University of Science and Technology of China

Software Architecture

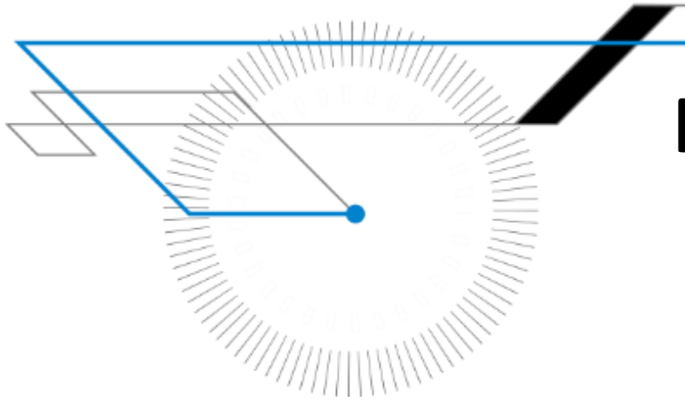
SSE USTC Qing Ding
dingqing@ustc.edu.cn

Pattern and Style II

Outline

- Network-Centred Style
- Remote Invocation Architectures
- Interpreter
- Interceptor
- GUI Architectures (Interactive)
- Adaptable Architectures
- Transaction-Processing
- Others
- Heterogeneous Architectures

Network-Centred Style



Focus on communication.

Client-server and other Distributed Architectural Patterns

- At least **one** component has the role of ***server:***
  至少有一个组件具有服务器角色:等待并处理连接
  —waiting for and then handling connections.
- At least **one** component that has the role of ***client***, initiating connections to obtain some service.
  至少有一个具有客户端角色的组件，启动连接以获取某些服务

- **Three-tier model** for web-based client-sever applications:
  基于web的客户端-服务器应用程序的三层模型:
  —Server ‘in the middle’
  服务器在中间
  - server to client (web-based or not; likely communicating via the Internet)
    服务器到客户端(是否基于web;可能通过互联网通信)
  - client to a database server (usually / often via an **intranet**)
    客户端到数据库服务器(通常/经常通过内部网)

Client-Server Style

- 组件是客户端和服务器
- Components are clients and servers
- 服务器不知道客户端的数量或身份
- Servers do not know number or identities of clients
- 客户端知道服务器的身份
- Clients know server’s identity
- Connectors are RPC-based network interaction protocols
  连接器是基于RPC的网络交互协议

Client-Server

基本概念

- Basic concept:

客户端使用服务

- The **client** uses a service

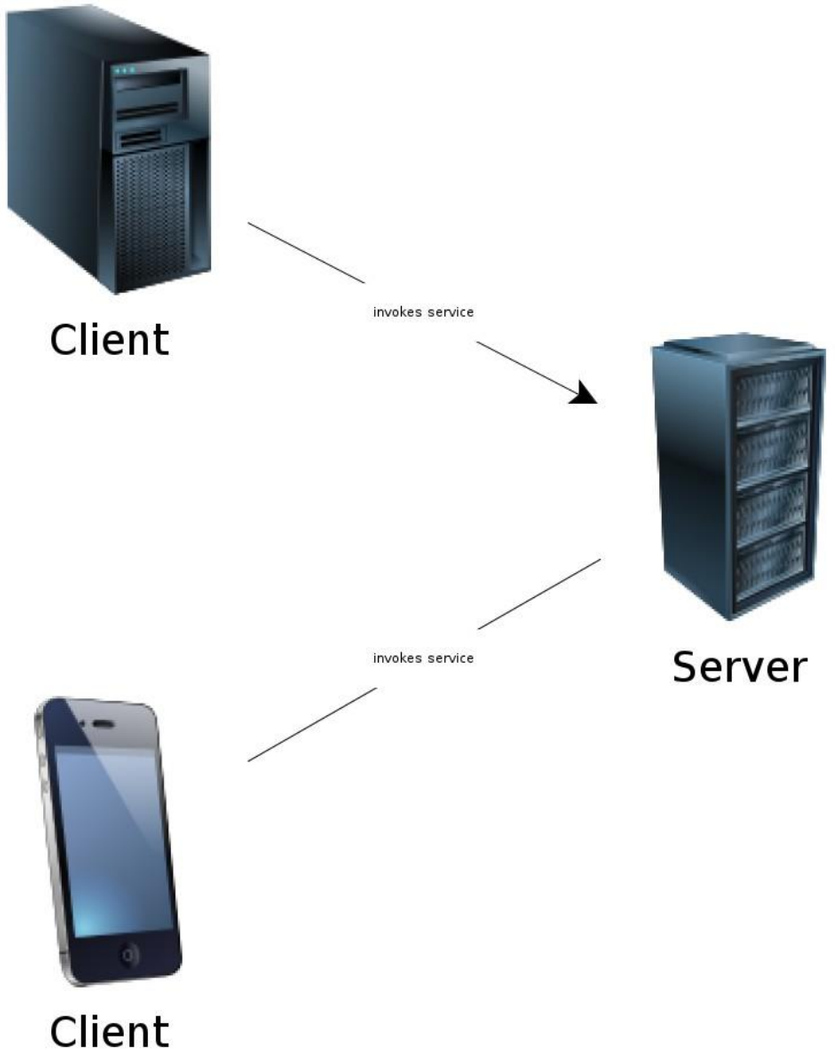
服务器提供服务，服务可以是任何资源

- The **server** provides a service The service can be any resource

例如，数据，文件，CPU，显示设备，通常通过网络连接，客户端是相互独立的

- E.g. data, file, CPU, display device Typically connected via a network Clients are **independent** from each other

Client-Server



Client-Server

- The server provides an abstract service
  服务器提供了一个抽象服务
- The implementation of the server decides how to fulfil the request
  服务器的实现决定如何满足请求，编程语言、操作系统的抽象
  Abstraction of programming language, operating system
- ⇒ **loose coupling** between client and server The location of the server is transparent
  客户机和服务器之间的松耦合，服务器的位置是透明的
- Sometimes the client also might become the server (and vice versa)
  有时客户端也可能成为服务器(反之亦然)
- ⇒ increases the coupling
  增加耦合

Client-Server

关注点分离
- **Separation of concerns (SoC)**

功能被清楚地划分为独立的组件
- Functionality is clearly split into separate components
- Also a motivation for the layered architecture style, where each layer is responsible for its own abstraction

这也是分层架构风格的一个动机，其中每一层负责自己的抽象
- Aspect oriented programming tries to separate cross-cutting concerns into separate modules

面向方面编程试图将横切关注点分离到单独的模块中

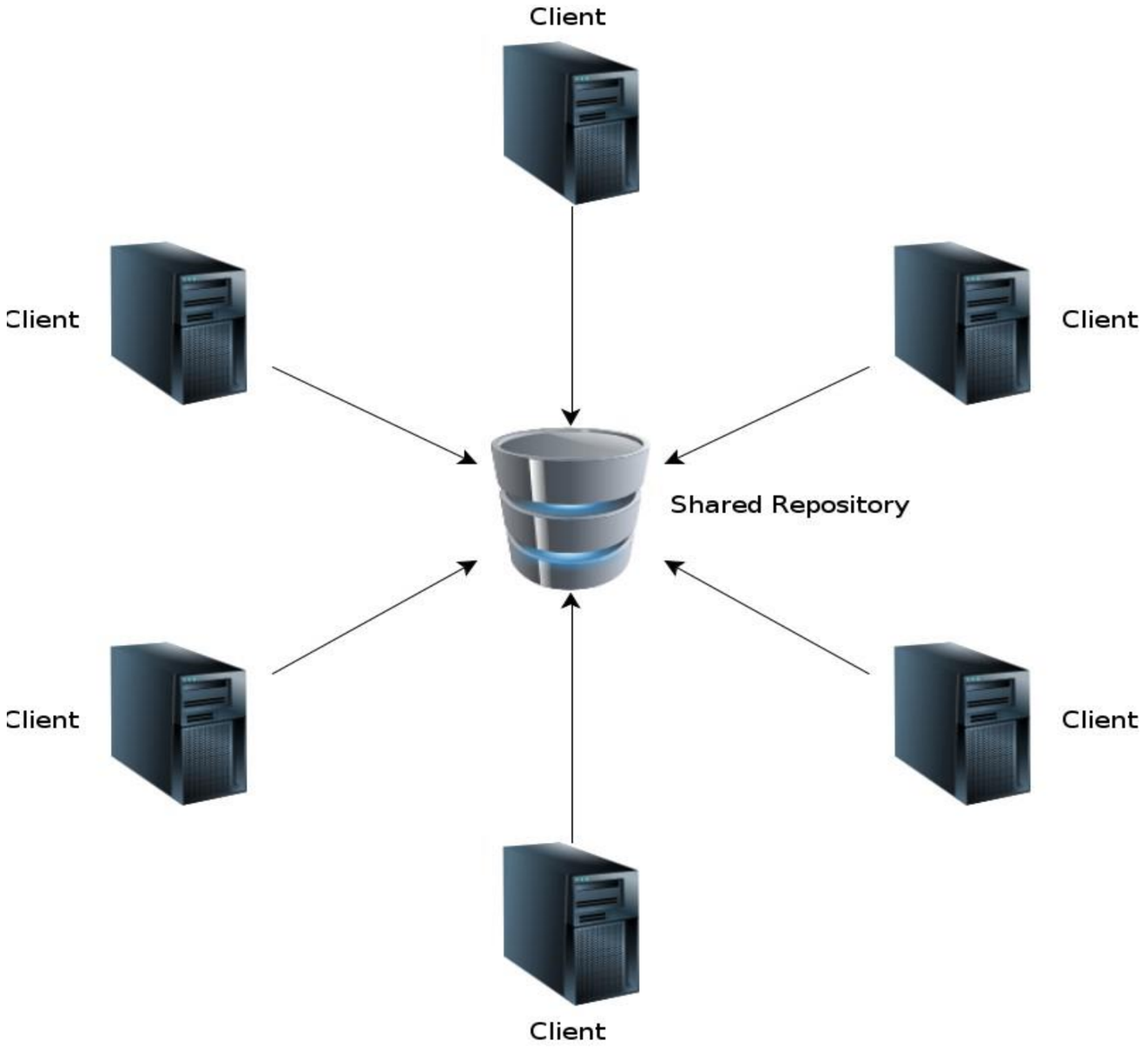
支持独立的可发展性
- Supports independent evolvability
- Achieved, if the communication between client and server is well designed

如果客户端和服务器之间的通信设计得很好，就可以实现

Client-Server

- Client-server pattern is used by other architectural styles
  其他体系结构样式使用客户机-服务器模式
- It can be used to realise a **shared repository**
  它可以用来实现共享存储库
- E.g. for the data-centric repository pattern
  例如，以数据为中心的存储库模式
- E.g. for filters which operate on a single shared data structure
  例如，用于在单一共享数据结构上操作的过滤器

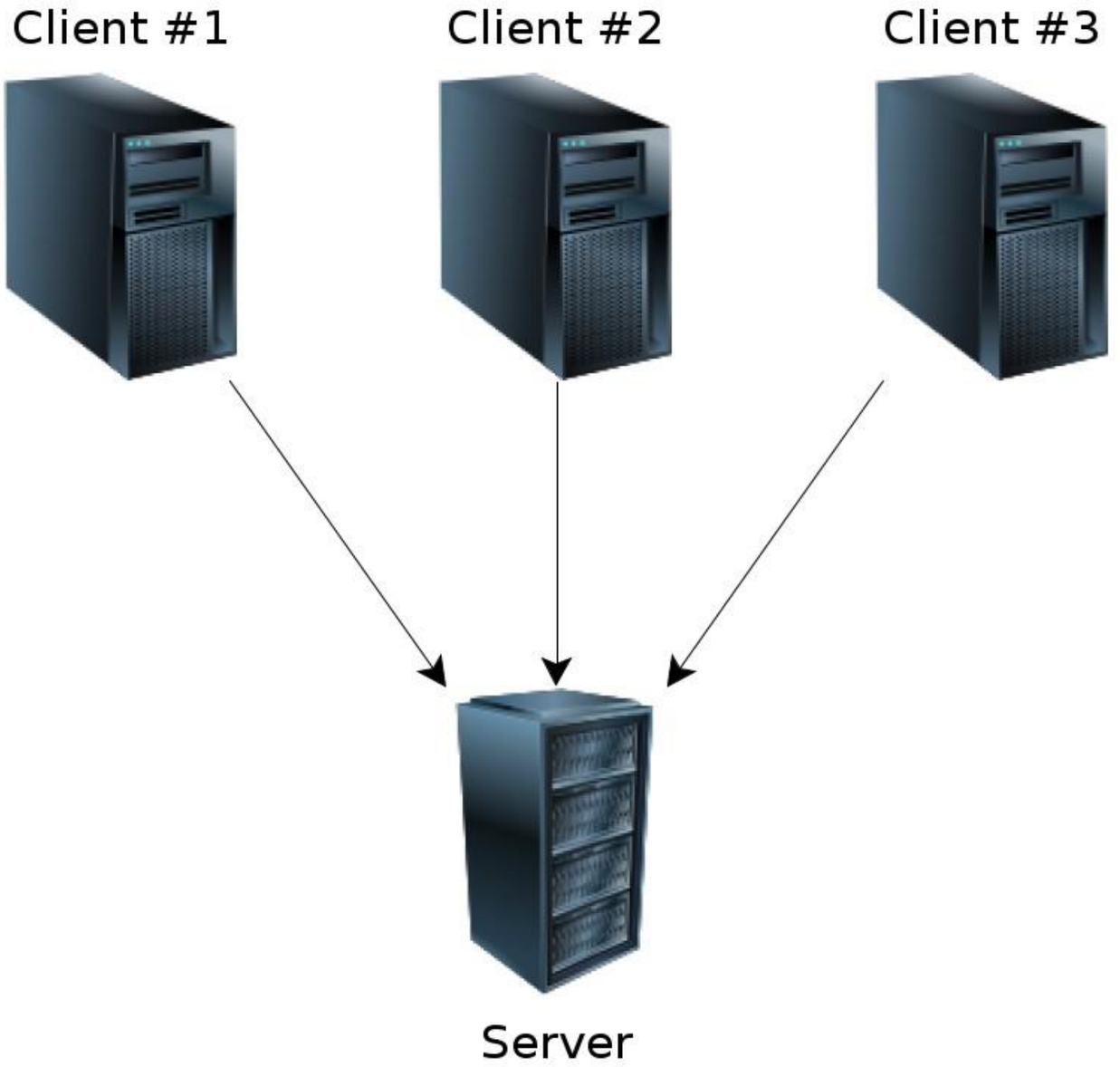
Client-Server - Shared Repository



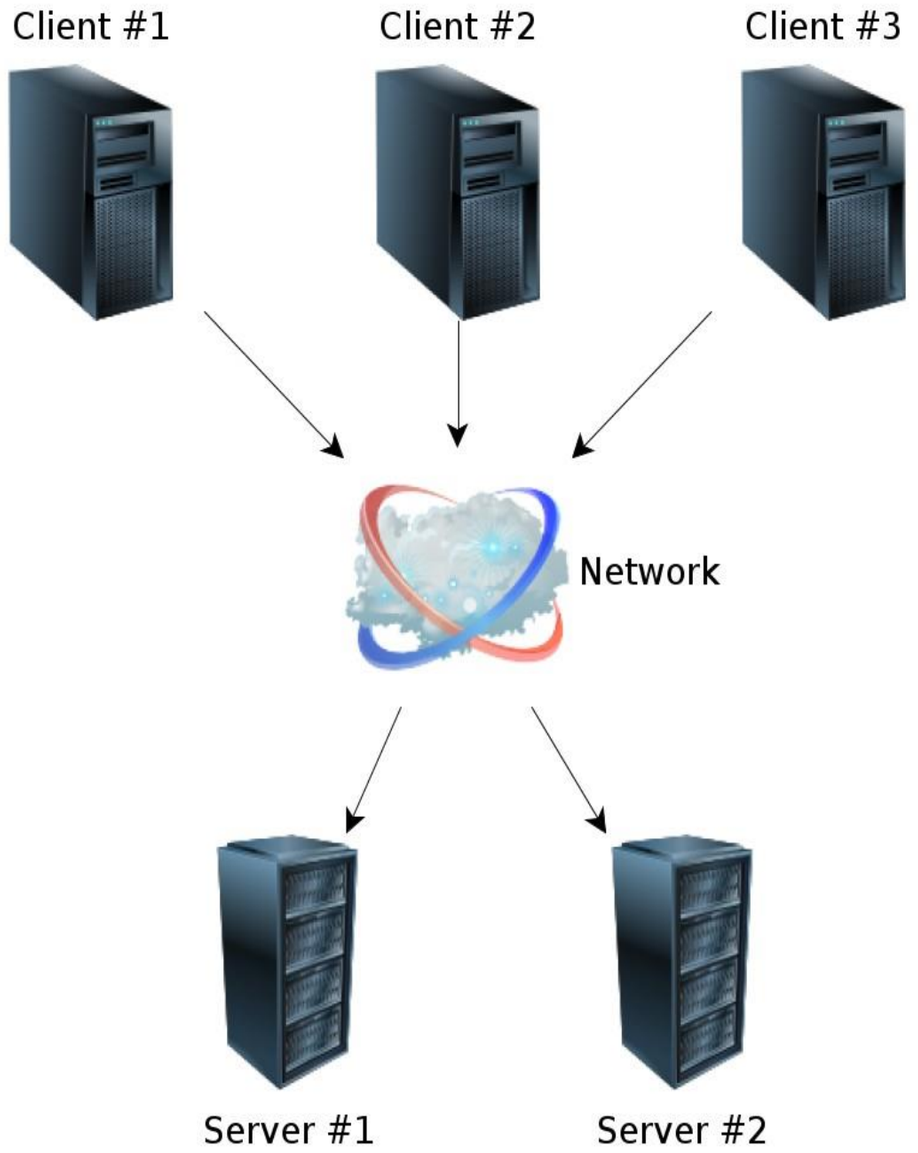
Client-Server

- Two basic types of topology of the server Single, 两种基本类型的服务器拓扑，单一的，集中的服务器，或者
  centralised server or
- 多个分布式服务器
  Multiple, **distributed** servers
- Centralised servers are easier to administer (install, 集中式服务器更容易管理(安装、部署、更新、维护、监控...)
  deploy updates, maintain, monitor, ...)
- Distributed servers scale better, but could introduce 分布式服务器伸缩性更好，但可能会引入复杂性(例如，需要两阶段提交)
  complexity (e.g. require two-phase commits)

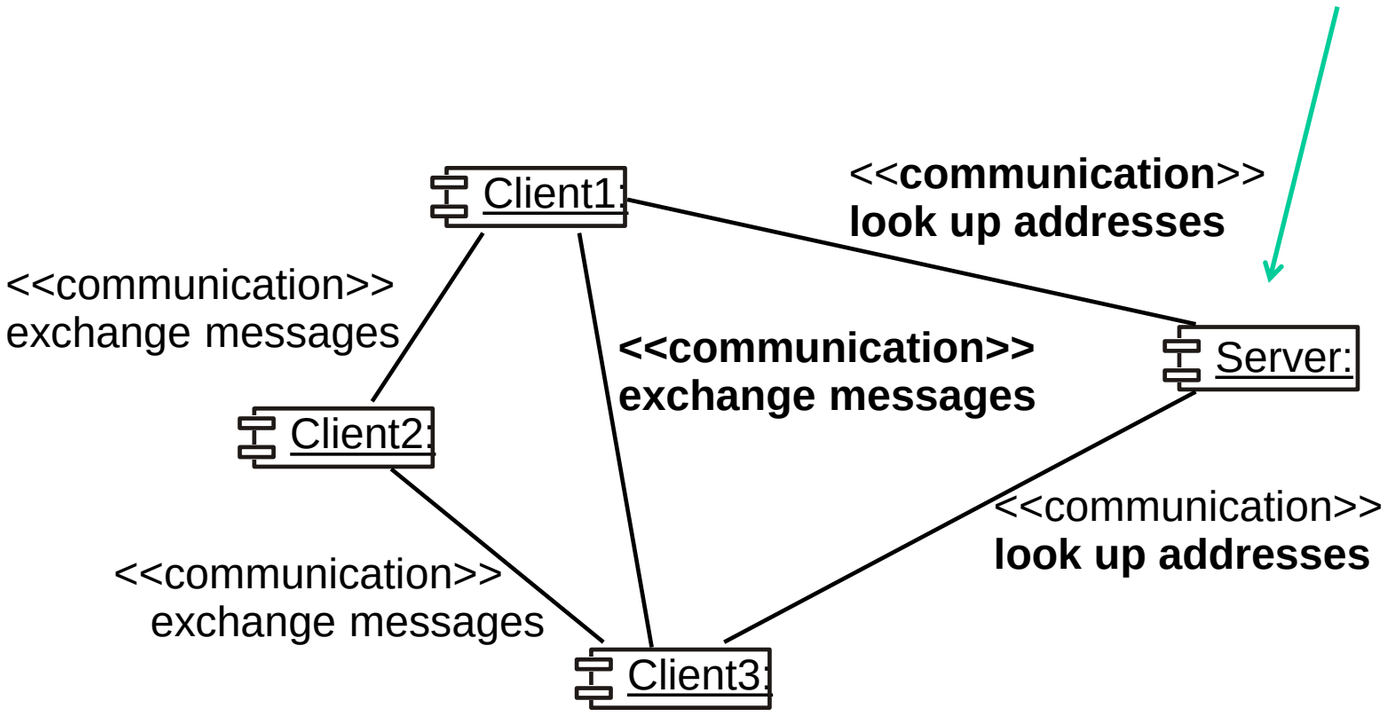
Client-Server -Centralised



Client-Server -Distributed



An Example Of A Distributed System



Client-Server -Scalability

扩展的两种基本类型

- **Two basic types of scaling:**
- **Vertically**, by increasing the computing power of a single machine (scale up)

垂直地，通过增加单机的计算能力，向上扩展（比较难）

- **Horizontally**, by adding more machines (scale out)

水平地，通过添加更多的机器(水平扩展)

- Note: scaling strategies are not specific to client-server architectures

注意:扩展策略不是特定于客户机-服务器架构的

Client-Server -Scalability

- Scaling vertically is often the only option
  垂直扩展通常是唯一的选择
- Especially if the system has not been designed from the ground up to allow distributed processing
  特别是当系统还没有被从头设计成允许分布式处理时
- Scaling vertically is made easier in a virtual environment (virtual machines instead of physical)
  在虚拟环境(虚拟机而不是物理环境)中，垂直扩展变得更加容易。
- Because the system could be transferred to a more powerful machine, without the system knowing
  因为这个系统可以在系统不知情的情况下被转移到一个更强大的机器上

Client-Server -Scalability

- Scaling horizontally should provide a high upper limit for scalability
  水平扩展应该为可伸缩性提供一个较高的上限
- Needs support from the system, not every system allows being distributed
  需要系统的支持，不是每个系统都允许分布式
- Might lead to a high communication overhead due to synchronisation between the nodes
  可能会由于节点之间的同步而导致较高的通信开销

Client-Server -Scalability

- Distributed servers need to be specifically designed to allow scaling horizontally
  分布式服务器需要专门设计以允许水平伸缩
- Will be typically more effort to develop Upgrading to a bigger machine might be cheaper
  通常会花费更多的精力去开发，升级到更大的机器可能更便宜
- Which strategy (scale up or scale out) is more suitable, depends on the actual system.
  哪一种策略(向上扩展还是水平扩展)更适合，这取决于实际的系统

Client-Server

- **Stateful vs. stateless**

有状态和无状态

  - If the client-server communication is stateful, the server keeps track of the application state

如果客户机-服务器通信是有状态的，服务器将跟踪应用程序的状态

  - Typically provides a handle or a session id

通常提供句柄或会话id

  - The client then may manipulate the state on the server, e.g.
    - Open file (returns file handle) Append line
    - Close file

然后客户端可以操作服务器上的状态，例如
·打开文件(返回文件句柄)追加行
·关闭文件

  - Easier for the clients, as they not need to manage the state ⇒ needed if coupled with thin clients

对客户端来说更容易，因为他们不需要管理状态->如果与瘦客户机耦合，则需要

Client-Server

- If the client-server communication is stateless, the client is responsible to keep track of the application state
  如果客户机-服务器通信是无状态的，则客户机负责跟踪应用程序的状态
- The server does not need to store or manage session specific data ⇒
  服务器不需要存储或管理特定于会话的数据
- typically coupled with rich clients
  通常与富客户端耦合
- Therefore the scalability of stateless servers are generally better than stateful
  因此，无状态服务器的可伸缩性通常比有状态服务器好

无状态：pool容器管理多个instance（整个生命周期），客户端有请求就分配一个instance提供服务，服务提供完就释放instance，如果instance分配完客户端需要排队等待
有状态：客户端与instance绑定（有100个客户就需要100个instance），服务提供完并不会释放instance，instance生命周期由客户管理

Client-Server

- **Cloud Computing**
  - The server is no longer in the organisations network, but somewhere in the Internet
  - Example: cloud services by Salesforce, Google, Microsoft
  - Scalability, security, reliability is expected to be handled by a specialised team
  - Loss of control, legal issues (data is exported to another country) Needs a working Internet connection

Client-Server -Advantages

概念上很简单

- Conceptually simple
- Clear separation of responsibilities, eases evolvability, helps testability

清晰的职责分离，简化了可进化性，有助于测试性

- Good scalability (especially, if stateless)

良好的可伸缩性(特别是在无状态时)
-优秀的可伸缩性(如果服务器可以扩展)

  - Excellent scalability (if server can be scaled out)

- Good for security, as data can be held at the server with restricted access

有利于安全性，因为数据可以在访问受限的情况下保存在服务器上

Client-Server - Disadvantages

- Risk of bad usability/performance, if the communication between client and server is slow, or has a high latency
  如果客户端和服务器之间的通信很慢，或者有很高的延迟，那么存在可用性/性能差的风险
- Need to develop/agree on a protocol between client and server
  需要在客户端和服务器端之间开发/达成协议
- For stateful, centralised servers scalability is limited
  对于有状态的集中服务器，可伸缩性是有限的
- Integrability into existing systems might not be possible (e.g. if the communication is not possible, or not allowed)
  可集成到现有系统中可能是不可能的(例如，如果不可能或不允许通信)

Other Distributed Client-server Architectural Patterns

对等模式

- **Peer-to-Peer pattern**. A system where:
  分布在多个主机上的各种软件组件
  —various software components **distributed** over several hosts.
  主机:客户端和服务器(彼此)
  —Hosts: both clients and servers (to each other)
  —**Any two components** can set up a communications channel through which communications is accomplished.
  任何两个组件都可以设置一个通信通道，通过该通道实现通信

变体

- **Variation:**
  有时候同伴需要能够找到彼此
  —Sometimes peers need to be able to find each other;
  可能需要一个包含位置信息的服务器
  — May need a server containing **location information**

Peer-to-Peer Style

- State and behavior are distributed among peers which can act as either clients or servers.
  状态和行为分布在可以充当客户机或服务器的对等点之间
- Peers: independent components, having their own state and control thread.
  对等方:独立的组件，有自己的状态和控制线程
- Connectors: Network protocols, often custom.
  连接器:网络协议，通常是定制的
- Data Elements: Network messages
  数据元素:网络消息

Peer to peer

客户机和服务器之间的分离被删除

- Separation between client and server is removed

每个客户机同时是一个服务器，称为对等点

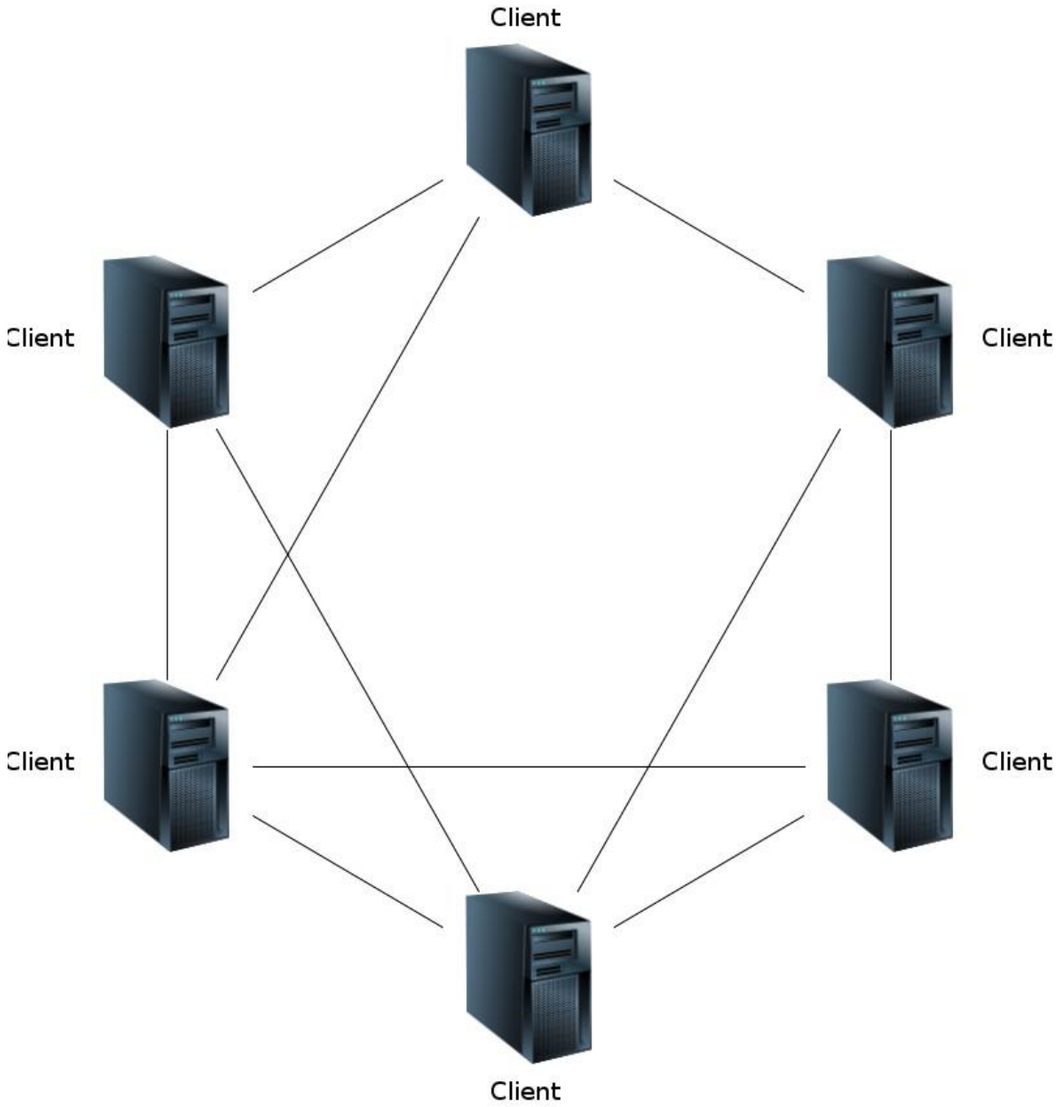
- Each client is a server at the same time, called **peer**
- The goal is to distribute the **processing or data** among many peers

目标是将处理或数据分布在许多对等点上

没有中央管理或协调

- No central administration or coordination

Peer to peer



Peer to peer

- Each peer provides services and consumes services
  每个对等点提供服务并使用服务
- Communication might occurs between all peers
  通信可能发生在所有对等点之间
- Number of peers is dynamic
  对等点的数量是动态的
- Each peer has to know how to access other peers (discover, search, join)
  每个对等点必须知道如何访问其他对等点(发现、搜索、加入)

Peer to peer

- Once a peer is initialised, it needs to be come part of the network
  一旦对等点被初始化，它就需要成为网络的一部分
- A **bootstrapping** mechanism is needed:
  - For example via a broadcast message
  - For example a public list of network addresses

需要一个自举机制：
· 例如通过广播消息
· 例如网络地址的公开列表

Peer to peer - centralised P2P

中心化点对点

Centralised peer to peer

- 有些方面是集中的
  Some aspects are centralised
- 例如，中心组件跟踪可用的对等点
  For example, a central component keeps track of the available peers

Peer to peer - hybrid P2P

混合点对点

- **Hybrid peer to peer**
- Not all peers are equal, some have additional responsibilities

不是所有的对等点都是平等的，有些有额外的责任

- They are called supernodes

它们被称为超级节点

- Example: Skype uses a peer-to-peer protocol, but also uses supernodes and a central login servers

例如:Skype使用点对点协议，但也使用超级节点和中央登录服务器

Peer to peer - Advantages

- Typically, excellent scalability, as the computation can be distributed
  通常具有优秀的可伸缩性，因为计算可以分布式进行
- Good for reliability, as data can be replicated over multiple peers
  很好的可靠性，因为数据可以在多个对等点上复制
- No single point of failure
  无单点故障

Peer to peer - Disadvantages

- Quality of service is not deterministic, cannot be guaranteed
  服务质量是不确定的，不能保证
- E.g. high latency
  高延迟
- Very complex, hard to maintain and to test
  非常复杂，难以维护和测试

How Does The Client-server Architectural Pattern Subscribe To Principles Of Good Architectural Design?

客户机-服务器体系结构模式如何遵循良好的体系结构设计原则

- 1. *Divide and conquer*: Dividing the system into client and server processes is a **strong** way to divide the system.

分治法:将系统划分为客户端和服务器进程是划分系统的一种有效方法。
➔每个可以被分别开发

— ➔ Each can be separately developed.

增加内聚性:每个服务器都可以向客户端提供内聚性服务

- 2. *Increase cohesion*: Server can **each** provide **cohesive services** to clients.
-
- 3. *Reduce coupling*: There is usually only **one** communication channel exchanging simple messages.

减少耦合:通常只有一个通信通道交换简单的消息

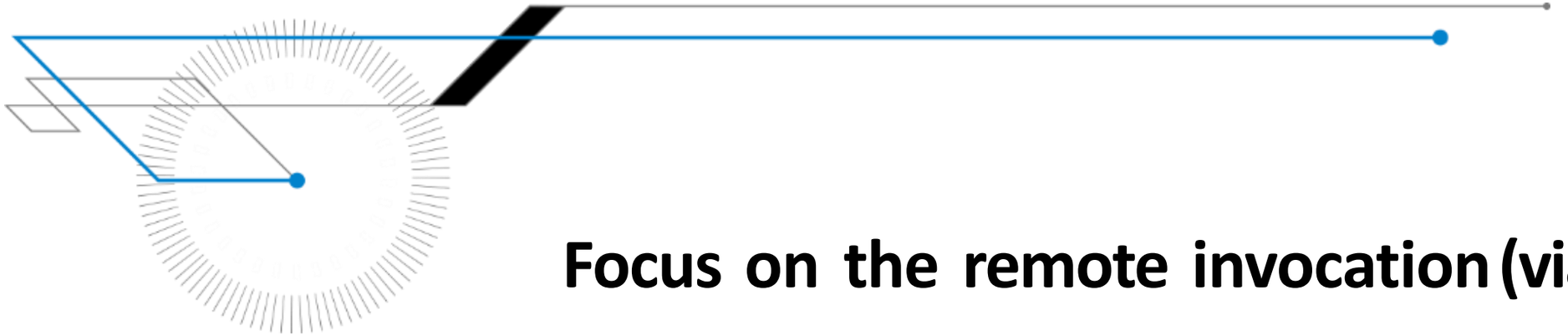
增加抽象:独立的分布式组件通常是很好的抽象。这句话对你来说意味着什么?抽象?

- 4. *Increase abstraction*: Separate distributed components are often good abstractions. **What does this sentence mean to you? Abstractions??**

增加重用性:找到合适的框架来构建好的分布式系统通常是可能的。示例架构已经存在。然而，客户机-服务器系统通常是与应用程序相关的

- 6. *Increase reuse*: It is often possible to find suitable **frameworks** on which to build good distributed systems. Sample architectures already exist. However, client-server systems are often **very application specific**.

Remote Invocation Architectures



Focus on the remote invocation (via network).

Remote invocation and service architectures

- Remote invocation architectures involve distributed processing components
  远程调用体系结构涉及分布式处理组件
- Typically, a client component invokes a method (function) on a remote component
  通常，客户端组件调用远程组件上的方法(函数)

socket：service端listen某个端口port，client使用open方法建立连接，client使用send方法向service请求数据，使用receive方法接受数据

RPC家族：CORBA分布式中间件
DCOM-- .net
RMI-- JavaEE
Web Service

Remote invocation and service architectures

- Advantages: increased performance through distributed computation
  优点:通过分布式计算提高性能
- Only if network is reliable and fast and the communication overhead is manageable
  只有当网络是可靠的和快速的，并且通信开销是可管理的
- Disadvantages: often, tightly coupling of components
  缺点:通常是组件的紧密耦合
- Managing of addressability (recollect unique identity of objects) increases communication overhead
  对可寻址性(回收对象的唯一标识)的管理增加了通信开销

Remote invocation and service architectures

- Service architectures introduce a special component where services are registered
  服务体系结构引入了注册服务的特殊组件
- Any component interested in a service asks that component for the address of the service
  任何对服务感兴趣的组件都会向该组件询问服务的地址
- It tries to solve the addressability problem
  它试图解决可寻址性问题
- If communication protocols are standardized then services can be integrated on-the-fly
  如果通信协议是标准化的，那么服务就可以即时集成

Remote invocation and service architectures

使用Web协议进行通信

- Use Web protocols for communication
- However, the addressability is still not managed because you have to know how to address objects in a remote service

但是，由于您必须知道如何在远程服务中寻址对象，因此仍然没有对寻址能力进行管理

- Web services are in essence only remote procedure calls using Web protocols

Web服务本质上只是使用Web协议的远程过程调用

- Same advantages and disadvantages apply

优点和缺点是一样的

文本文件关注交互性

Broker

代理：希望某种透明性时，采用代理技术（透明性：用户不知道下层的实现细节）

- **Separate communication** from the application functionality Support for distributed systems

将通信与应用程序功能分离，对分布式系统的支持

- The broker hides the communication from the components of the system

代理对系统的组件隐藏通信

- The broker coordinates the communication

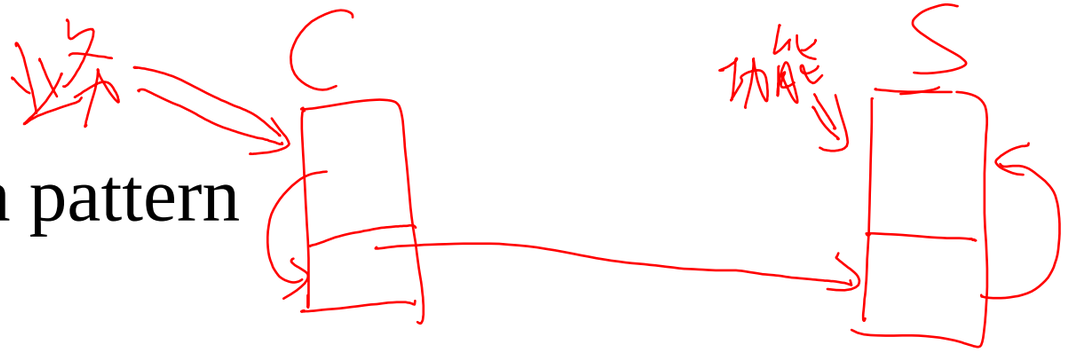
代理协调通信

- A broker can be used to transparently change a non-distributed system into a distributed one

代理可用于透明地将非分布式系统更改为分布式系统

- For example: The client interacts with a remote object via a broker

例如:客户端通过代理与远程对象交互



- *Note*: Similar to the proxy design pattern

注:类似于代理设计模式

The Broker Architectural Pattern

- Here, we **transparently distribute** **aspects** of the software system to different nodes

  在这里，我们透明地将软件系统的各个方面分发给不同的节点

  对象调用方法的其他对象，而不知道对象位于远程

  —Objects call method's other objects **w/o knowing** object is remotely located.

  客户端不“关心”远程对象在哪里

  —Client does not '**care**' where the remote object is.

  —**CORBA:** well-known open standard allowing you to build this kind of architecture.

  CORBA:众所周知的开放标准，允许您构建这种体系结构

  公共对象请求代理体系结构

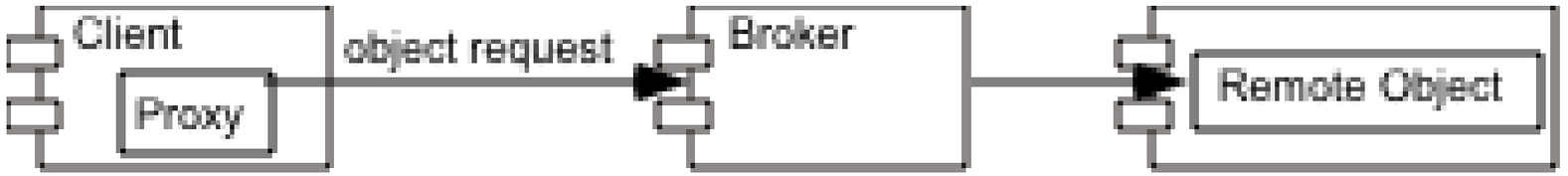
  - (Common Object Request Broker Architecture)
  - (Microsoft has its own architecture: COM, DCOM (old))

  可以使用“代理设计模式”，让代理对象调用代理，代理决定所需对象的位置

  - **'Proxy design pattern'** can be used such that a **proxy object** calls the broker, which determines where the desired object is located.

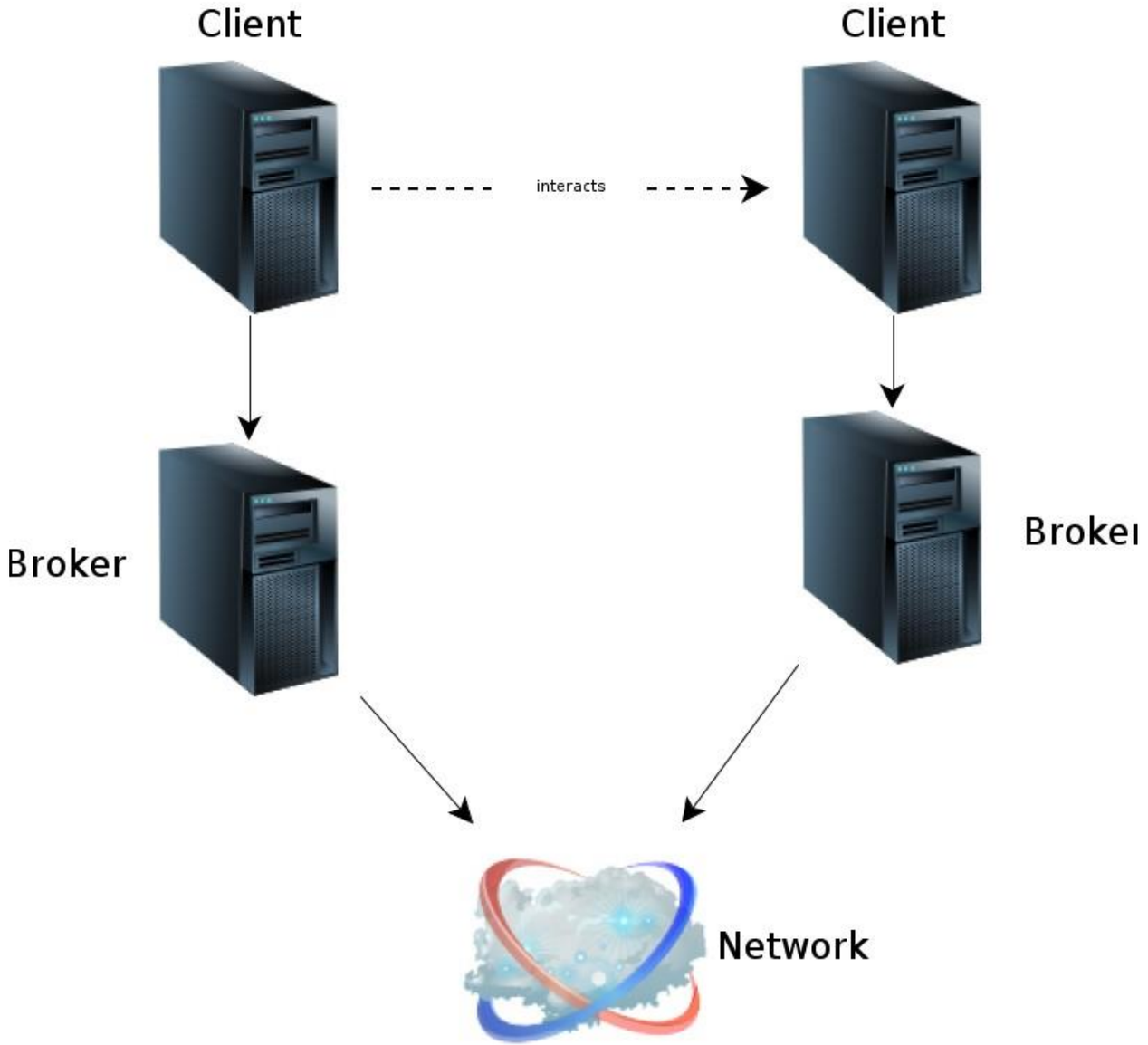
Example of a Broker system

Very popular design pattern. Simple and very effective.
非常流行的设计模式。简单而有效



Note that all these architectural patterns are illustrated using ‘components.’
注意，所有这些架构模式都使用“组件”来说明

Broker



Broker - Tasks of a broker

代理的任务

- **The task of the broker**
- Find the appropriate server/service Forward request
  找到合适的服务器/服务，将请求转发给服务器
  to the servers, and
- Report the result (or error message/exception) back to
  将结果(或错误消息/异常)报告给客户端
  the client
- The communication is abstracted away from the
  通信被从客户端抽象出来
  client

Broker - Relationship to other patterns

- The broker pattern has relationships with other patterns
  代理模式与其他模式有关系
- E.g. the broker can be implemented as separate layer
  例如，代理可以作为单独的层实现
- E.g. the broker uses a client/server infrastructure
  例如，代理使用客户机/服务器基础设施

Broker -Advantages

- Decoupling of components (the networking aspect)
  组件的解耦(网络方面)有助于灵活性、可维护性和可进化性
  Helps flexability, maintainability and evolvability
- Allows a system to be distributed, even if it has not
  允许一个系统分布式，即使它一开始并没有为此设计
  been designed for this in the first place

Broker -Disadvantages

- Network communication might introduce new types of errors
  网络通信可能会引入新的错误类型
- Due to network latencies and limited bandwidth the behaviour might change
  由于网络延迟和有限的带宽，行为可能会改变

Broker Architecture And How This Architectural Design Pattern Subscribes To Design Principles

分而治之:远程对象可以独立设计

- 1. *Divide and conquer*: The remote objects can be independently designed.

增加可重用性:通常可以设计远程对象，以便其他系统也可以使用它们。我们在互联网上经常看到这种情况

- 5. *Increase reusability*: It is usually possible to **design the remote objects** so that other systems can use them too. We see this all the time on the Internet.

设计灵活性:可以根据需要更新代理，或者代理可以与不同的远程对象通信

- 7. *Design for flexibility*: The brokers can be updated as required, **or** the proxy can communicate with a different remote object.

设计可移植性:您可以为新平台编写客户机，同时仍然访问其他平台上的代理和远程对象

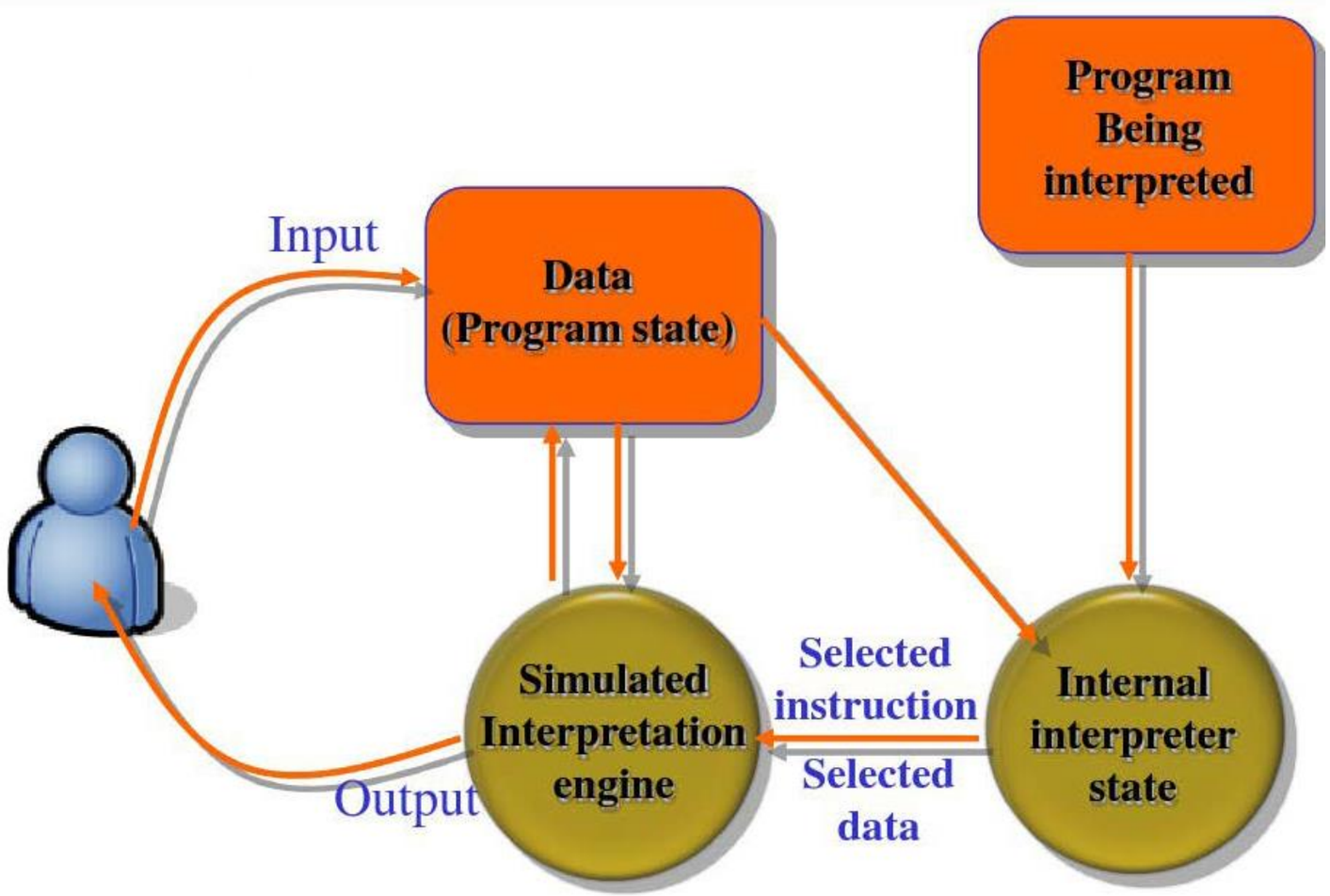
- 9. *Design for portability*: You can write clients for new platforms while still accessing brokers and remote objects on other platforms.

防御性设计:可以在远程对象中提供谨慎的断言检查

- 11. *Design defensively*: You can provide careful assertion checking in the remote objects.
- **Note that there are fewer design principles satisfied. Does NOT make this pattern inferior! (coherence; coupling; obsolescence, more…)** 注意，这里满足的设计原则较少。不要让这个模式逊色!(一致性;耦合;过时,更多...)

Interpreter Style

解释器：输入根据不同的用户解释成不同的东西
围绕一个核心Simulated interpretation engine，类似于rule-based的系统



Interpreters

- 体系结构是基于虚拟机在软件中产生的
  Architecture is based on a **virtual machine** produced in software
- Special kind of a **layered architecture** where a layer is implemented as a true language interpreter
  一种特殊的分层架构，其中一个层被实现为一个真正的语言解释器
- 组件
  Components
  - 正在执行的“程序”及其数据
    “Program” being executed and its data
  - 解释引擎及其状态
    Interpretation engine and its state
- 示例：Java虚拟机
  Example: Java Virtual Machine
  - Java代码被转换为平台无关的字节码
    Java code translated to platform independent bytecode
  - JVM是特定于平台的，它解释字节码
    JVM is platform specific and interprets the bytecode

Interpreter Style

适合小系统，单机运行

- Interpreter parses and executes input commands, updating the state maintained by the interpreter

解释器解析并执行输入命令，更新解释器维护的状态

  - Components: Command interpreter, program/interpreter state, user interface.

组件:命令解释器，程序/解释器状态，用户界面

  - Connectors: Typically very closely bound with direct procedure calls and shared state.

连接器:通常与直接过程调用和共享状态紧密绑定

  - Highly dynamic behavior possible, where the set of commands is dynamically modified.

高度动态行为，其中命令集是动态修改的

  - System architecture may remain constant while new capabilities are created based upon existing primitives.

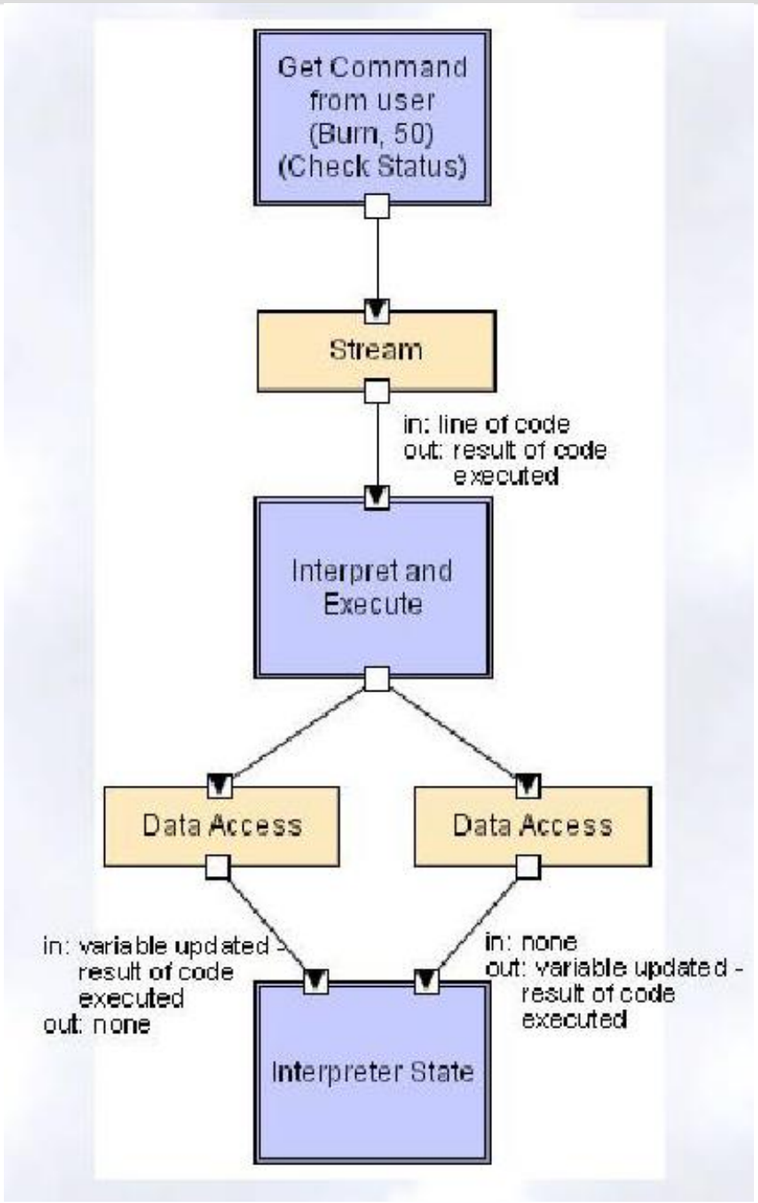
当基于现有原语创建新功能时，系统架构可能保持不变

  - Superb for end-user programmability; supports dynamically changing set of capabilities

极佳的终端用户可编程性;支持动态更改功能集

  - Lisp and Scheme

Interpreter LL



Interpreters

- An interpreter takes a string of characters, and converts it into actual code that is then executed

解释器接受一串字符，并将其转换为实际的代码，然后执行

- Always used to build virtual machine

一直用于构建虚拟机

  - Basic
  - Java virtual machine
  - 目前基于Web开发的各种脚本语言

Interceptor -Overview

拦截器：主要实现单一原则，有容器作为支撑可以实现拦截

- 动机:将功能分离为单独的组件
- Motivation: Separate functionality into a separate components
- Allow integration of additional services without changing the core components
  允许在不更改核心组件的情况下集成其他服务
- Interface provided by a component, which allows another component to register itself
  组件提供的接口，允许另一个组件注册自己
- 并在特定事件中调用
- And be invoked at specific events
- Open-closed design principle (open for extension, but closed for modifications)
  开闭设计原则(对扩展开放，对修改关闭)
- 用于处理横切关注点
- Used to address cross-cutting concerns

Interceptor -Example

例如:日志记录

- Example: Logging

系统由两个组件组成:客户端和服务器

- System consists of two components: a client and a server
- Each component provides a callback interface when some event occurs

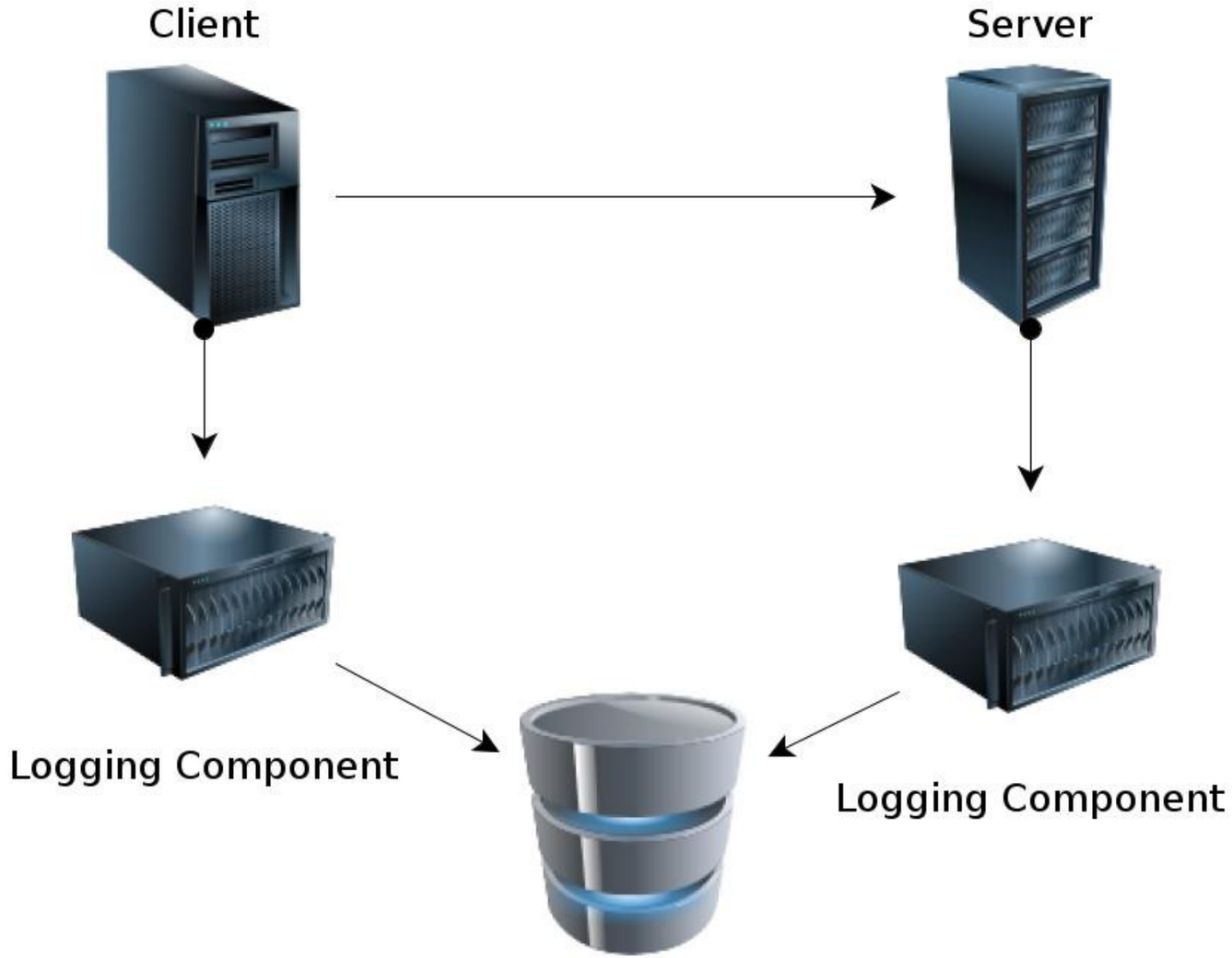
当某些事件发生时，每个组件提供一个回调接口

日志组件在每个组件上注册自己

- A logging components registers itself at every component
- The logging components store the log into a shared repository

日志组件将日志存储到共享存储库中

Interceptor -Example



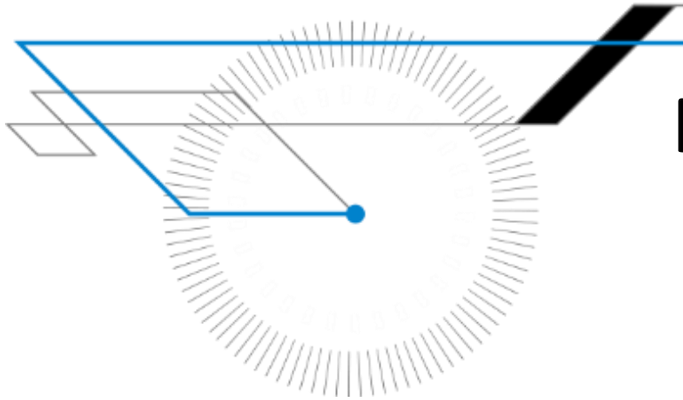
Interceptor -Advantages

- 拦截器组件可以重用(可重用性)
  Interceptor components can be reused (reusability)
- 优秀的灵活性
  Excellent flexability
- Clear separation of concerns, loose coupling ⇒ maintainability, evolvability
  明确的关注点分离，松耦合->可维护性，可进化性

Interceptor -disadvantages

- Can get quite complex quickly
  能很快变得相当复杂
- Potential cascading callbacks, endless loops
  潜在的级联回调，无尽的循环
- Sometimes event non-deterministic
  有时事件不确定
- Core components need to provide the callback interface
  核心组件需要提供回调接口
- May lead to bad testability
  可能导致不良的可测试性

GUI Architectures



Focus on user interaction.

模型视图控制器

Model View Controller (MVC)

- Motivation: reusability and separation of concerns
  动机:可重用性和关注点分离，三种角色:模型、视图和控制器
  Three roles: model, view & controller
- **Model**: encapsulates the behaviour and data of the
  模型:封装应用程序域的行为和数据
  application domain
  视图:呈现模型
- **View**: renders the model for presentation
- **Controller**: Reacts on user input, modifies the model
  控制器:对用户输入作出反应，修改模型并分派给视图
  and dispatches to the view
- Both, controller and view, depend on the model
  控制器和视图都依赖于模型，控制器和视图是UI的一部分
  Controller and view are part of the UI

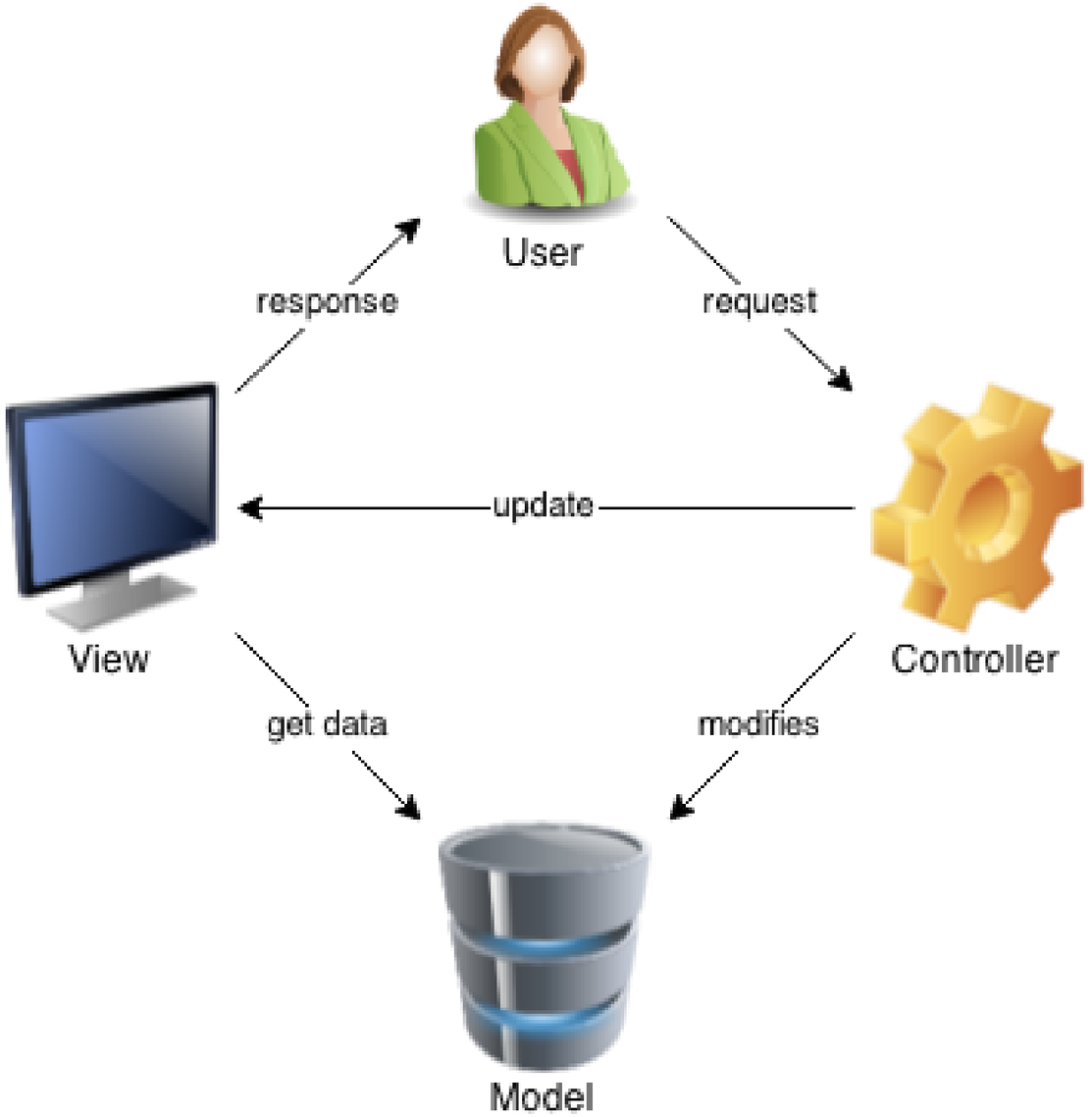
Model View Controller (MVC)

- MVC is often used for web applications (but not exclusively) Many existing frameworks:

MVC通常用于web应用(但不限于)许多现有的框架:

- AngularJS, JavaServer Faces (JSF), Struts, CakePHP, Django, Ruby on Rails, ...

Model View Controller (MVC)



MVC -Model

- **Model**
- 封装应用程序状态
- Encapsulates the application state
- 对状态查询的响应
- Response to state queries
- 暴露应用程序功能
- Exposes application functionality
- 通知变更视图(可选)
- Notify view of changes (optionally)
- Note: Notification only necessary, if the model and view realise an observer pattern
  注意:只有当模型和视图实现了观察者模式时，才需要通知

MVC -View

- **View**
- 展示模型
- Renders the model
- 从模型请求更新
- Requests updates from model
- 为控制器准备用户界面
- Prepares the user interface for the controller
- 通常多个视图
- Usually multiple views

MVC -Controller

- **Controller**
- 操纵模型
- Manipulates the model
- 触发应用程序行为
- Triggers application behaviour
- 选择响应视图(可选)
- Selects view for response (optionally)

MVC -Relationships

- The model-view-controller pattern does not replace a n-tier architecture
  模型-视图-控制器模式不会取代n层体系结构
- Model is part of the n-tier pattern and the MVC pattern
  模型是n层模式和MVC模式的一部分
- The model communicates with lower abstraction layers (e.g. data access layer)
  模型与较低的抽象层进行通信(例如数据访问层)
- The model might use a notification pattern to inform the view of changes
  模型可能使用通知模式来通知视图更改
- The lesser known presentation-abstraction-control pattern is similar to MVC
  较少为人所知的表示-抽象-控制模式类似于MVC

MVC -Advantages

- Separation of concerns, helps reusability
  关注点分离，有助于重用性
  controller最容易重用,view也可重用，model不能重用
- Multiple user interfaces without changes to the model, e.g. mobile and Web
  多用户界面，而不改变模型，如移动和网络
- Helps configurability (as interface changes are easier, with less expected side effects than changes to the application logic)
  有助于可配置性(因为界面更改更容易，与应用程序逻辑更改相比，预期的副作用更少)

MVC - Disadvantages

- Increases the complexity by additional components
  附加组件增加复杂性，并不是所有系统都适用于MVC模式
  Not all systems are applicable to MVC pattern
- If updates to the view are based on notifications, it
  如果对视图的更新是基于通知的，那么可能很难找到错误
  might be hard to find errors
- In these cases, it is hard to ensure a good usability
  在这些情况下，很难确保良好的可用性(当更新发生时没有控制)
  (no control when an update happens)

PAC

表示-抽象-控制

- Presentation--Abstraction--Control

一种面向交互的软件架构，在某种程度上类似于模型-视图-控制器(MVC)，因为它将一个交互系统分割成三种类型的组件，负责应用程序功能的特定方面

  - It is an interaction--oriented software architecture, and is somewhat similar to model–view–controller (MVC) in that it separates an interactive system into three types of components responsible for speciOic aspects of the application's functionality.
  - The **abstraction** component retrieves and processes the data

抽象组件检索和处理数据

  - The **presentation** component formats the visual and audio presentation of data, and

表示组件对数据的可视和音频表示进行格式化

  - The **control** component handles things such as the Olow of control and communication between the other two components .

控制组件处理其他两个组件之间的低控制和通信等事情

PAC

MVC和PAC之间的区别

- Difference between MVC and PAC
  - The Controllers of MVC focus on the input from and the output to Views,

MVC的控制器关注于视图的输入和输出

  - While the Controllers of PAC focus on the communication and coordination among agents and the ones inside agents.

而PAC的控制器主要负责代理之间和代理内部的沟通协调

  - PAC divides the systems into layered and loose coupled agents,

PAC将系统划分为分层的和松散耦合的代理

  - While MVC focuses on the separation of Models and Views, in which there are no layered agents.

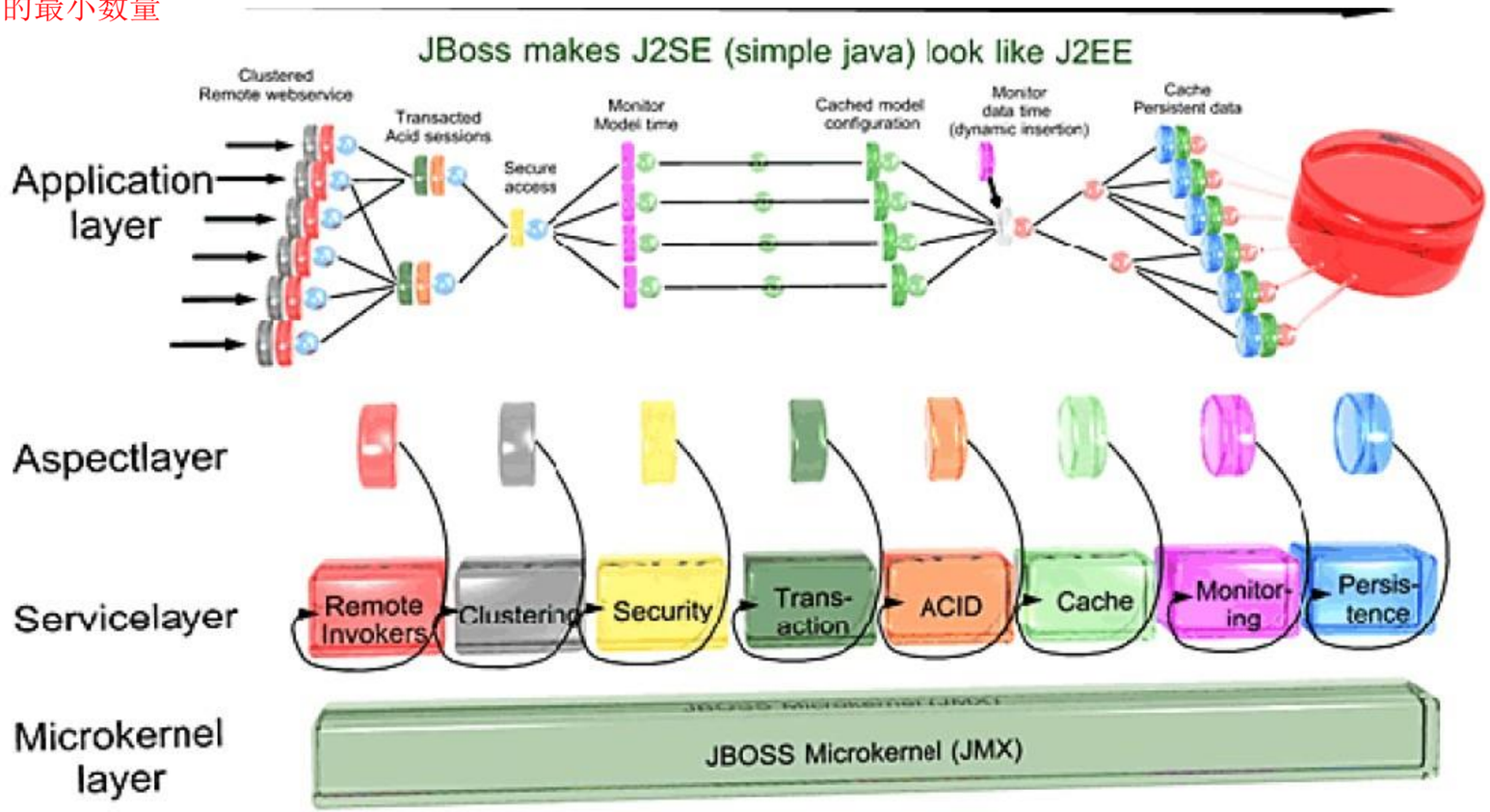
而MVC侧重于模型和视图的分离，其中没有分层的代理

Adaptive Styles – Micro Kernel

- Micro Kernel

– is the near--minimum amount of software that can boot up the necessary services of a software system.

微内核是能够启动一个软件系统所需服务的软件的最小数量

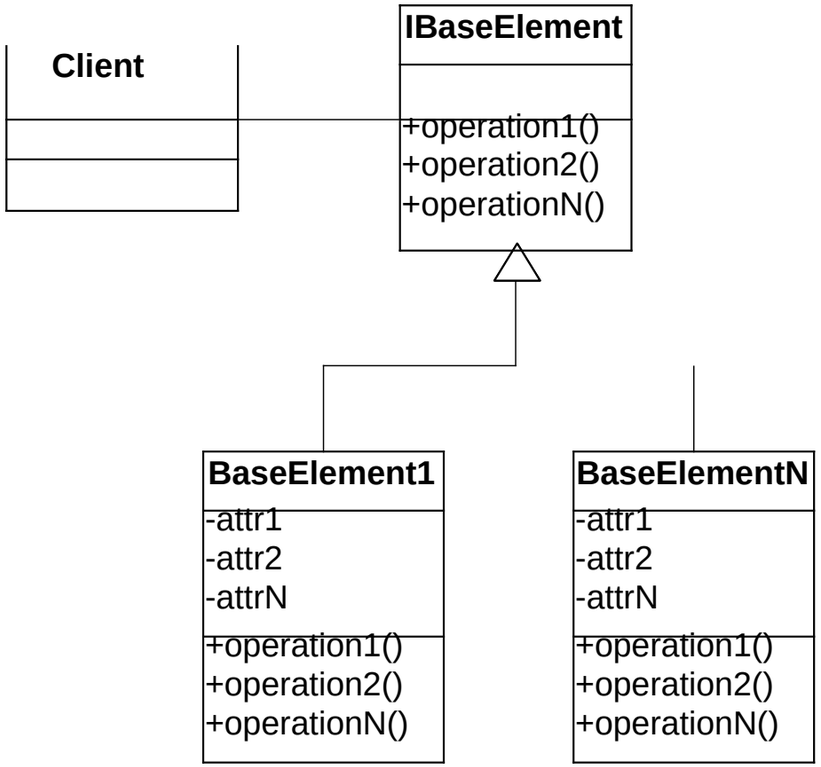


Adaptive Styles – Reflection

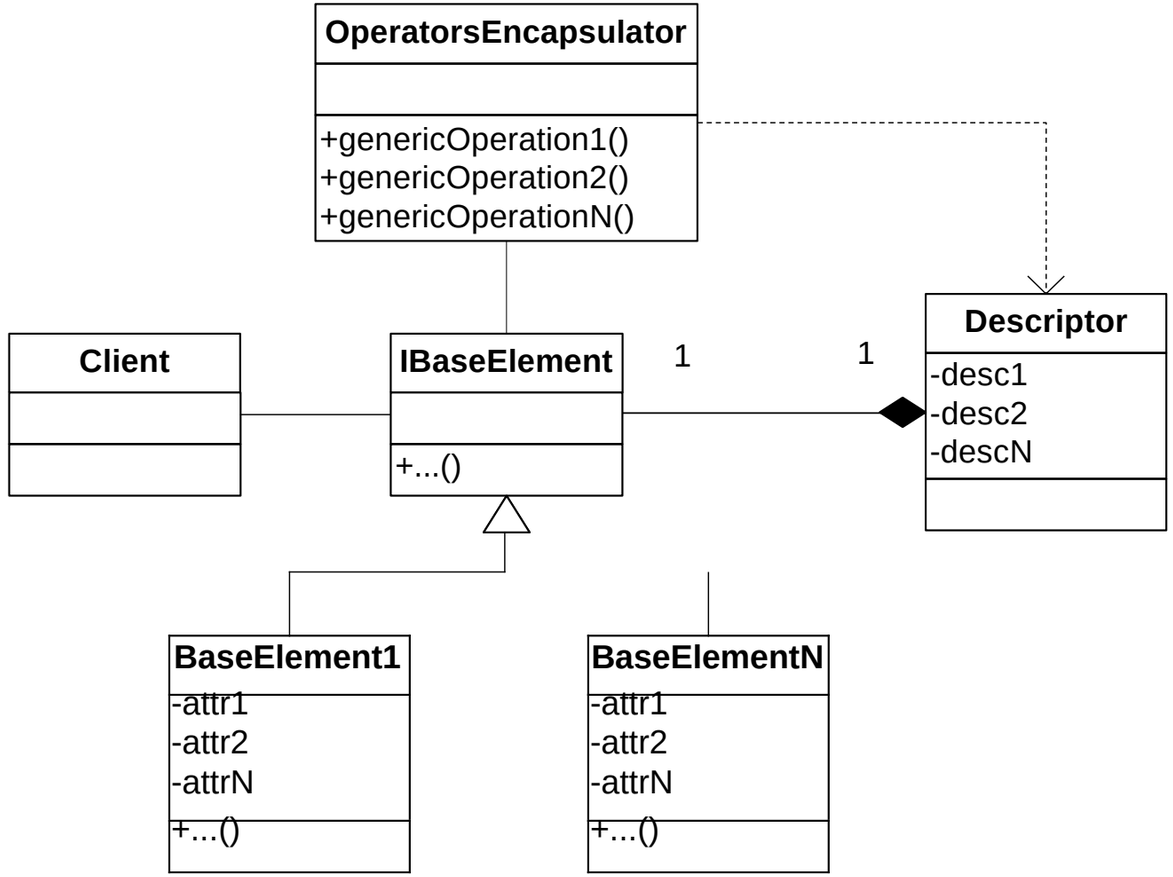
- Reflection
  - Has meta information about other elements of a system.
    有关于系统其他元素的元信息
  - The meta information is used as the basic element to communicate with other element.
    元信息用作与其他元素通信的基本元素
  - The elements described by meta information are base elements.
    元信息描述的元素是基本元素
  - The base elements have many common attributes, onto which the uniOied process is applied.
    基本元素有许多公共属性，uniOied流程将应用于这些属性

底层由java反射机制实现
Class a = Class.forName("具体类名"), 该类名以key-value对的形式保存在Descriptor（配置文件）中，修改配置文件比较方便

Adaptive Styles – Reflection



（a）普通继承–实现架构



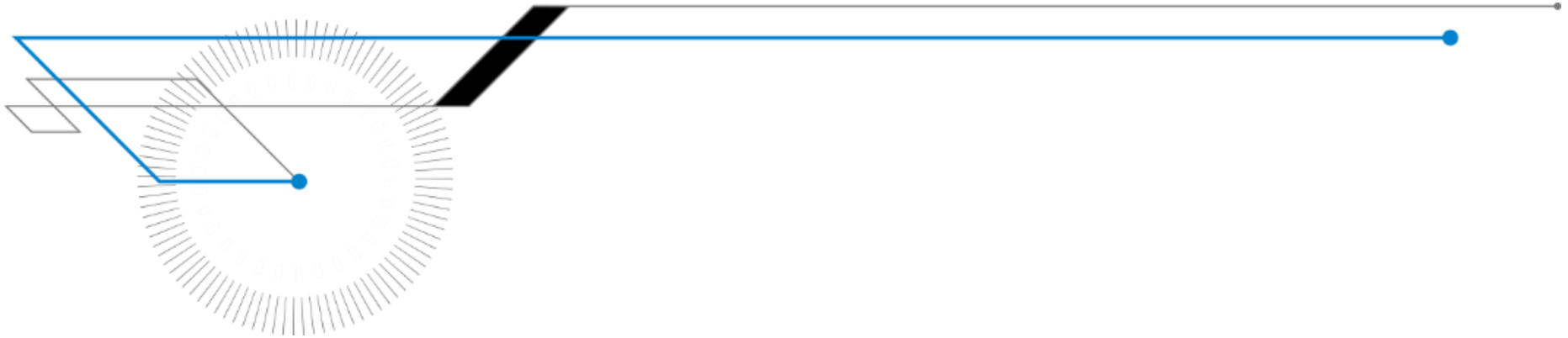
（b）反射架构

Adaptive Styles – Reflection

- Advantage
  - It is quite easy to add new elements or attributes
    添加新元素或属性非常容易
  - It supports various modification
    支持各种修改

- Disadvantage
  - Low performance
    低性能
  - Complex structure of classes
    类的复杂结构

Transaction-Processing



Transaction-Processing Architectural Pattern

一个进程一个接一个地读取一系列输入

- **A process reads a series of inputs one by one.**
  - Each input describes a *transaction* – a command that typically (for example) might **change** some data stored by the system

每个输入描述一个事务--一个通常(例如)可能更改系统存储的某些数据的命令

通常情况下，事务是一个接一个出现的。通常原子

  - Normally transactions come in **one-by-one**. generally **atomic**.
    - Do this, then that, then another thing. Add, Change, Delete…

做这个，再做那个，再做另一件事。添加、更改、删除…

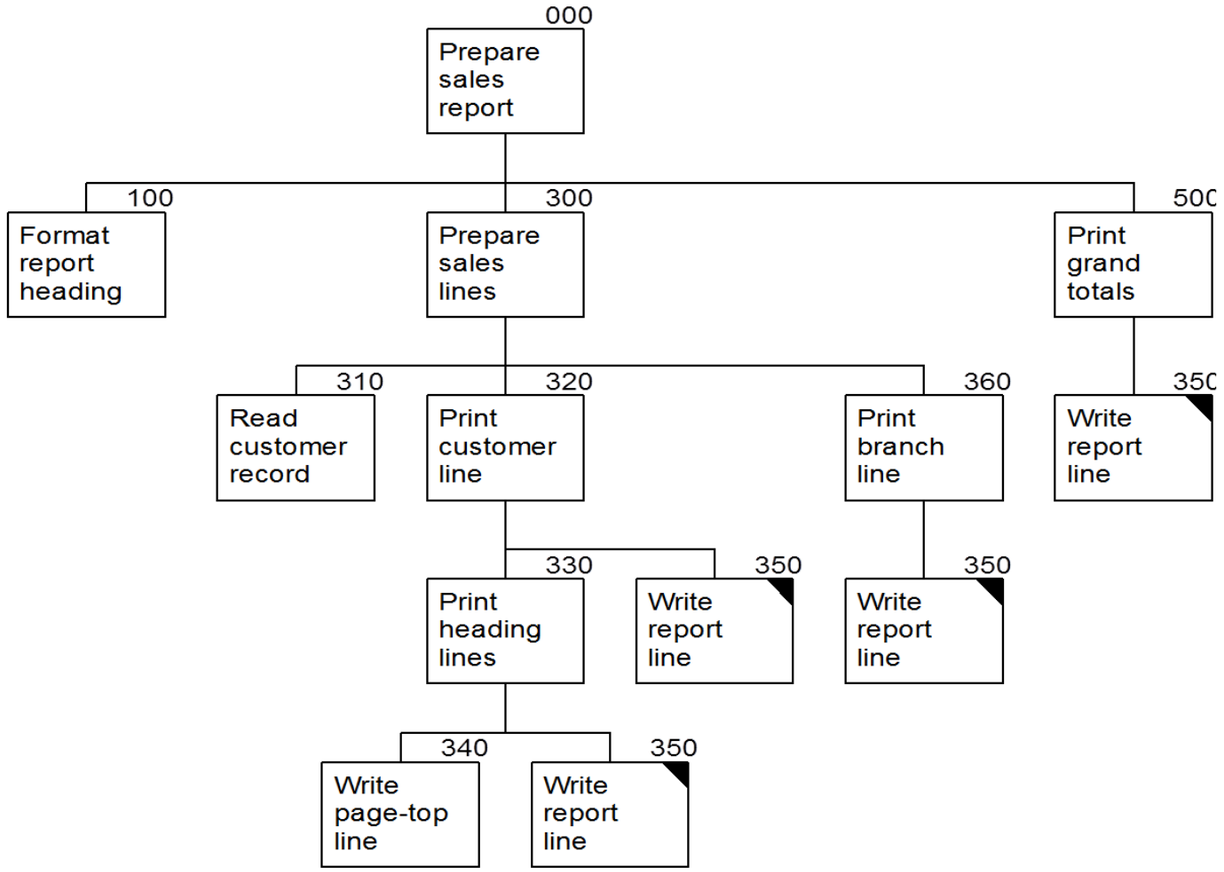
事务分配器简单地处理事务，并将该事务“交给”特定的事务处理程序，该事务处理程序专门用于“处理”此类事务

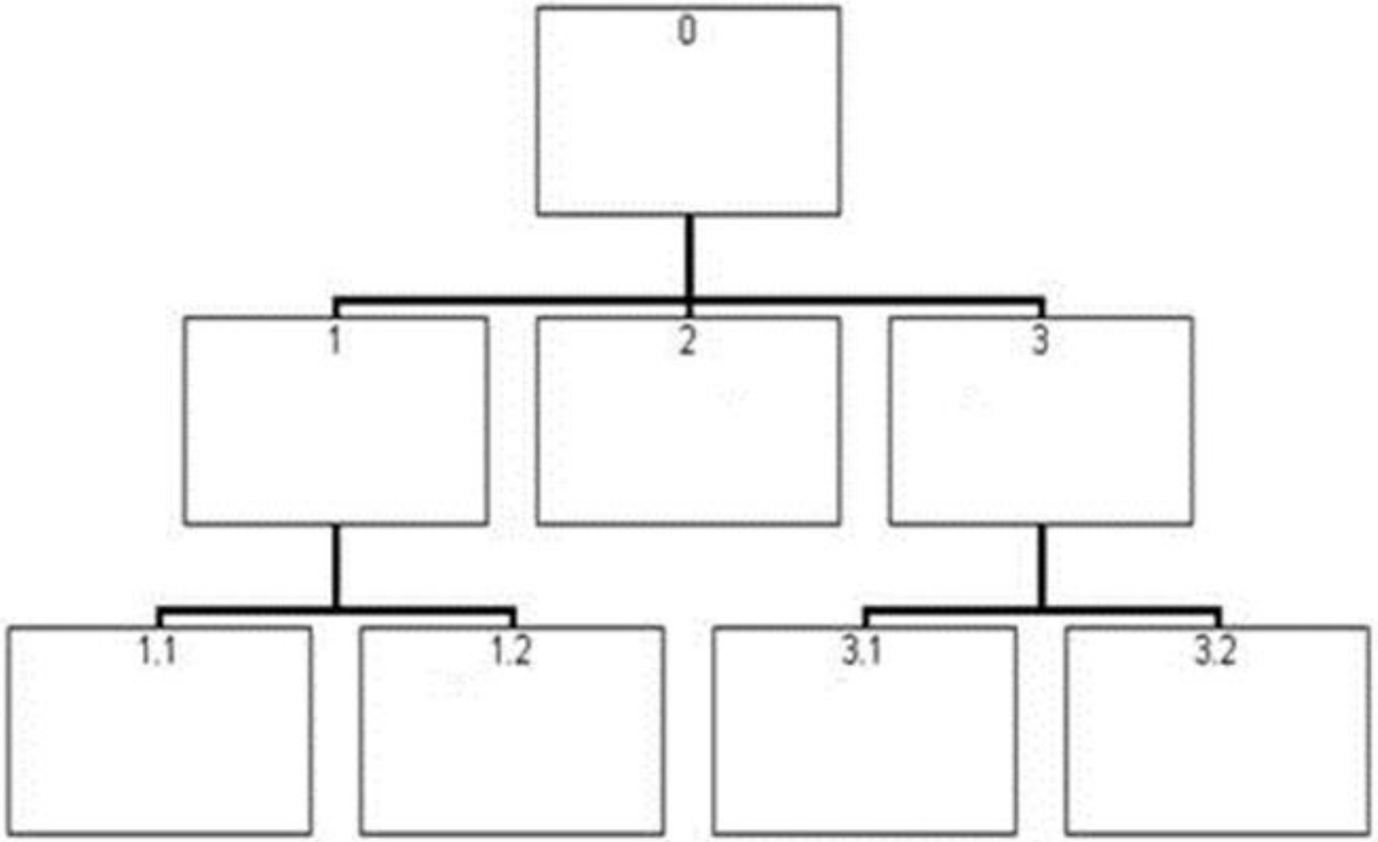
  - A ***transaction dispatcher*** briefly processes a transaction and ‘hands’ that transaction to a specific ***transaction handler*** designed to specifically ‘handle’ that kind of transaction.

事务处理程序是专门为处理特定类型的事务而设计和实现的

  - The ***transaction handle***r is specifically designed and implemented to handle ‘a’ specific type of transaction.

The numbering sequence for the report-preparation program




0
1
2
3
1.1
1.2
3.1
3.2

Transaction Dispatchers – Some Complexity

- Comments:

在线程环境中，许多事务可能在“进程中”，要修改的数据必须适当地锁定和释放。额外的复杂性

- In a threaded environment, where many transactions may be ‘in process,’ data to be modified must be *locked* and *released* as appropriate. Additional complexity.

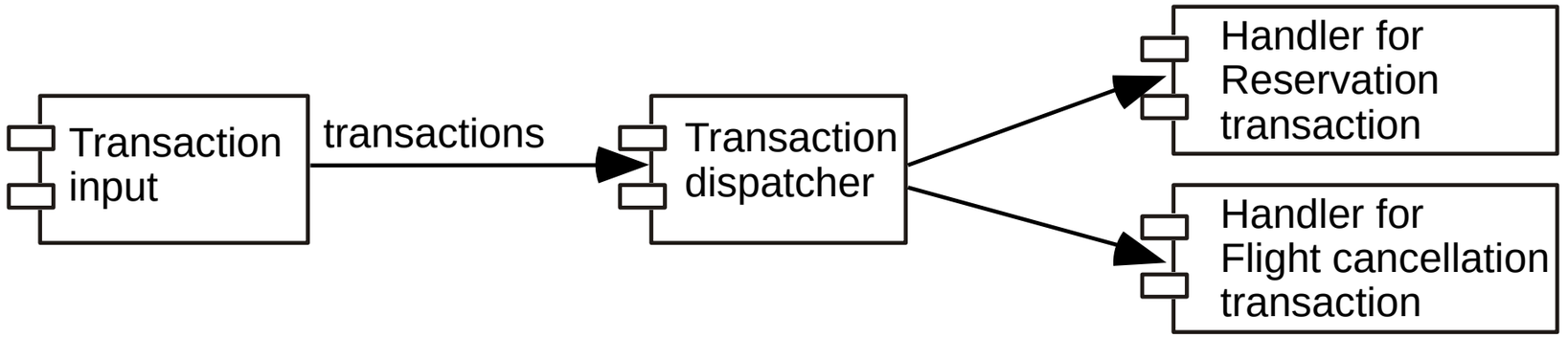
尤其复杂的是，当应用程序需要在更新事务之前执行查询，同时确保数据没有被更改

- Particularly complicated when an application needs to *perform a query* ***prior*** to an update transaction all the while ensuring that the data is not changed…

- See database books on the details. (record / attribute lockout)

Example of a Transaction-Processing System

At a higher level – the application architecture rather than a program architecture:
在更高的层次上--应用程序架构，而不是程序架构



认识到这些“组件”可能存在于本地设备或远程设备上

Recognize that these ‘components’ might exist on a local device or remotely.

“组件”可能只是某个设计子系统的一个实现

The ‘component’ may simply be an implementation of some design subsystem.

The Transaction-Processing Architecture And Design Principles (As Usual, These Are Very Good..)

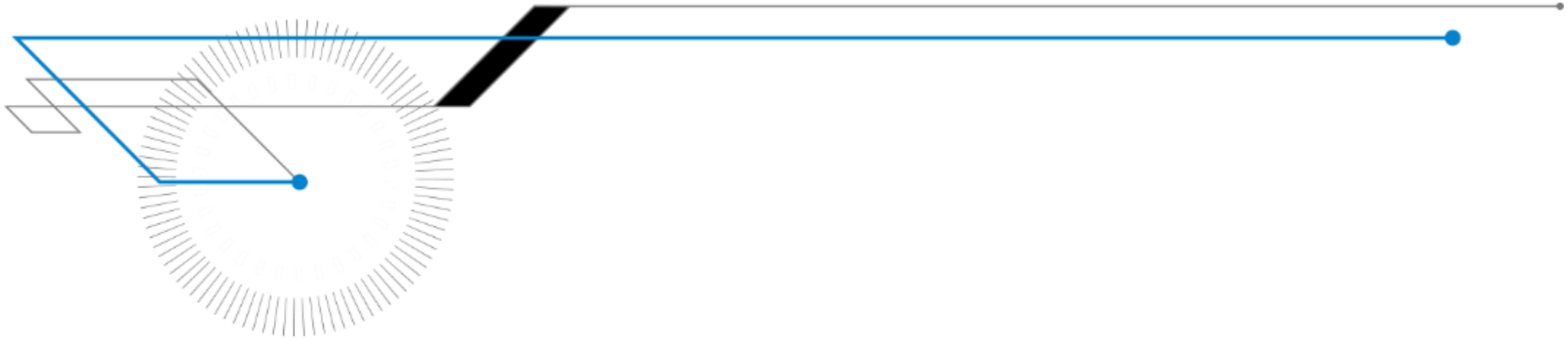
- 1. *Divide and conquer*: The **transaction handlers** are **suitable system divisions** that can be given to separate software engineers for detailed design and development.
  分而治之:事务处理程序是合适的系统划分，可以给独立的软件工程师进行详细的设计和开发
- 2. ***Increase cohesion***: Transaction handlers are **naturally** cohesive units.
  增加内聚性:事务处理程序天生就是内聚单元
- A 'hander' accommodates only 'that' transaction.'
  处理程序只接受那种事务
- 3. *Reduce coupling*: Separating the dispatcher from the handlers clearly **reduces coupling**.
  减少耦合:将调度程序与处理程序分离可以明显减少耦合
- 7. *Design for flexibility*: One may **readily add new** transaction handlers to handle additional transactions.
  为灵活性而设计:可以随时添加新的事务处理程序来处理其他事务
- 11. *Design defensively*: One may add **assertion checking** in each transaction handler and/or in the dispatcher.
  防御性设计:可以在每个事务处理程序和/或分配器中添加断言检查

注意:这里也缺少一些原则...设计决定！！

- Note: several principles missing here too….Design Decisions!!

Transaction关注容错

Others Architectures



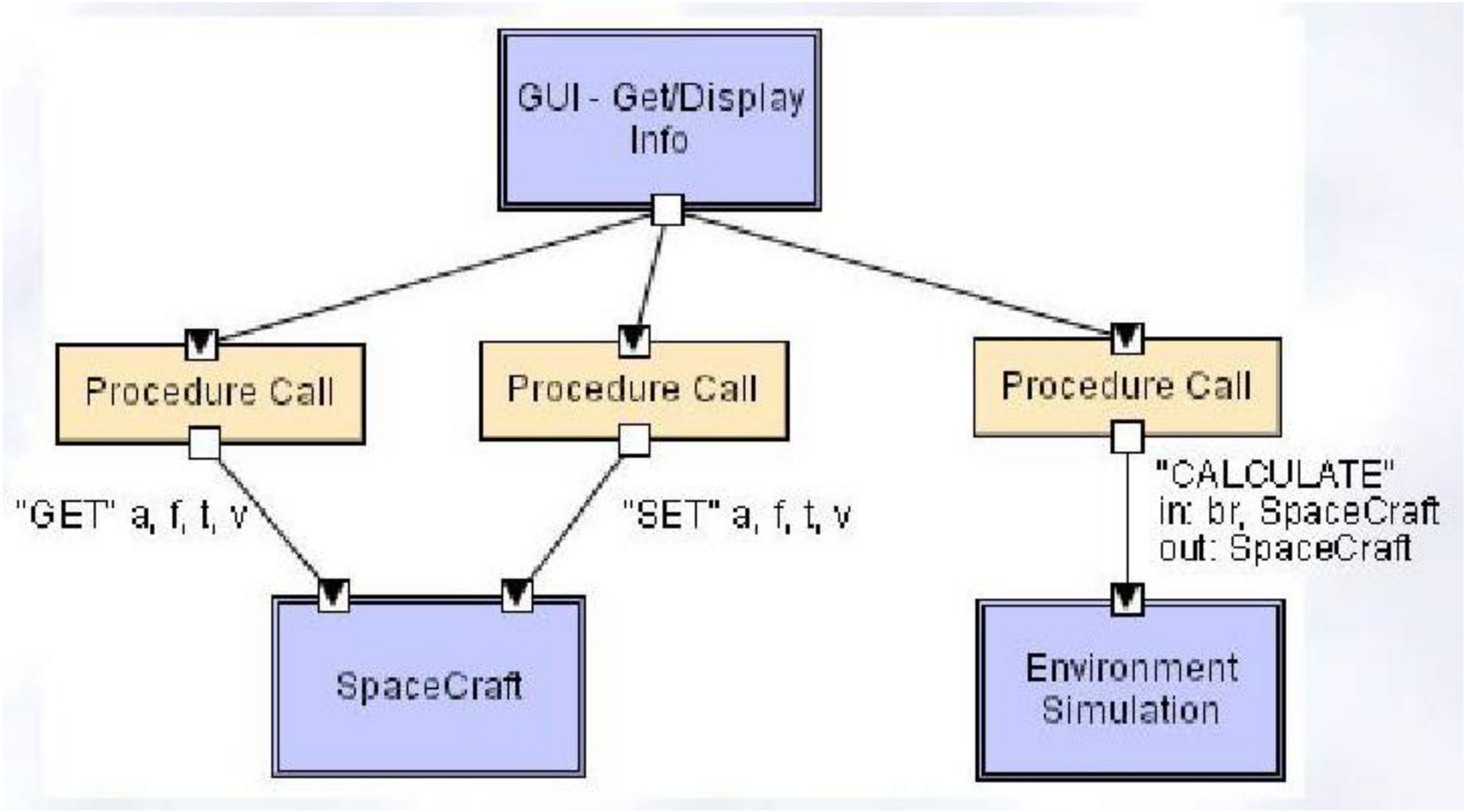
Object-Oriented Style

- Components are objects 组件是对象
  - Data and associated operations 数据及相关操作
- Connectors are messages and method invocations 连接器是消息和方法调用
- Style invariants 风格不变性
  - Objects are responsible for their internal representation integrity 对象负责其内部表示的完整性
  - Internal representation is hidden from other objects 内部表示对其他对象是隐藏的
- Characteristic 特征
  - Preserving the integrity of data representation (persistent objects) 保持数据表示的完整性(持久化对象)
  - Information hiding (encapsulation) 信息隐藏(封装)
  - Combining functions with data 将函数与数据结合

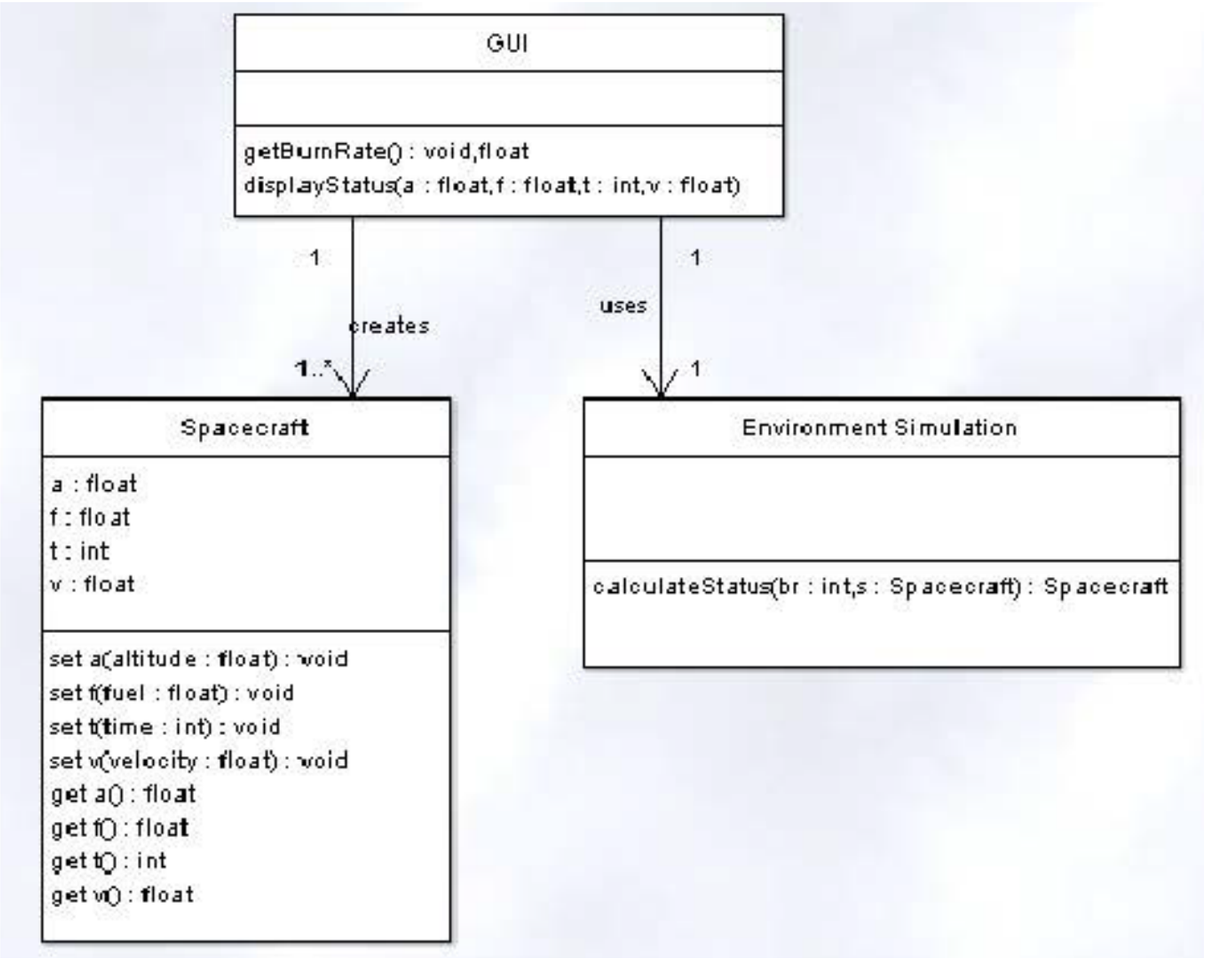
Object-oriented design

- Advantages
  - 对象内部的“无限延展性”
    "Infinite malleability" of object internals
  - 系统分解为一系列交互的代理
    System decomposition into sets of interacting agents
- Disadvantages
  - 对象必须知道服务器的身份
    Objects must know identities of servers
  - 对象方法调用中的副作用
    Side effects in object method invocations

Object-Oriented LL



OO/LL in UML


GUI
getBurnRate() : void,float
displayStatus(a : float,f : float,t : int,v : float)
1
creates
1..*
1
uses
1
Spacecraft
a : float
f : float
t : int
v : float
set a(altitude : float) : void
set f(fuel : float) : void
set t(time : int) : void
set v(velocity : float) : void
get a() : float
get f() : float
get t() : int
get v() : float
Environment Simulation
calculateStatus(br : int,s : Spacecraft) : Spacecraft

Implicit invocation

- The implicit invocation is event-driven, based on the notion of broadcasting (广播）
  隐式调用是事件驱动的, 基于广播的概念(广播)
- Instead of invoking a procedure directly, a component announces (通知） that one or more events have taken place.
  而不是直接调用一个过程, 一个组件通知一个或多个事件发生了
- Registering the procedure
  注册过程
- Data exchange in this type of system must be done through shared data in a repository.
  这种类型的系统中的数据交换必须通过存储库中的共享数据来完成

Implicit Invocation Style

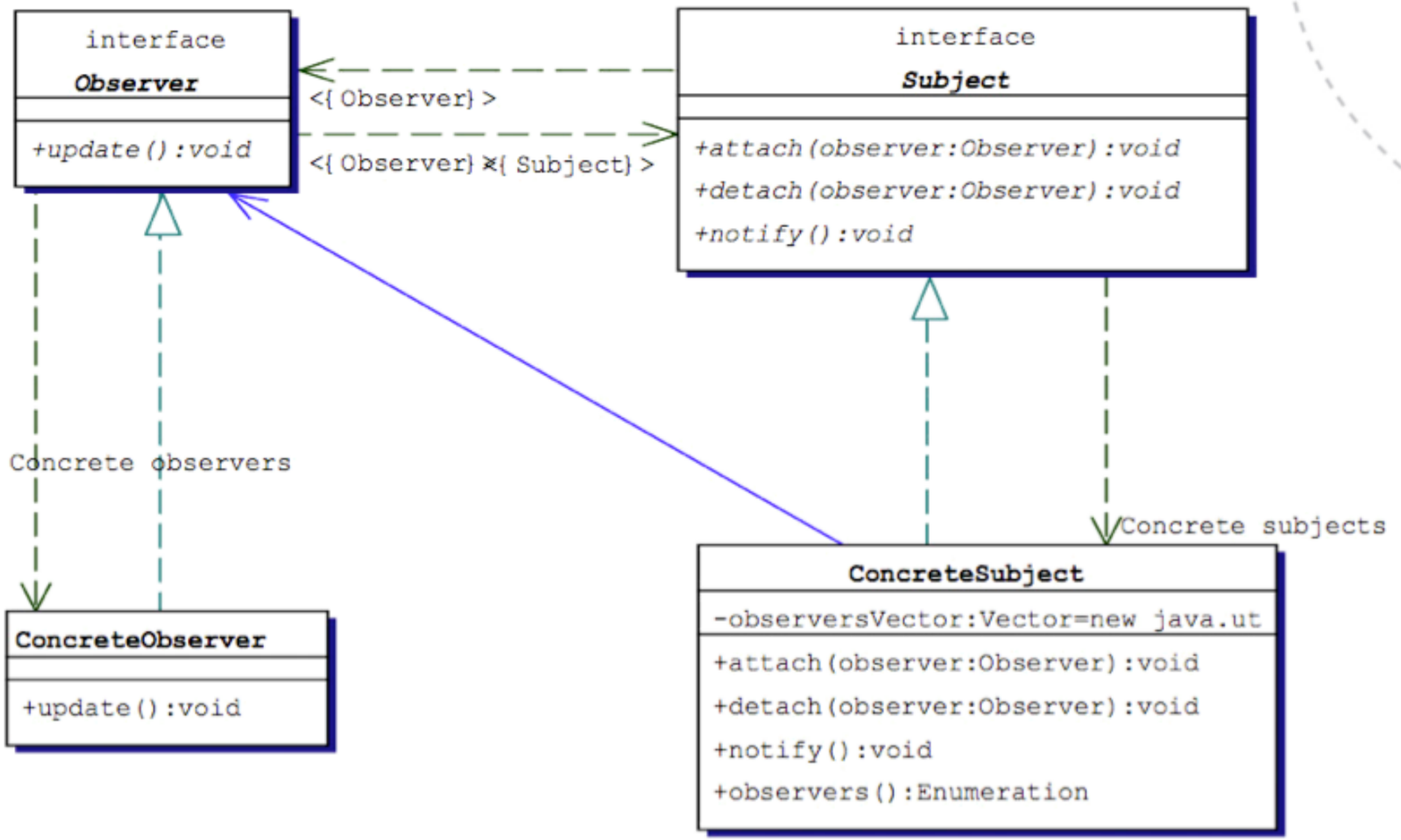
事件声明，而不是方法调用

- Event announcement instead of method invocation
  - "侦听器”注册感兴趣的方法并将其与事件关联起来
  - "Listeners" register interest in and associate methods with events
  - 系统隐式调用所有注册的方法
  - System invokes all registered methods implicitly
- 组件接口是方法和事件
- Component interfaces are methods and events
- Two types of connectors
  - 两种连接器
  -响应事件的调用是显式的或隐式的
  - Invocation is either explicit or implicit in response to events
- Style invariants
  - “广播员”不知道他们的事件的影响
  - "Announcers" are unaware of their events‘ effects
  - 对于响应事件的处理没有任何假设
  - No assumption about processing in response to events

Observer pattern
注册-监听模式

The Observer pattern is used where you need to update several objects when one object is changed. This pattern is used to solve issues related to high coupling.

观察者模式用于在一个对象发生更改时需要更新多个对象的情况。此模式用于解决与高耦合相关的问题。



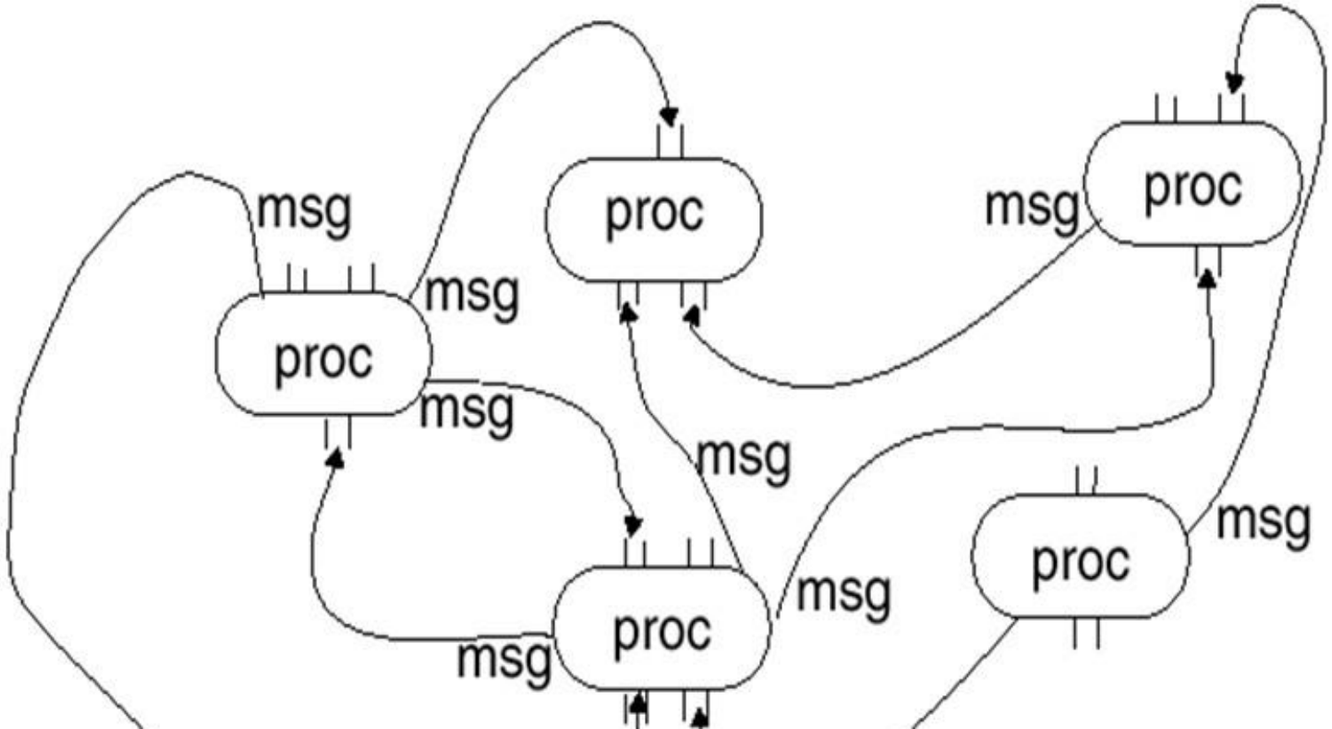
Implicit invocation

- Advantages
  - Easy reuse of components from other systems
    容易重用来自其他系统的组件
  - Especially useful for user interfaces
    对用户界面特别有用
  - System evolution
    系统进化
    · 在系统构造时和运行时都是如此
    - Both at system construction-time & run-time
- Disadvantages
  - The response to an event is not certain.
    事件的响应是不确定的
  - Difficulty to test the system for all possible sequences of events.
    很难测试系统的所有可能的事件序列

Other Styles - Process control

- In process control model
  - The system is divided into multiple independent processes
  - The processes synchronously or asynchronously communicate with each other
  - Communication linkages determine the topological nature of system

-系统被划分为多个独立的进程
-进程之间同步或异步通信
-通讯连接决定系统的拓扑性质



Process control

- Process Control systems are very different from function- or object-based designs.
过程控制系统与基于功能或对象的设计非常不同
- Process control system are characterized not only by the type of component, but also by the relationships that hold among them.
过程控制系统的特征不仅在于部件的类型，而且在于它们之间的关系
  - 核动力系统中对核燃料棒中原子分裂状态的控制；
  - 空调系统的温度控制
- The most common software-based control system involves a closed loop in one of two forms, feedback and feedforward.
最常见的基于软件的控制系统包括两种形式之一的闭环，反馈和前馈

Other Styles - Process control

有标准的控制流

- Advantage
  - 问题的分解是直接的
  - 系统的演进和集成容易
  - 可以通过并行来提高性能
  - The decomposition of problem is straightforward
  - The evolution and integration of system is easy
  - Performance can be improved by parallelizing

对通讯的实时性要求较高

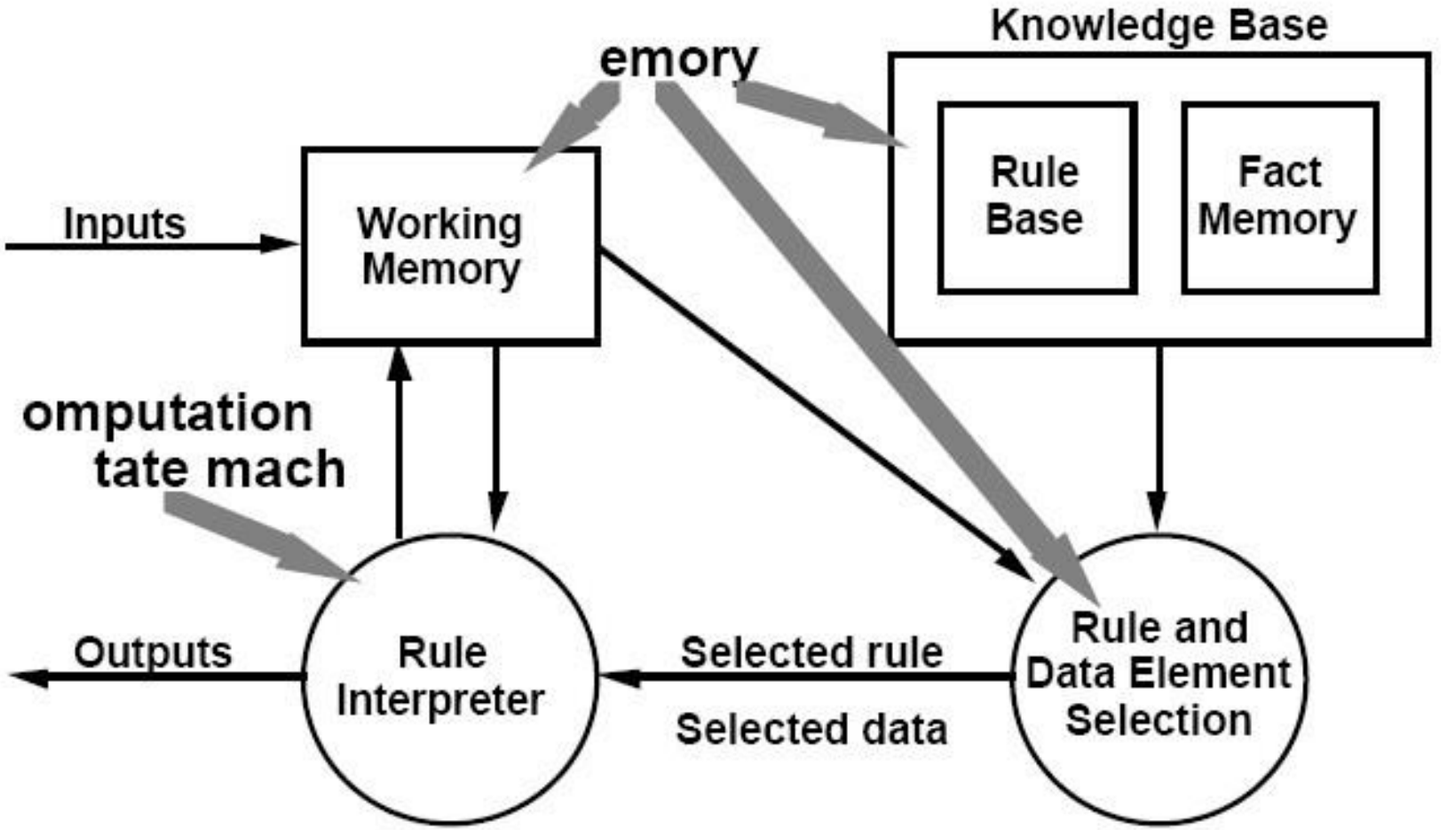
- Disadvantage
  - 沟通的时间是难以控制的
  - 流程之间的通讯成本相当高
  - The timing of communication is difficult to control
  - The cost of communication between processes is rather high

Other Styles - Rule-based Arch.

对输入进行解析，根据规则确定系统状态，类似于之前的解释器架构

- Ruled--based architecture
  - Encodes the knowledge of human experts into rules
  - The rules are executed or activated when the speciOied conditions are satisfied

-将人类专家的知识编码为规则
-当指定的条件满足时，规则被执行或激活



Rule-Based Style

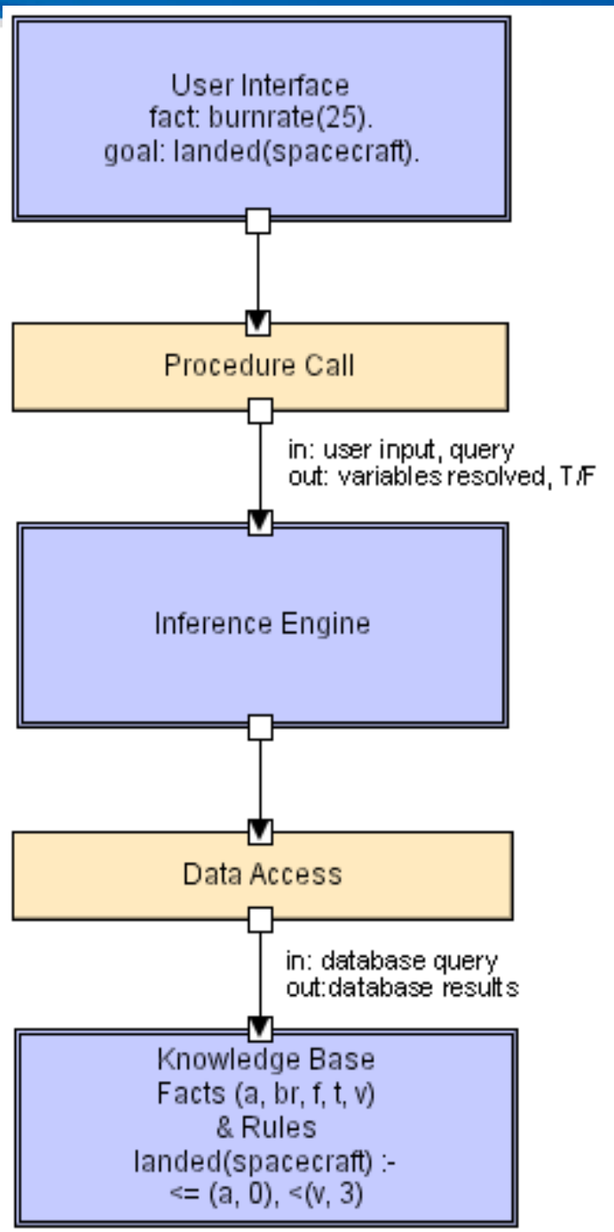
- Inference engine parses user input and determines whether it is a fact/rule or a query. If it is a fact/rule, it adds this entry to the knowledge base. Otherwise, it queries the knowledge base for applicable rules and attempts to resolve the query.

推理引擎解析用户输入并确定它是事实/规则还是查询。如果它是一个事实/规则，它将该条目添加到知识库中。否则，它将查询知识库中的适用规则，并尝试解析查询

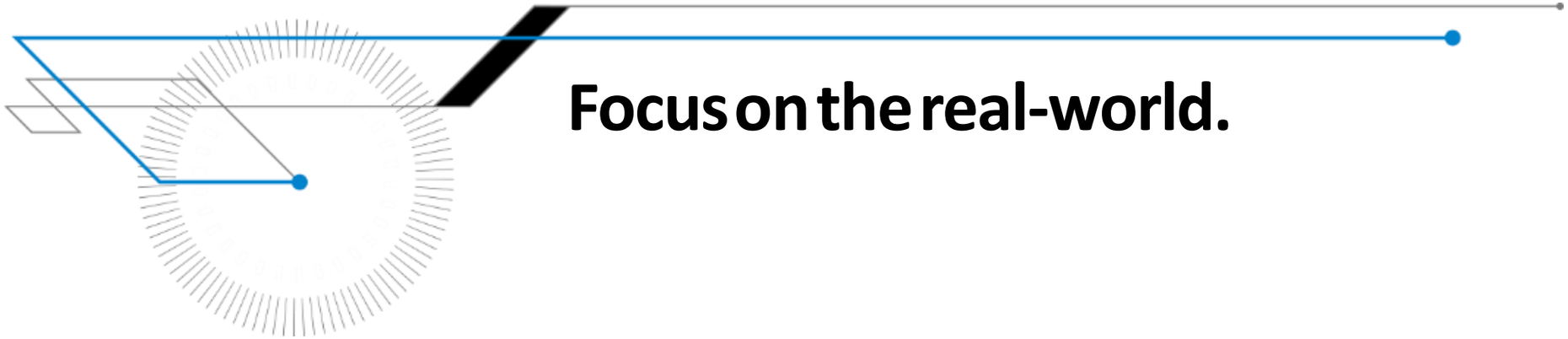
Rule-Based Style (cont'd)

- Components: User interface, inference engine, knowledge base
  组件:用户界面，推理引擎，知识库
- Connectors: Components are tightly interconnected, with direct procedure calls and/or shared memory.
  连接器:组件通过直接的过程调用和/或共享内存紧密地相互连接
- Data Elements: Facts and queries
  数据元素:事实和查询
- Behavior of the application can be very easily modified through addition or deletion of rules from the knowledge base.
  通过从知识库中添加或删除规则，可以非常容易地修改应用程序的行为
- Caution: When a large number of rules are involved understanding the interactions between multiple rules affected by the same facts can become *very* difficult.
  注意:当涉及大量规则时，理解受相同事实影响的多个规则之间的交互将变得非常困难

Rule Based LL



Heterogeneous Architectures



Focus on the real-world.

Heterogeneous architectures

没有一个真正的系统严格遵循单一的风格

- No real system follows strictly only a single style
- Styles themselves are not that strictly separated but they are blurred Architectures might be conceptually heterogeneous

样式本身并不是严格分离的，但它们是模糊的。架构在概念上可能是异构的

n层架构是分层架构

- N-tier architectures are layered architectures
- N-tier architectures are typically data-centric architectures Thin clients involve some sort of notification architecture

n层体系结构通常是以数据为中心的体系结构，瘦客户机涉及某种通知体系结构

Heterogeneous architectures

- Architectures might be structurally heterogeneous
  架构在结构上可能是异构的，总体架构遵循一种风格
  Overall architecture follows one style
- 单个组件遵循其他样式
  Single components follow other styles
- The Web has a 2-tier architecture: browser and
  Web有两层架构:浏览器和服务器
  server
- The browser itself has a notification architecture to
  浏览器本身有一个处理用户事件的通知体系结构
  handle user events

Heterogeneous architectures

- Architectures might be heterogeneous at execution level At execution level components follow different styles

  执行级别的架构可能是异构的，执行级别的组件遵循不同的样式

- E.g. notification architectures might include remote procedure calls if components are distributed

  例如，如果组件是分布式的，通知架构可能包括远程过程调用

- Service architectures require networking architecture which is a layered architecture, etc..

  服务架构需要网络架构，网络架构是分层的架构，等等

- In real systems architectures are heterogeneous at all levels!

  在实际系统中，架构在所有层次上都是异构的

Heterogeneous architectures

- Web-based search engine

基于网络的搜索引擎
•概念上:以数据为中心，分层，三层
•结构上:分层(网络)，三层，通知
•执行:分布式、面向服务、带有回调的通知...

- Conceptually: data-centric, layered, 3-tier
- Structurally: layered (network), 3-tier, notification
- Execution: distributed, service-oriented, notification with callbacks, ...

Example: Web search engine architecture

爬虫:多种和分布式
· 例如，你有分层的、面向服务的通知
· URLServer通知并同步要获取url的爬虫程序
· 从storeserver中获取并存储的网页
· Storeserver:以数据为中心的架构
· searcher作为网络服务器运行

- Crawlers: multiple and distributed
- I.e. you have layered, service-oriented, notification
- URLServer notifies and syncronizes crawlers that URLs to fetch
- Web pages fetched and stored in storeserver
- Storeserver: data-centric architecture
- The searcher runs as a Web server

Conclusion

- Architectural styles provide patterns
- Why one should know these styles?
- They provide a common vocabulary
- They provide blueprints on how a system can be designed
- They provide a well known structure and behaviour

· 架构风格提供模式
· 为什么要知道这些风格?
· 它们提供通用的词汇
· 他们提供如何设计系统的蓝图
· 它们提供了众所周知的结构和行为

Table 1. Patterns' Impact on Usability, Security, Maintainability and Efficiency

模式影响可用性、安全性、可维护性和效率

	Usability	Security	Maintainability	Efficiency
Layers	*Neutral*	*Key Strength:* Supports layers of access.	*Key Strength:* Separate modification and testing of layers, and supports reusability	*Liability:* Propagation of calls through layers can be inefficient
Pipes and Filters	*Liability:* Generally not interactive	*Liability:* Each filter needs its own security	*Strength:* Can modify or add filters separately	*Strength:* If one can exploit parallel processing *Liability:* Time and space to copy data
Blackboard	*Neutral*	*Liability:* Independent agents may be vulnerable	*Key Strength:* extendable *Key Liability:* Difficult to test	*Liability:* Hard to support parallelism
Model View Controller	*Key Strength:* Synchronized views	*Neutral*	*Liability:* Coupling of views and controllers to model	*Liability:* Inefficiency of data access in view
Presentation Abstraction Control	*Strength:* Semantic separationo	*Neutral*	*Key Strength:* Separation of concerns	*Key Liability:* High overhead among agents
Microkernel	*Neutral*	*Neutral*	*Key Strength:* Very flexible, extensible	*Key Liability:* High overhead
Reflection	*Neutral*	*Neutral*	*Key Strength:* No explicit modification of source code	*Liability:* Meta-object protocols often inefficient
Broker	*Strength:* Location Transparency	*Strength:* Supports access control	*Strength:* Components easily changed	*Neutral:* Some communication overhead

Table 2. Patterns' Impact on Reliability, Portability, and Implementability

	Reliability	Portability	Implementability
Layers	*Strength*: Supports fault tolerance and graceful undo	*Strength*: Can confine platform specifics in layers	*Liability*: Can be difficult to get the layers right
Pipes and Filters	*Key Liability*: Error handling is a problem	*Key Strength*: Filters can be combined in custom ways	*Liability*: Implementation of parallel processing can be very difficult
Blackboard	*Neutral*: Single point of failure, but can duplicate it	*Neutral*	*Key Liability*: Difficult to design effectively, high development effort
Model View Controller	*Neutral*	*Liability:* Coupling of components	*Liability:* Complex structure
Presentation Abstraction Control	*Neutral*	*Strength:* Easy distribution and porting	*Key Liability:* Complexity; difficult to get atomic semantic concepts right
Microkernel	*Strength:* Supports duplication and fault tolerance	*Key Strength:* Very easy to port to new hardware, OS, etc	*Key Liability:* Very complex design and implementation
Reflection	*Key Liability:* Protocol robustness is key to safety	*Strength:* If you can port the meta-object protocol	*Liability:* Not well supported in some languages
Broker	*Neutral:* Single point of failure mitigated by duplication	*Key Strength:* Hardware and OS details well hidden	*Strength:* Can often base functionality on existing services.

When There's No Experience to Go On...

- The first effort you should make in addressing a novel design challenge is to attempt to determine that it is genuinely a novel problem.

在处理一个新颖的设计挑战时，您应该做的第一个努力是尝试确定它确实是一个新颖的问题

- Basic Strategy
  - 分散--摆脱不充分的以前的方法，发现/承认各种新的想法，提供一个潜在可行的解决方案
  - Divergence – shake off inadequate prior approaches and discover/admit a variety of new ideas that offer a potentially workable solution
  - 转化--结合分析和选择这些可能的创意。对问题陈述有新的理解和改变
  - Transformation – combine analysis and selection of these potential ides. New understanding and changes to the problem statement
  - 收敛--选择和进一步细化想法
  - Convergence – select and further refine ideas
- Repeatedly cycling through the basic steps until a feasible solution emerges.

重复循环基本步骤，直到出现可行的解决方案

Analogy Search 类比搜索

- Examine other fields and disciplines unrelated to the target problem for approaches and ideas that are analogous to the problem.
  检查与目标问题无关的其他领域和学科，寻找与目标问题类似的方法和思想
- Formulate a solution strategy based upon that analogy.
  在这个类比的基础上制定解决方案策略
- A common “unrelated domain” that has yielded a variety of solutions is nature, especially the biological sciences.
  一个共同的“不相关的领域”产生了各种各样的解决方案，那就是自然，特别是生物科学。
  - E.g., neural networks
    ◆如,神经网络

Brainstorming 头脑风暴 头

- Technique of rapidly generating a wide set of ideas and thoughts pertaining to a design problem
  与设计问题相关的快速产生广泛想法和想法的技术
  - without (initially) devoting effort to assessing the feasibility.
    没有(最初)投入精力去评估可行性
- Brainstorming can be done by an individual or, more commonly, by a group.
  头脑风暴可以由个人完成，或者更常见的是由一群人完成
- Problem: A brainstorming session can generate a large number of ideas… all of which might be low-quality.
  问题:头脑风暴会议可以产生大量的想法，但所有的想法都可能是低质量的
- Chief value: identifying categories of possible designs, not any specific design solution suggested during a session.
  主要价值:确定可能的设计类别，而不是在会议中建议的任何具体设计方案

“Literature” Search

- Examining published information to identify material that can be used to guide or inspire designers
- Digital library collections make searching much faster and more effective
  - IEEE Xplore
  - ACM Digital Library
  - Google Scholar
- The availability of free and open-source software adds special value to this technique.

Morphological Charts

- The essential idea:
  - identify all the primary functions to be performed by the desired system
  - for each function identify a means of performing that function
  - attempt to choose one means for each function such that the collection of means performs all the required functions in a compatible manner.
- The technique does not demand that the functions be shown to be independent when starting out.
- Sub-solutions to a given problem do not need to be compatible with all the sub-solutions to other functions in the beginning.

Removing Mental Blocks

- If you can’ t solve the problem, change the problem to one you can solve.
  - If the new problem is “close enough” to what is needed, then closure is reached.
  - If it is not close enough, the solution to the revised problem may suggest new venues for attacking the original.

Controlling the Design Strategy

- Exploring diverse approaches
  - Potentially chaotic
  - be care in managing the activity
- Identify and review *critical* decisions
- Relate the costs of research and design to the penalty for taking wrong decisions
- Insulate uncertain decisions
- Continually re-evaluate system “requirements” in light of what the design exploration yields