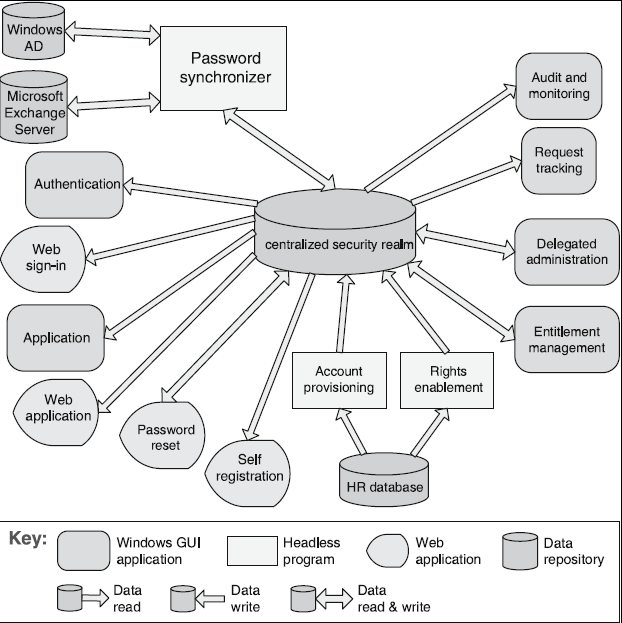


## Shared-Data Pattern （掌握）

* **Context:** 各种计算组件需要共享和操作大量数据。
* **Problem:** 系统如何存储和操作由多个独立组件访问的持久数据？
* **Solution:** 在共享数据模式中，交互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。

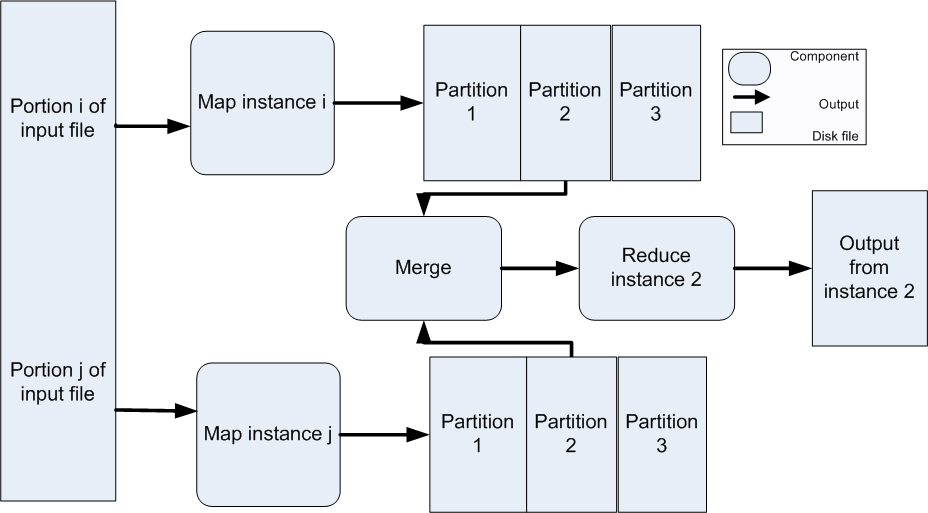
**Shared Data Solution**

* **Overview:** 数据访问器之间的通信由共享数据存储调解。
* **Elements:**
  + Shared-data store.（共享数据存储）
  + Data accessor component.（数据访问器组件。）
  + Data reading and writing connector. （数据读写连接器）
* **Relations:** 附件关系确定哪些数据访问器连接到哪些数据存储。
* **Constraints:** 数据访问器仅与数据存储交互。
* **Weaknesses:** 
  + 共享数据存储可能是性能瓶颈。（performance bottleneck）
  + 共享数据存储可能是单点故障。（a single point of failure）
  + 数据的生产者和消费者可以紧密耦合。（tightly coupled）



## Map-Reduce Pattern （掌握）

* **Context:** 需要快速分析大量数据。
* **Problem:** 有效地执行大型数据集的分布式和并行排序，并为程序员指定要完成的分析提供简单的方法。
* **Solution:** The map-reduce pattern requires three parts:
  + 专用基础架构负责根据需要分配数据。
  + 用于过滤数据以检索项目的映射。
  + 结合了地图结果的reduce。
* **Overview:** map-reduce模式提供了一个分析大型分布式数据集的框架。 地图执行分析的提取和变换部分，reduce执行结果的加载。
* **Elements:** 
  + Map.
  + Reduce.
  + 基础结构是负责部署映射和减少实例，在它们之间引导数据以及检测故障并从中恢复的框架。
* **Relations:** 
  + 部署。Deploy on.
  + 实例化，监视和控制是基础结构与map和reduce实例之间的关系。
* **Constraints:** 
  + 要分析的数据必须作为一组文件存在。
  + 映射函数是无状态的，不会相互通信。
* **Weaknesses:** 
  + 高架。
  + 如果无法将数据集划分为类似大小的子集，则并行性的优势就会丧失。
  + 需要多次减少的操作很难协调。



## Multi-Tier Pattern （掌握）

* **Context:**将系统的基础结构分配到不同的子集中。
* **Problem:**我们如何将系统分成若干计算独立的执行结构 - 由一些通信媒体连接的软件和硬件组？
* **Solution**:许多系统的执行结构被组织为一组逻辑分组。 每个分组称为一个层。

**Multi-Tier Solution**

* **Overview:** 系统的执行结构被组织为一组逻辑分组。
* **Elements:** 
  + Tier.(层)
* **Relations:** 
  + Is part of.
  + Communicates with.
  + Allocated to.
* **Constraints:** 软件组件恰好属于一个层。
* **Weaknesses:** 大量的前期成本和复杂性。



# Relationships Between Tactics and Patterns （理解）

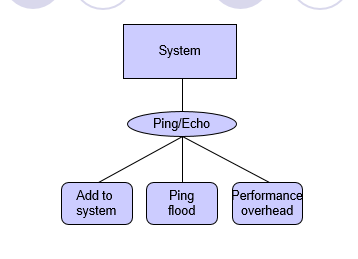
* 模式是根据战术建立的; 如果一个模式是一个分子，一个策略就是一个原子。
* MVC, for example utilizes the tactics:
  + Increase semantic coherence（增加语义连贯性）
  + Encapsulation（封装）
  + Use an intermediary（中介）
  + Use run time binding

**Tactics Augment Patterns（策略增强模氏）**

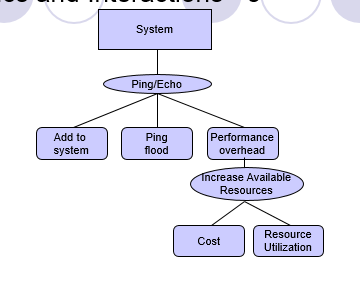
* 模式解决了一个特定的问题，但是在其他品质方面是中立的或有弱点。
* 考虑代理模式
  + 可能存在性能瓶颈
  + 可能只有一个故障点
* Using tactics such as
  + 增加资源有助于提高绩效
  + 维护多份副本将有助于提供可用性

**Tactics and Interactions**

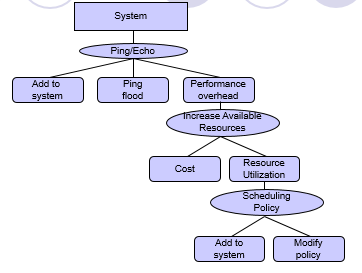
* 每种策略都有优点（其存在的原因）和缺点 - 副作用。
* 使用战术可以帮助减轻这些缺陷。
* Ping / Echo是一种常见的故障检测策略。
* Ping / Echo的常见副作用是：
  + 安全性：如何防止ping泛洪攻击？
  + 性能：如何确保ping / echo的性能开销很小？
  + 可修改性：如何将ping / echo添加到现有架构中？



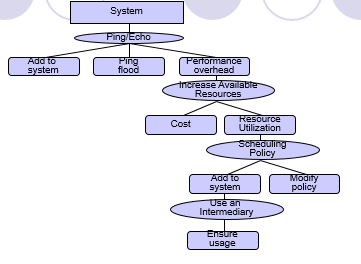
* 解决性能副作用的策略是“增加可用资源”。
* 增加可用资源的常见副作用是：
  + 成本：增加资源成本
  + 绩效：如何有效利用增加资源？



* 解决有效利用资源副作用的策略是“调度策略”。
* 调度策略的常见副作用是：
  + 可修改性：如何将调度策略添加到现有体系结构中
  + 可修改性：如何在未来改变调度策略？



* 解决向系统添加调度程序的策略是“使用中介”。
* 使用中介的常见副作用是：
  + 可修改性：如何确保所有通信都通过中介？



* 解决所有通信通过中间人的问题的策略是“限制通信路径”。
* 限制通信路径的常见副作用是：
  + 性能：如何确保中介的性能开销不会过大？

**注意：这个设计问题现在变得递归了！**

**How Does This Process End?**

* 每次使用战术都会引入新的担忧。
* 每一个新的关注都会引发新的策略。
* 我们是在无限进行吗？
* 不会。最终，每种战术的副作用都变得小到可以忽略不计。

# Summary

* 架构模式
  + 是一套在实践中反复发现的设计决策，
  + 具有允许重复使用的已知属性，并且
  + 描述了一类架构。
* 战术比模式更简单
* 关于真实系统的模式是不明确的，所以它们必须用策略来增强。
  + 当满足特定系统的要求时，扩充结束。