軟體架構師

報告人： 劉祖弘

日　期： 2017/10/18

Agenda

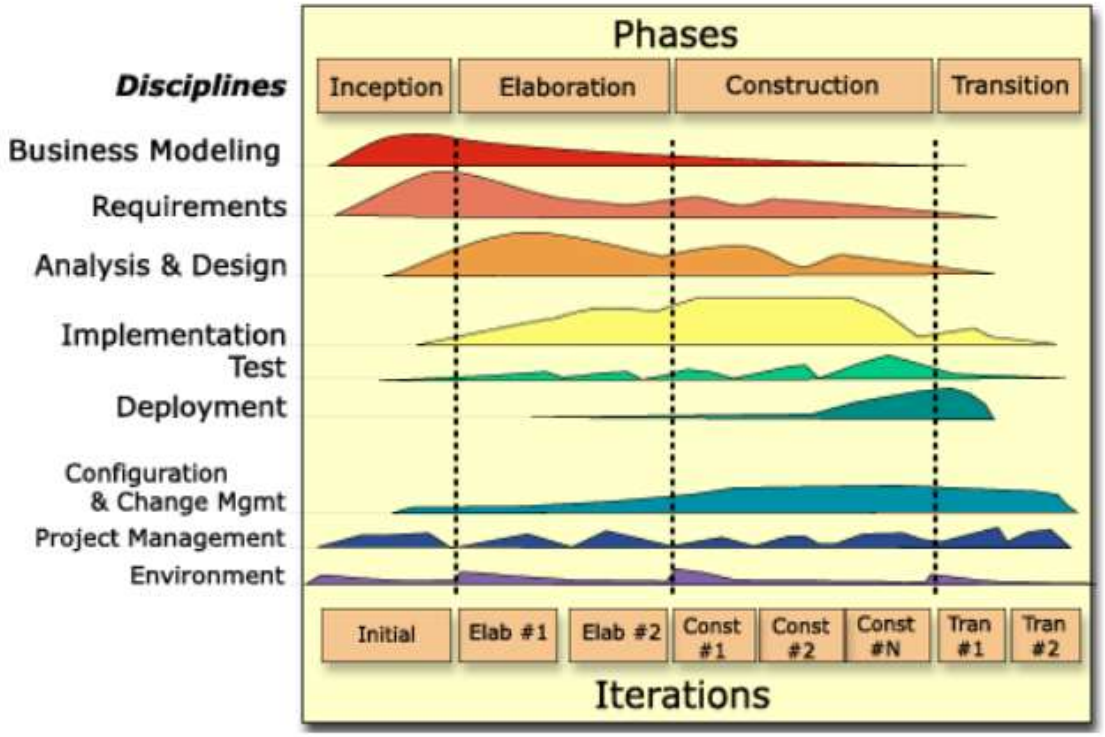
1. 前言 – RUP & UML
2. 軟體架構師的職責、挑戰、角色關係，與設計師、分析師的區別
3. 商業塑模 – Business Modeling
4. 架構性分析 – Architectural Analysis
5. 架構設計 – Architecture Design
6. 設計模式 – Design Pattern
7. Design Resilience
8. Run-Time Architecture
9. Design Distributed System
10. Design Scalability

1. 前言 – RUP & UML

PIC

RUP (Rational Unified Process)

- RUP is a **software engineering process**.
- It provides a **disciplined approach** to assigning tasks and responsibilities within a development organization.
- Its goal is to ensure the production of **high-quality** software that meets the needs of its end users within a predictable schedule and budget.



來源：http://sce.uhcl.edu/helm/rationalunifiedprocess/

RUP 三大特點、六大最佳實務

三大特點

1. **使用案例**驅動
2. 以**架構**為中心
3. **反覆**漸進式開發

六大最佳實務

1. **迭代式**開發 (Develop iteratively)
2. 管理**需求** (Manage requirements)
3. **可視化**建模 (Model visually)
4. 使用**組件** (Use components)
5. 驗證軟體**品質**(Verify quality)
6. 控制軟體**變更** (Control changes)

RUP 四個階段

1. 初始階段 (Inception)

進行可行性研究，定義出專案大小及涵蓋範圍，評估專案所需的能力、時程與經費，以及資訊系統預期達到之效益、了解商業模型及需求。

2. 精細規劃階段 (Elaboration)

擬定專案計畫、系統特性與架構、確認商業模型及需求、進行系統分析與設計。

3. 建構階段 (Construction)

建構產品並進行單元測試、整合測試。

4. 移轉階段 (Transition)

將產品分批交付給客戶進行驗收測試，並進行使用者訓練。

RUP 九個核心流程

6 個核心過程工作流 (Core Process Workflows)

1. 商業建模 (Business Modeling)
2. 需求 (Requirements)
3. 分析和設計 (Analysis & Design)
4. 實作 (Implementation)
5. 測試 (Test)
6. 部署 (Deployment)

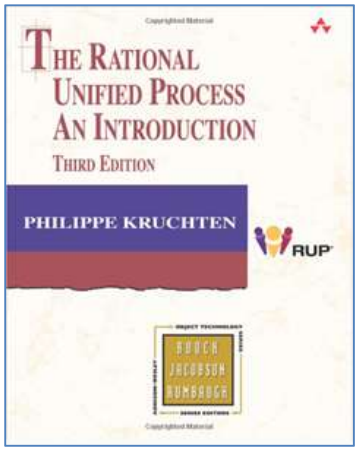
3 個核心支持工作流 (Core Supporting Workflows)

1. 配置和變更管理 (Configuration & Change Management)
2. 專案管理 (Project Management)
3. 環境 (Environment)

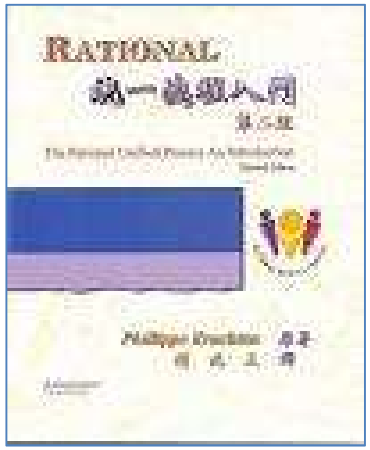
RUP 九個核心流程

1. 該程序方法論結合了眾多成功的方法論，提供完整的軟體開發流程。
2. 結合採用**反覆式 (Iterative)** 開發流程失敗，可降低開發的風險。
3. **UML**作為其視覺化軟體分析與設計語言，使軟體開發人員之間的溝通與資訊的交換無礙。
4. 以UML中的**使用者案例** (Use-Case) 作為整個Rational Unified Process 的核心。
5. 受到**工具**的支援：程序經過良好的支援，且受到好的工具支援。例如，Rational Rose、Rational Unified Process。
6. 對於軟體開發各階段中所參與之**角色**皆定義每其應有之責任與活動。

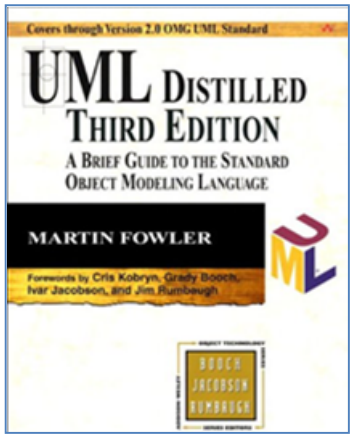
RUP & UML 經典書籍



書名	The Rational Unified Process: An Introduction (3rd Edition)
作者	Philippe Kruchten
出版日期	2003-12-20
ISBN	978-0321197702



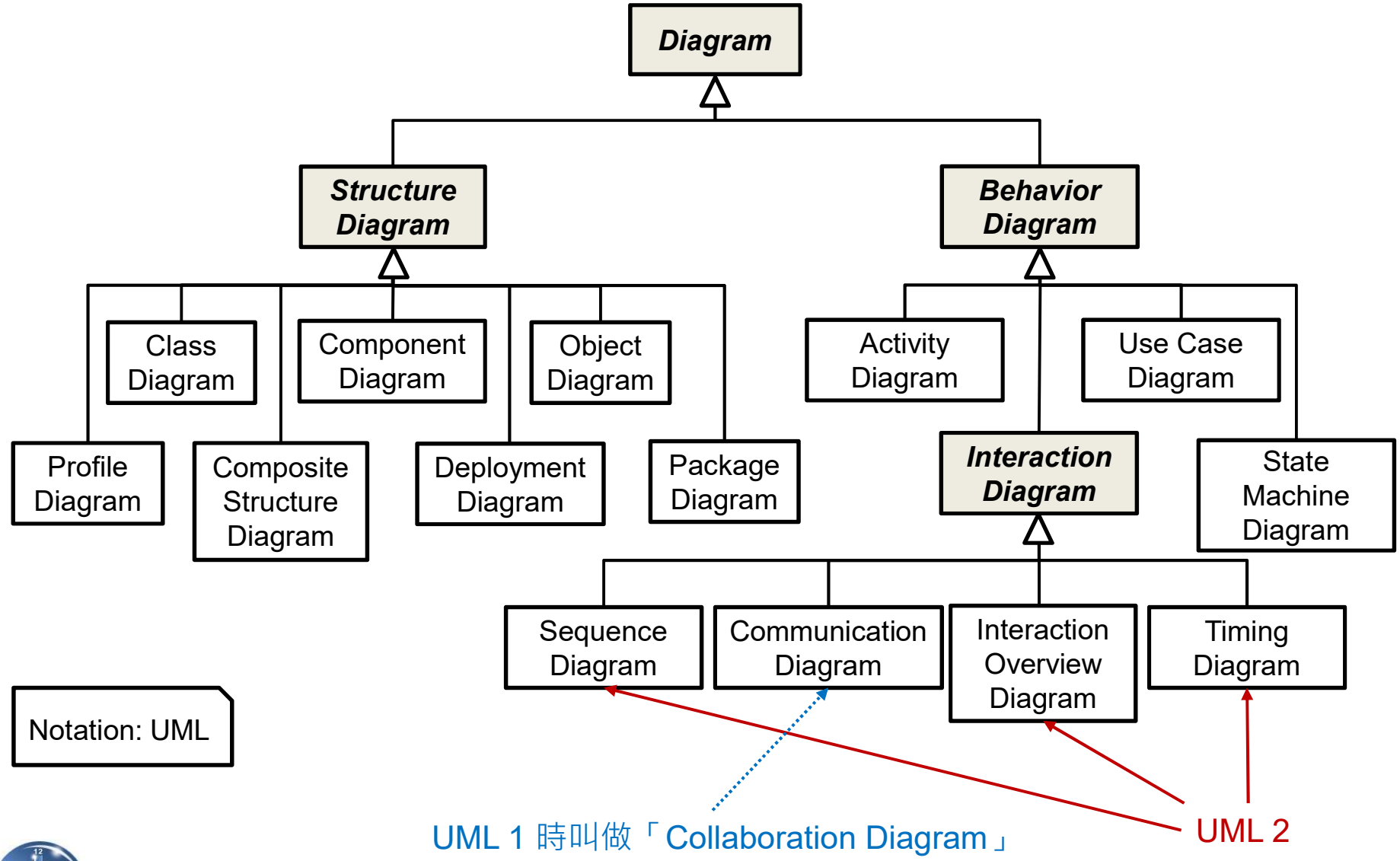
書名	統一流程入門 第二版
作者	趙光正
出版日期	2002-01-22
ISBN	957-8675-79-8



書名	UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition)
作者	Martin Fowler
出版日期	2003-09-25
ISBN	978-0321193681

書名	UML精華第三版：標準物件模型語言
作者	趙光正
出版日期	2005-01-07
ISBN	9789861540399

UML 2 diagrams



UML (Unified Modeling Language)

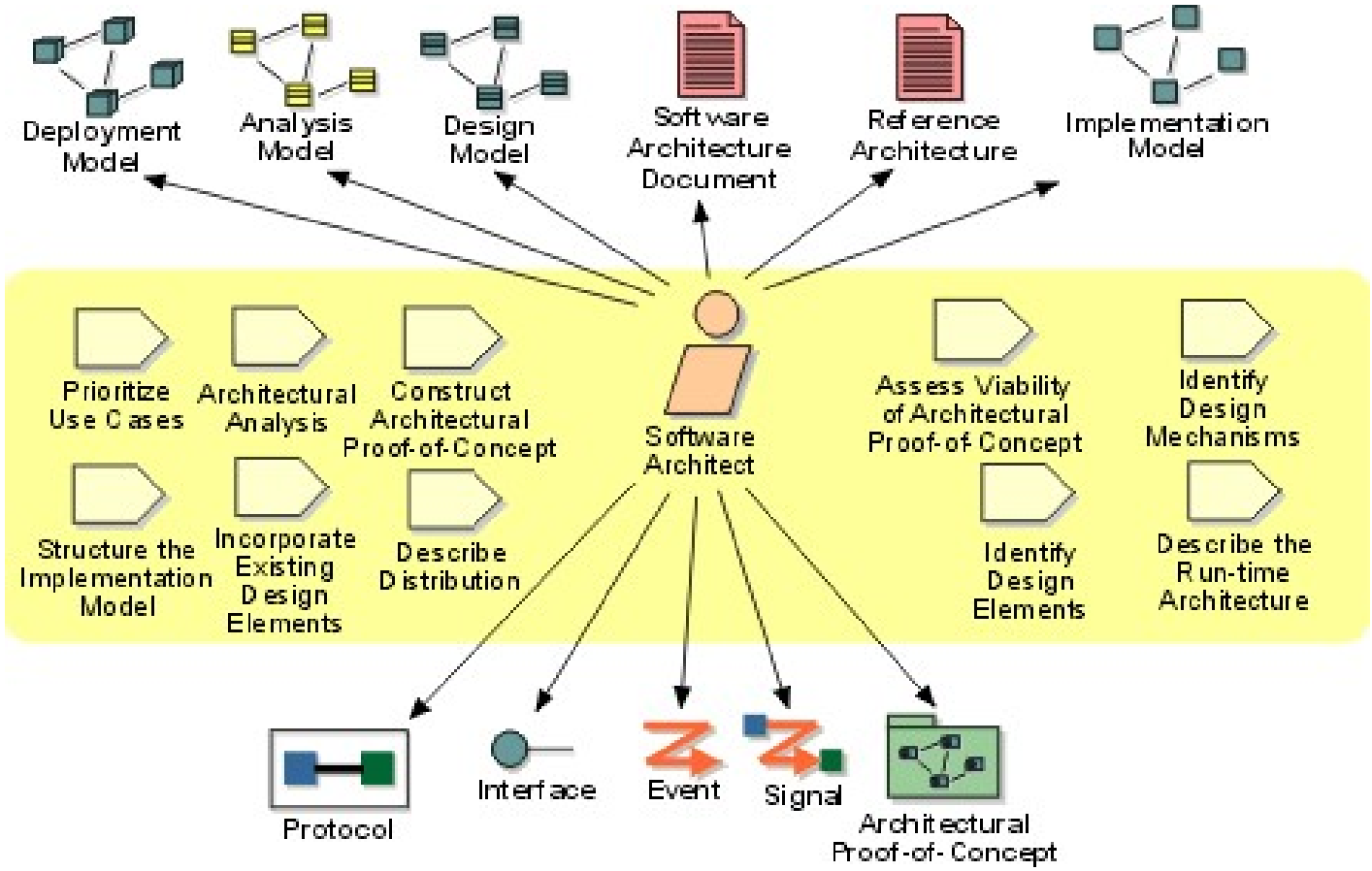
Grady Booch　　Ivar Jacobson　　James Rumbaugh

1. 在軟體工程領域中，一種一般用途、輔助開發的建模語言，試圖提供一種標準方式來將系統設計工作視覺化。
2. **1994 ~ 1995 年**期間，由當時任職於 Rational Software 的 **Grady Booch**、**Ivar Jacobson** 與 **James Rumbaugh** 共同發展，以統一並標準化當時幾種不同的表示系統與軟體設計方法。
3. **1997** 年 UML 1.0 提案給 OMG。目前最新版本是 UML 2.5 (2015 年 3月)。
4. UML 雖還不是一個工業標準，但在 OMG 的推廣和資助下，UML 正在逐漸成為**工業標準**。
5. **建模**的工作並非只有畫圖而已，還包含了**文件** (例如以文字紀錄 use case)。

2. 軟體架構師的職責、挑戰、角色關係，與設計師、分析師的區別

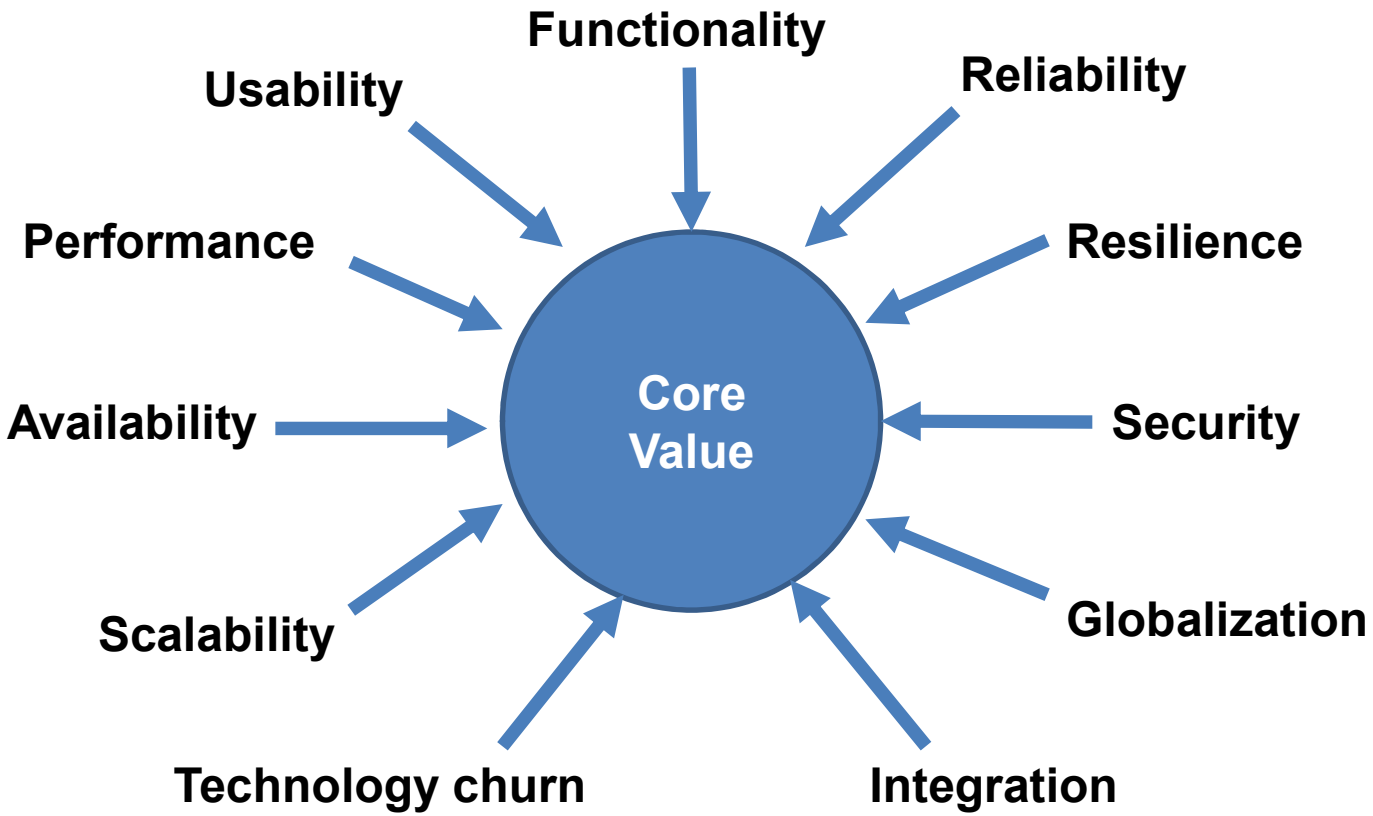
軟體架構師的職責

Responsible for software architecture, which includes the key technical decisions that constrain the overall design and implementation for the project



來源：http://sce.uhcl.edu/helm/rationalunifiedprocess/process/workers/wk_archt.htm

To build a mission critical or high value system, these challenges will follow you.

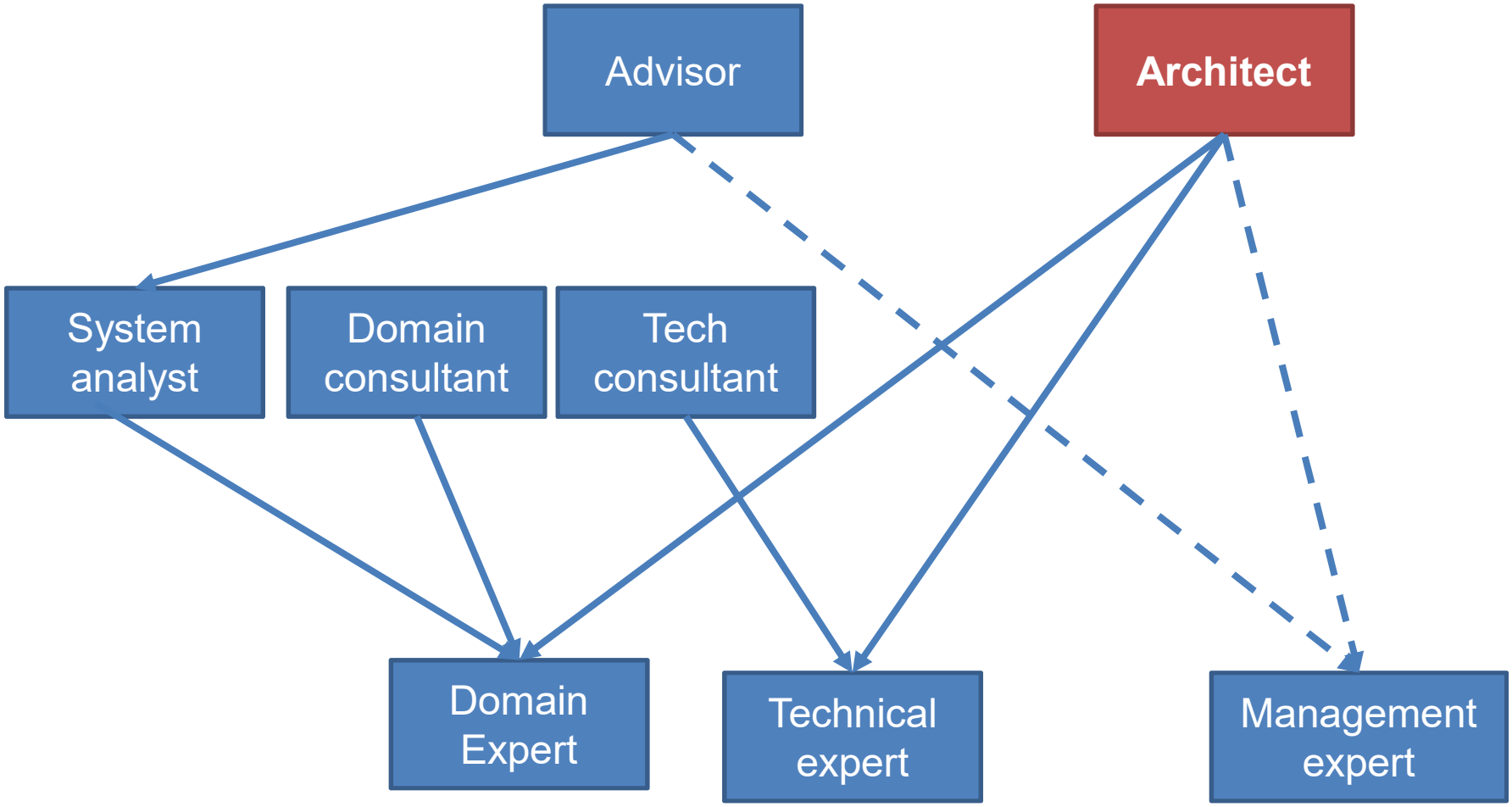


RUP Role Sets

- **Analysts**
  - System Analyst
  - Business Designer
  - Business-Model Reviewer
  - Business-Process Analyst
  - Requirements Reviewer
  - Requirements Specifier
  - Test Analyst
  - User-Interface Designer
- **Developers**
  - Capsule Designer
  - Code Reviewer
  - Database Designer
  - Implementer
  - Integrator
  - **Software Architect**
  - Architecture Reviewer
  - Design Reviewer
  - Designer
  - Test Designer
- **Tester**
  - Testers
- **Managers**
  - Process Engineer
  - Project Manager
  - Change Control Manager
  - Configuration Manager
  - Deployment Manager
  - Project Reviewer
  - Test Manager
- **Additional Role Set**
  - Stakeholder
  - Any Role
  - Course Developer
  - Graphic Artist
  - Tool Specialist
  - System Administrator
  - Technical Writer

PIC

軟體架構師與其他角色的關係



必須至少專精一項以上的技術

- Communication (Codec)
- Media processing (Codec)
- Image recognition
- Voice recognition
- 3D Graphics/Animation/Visual Effect
- Data Mining
- Natural Language Processing
- Parallel processing
- …

什麼是「軟體架構」？

“Software architecture encompasses the set of significant decisions about the organization of a software system” - Grady Booch (UML創始人).

- Structural elements and their interfaces
- Behavioral collaborations among those elements
- Organization of these structural and behavioral elements
- Architectural style
- Key technologies

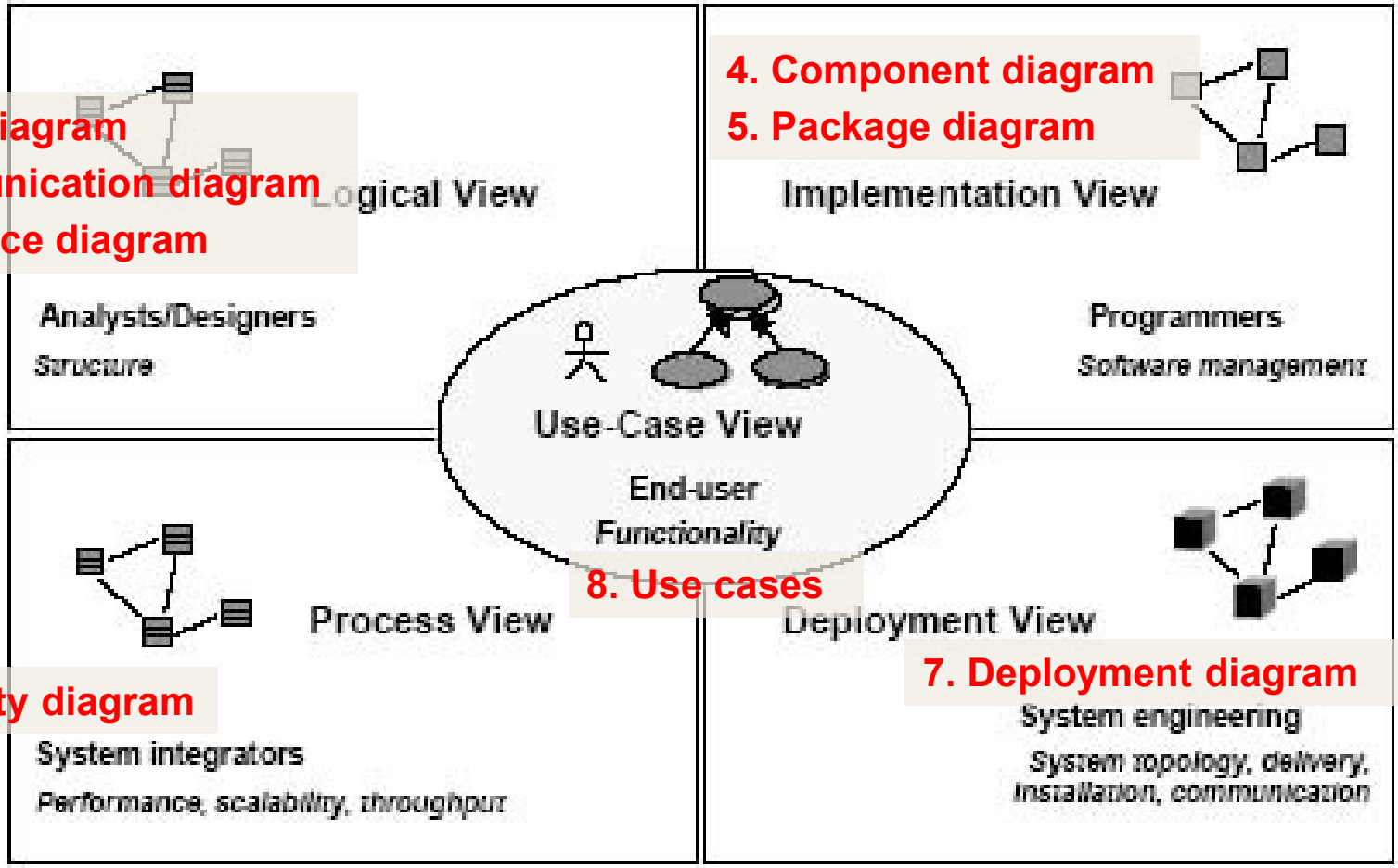
Grady Booch

Characteristics of Good Architecture

- Simple
  - Clear **Separation of Concerns** (SoC)
- Approachable
- Resilient
- Balanced distribution of responsibilities
- Balances economic and technology constraints

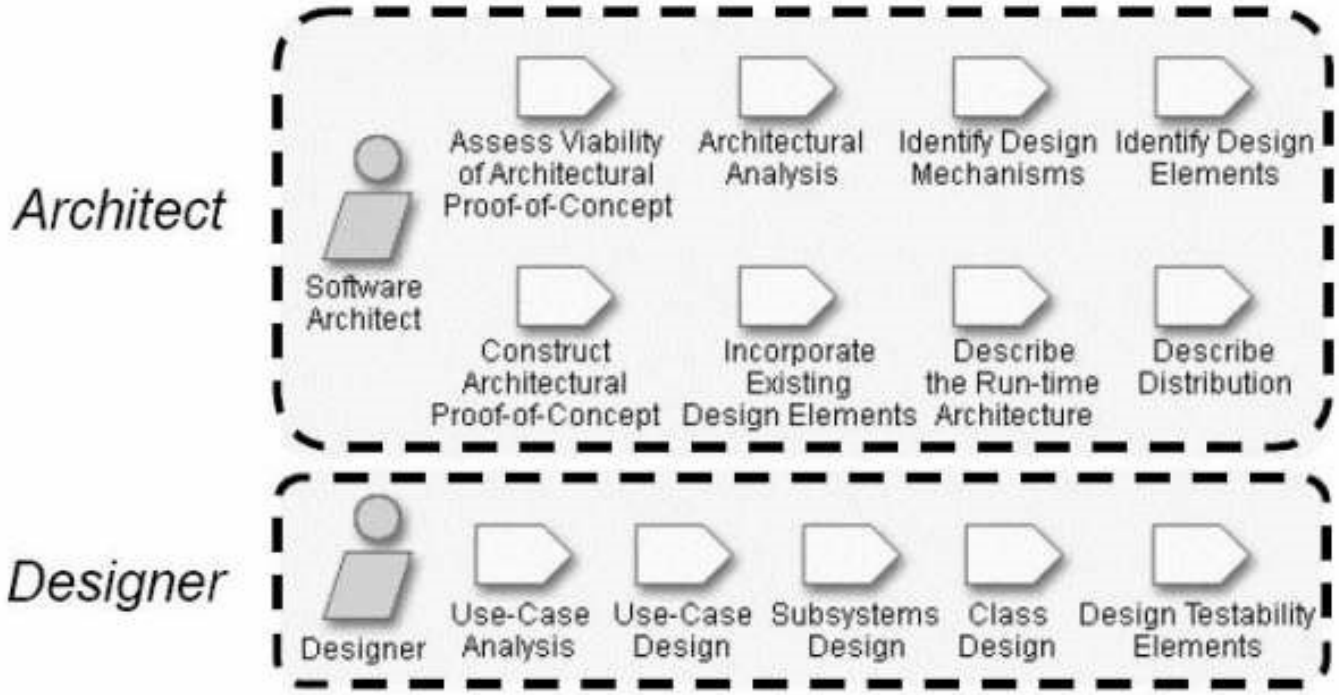
PIC

Software Architecture: The "4+1 View" Model



軟體架構師 vs. 軟體設計師

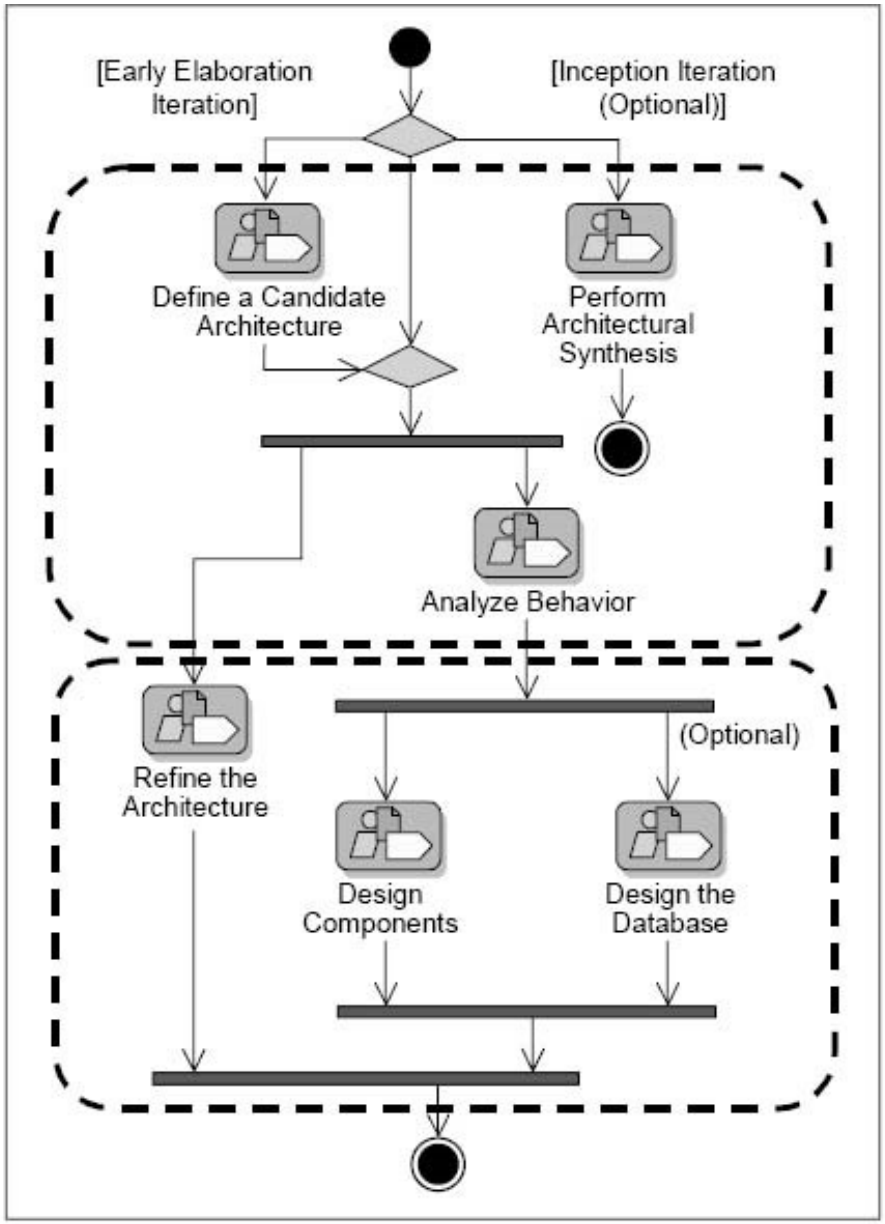
- Tends to be a generalist rather than a specialist, knowing many technologies at a high level rather than a few technologies at the detail level. (宏觀瞭解許多技術，但對於細節不是那麼深入)
- Makes broader technical decisions, and therefore broad knowledge and experience, as well as communication and leadership skills, are key. (因為必須制訂廣泛的技術決策，因此必須具備廣泛的知識、經驗、溝通技巧以及領導能力)



分析 vs. 設計

Analysis 主要是了解問題所在點，並清楚描述問題。

Analysis



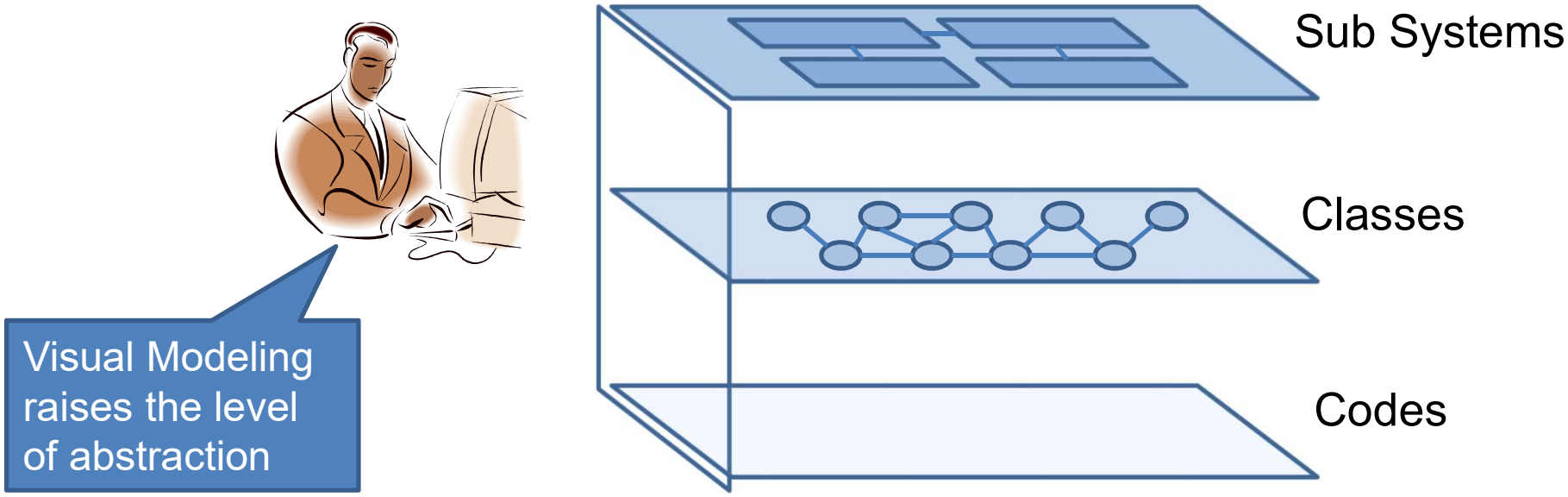
Design

Design 著重於問題解決方法

- Applying Levels of Abstraction iteratively
  - Have mastered the ability to conceptualize a solution at distinct levels of abstraction.
  - Attack various aspect one by one.
  - Continuously evolving and refining the layers into cohesive model that can ultimately be implemented.
- Construct multiple views within each level.
- Maintain consistency.
- Know detail implementations.

Levels of Abstraction

- Detect and assess architectural changes.
- Communicate accepted architectural changes.
- Synchronize your model and source code during every iteration.



To Be A Successful Architect

- **Expertise**
  - In the problem domain and the software engineering
- **Leadership**
  - To earn trust, to motivate, and to drive the technical effort across the various teams
- **Communication**
  - To persuade, and to mentor team members
- **Goal-orientation and Pro-activity (主動積極)**
  - Focus on results

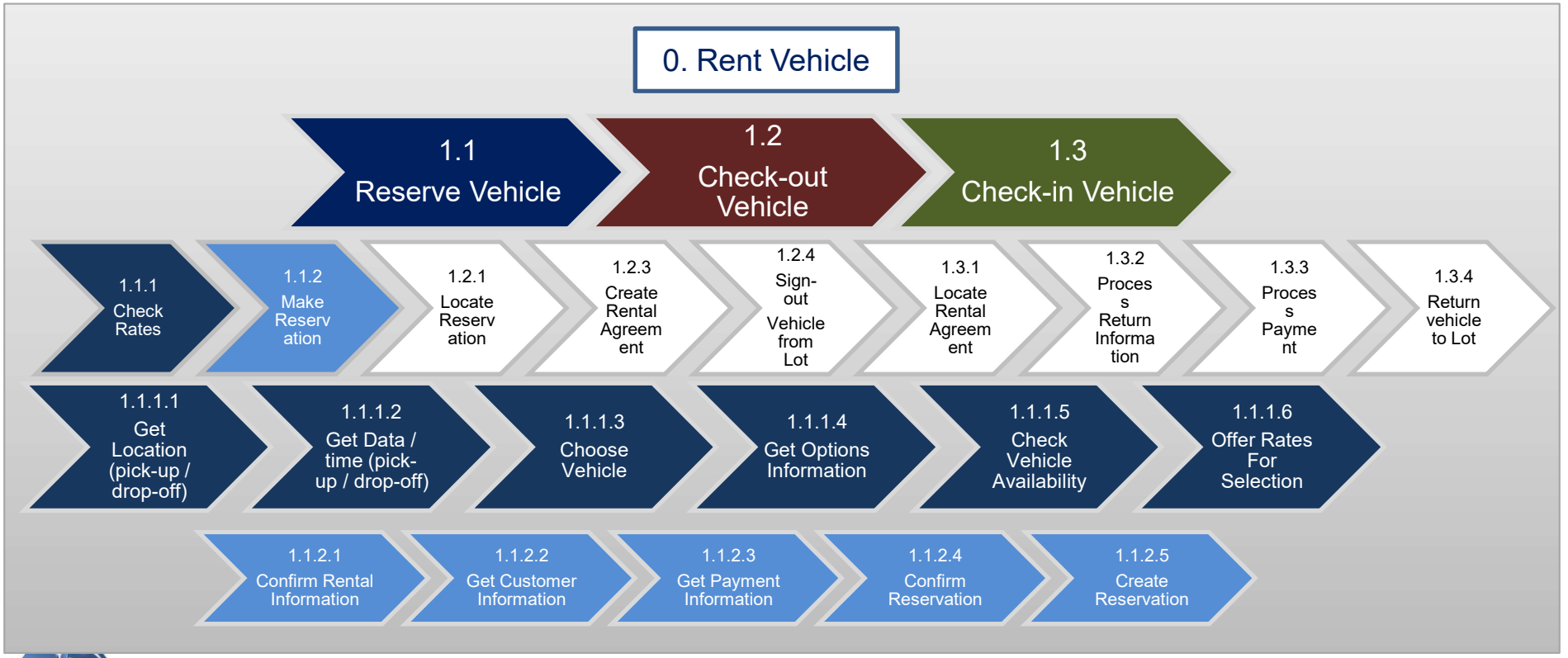
PIC

3. 商業塑模 – BUSINESS MODELING

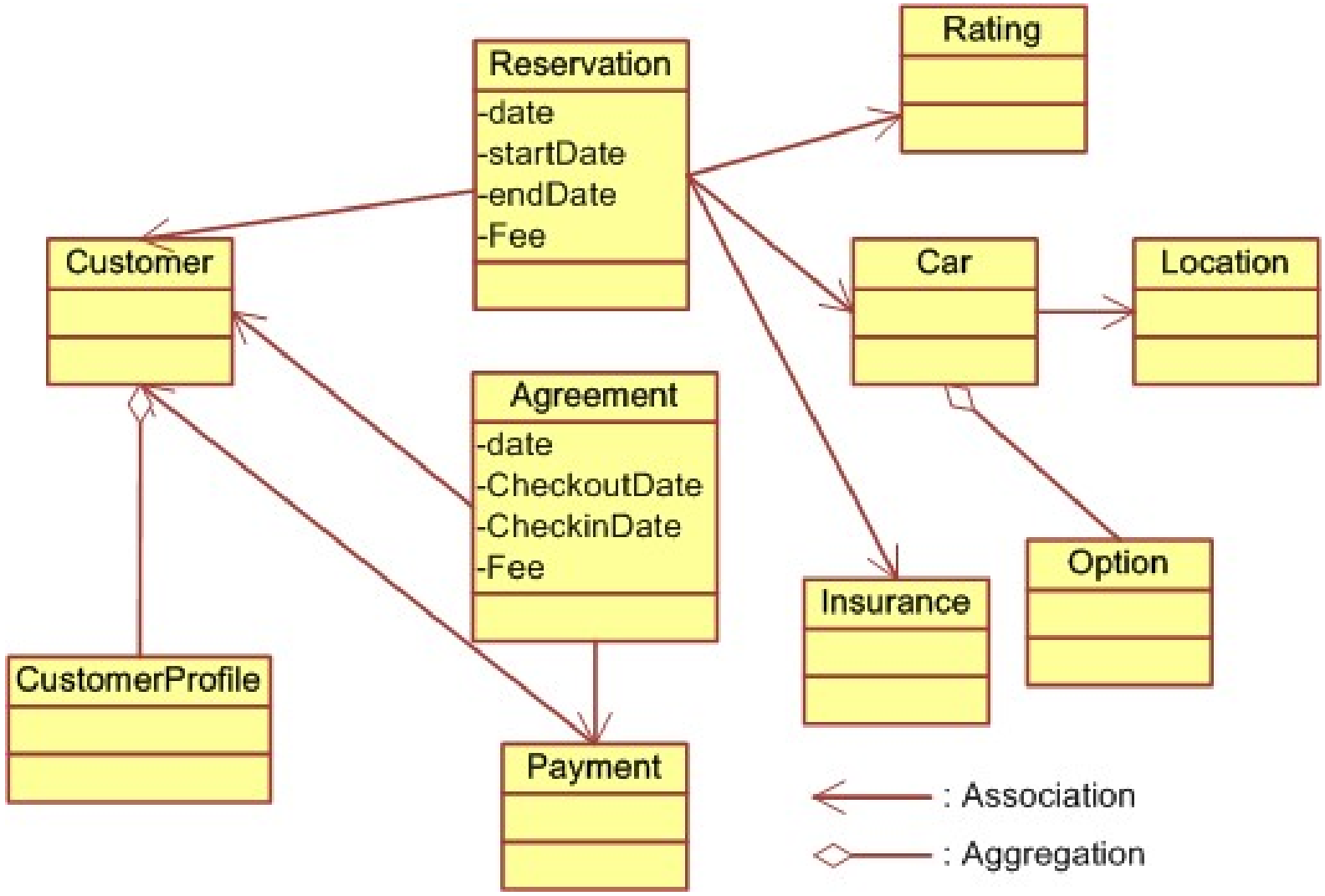
PIC

- Process model
- Object model
- Patterns of business object

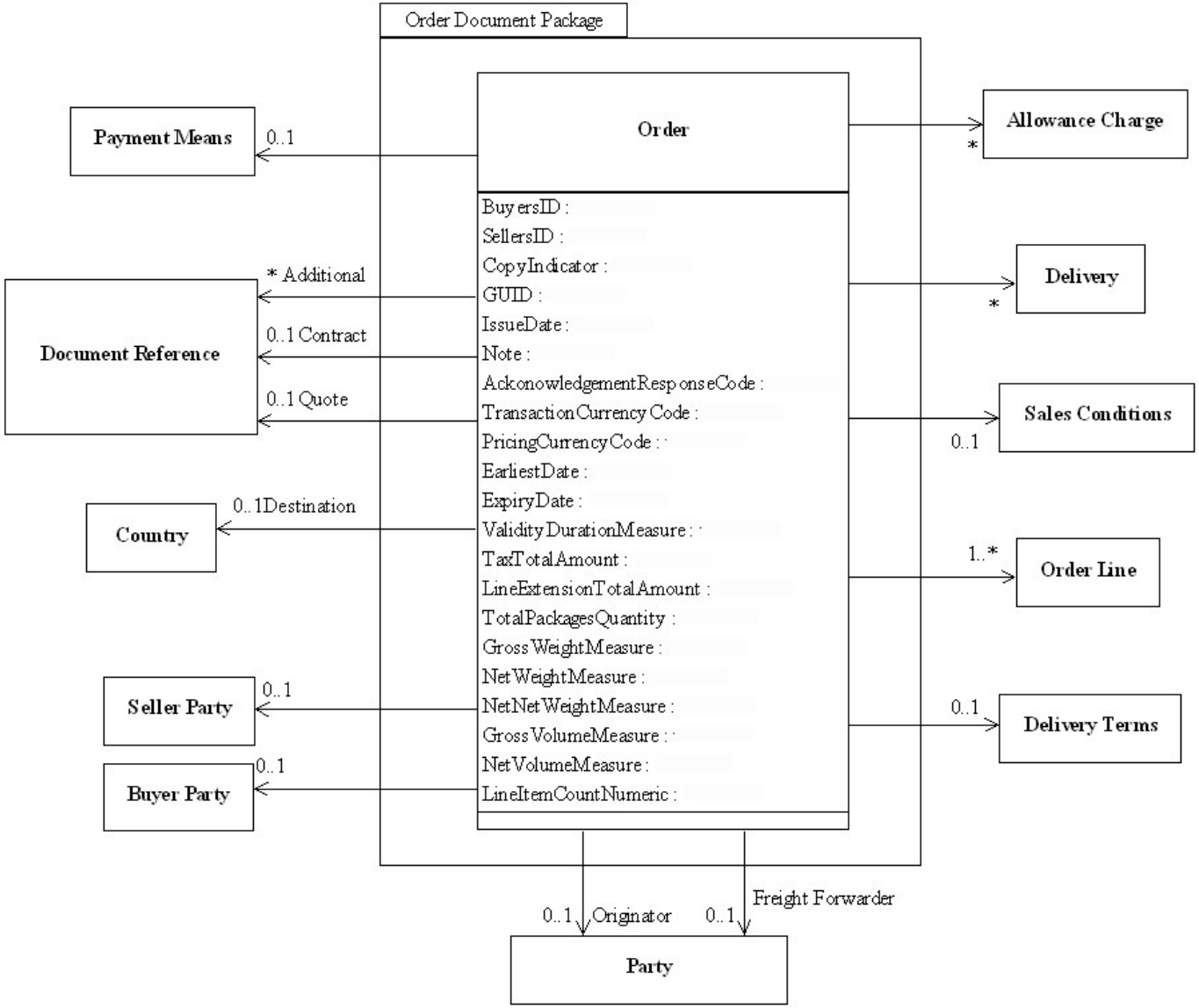
Rent Vehicle
Business Process



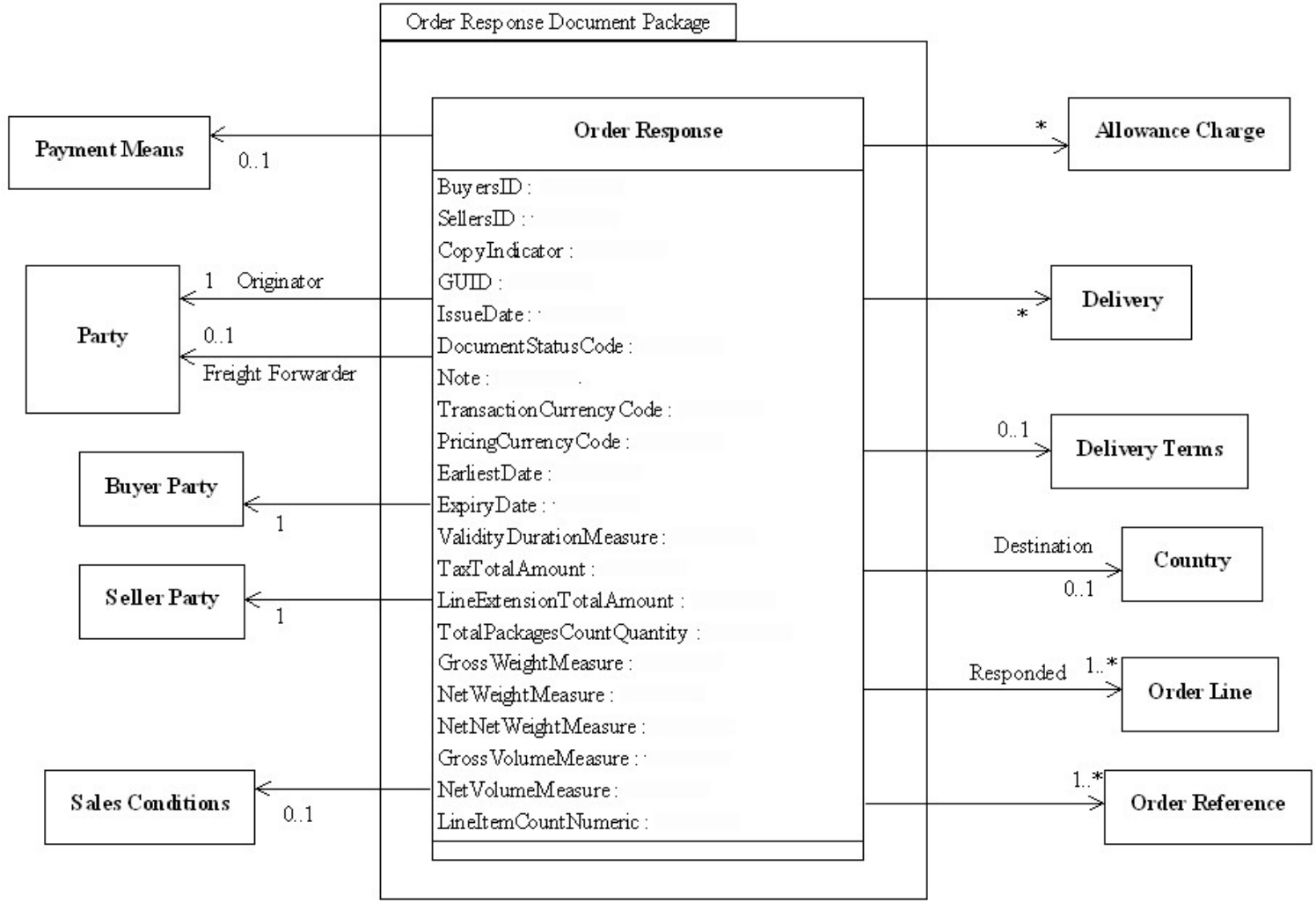
Object of Car Rental



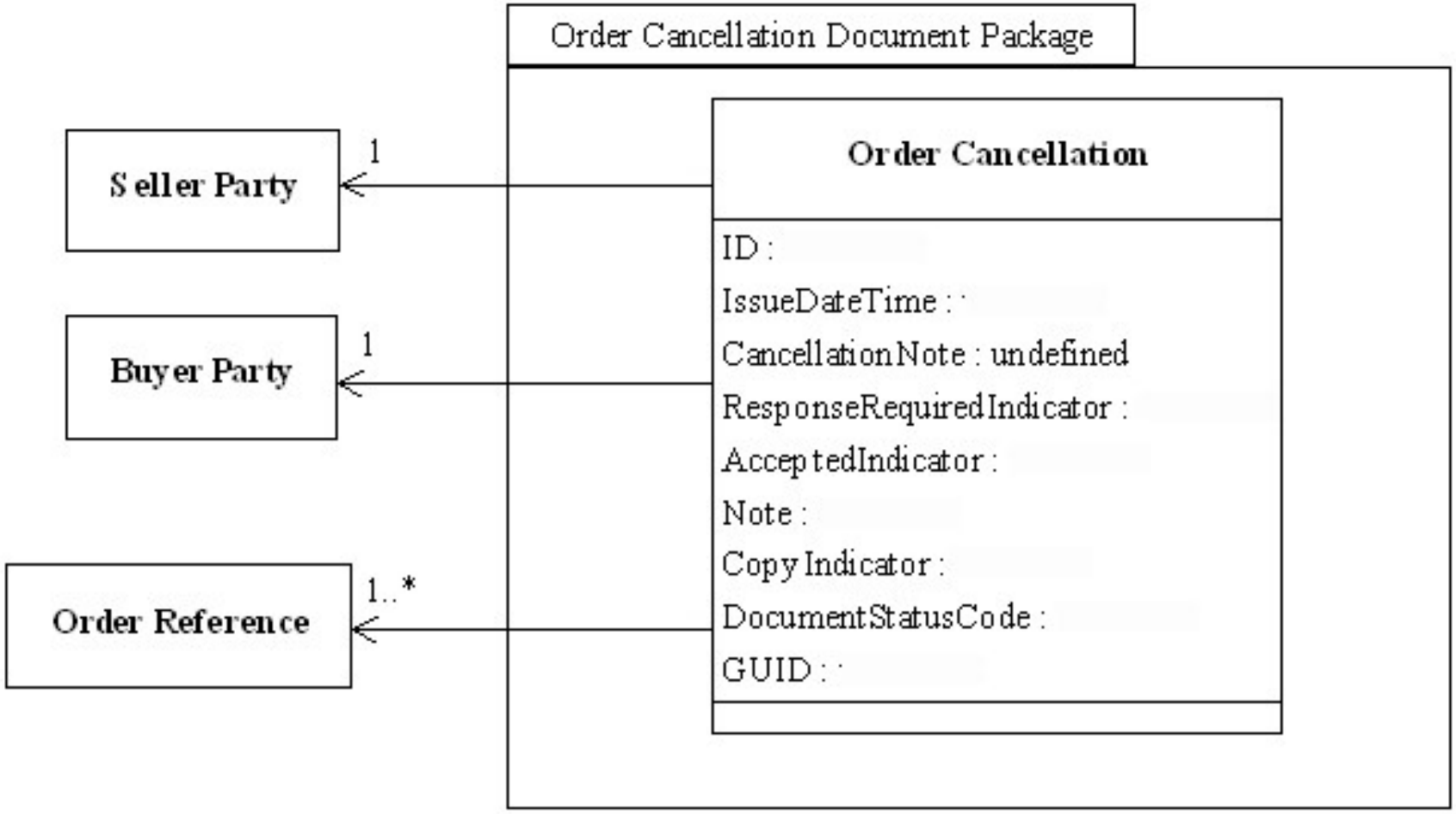
Order – from OASIS Universal Business Language (UBL 1.0)



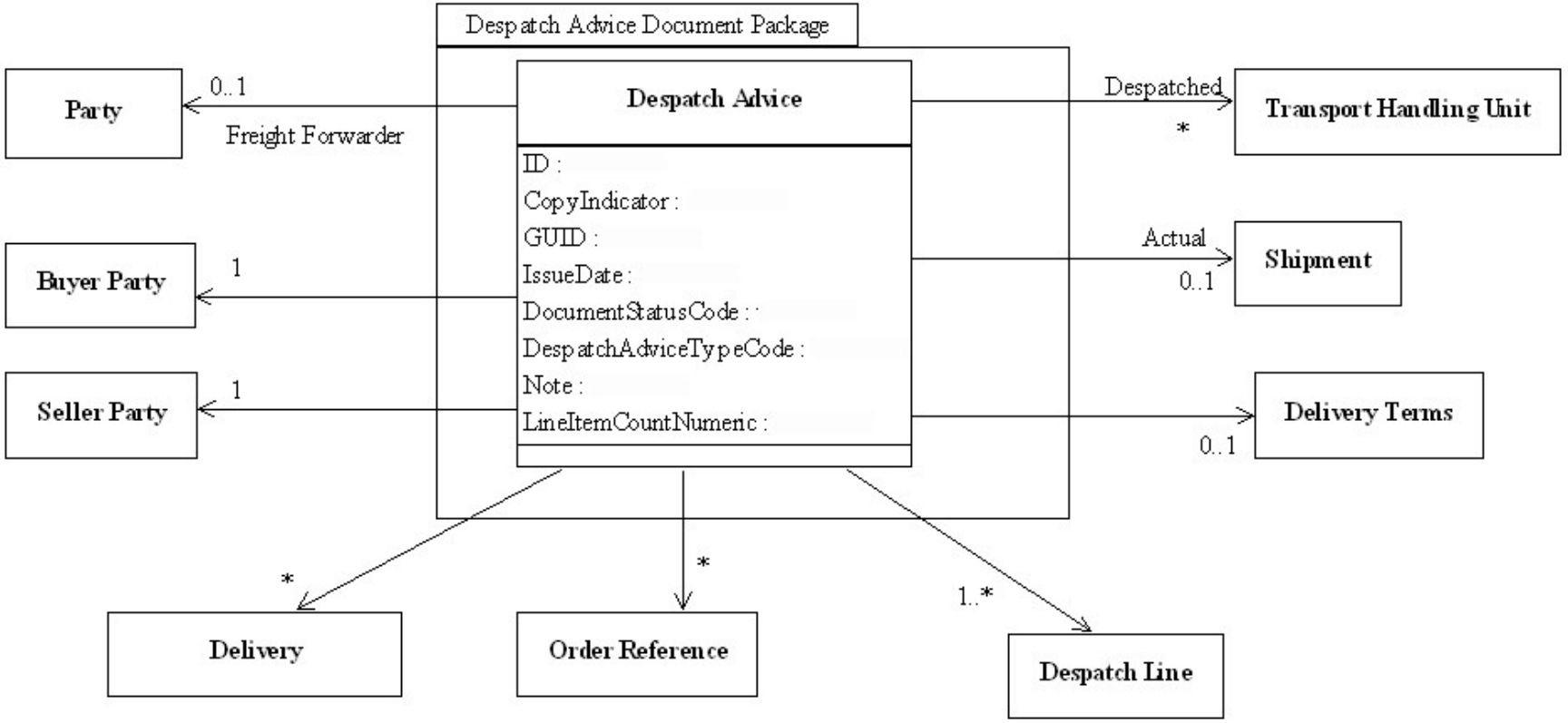
Order Response



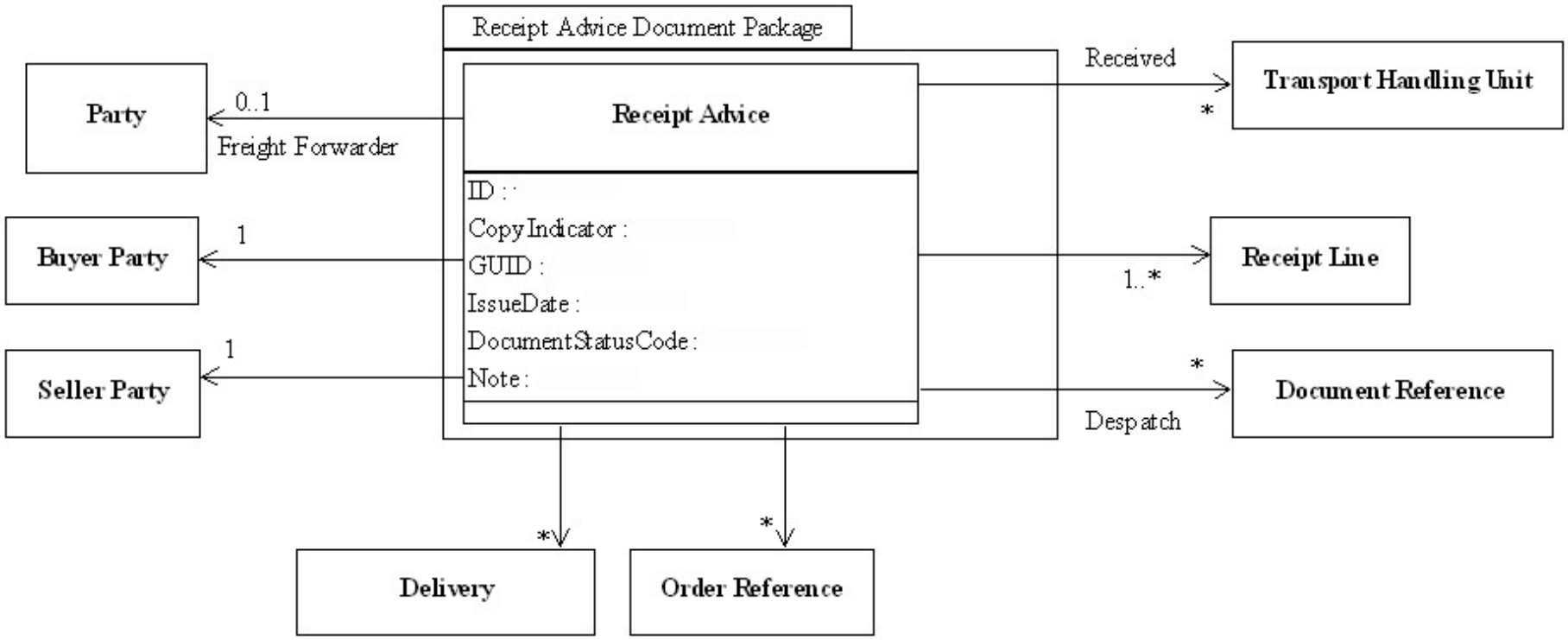
Order Response



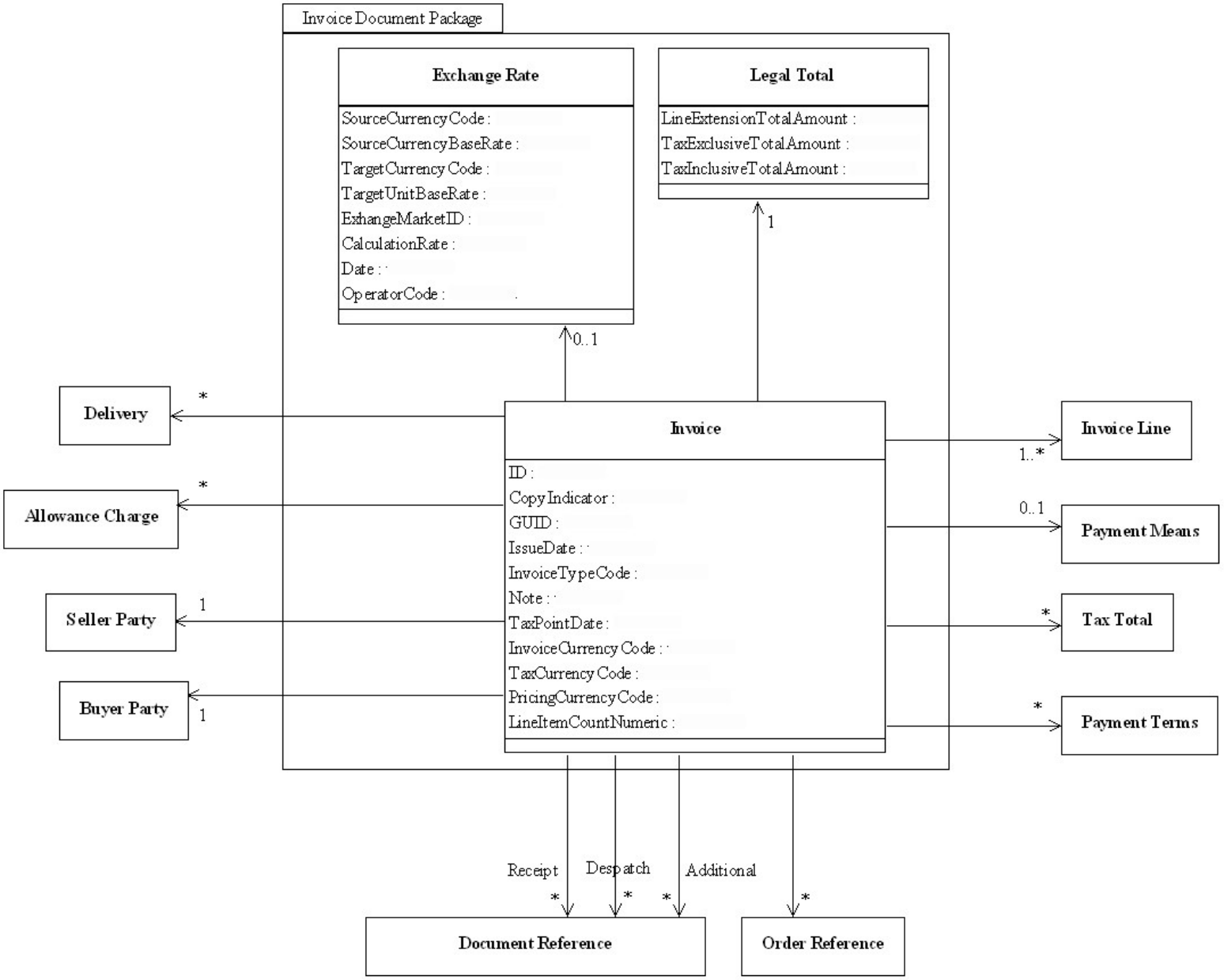
Despatch Advice



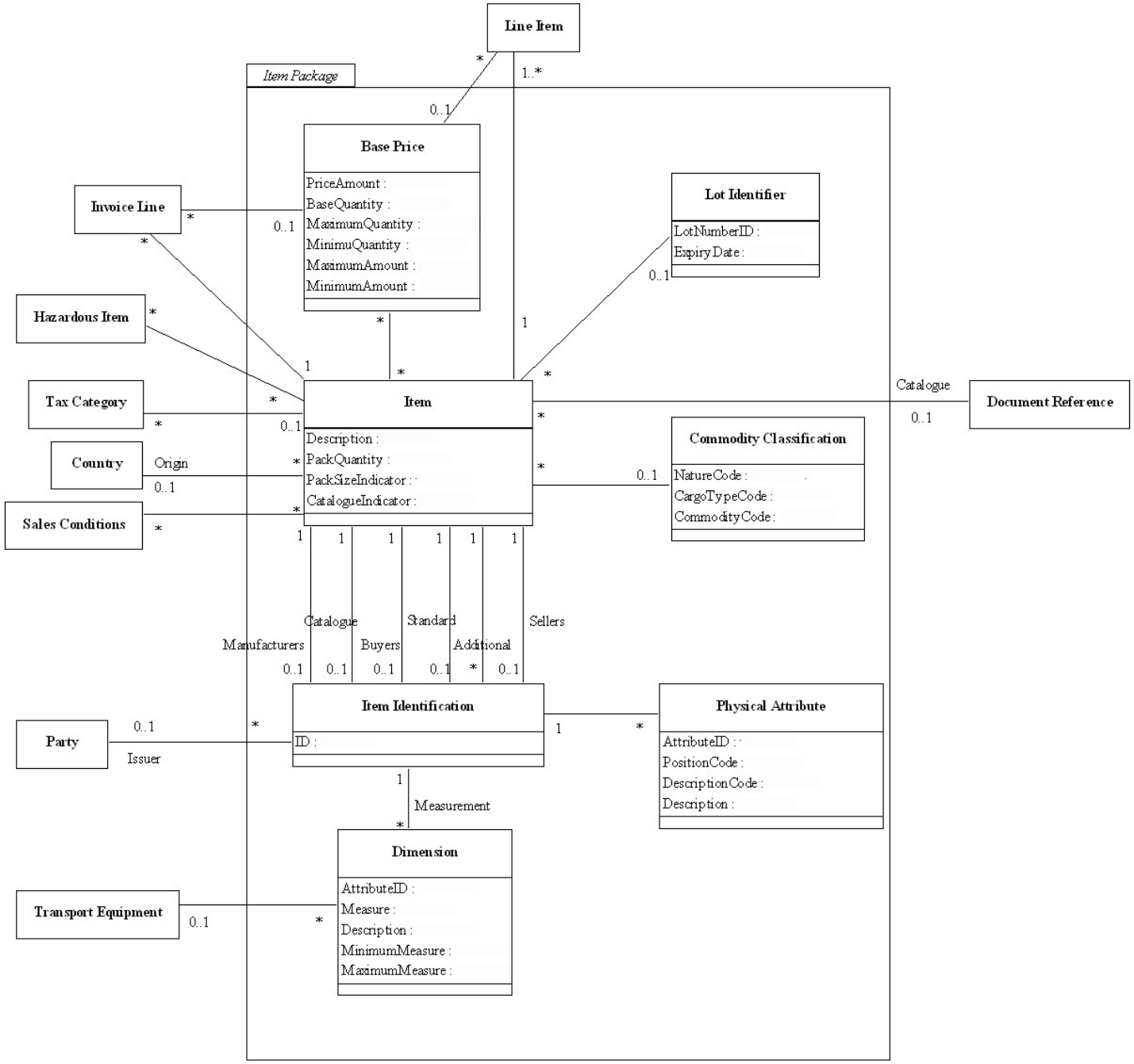
Receipt Advice



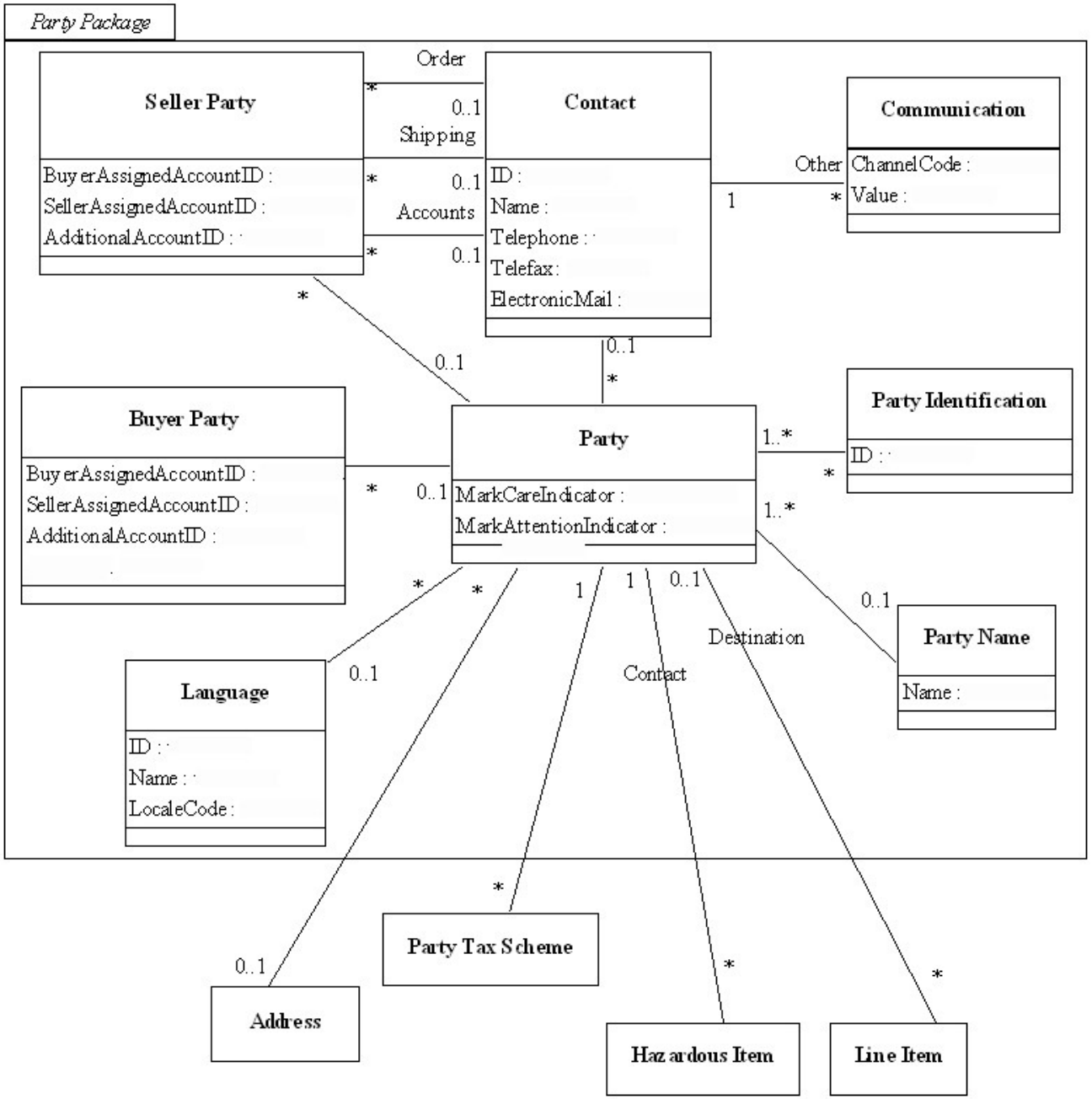
Invoice



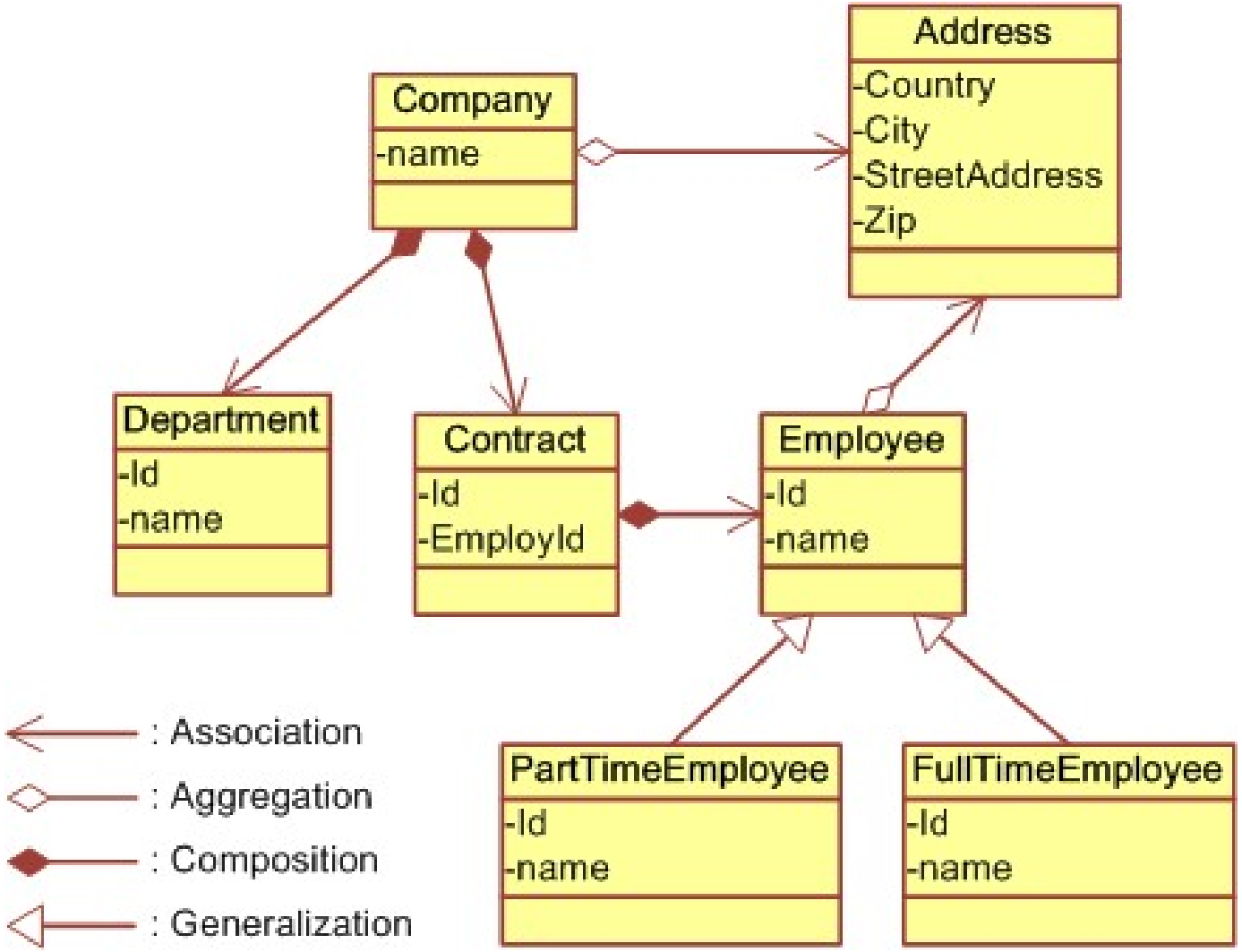
Item


Line Item
Item Package
Base Price
PriceAmount :
BaseQuantity :
MaximumQuantity :
MinimumQuantity :
MaximumAmount :
MinimumAmount :
Invoice Line
Lot Identifier
LotNumberID :
ExpiryDate :
Hazardous Item
Tax Category
Item
Description :
PackQuantity :
PackSizeIndicator :
CatalogueIndicator :
Catalogue
Document Reference
Country
Origin
Commodity Classification
NatureCode :
CargoTypeCode :
CommodityCode :
Sales Conditions
Manufacturers
Catalogue
Buyers
Standard
Additional
Sellers
Item Identification
ID :
Physical Attribute
AttributeID :
PositionCode :
DescriptionCode :
Description :
Party
Issuer
Measurement
Dimension
AttributeID :
Measure :
Description :
MinimumMeasure :
MaximumMeasure :
Transport Equipment

Party

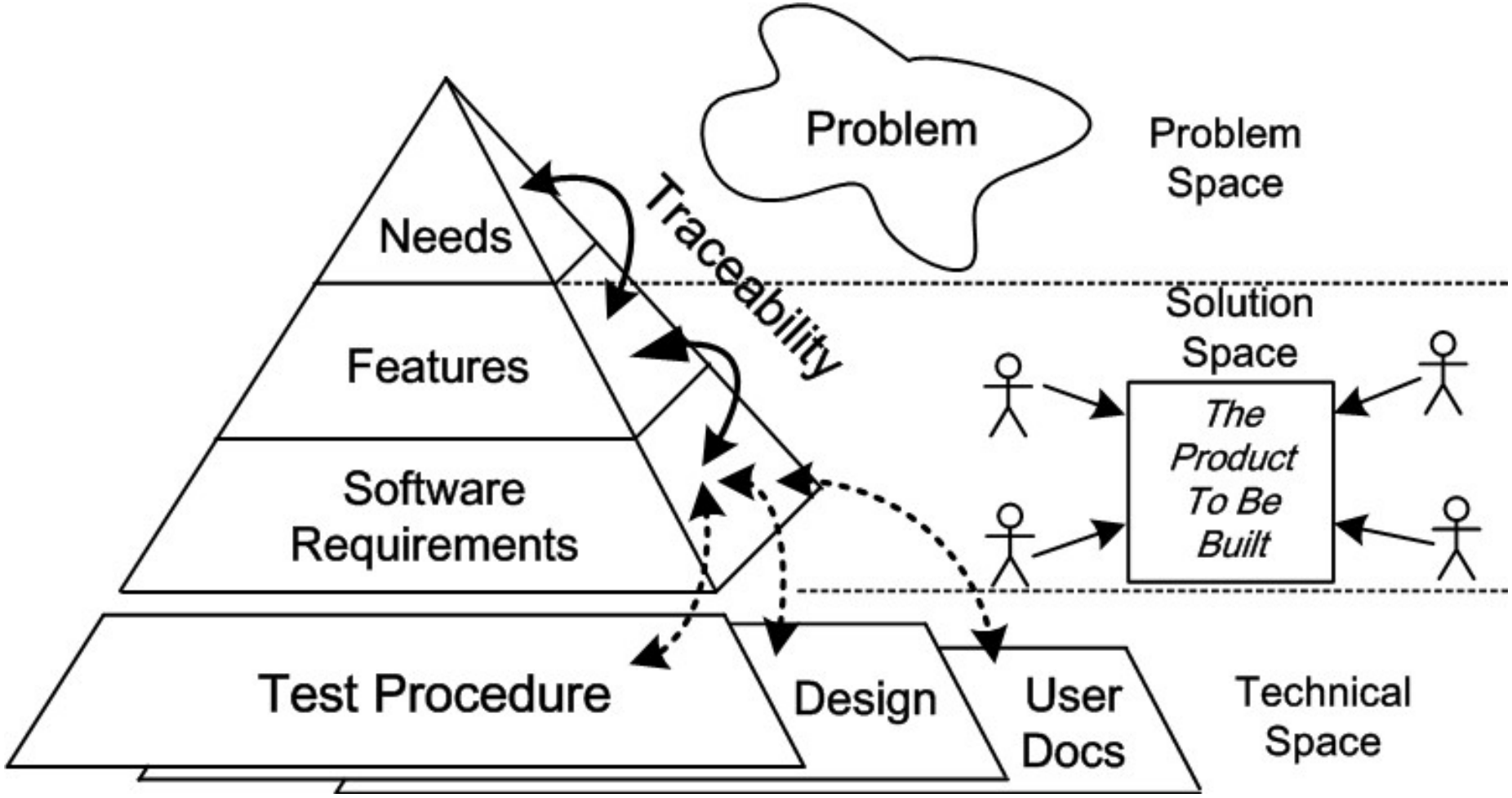

Party Package
Seller Party
BuyerAssignedAccountID :
SellerAssignedAccountID :
AdditionalAccountID :
Order
0..1
Shipping
0..1
Accounts
0..1
Contact
ID :
Name :
Telephone :
Telefax :
ElectronicMail :
Other
1
*
Communication
ChannelCode :
Value :
0..1
*
0..1
*
Buyer Party
BuyerAssignedAccountID :
SellerAssignedAccountID :
AdditionalAccountID :
*
0..1
Party
MarkCareIndicator :
MarkAttentionIndicator :
1..*
*
Party Identification
ID :
1..*
*
0..1
Party Name
Name :
Destination
Contact
1
1
0..1
*
*
0..1
Language
ID :
Name :
LocaleCode :
*
0..1
Address
Party Tax Scheme
*
Hazardous Item
*
Line Item

Be careful of the meaning…



Levels of Requirements

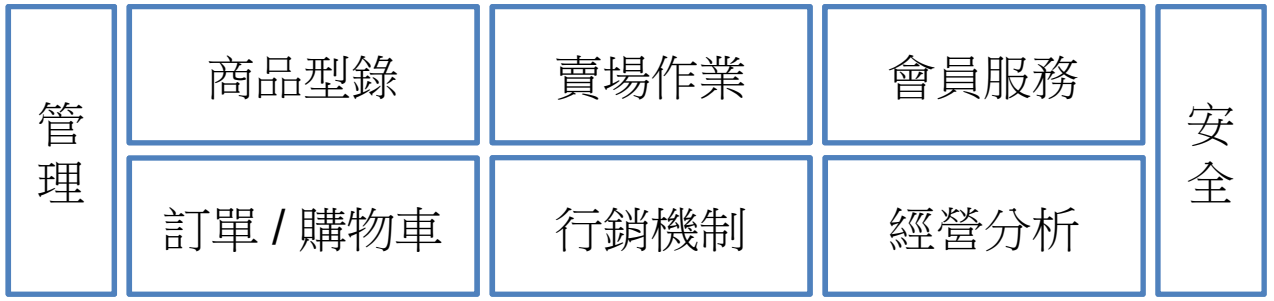
From IBM-Rational



參考：http://www.ibm.com/developerworks/rational/library/04/r-3217/

Features Sample – 電子商務服務

- 型錄
- 購物車 / 訂單
- 銷售管理(促銷、競標、團購)
- 行銷
- 會員/個人化服務
- 商業分析
- 安全、穩定可靠、容易使用、容易維運

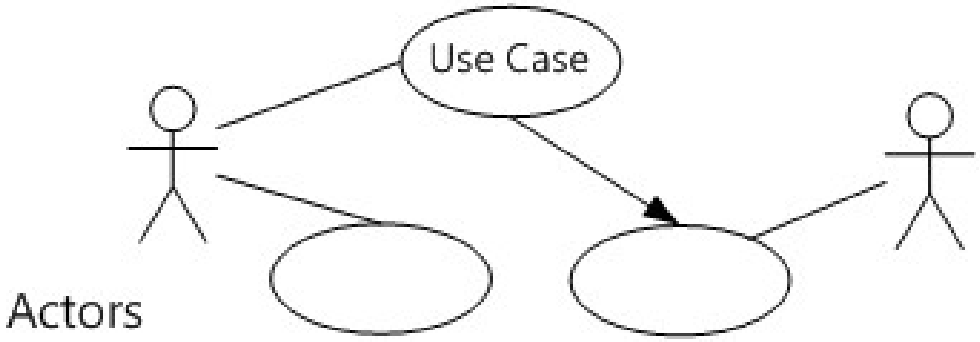


「訂單 / 購物車」Feature

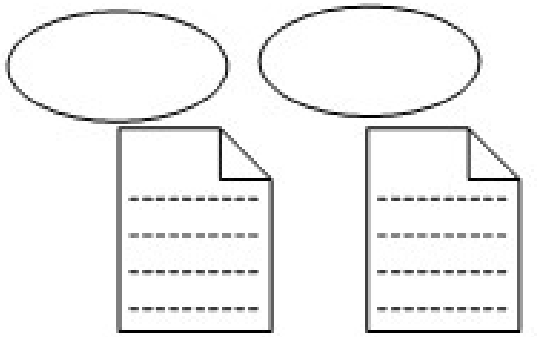
- 提供購物車存續功能
- 提供快速訂購服務
- 快速安全的購物平台
  - Tx <= 10 seconds
  - SSL
- 提供多種銷售機制的條件參數，可以依據商品及客戶等級，計算不同的商品售價、付款方式與送貨方式。
- 訂單轉送至後台進行後續的金流與物流。
- 提供 Web Services 接收友商的訂單，並轉送至後台。

Software Requirement - Use Case Model

- Actors
- Use cases
- Use case diagram
- Use case specification
- Organized by packages



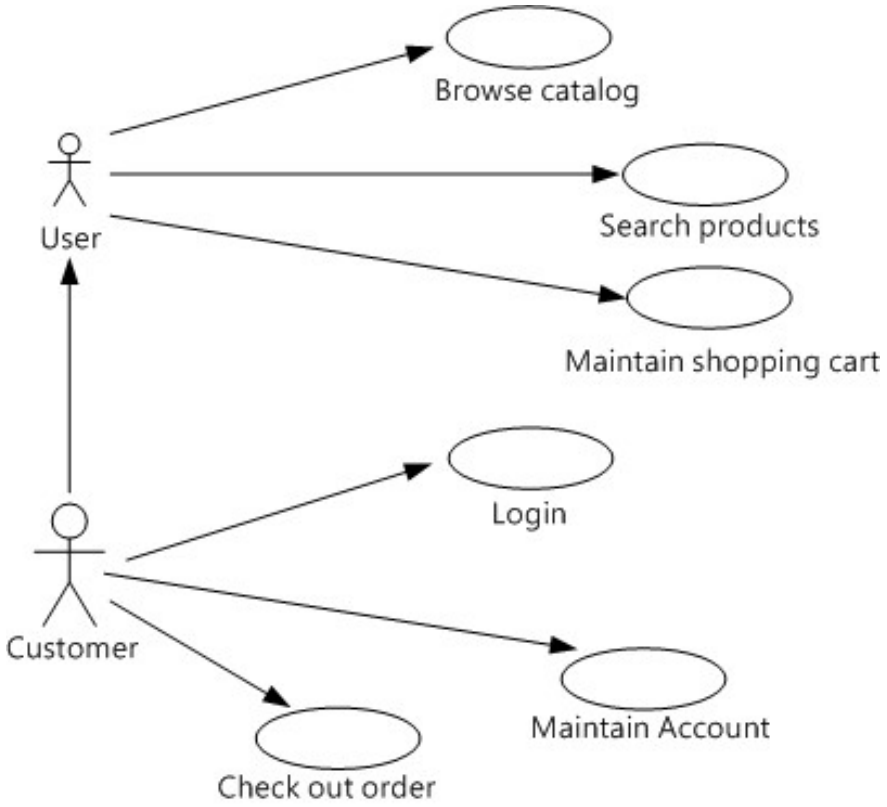
Use cases diagram



Use-case specification

Software Requirement - Use Case Diagram - eStore

- Use case diagram
  - provides the overall picture
- Use cases:
  - Browse catalog
  - Search products
  - Maintain shopping cart
  - Logon
  - Check out order



Software Requirement - Use Case Specification

2. Basic Flow

使用者在維護購物車畫面中選擇 Checkout

1. 系統顯示購物金額
   a) 每個項目金額，稅及總額
   b) 提醒使用者，運費需待輸入 shipping details 後才會計算出
2. 使用者確認採購項目金額
3. 使用者輸入送貨資料，ref 4.1-輸入送貨資料
4. 使用者輸入付款資訊，ref 4.2-輸入付款資料
5. 系統顯示訂單明細及總金額
6. 使用者確認交易
7. 系統處理訂單 (如果交易失敗，進入流程 3.3-交易失敗)
8. 系統顯示交易完成訊息，訂單編號及交貨訊息

3. Alternative Flows

3.1 送貨資料錯誤

1. 系統回到輸入送貨資料流程 ，並顯示失敗訊息
   a) 使用紅色*提示錯誤的欄位
   b) 保留使用者輸入的資料

3.2 付款資料錯誤

2. 系統回到輸入付資資料流程 ，並顯示失敗訊息
   c) 使用紅色*提示錯誤的欄位
   d) 保留使用者輸入的資料

3.3 交易失敗

1. 輸入的資料沒通過檢查，系統顯示錯誤訊息，並讓使用者修正輸入的資料

4. Subflows

4.1 輸入送貨資料

1. 系統顯示送貨資料(shipping details)的輸入畫面
2. 使用者輸入下列送貨資料
   A. 收件人名稱
   B. 地址
   C. 連絡電話
   D. 送貨方式: 緊急空運，正常空運，海運
3. 系統檢查送貨資料，(如果輸入錯誤，進入流程 3.1-送貨資料錯誤)
4. 系統顯示運費金額

4.2 輸入付款資訊

1. 系統顯示付款資訊輸入畫面
2. 使用者輸入下列付款資料
   A. 信用卡類別
   B. 名稱
   C. 卡號
   D. 有效期
3. 系統檢查付款資料(如果輸入錯誤，進入流程 3.2-付款資料錯誤)

Software Requirement - Detailed Use Case Specification

2. Basic Flow

使用者在維護購物車畫面中選擇 Checkout

1. 系統顯示購物金額
   a) 每個項目金額，稅及總額
2. 使用者輸入送貨資料, ref 4.1-輸入送貨資料
3. 使用者輸入付款資訊, ref 4.2-輸入付款資料
4. 系統計算包含運費、折扣、稅的總金額
   a) 允許接受其他系統傳來的訂單
5. 使用者確認交易
   a) 其他系統確認交易
6. 系統處理訂單 (如果交易失敗, 進入流程 3.3-交易失敗)
7. 系統下訂單給供應商, 請供應商出貨
8. 系統使用 email 通知客戶出貨資訊
9. 系統顯示交易完成訊息，訂單編號及交貨訊息

3. Alternative Flows

3.1 送貨資料錯誤

1. 系統回到輸入送貨資料流程，並顯示失敗訊息
   a) 使用紅色*提示錯誤的欄位
   b) 保留使用者輸入的資料

3.2 付款資料錯誤

2. 系統回到輸入付資資料流程，並顯示失敗訊息
   a) 使用紅色*提示錯誤的欄位
   b) 保留使用者輸入的資料

3.3 交易失敗

1. 輸入的資料沒通過檢查, 系統顯示錯誤訊息，並讓使用者修正輸入的資料

7. Post-Conditions

7.1 Web 系統清除購物車內容

7.2 Web 系統清除客戶的 Credit Card 資料

8. Extension Points

8.1 在 subflow 4.1 允許增加其他的送貨方式

8.2 在 subflow 4.2 允許增加其他的付款方式，並與 subflow 4.1 連動

8.3 在 basic flow 的步驟 4，允許不同情況的計算規則

9. Special Requirements

9.1 Performance

9.1.1 交易時間最長 10 seconds

9.2 Usability

9.2.1 提供快速訂購服務

9.2.2 顯示交易處理中的提示訊息

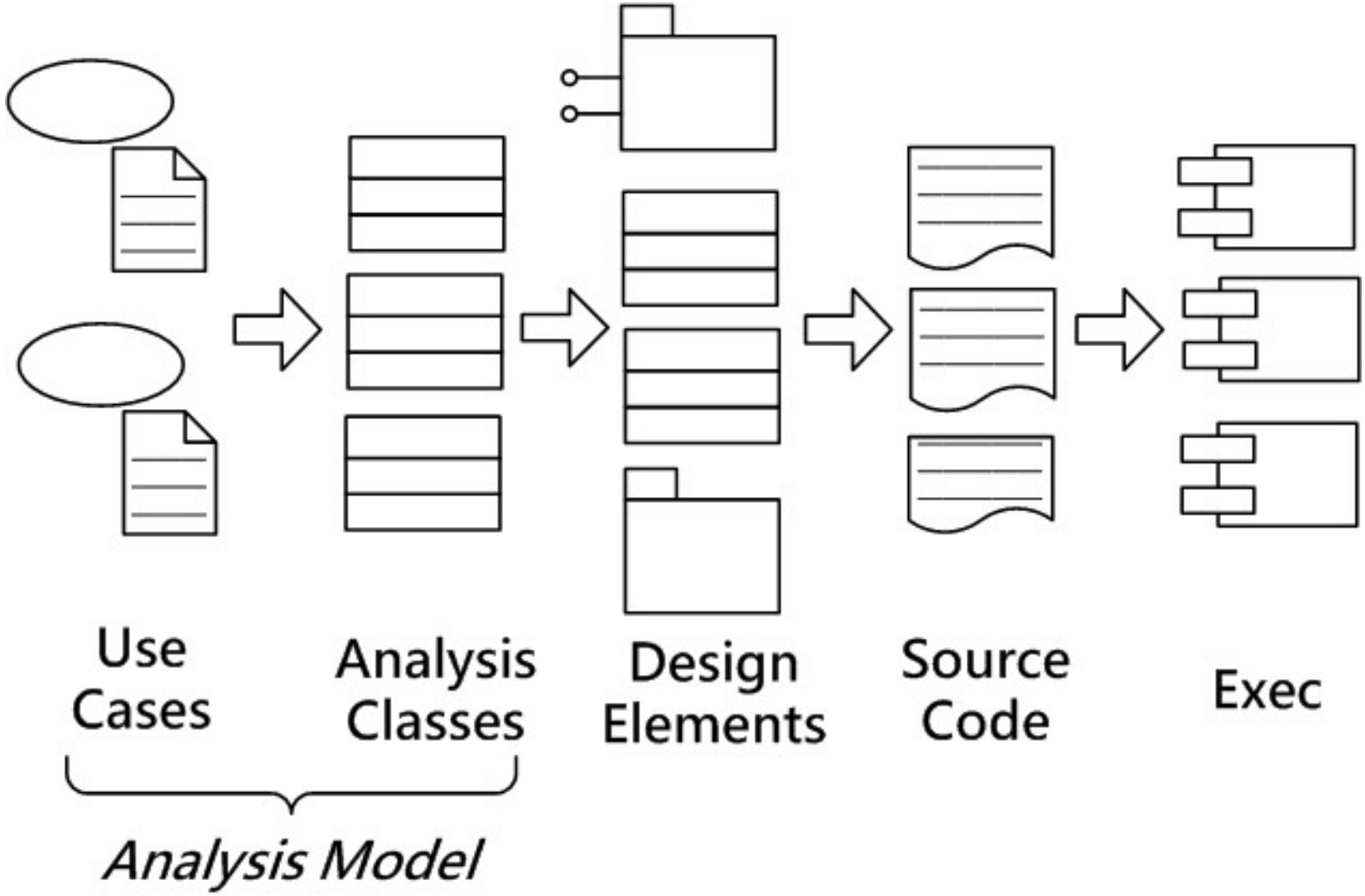
9.3 Security

9.3.1 使用 SSL 進行交易

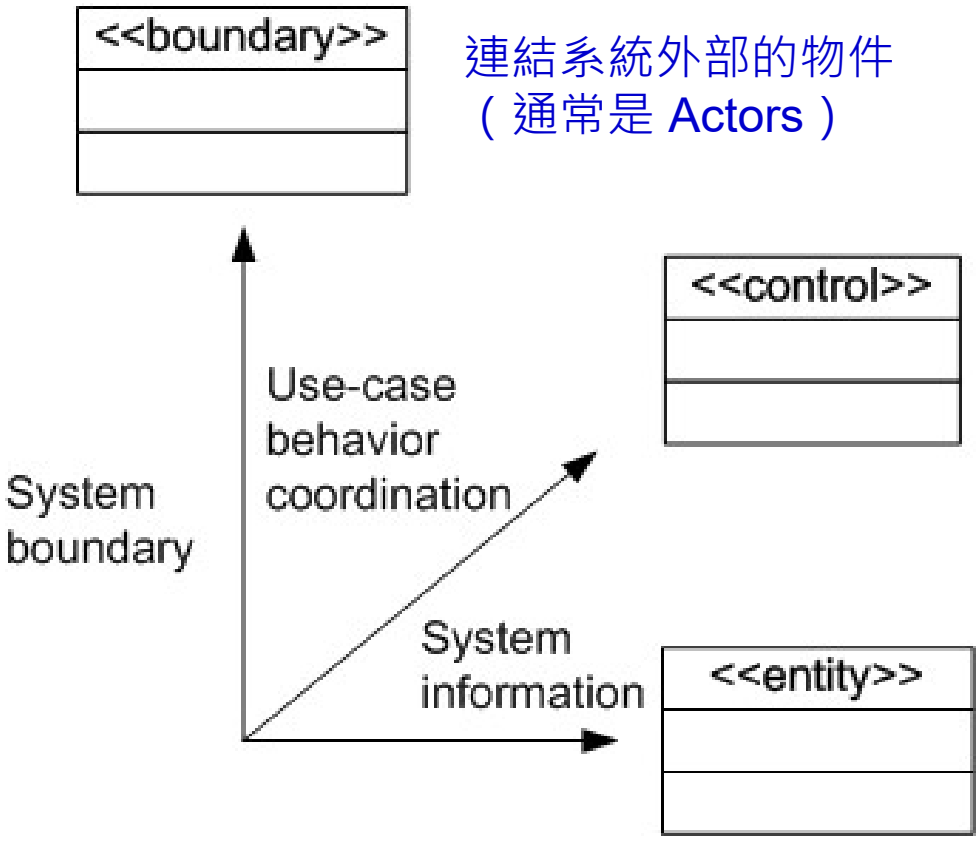
9.3.2 系統端於交易後不能保留客戶的信用卡資料

Non-functional Requirements。
由 Architect 填寫。

Analysis Model: A First Step Towards Executables



Analysis Class Stereotypes



連結系統外部的物件
（通常是 Actors）

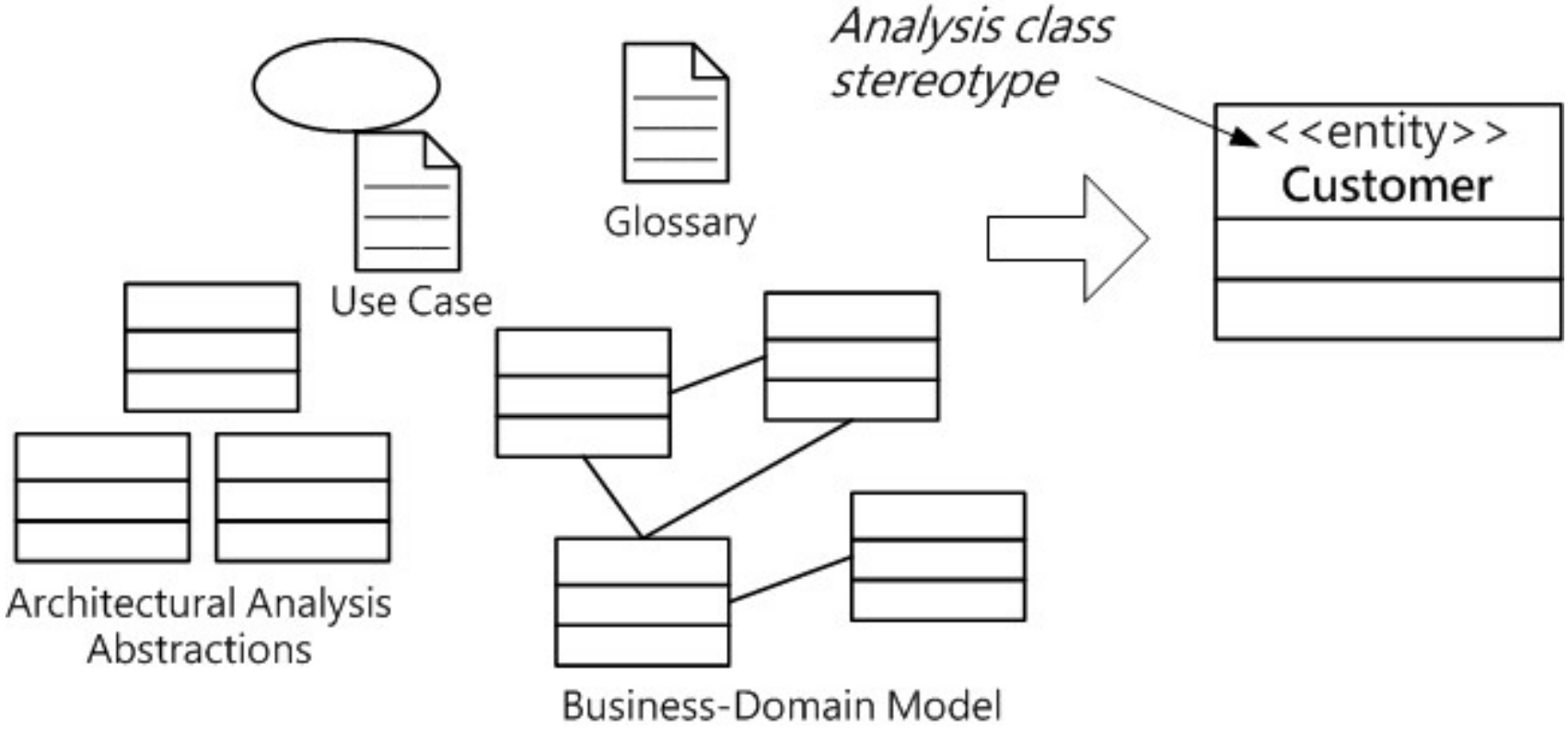
用來控制
1. Process (較長)
2. Workflow (較短)
的物件。

即 Business Object，如：會員、商品、訂單...等。

※ 建議所有使用外部 Library 的地方都建立 boundary class。

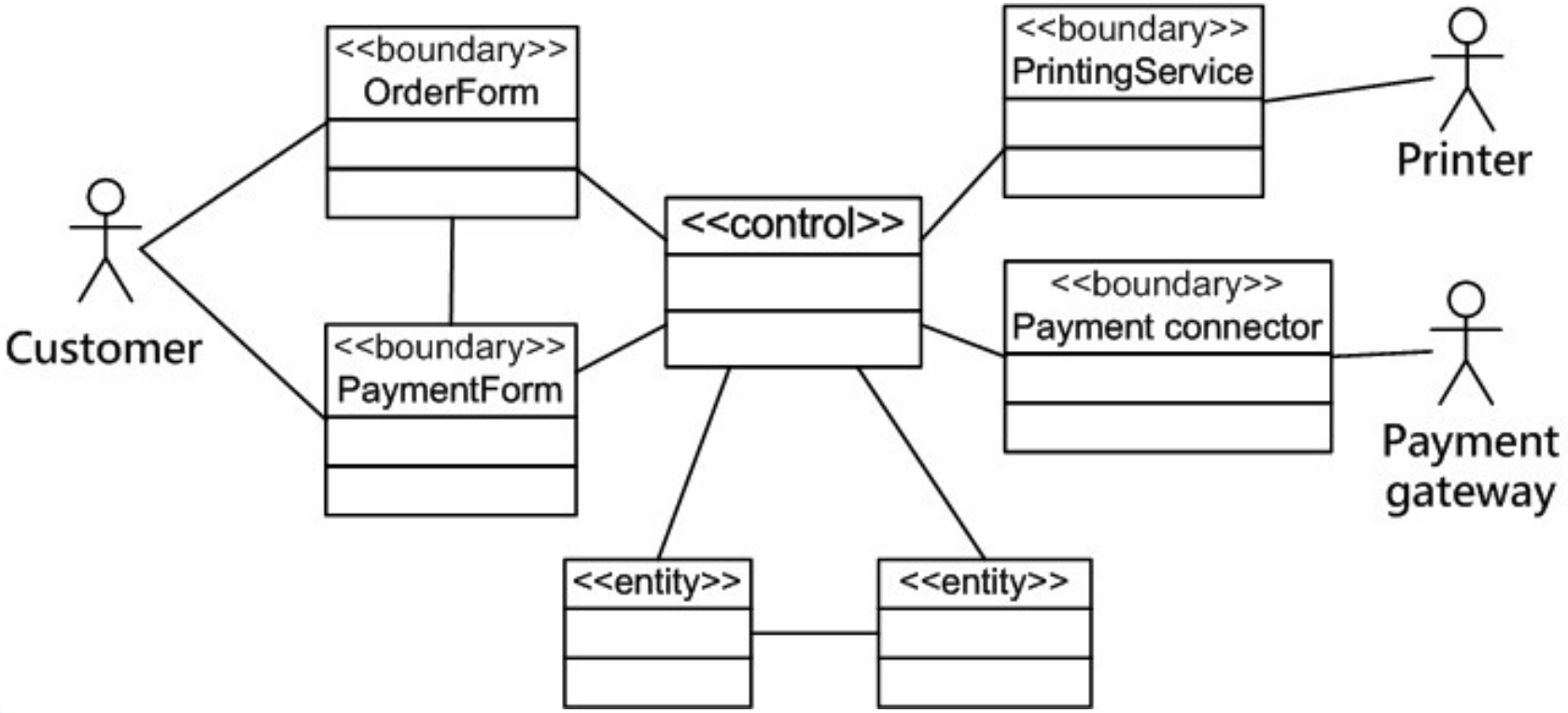
Entity Class

- Store and manage information in the system
- Responsible for realize use case's behaviors
- Such as customer, product, order, shopping cart



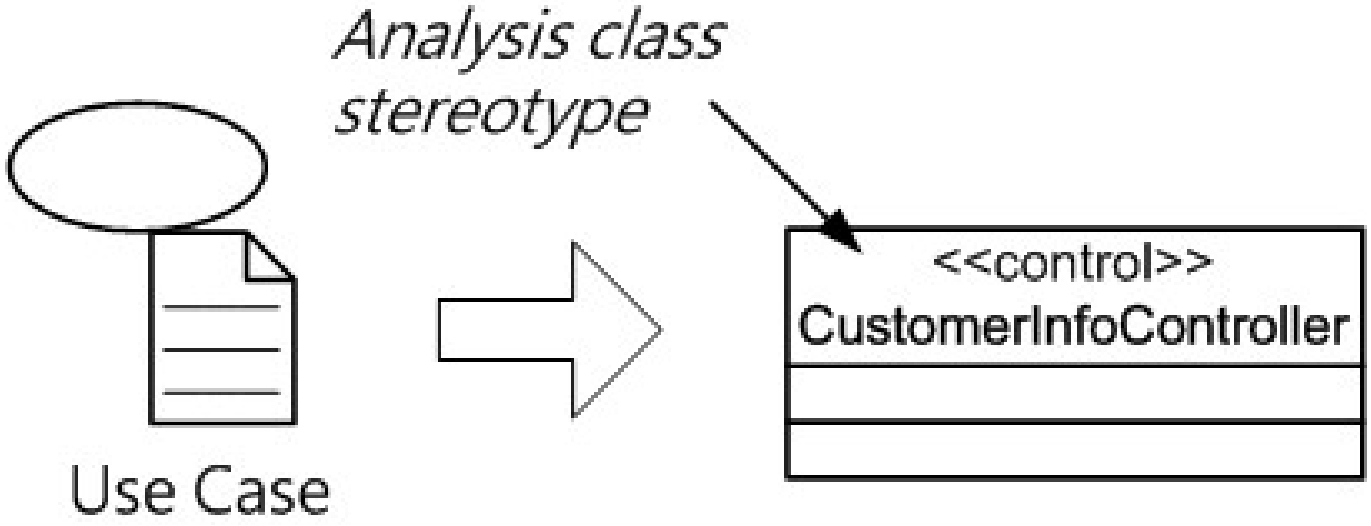
Boundary Class

- Boundary classes are interfaces between actors and the system, such as:
  User interface, device, library, other system interfaces

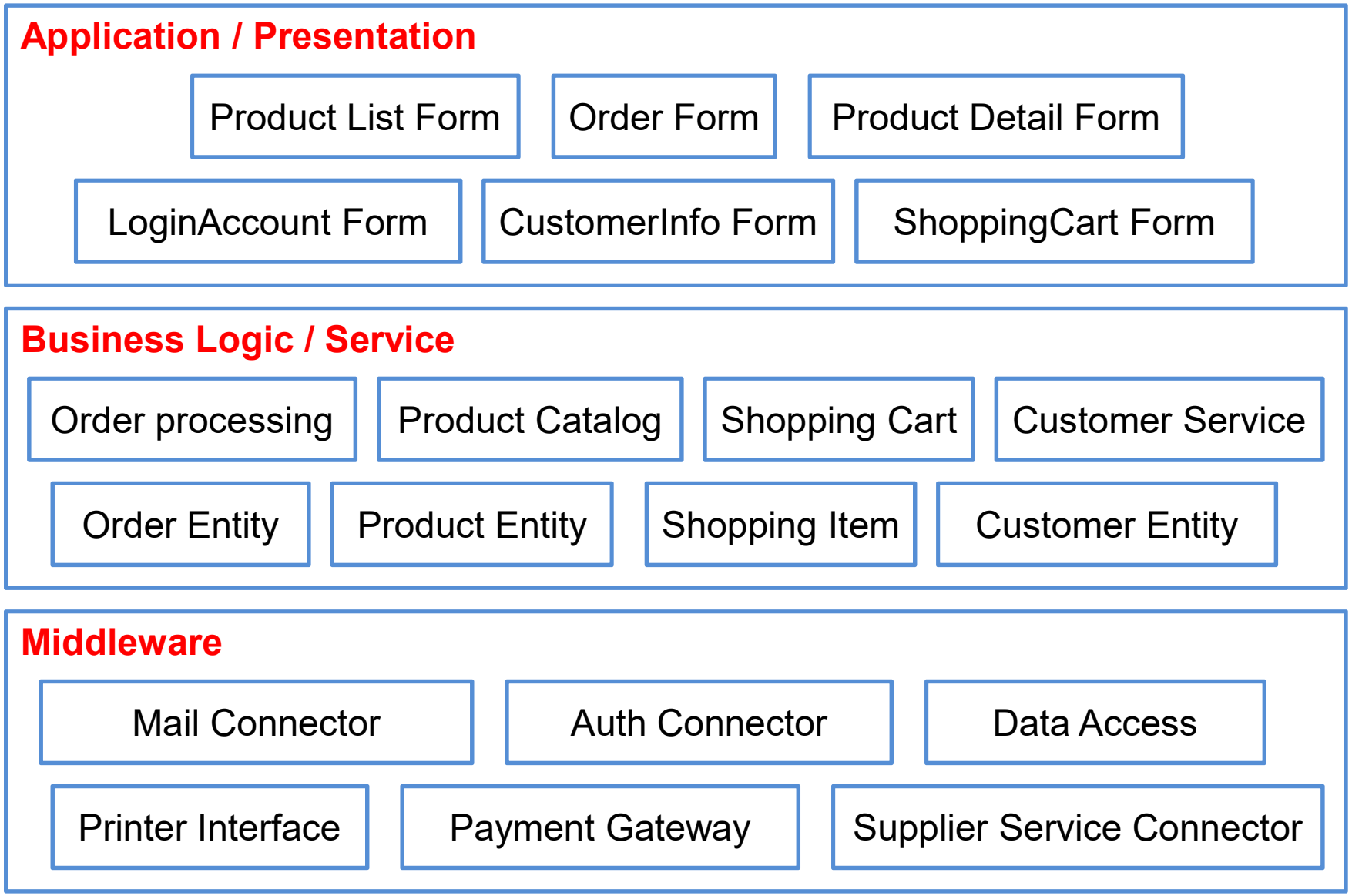


Control Class

- Use case behavior coordinator.
  - Get input data from boundary classes.
  - Invoke entities operation, and do some business logic.
  - Output data to boundary classes.
- Control classes separate boundary and entity classes.
- Control classes sample: business logic, transaction.



Analysis Model to Application Architecture



4. 架構性分析 – ARCHITECTURAL ANALYSIS

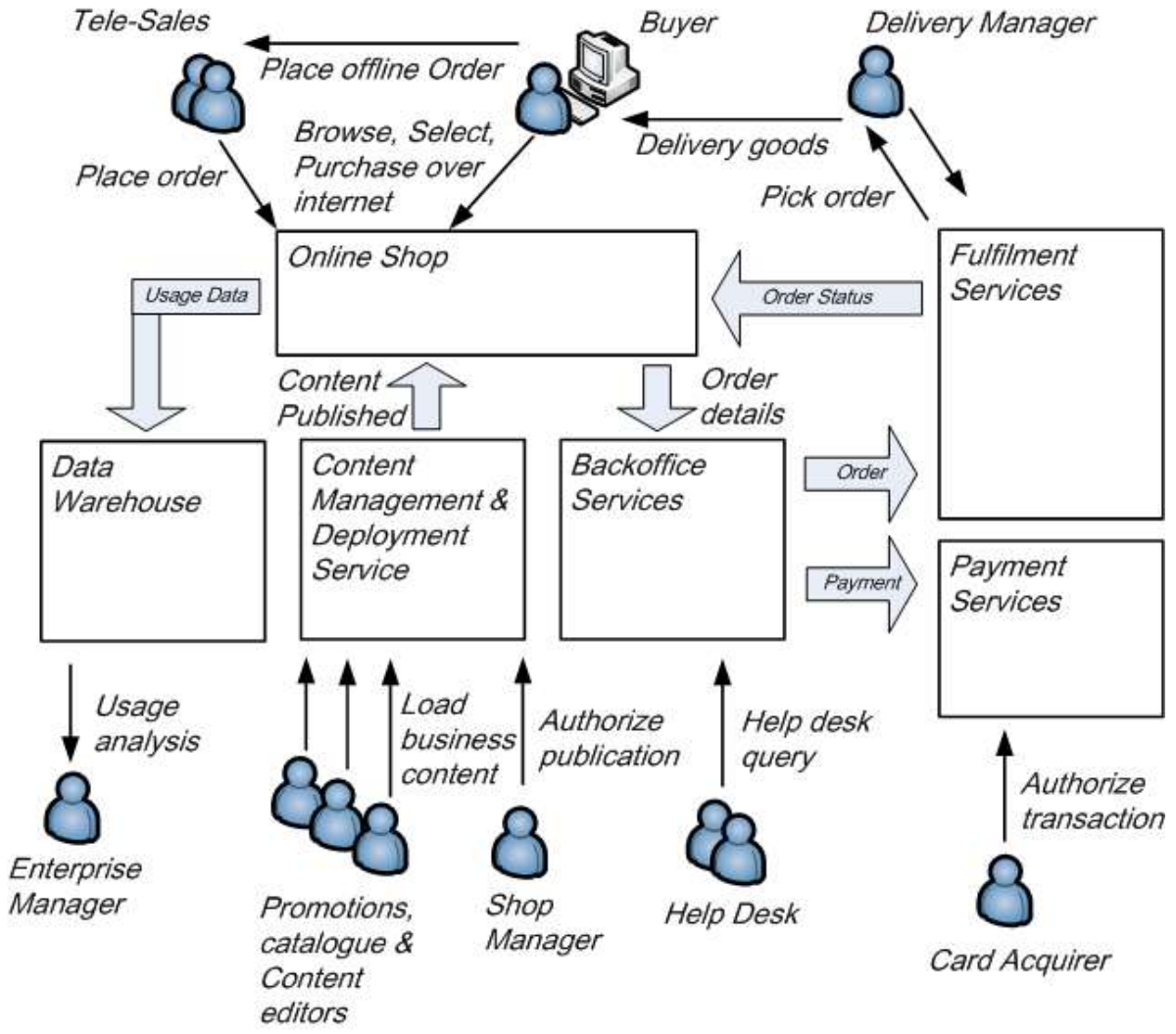
PIC

Architectural Analysis

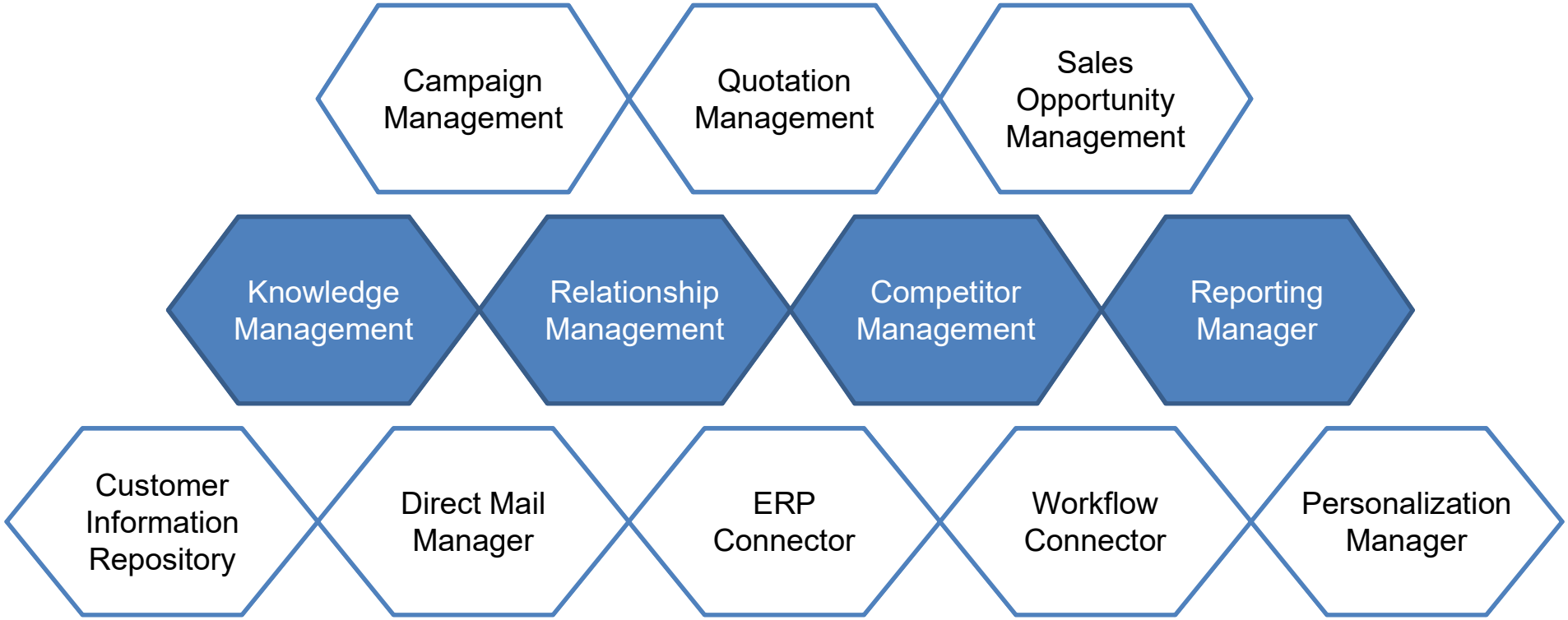
- Define High-Level Architecture
- Identify Architecture Style
- Synthesize a candidate architecture
- Select Assets and Technologies
- Choose Solution Mechanism

PIC

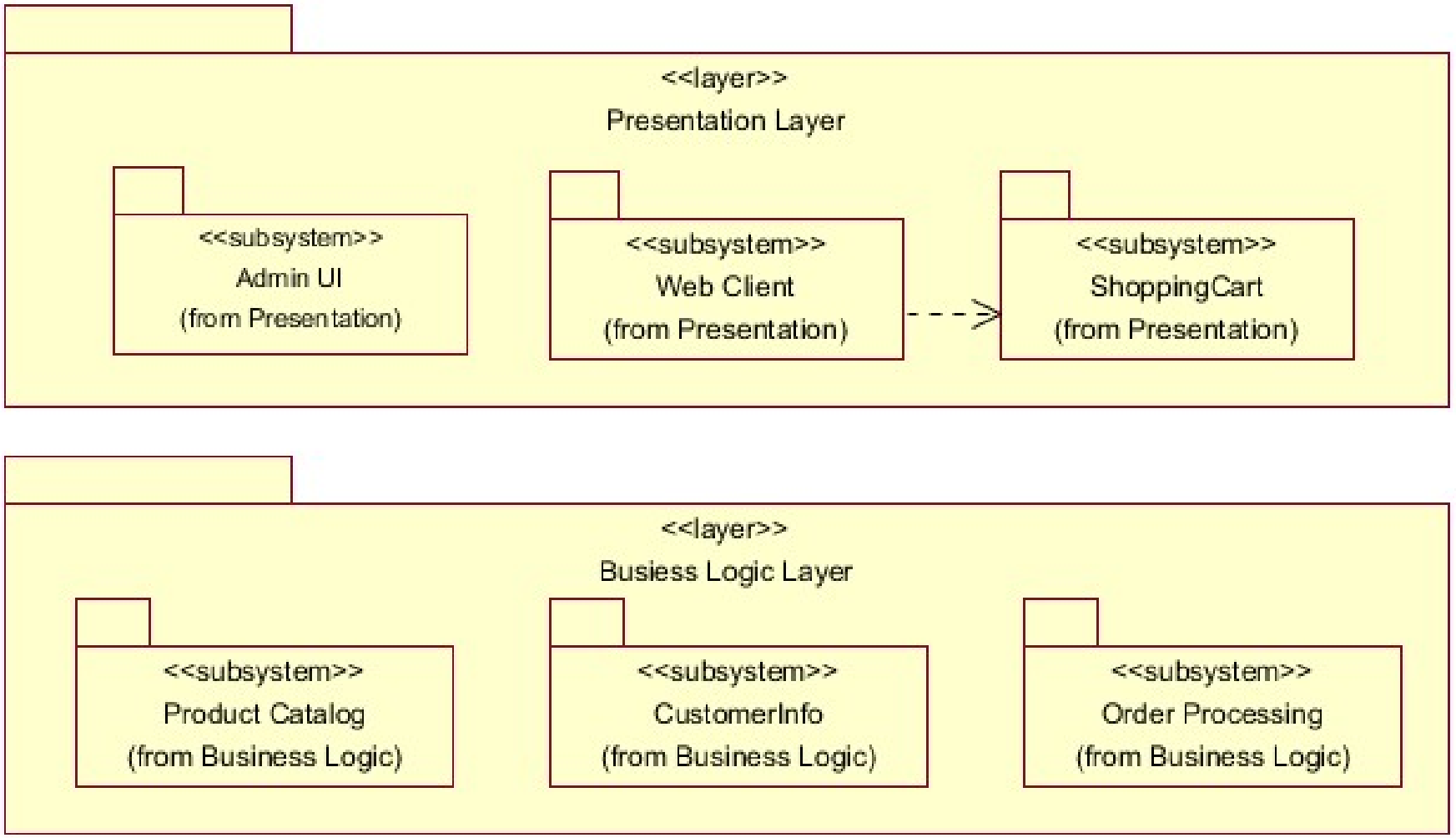
Online Shopping: Business, Process & Services



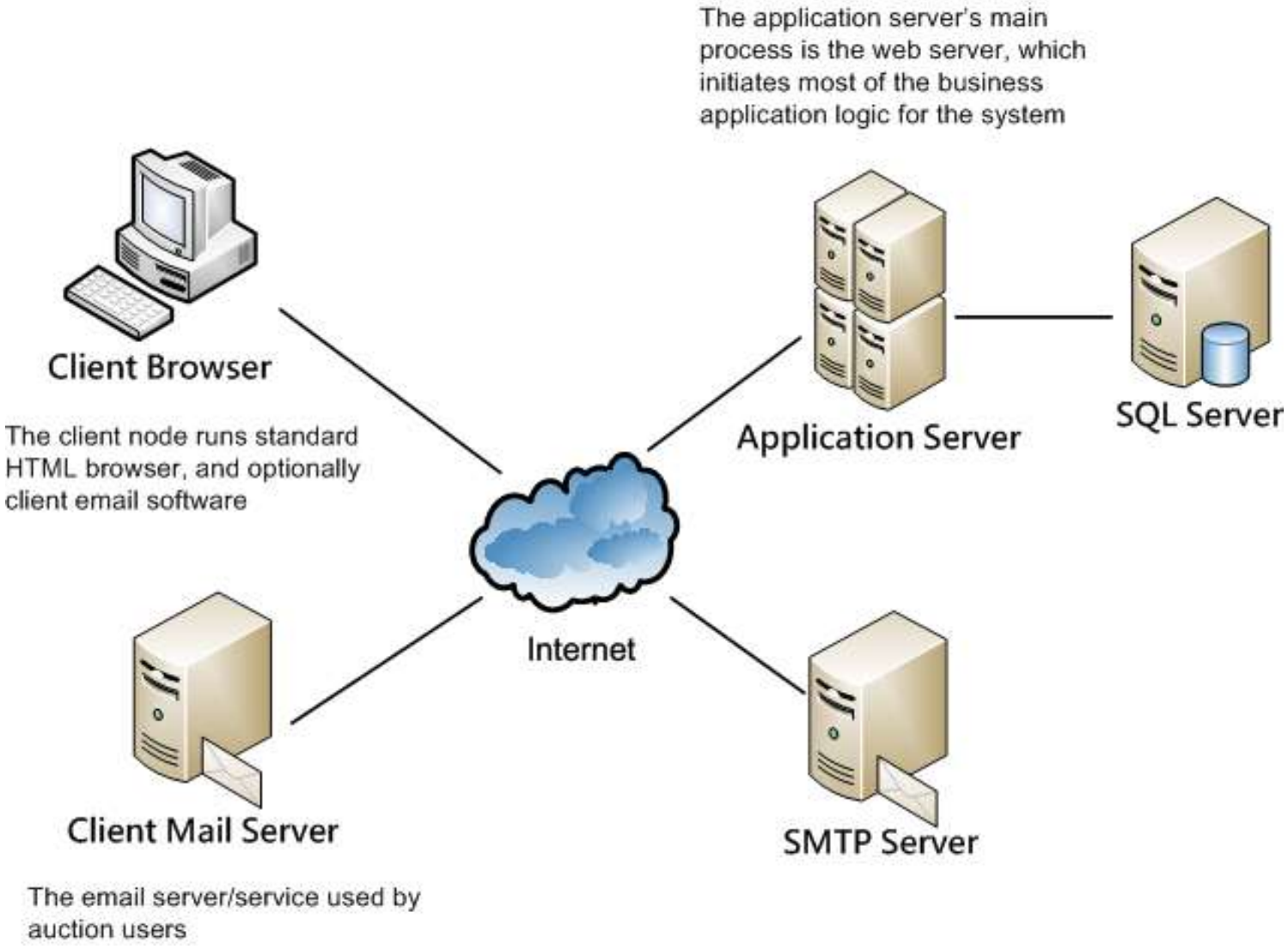
CRM Architecture



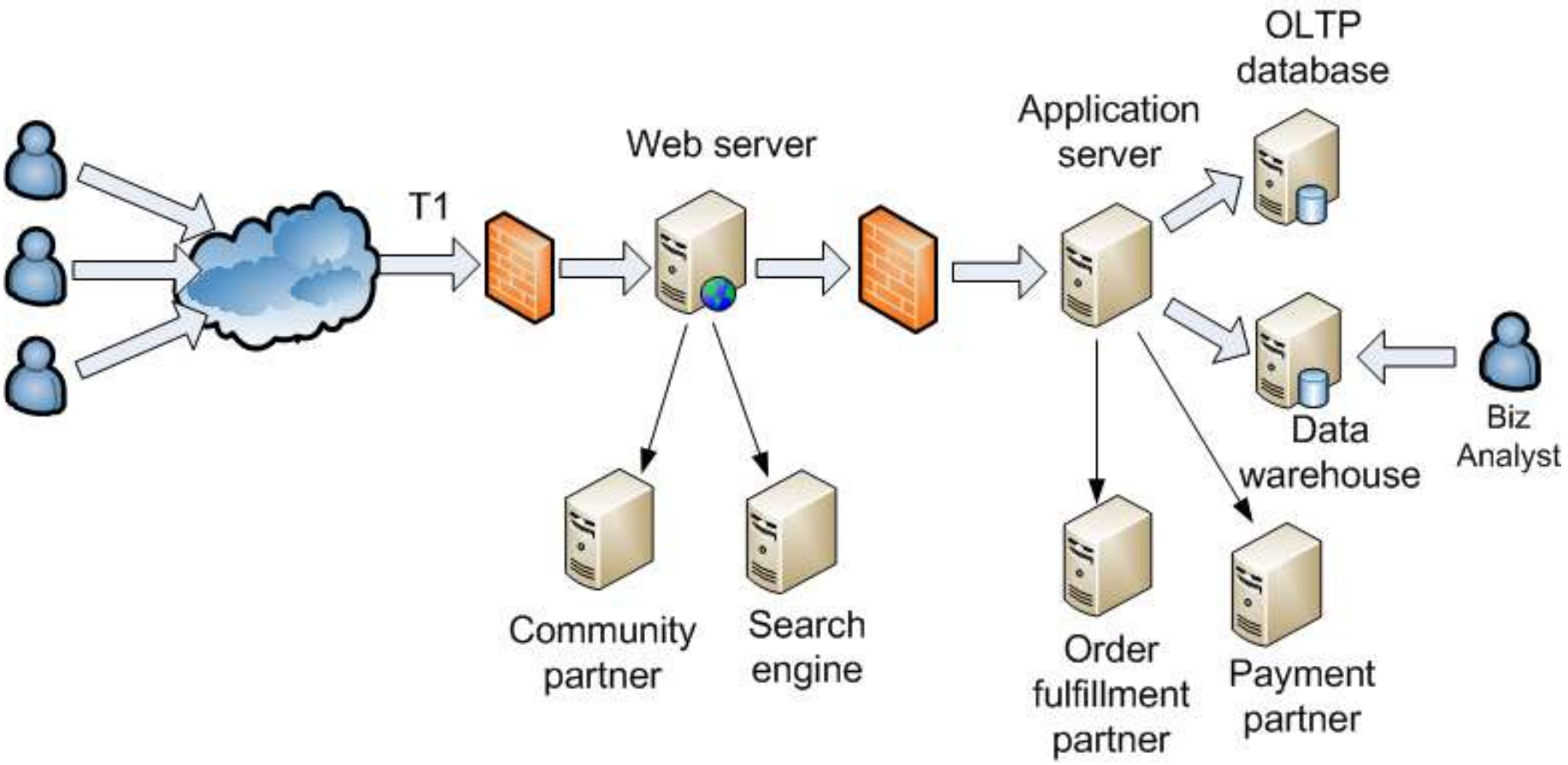
Applications & Business Services



System Architecture



3-Tiers System Architecture

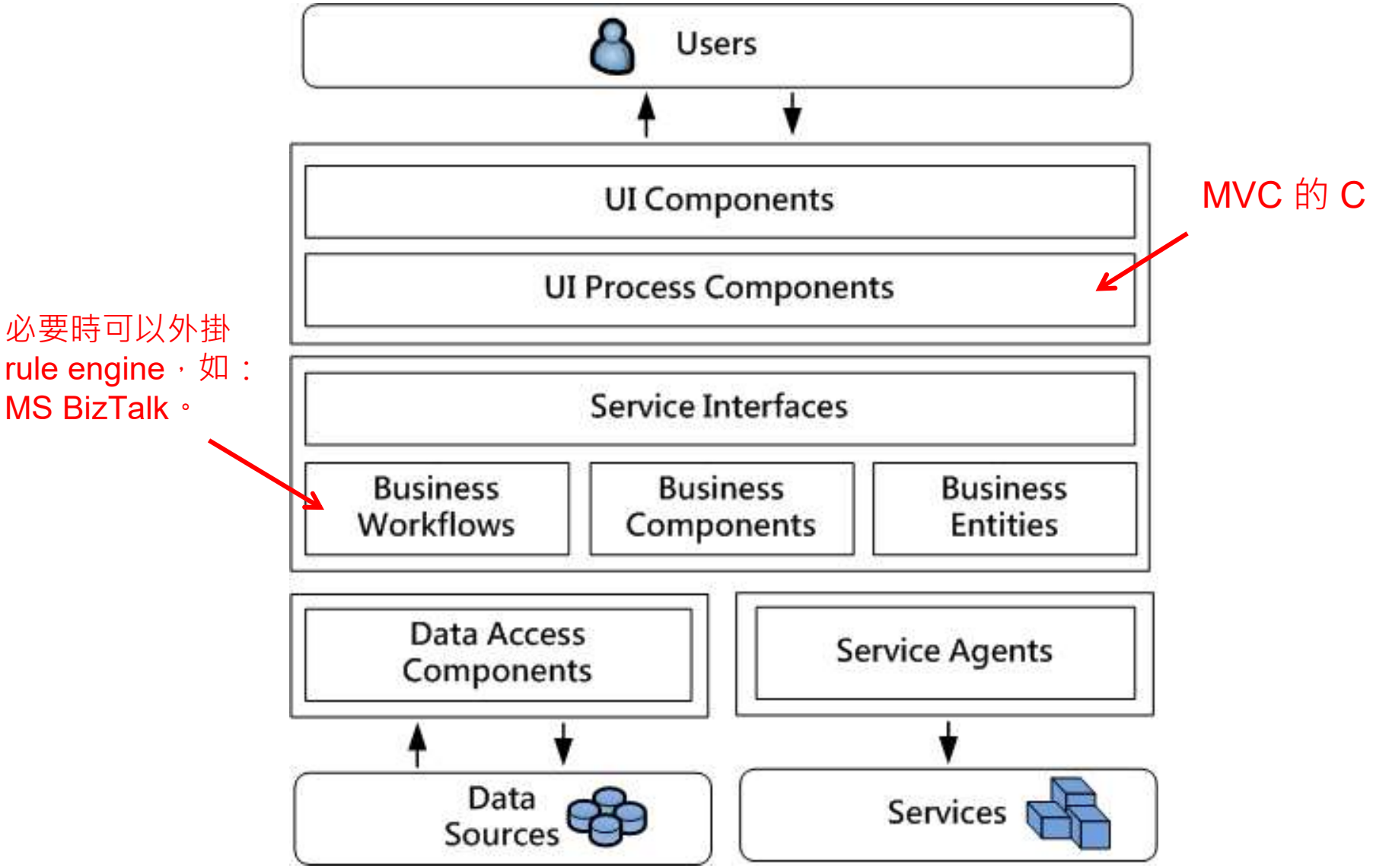


Architecture Style

Category	Pattern
Structure	Layers
	Pipes and Filters
	Blackboard
Distributed Systems	Broker
Interactive Systems	Model-View-Controller
	Presentation-Abstraction-Control
Adaptable Systems	Reflection
	Microkernel

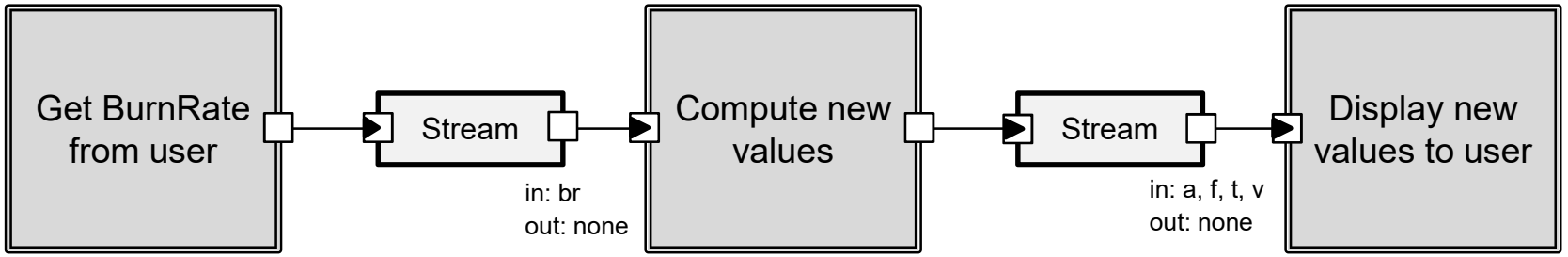
PIC

Layered Application Architecture



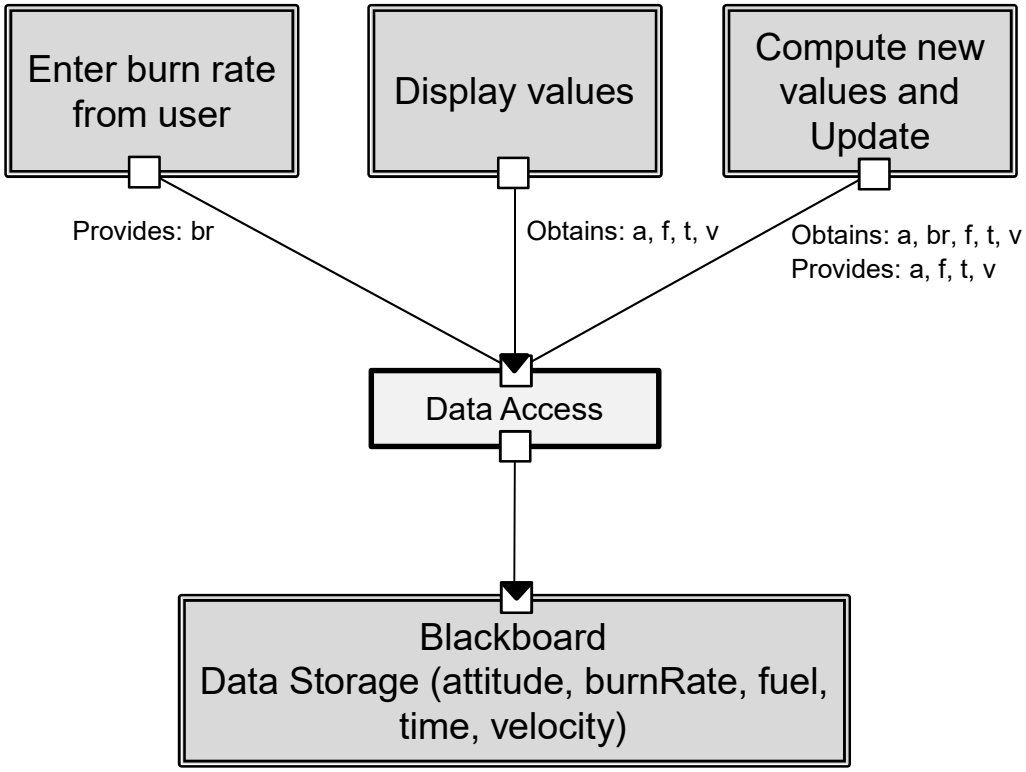
Pipe and Filter Style

- Examples
  - Unix shell — signal processing
  - Distributed systems — parallel programming
- Example: `ls invoices | grep -e August | sort`

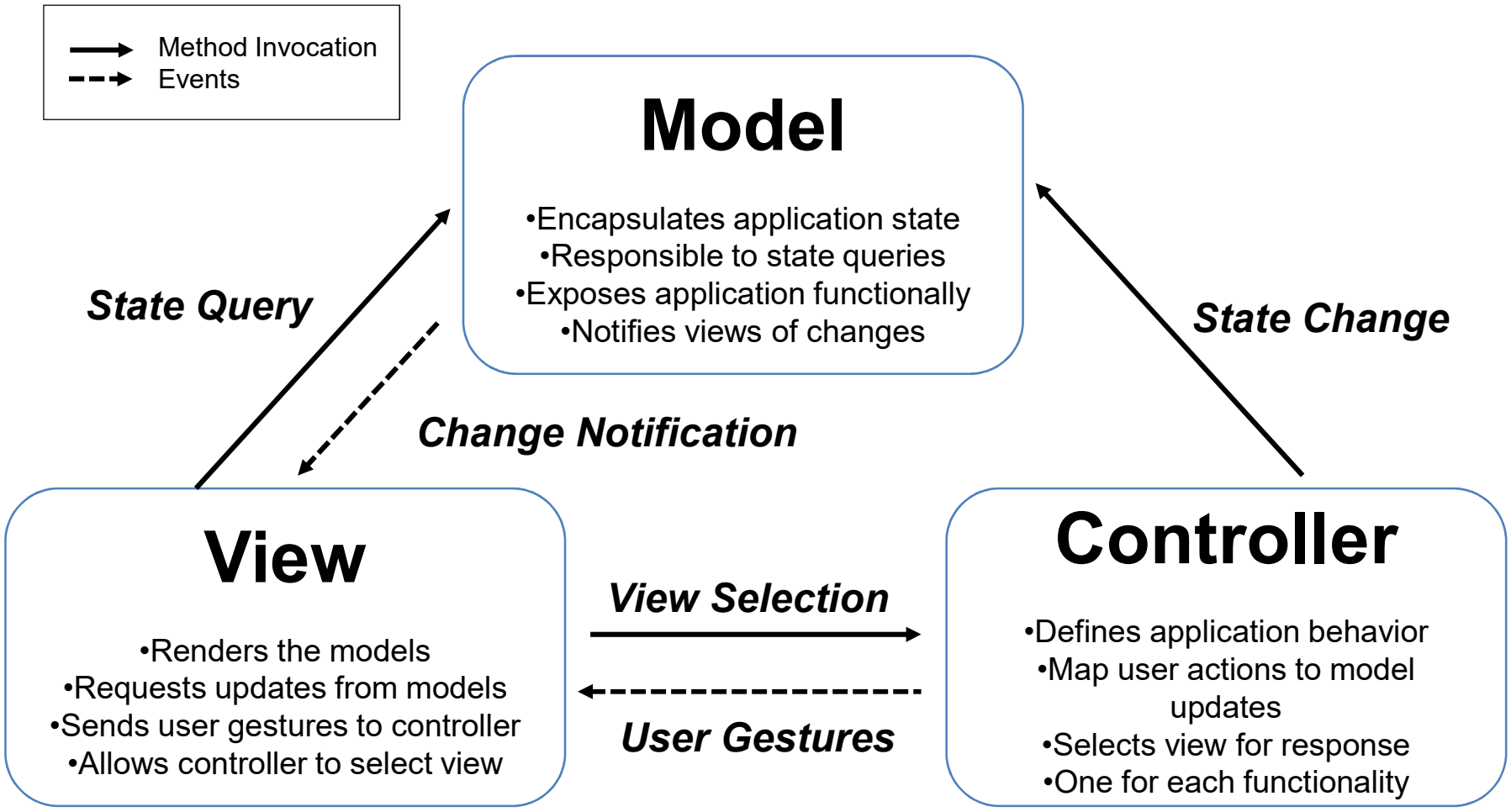


Blackboard Style

- 通常用於解決不確定性問題（uncertainty）的系統，如： AI、machine learning…等。
- 通常使用到許多統計分析理論與演算法。
- Sample
  - CIA threat assessment
  - 多重 sensors
  - 多重 agents
  - Central info store
    - Update & notify

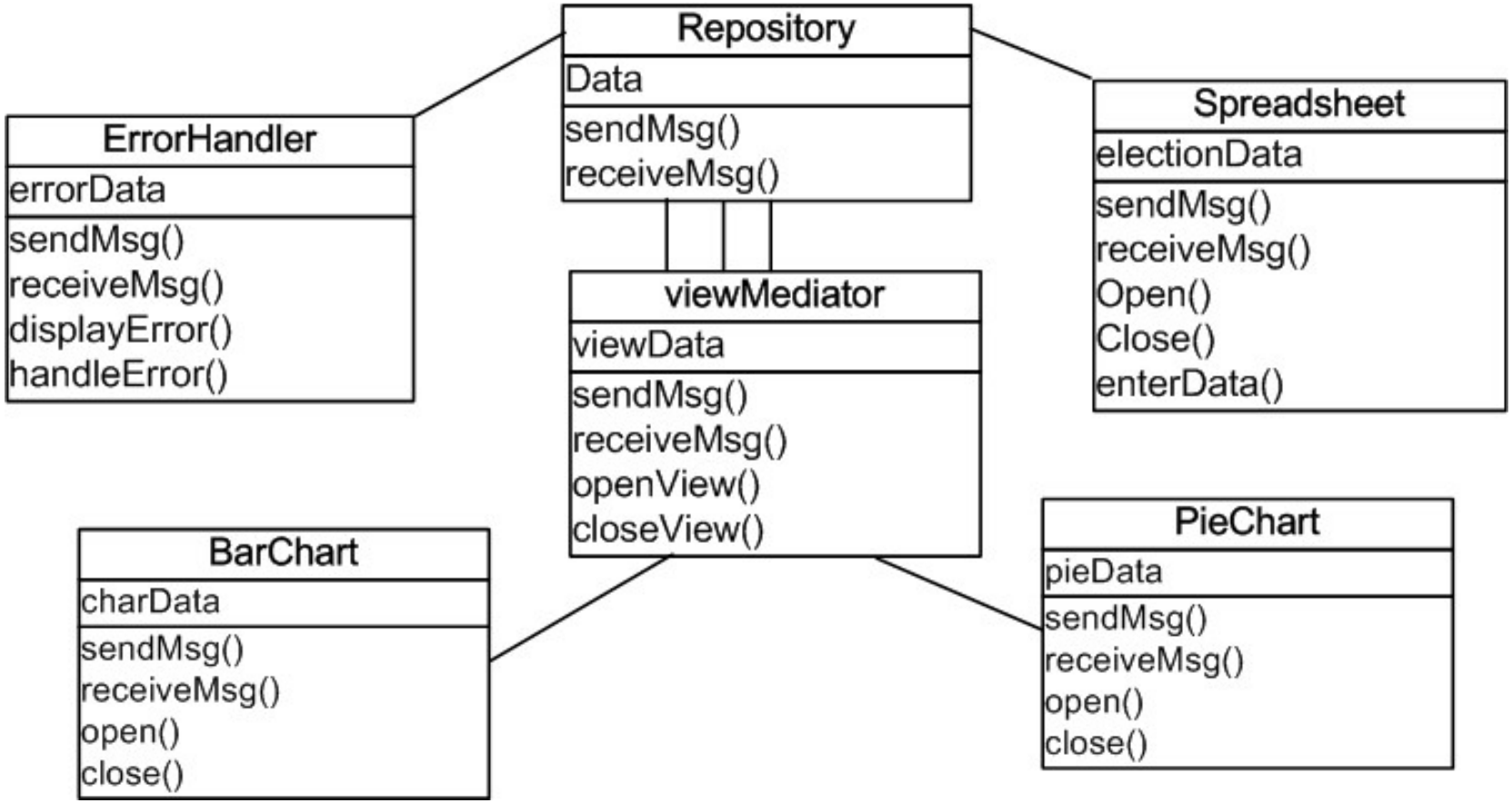


MVC Architecture Design



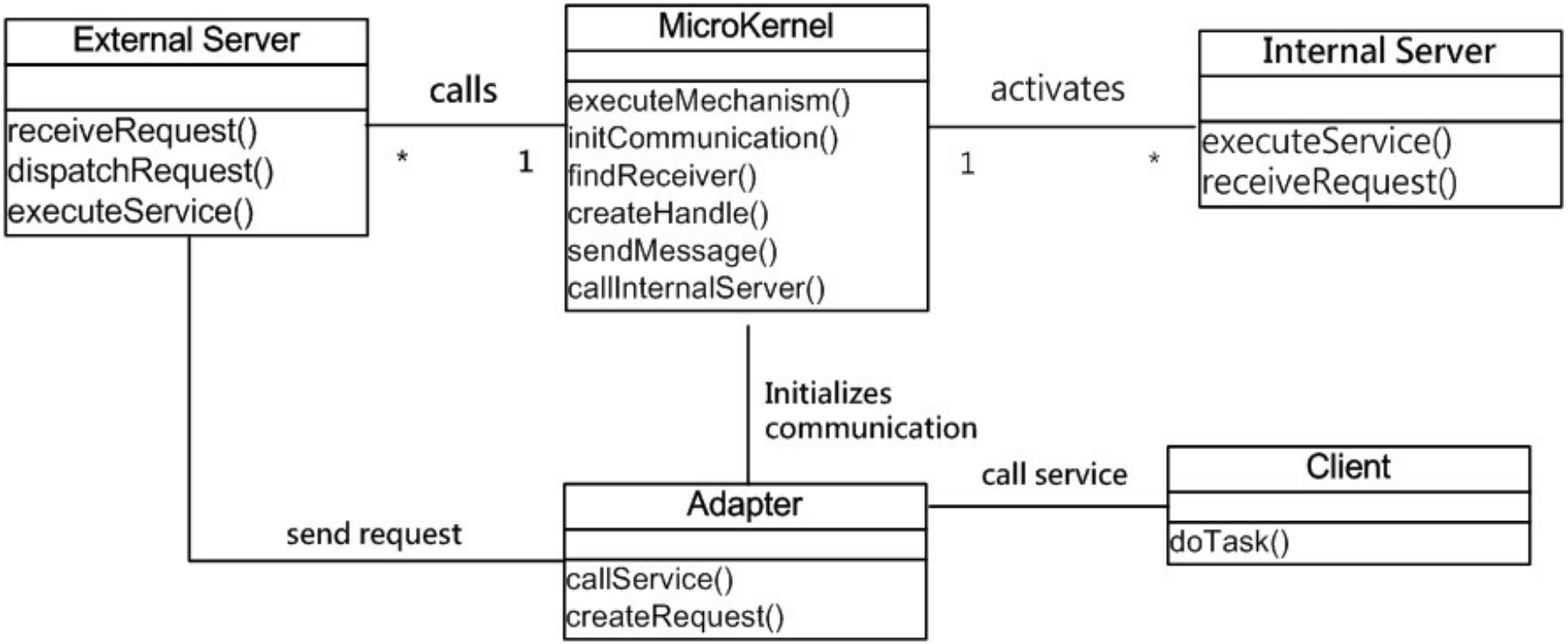
Presentation-Abstraction-Control

- 常用於統計圖表的軟體

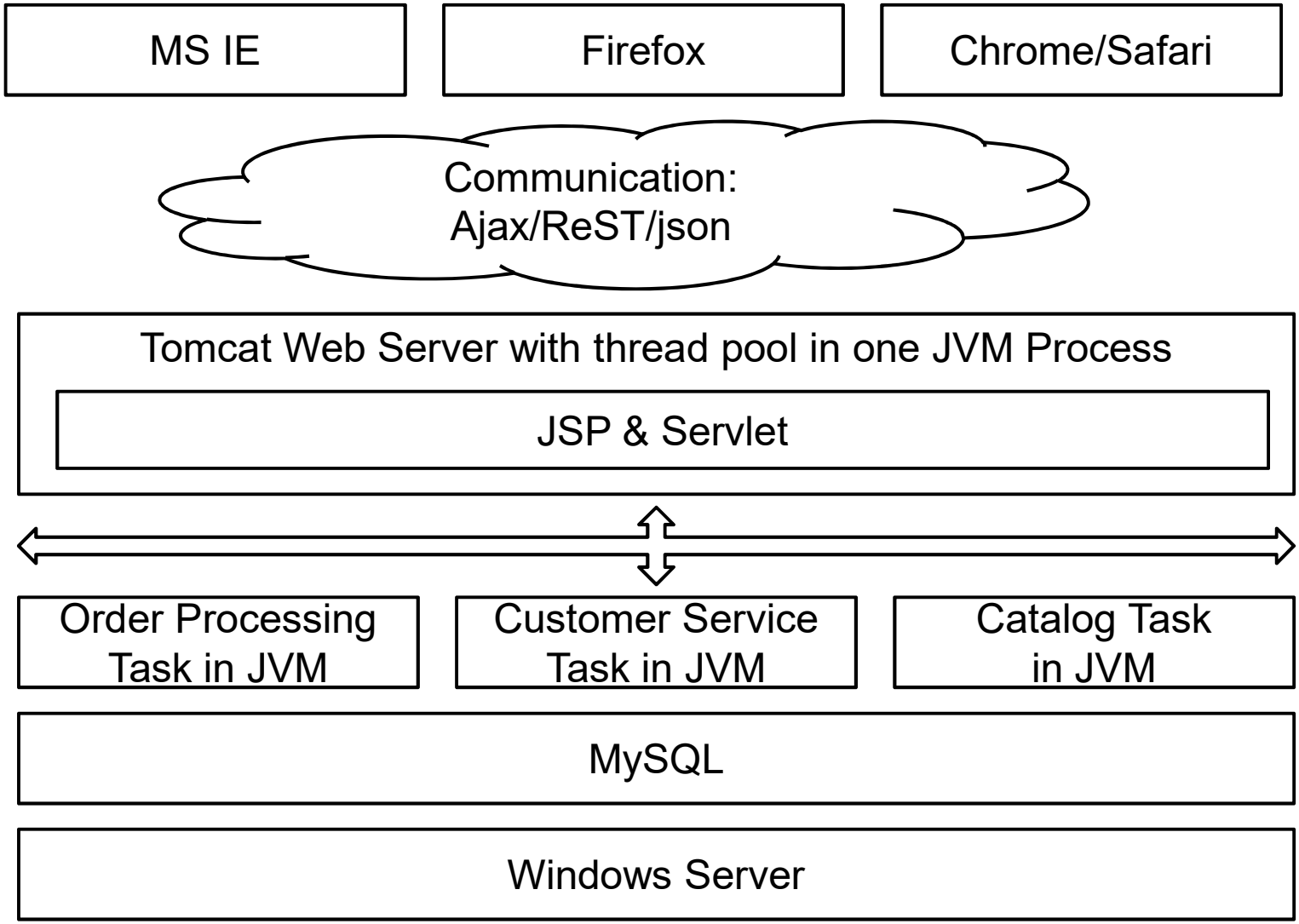


Microkernel

- Allows adaptability to changing system requirements.
- Separates a functional core from extended functionality and customer-specific parts.
- Example: Windows OS, Linux/Unix OS, Eclipse, MS Visual Studio/Office



Solution Architecture



Technology churn

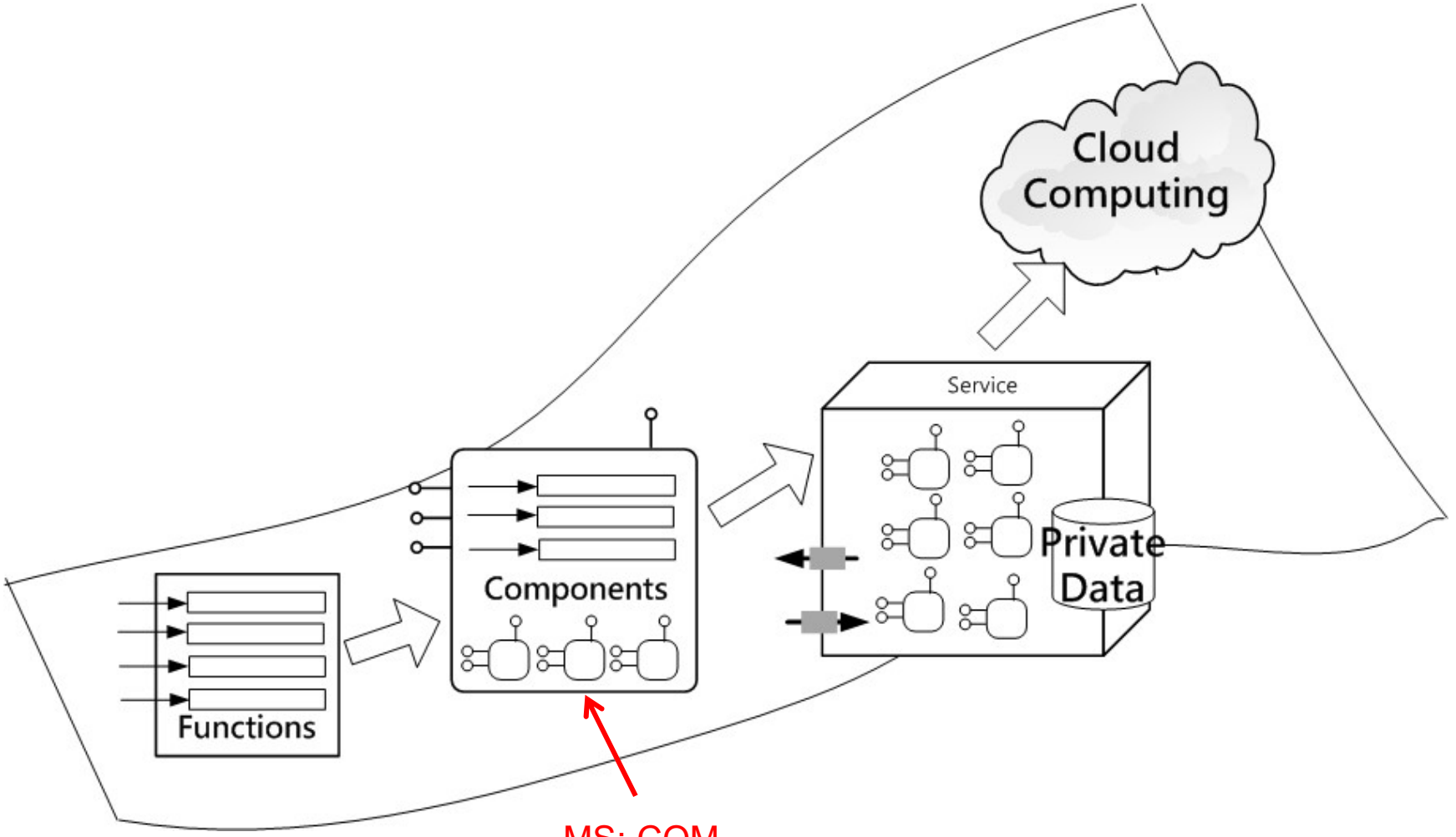
- Industry
  - Mobile devices
  - Smart devices
- Technology
  - Cloud computing
- Architecture
  - SOA
- Web
  - Ajax, HTML 5
- Authentication
  - OAuth (Facebook, Google)

PIC

Survey Available Assets

- Identify assets.
- Analyze the fit and gap between assets and needs.
- Locate and list assets that are potentially reusable on the project.
- Perform a preliminary evaluation to ensure that necessary support is potentially available.

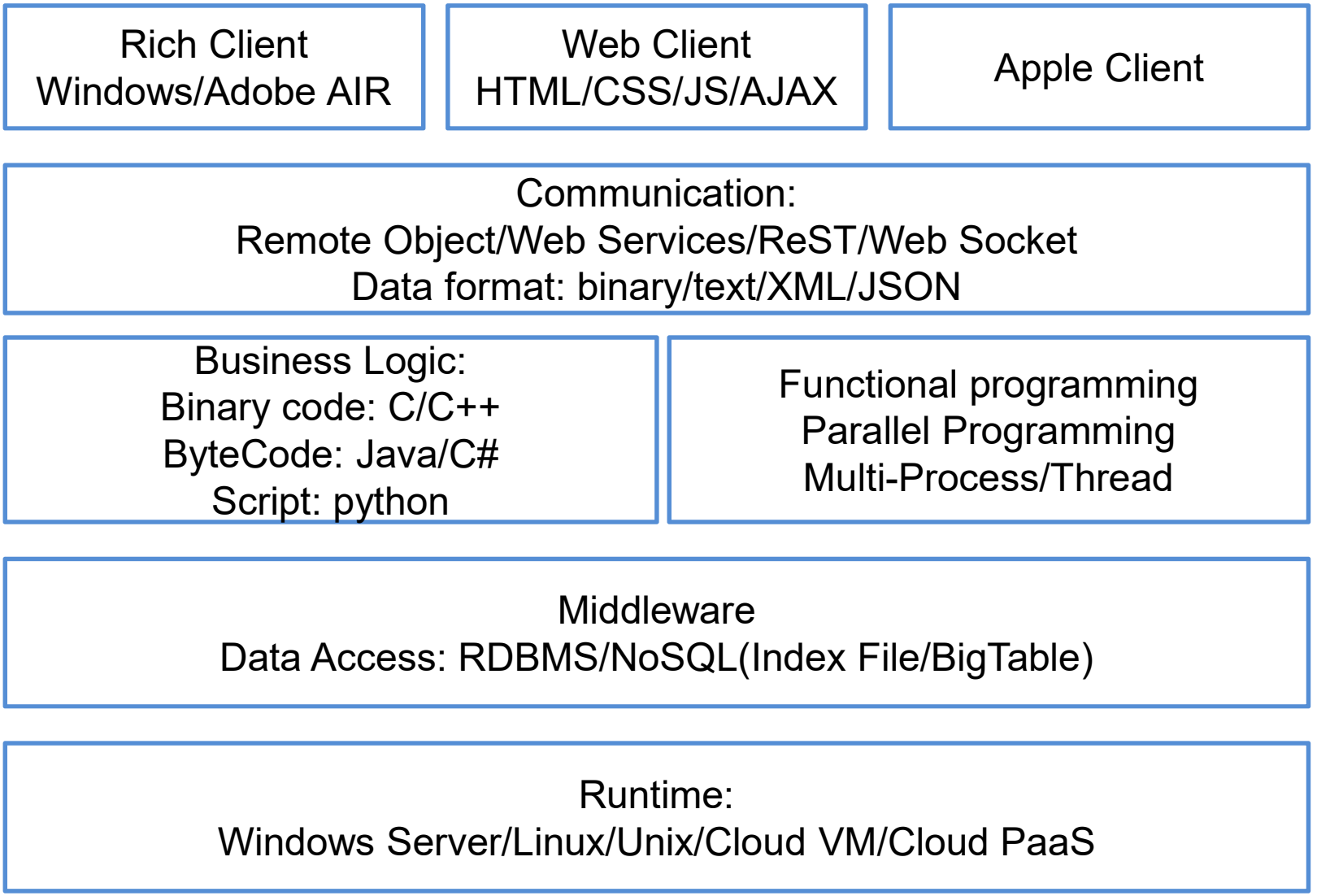
Technology Trend



Select Technologies

- Linux / Windows / Proprietary UNIX
  - .NET v.s Java v.s LAMP (Linux + Apache + MySQL + PHP)
- RDBMS or Mixed
- Open Source or WebSphere & WebLogic
- Plain HTML / Ajax / RIA / HTML 5
- Clustering / Grid computing / Cloud Computing
- Collaborate with existing Internet Services

Technology Architecture



Analyze Technology Requirements

- Persistency
  - Granularity
  - Volume
  - Duration
  - Access mechanism
  - Access frequency (creation/deletion, update, read)
  - Reliability
- Communication of Distributed System
  - Latency
  - Synchronicity (Sync & Async)
  - Message Size
  - Protocol

PIC

- 機制是對常見問題的一種共同性解決方案

架構需求 Architectural Requirement	分析機制 Analysis Mechanism	設計機制 Design Mechanism	實作機制 Implement Mechanism
Supportability	Auditing Error mgmt Remote mgmt Monitor	Instrument Log Exception Remote control Service control	WMI .NET Log

Solution Mechanism Sample

架構需求 **Architectural Requirement**	分析機制 **Analysis Mechanism**	設計機制 **Design Mechanism**	實作機制 **Implement Mechanism**
Persistent store	Storage Persistence	Cloud BigTable File I/O	Amazon Google
	RDBMS Persistence	Data Access Class Library	ADO .net
		Data access component	Data access application block
		O-R Mapping	NHibernate

Solution Mechanism List

考慮使用下列機制滿足架構性需求(或稱「非功能性需求」)

- Auditing
- Communication
- Debugging
- Error management
- Information exchange
- Licensing
- Globalization
- Localization
- Mega-data
- Memory management
- Pluggable component architecture
- Meta-data
- Persistence
- Process management
- Reporting
- Resource management
- Scheduling
- Security
- System management
- Transaction management
- Workflow

5. 架構設計 – ARCHITECTURE DESIGN

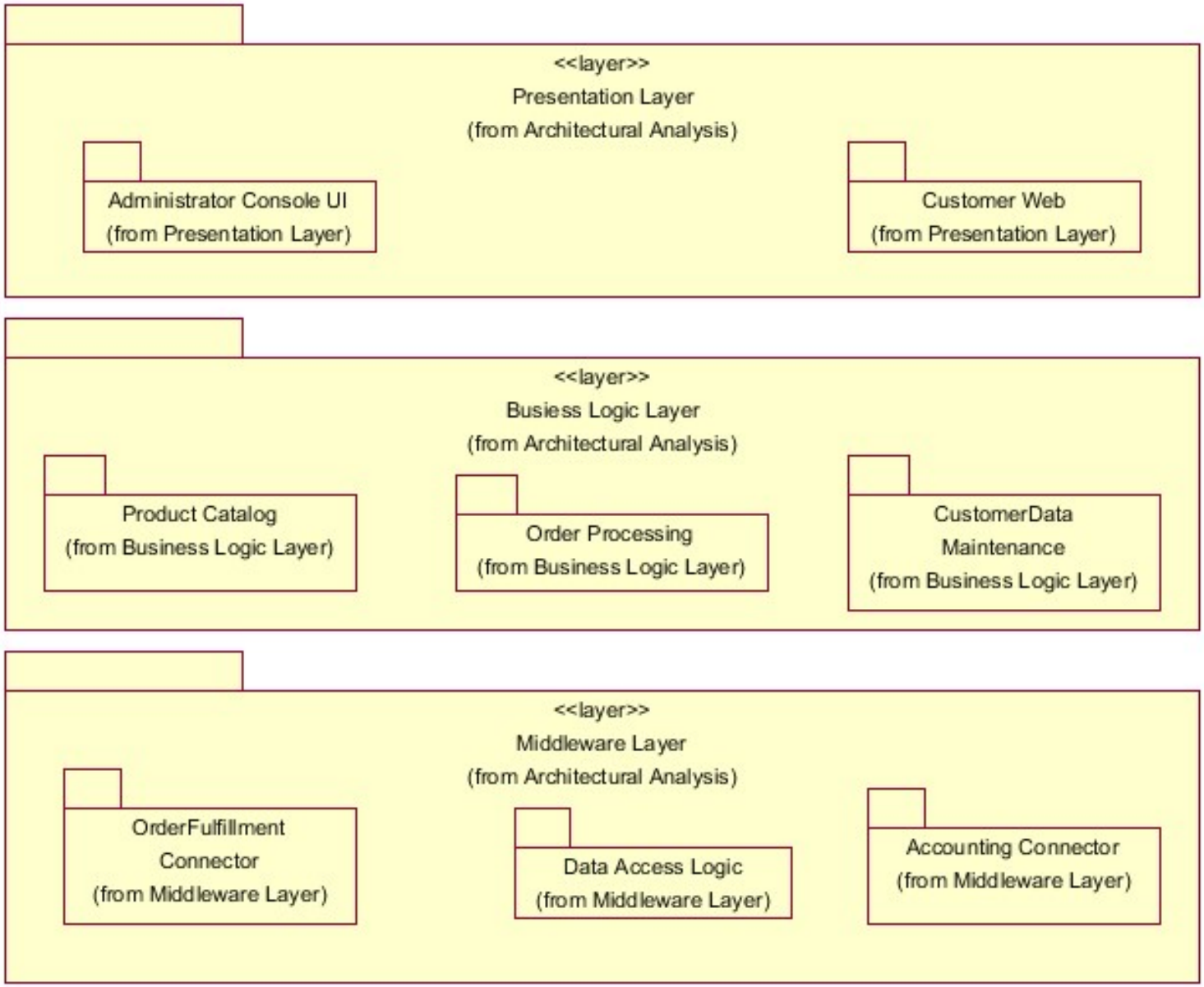
- Identify subsystems
- Identify Interfaces
- Construct an Architectural Model

- **Subsystem**: An important abstraction in logical design
- **Component**: A physical file, such as exe, DLL, jar, war
- **Service**: a executable runtime instance

Candidate Subsystems

- Key Abstractions/Analysis classes which may evolve into subsystems:
  - Classes providing complex services and/or utilities
  - Boundary classes (user interfaces and external system interfaces)
- Existing products or external system in the design
  - Communication software
  - Database access support
  - Types and data structures
  - Common utilities

eStore Subsystems



Identifying Interfaces

- Purpose
  - More flexible then class dependency.
  - Stable, well-defined interfaces are key to a stable, resilient architecture.
- Steps
  - Identify a set of candidate interfaces for all subsystems.
  - Look for similarities between interfaces
    - Merge or generalize
  - Define interface dependencies.
  - Map the interfaces to subsystems.
  - Define the behavior specified by the interfaces.
  - Package the interfaces.

Interface Guidelines

- Interface name
  - Reflect role in the system.
- Interface description
  - Express responsibilities.
- Operation definition
  - Name should reflect operation result.
  - Name of parameters and result should be descriptive.
- Clear & complete Interface specifications

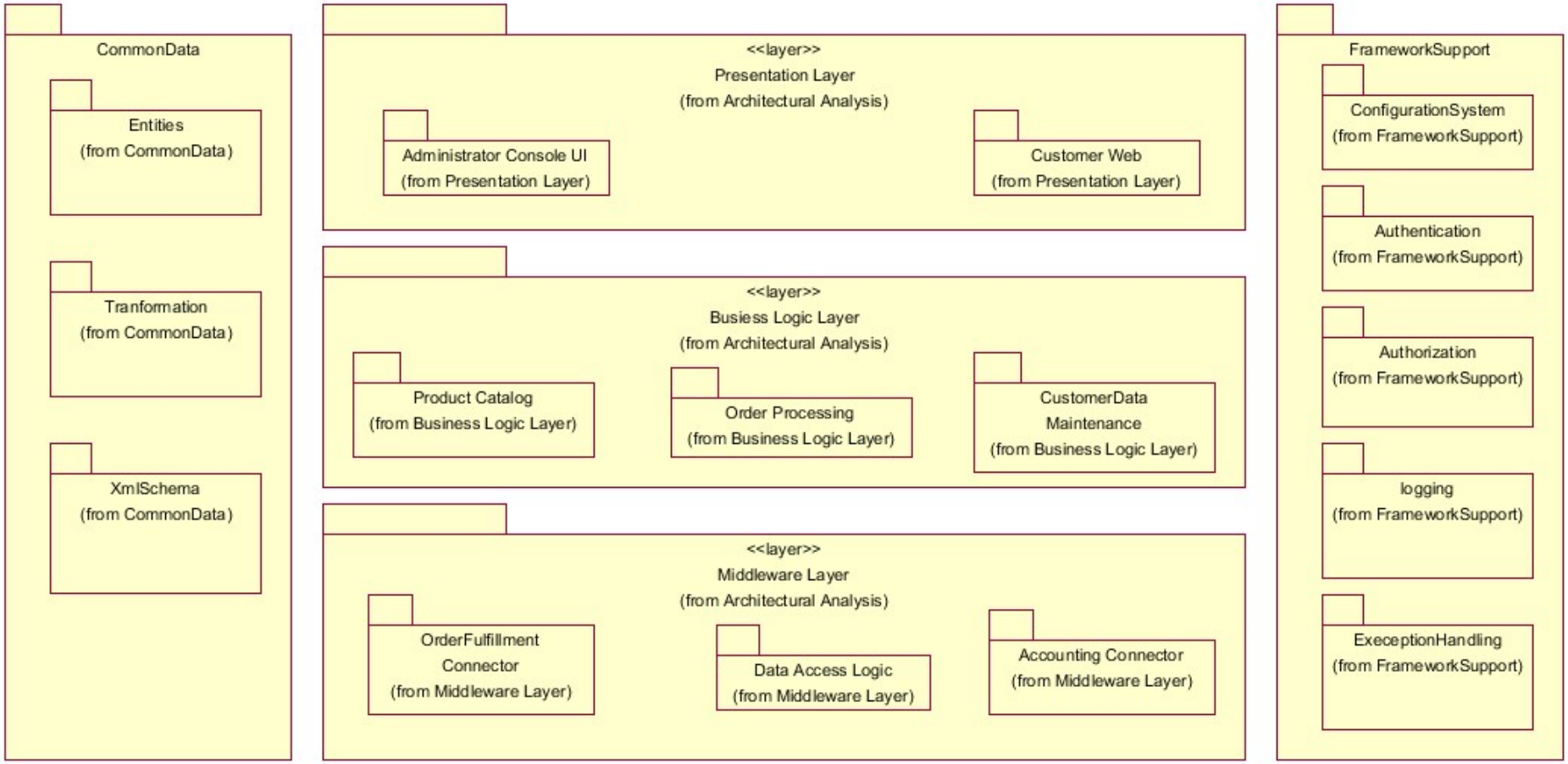
Interface Specification

- Parameters
- Return value
- Errors and Exceptions
- Pre-conditions and post-conditions
- Biz Rules
- Special algorithm
- Design patterns
- Non functional requirements
  - Security, performance, reliability,….
- Test specifications

PIC

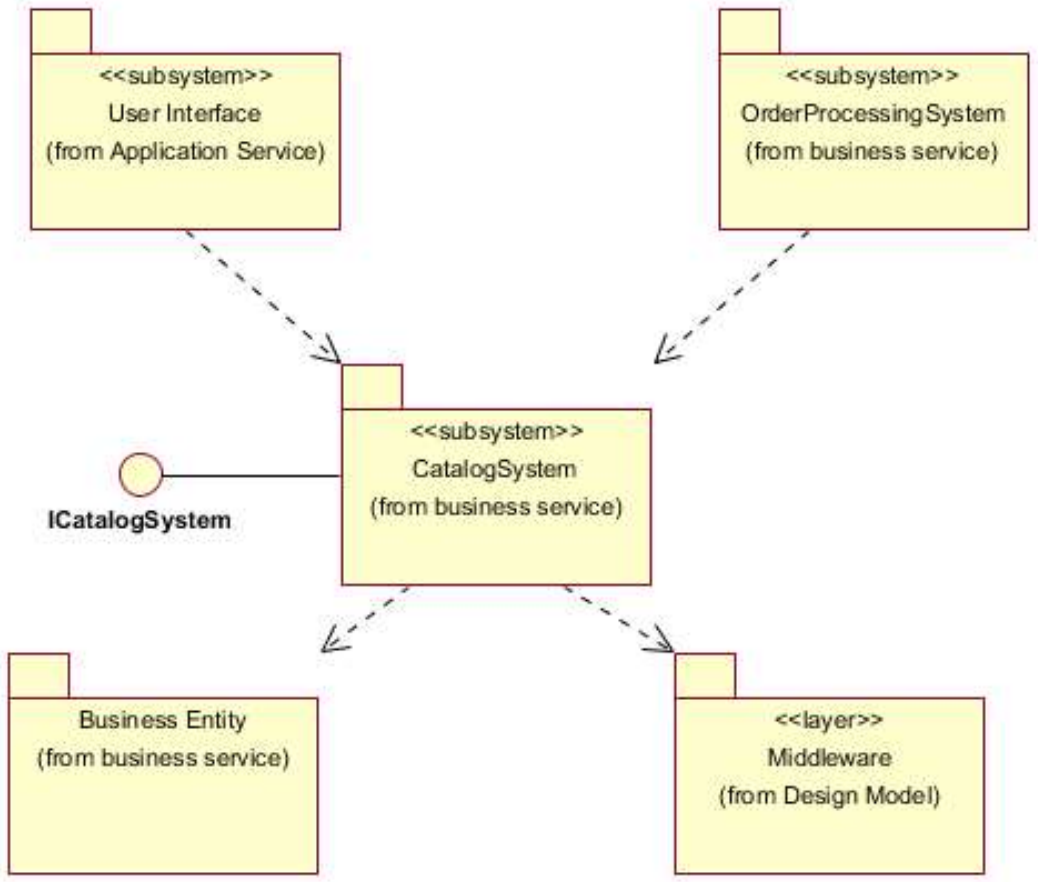
Layered Application Architecture with Subsystems

- Add infrastructure or technical subsystems



Subsystem Context

- Context diagram contain the subsystem, interface, and related subsystems



Tips

- Consider grouping two design elements in the same **package** if they are closely related.
- If an element is related to an optional service, group it with its collaborators in a separate subsystem.
- Consider moving two design elements in different packages if:
  - One is optional and the other mandatory.
  - They are related to different actors.

6. 設計模式 - DESIGN PATTERN

- Fundamental design pattern
- Platform specific design pattern

PIC

⇨ **Fundamental design pattern**

- Platform specific design pattern

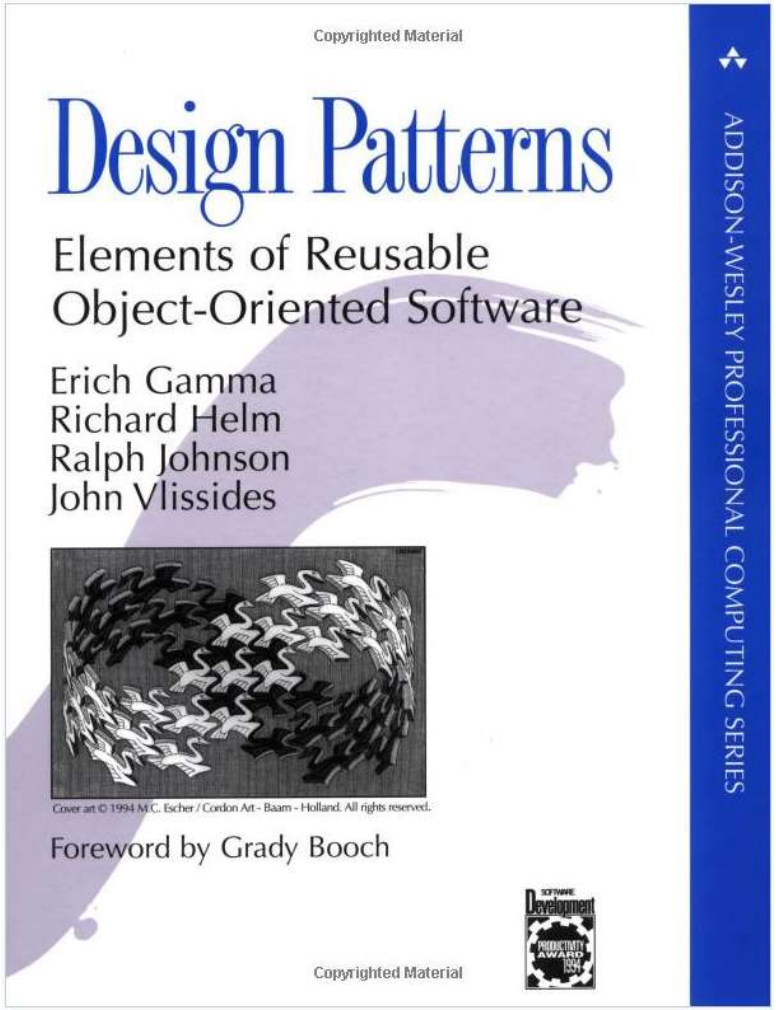
Why Use Design Pattern ?

- Best practice to solve recurring problems.
- Make software design understandable.
- Let software changeable.
  - Resilience

PIC

Design Pattern 經典

GoF: Gang of Four



This is easier to understand….

1. *Program to an **interface**, not an **implementation** (Gang of Four 1995:18)*
2. *Favor **object composition** over **class inheritance** (Gang of Four 1995:20)*

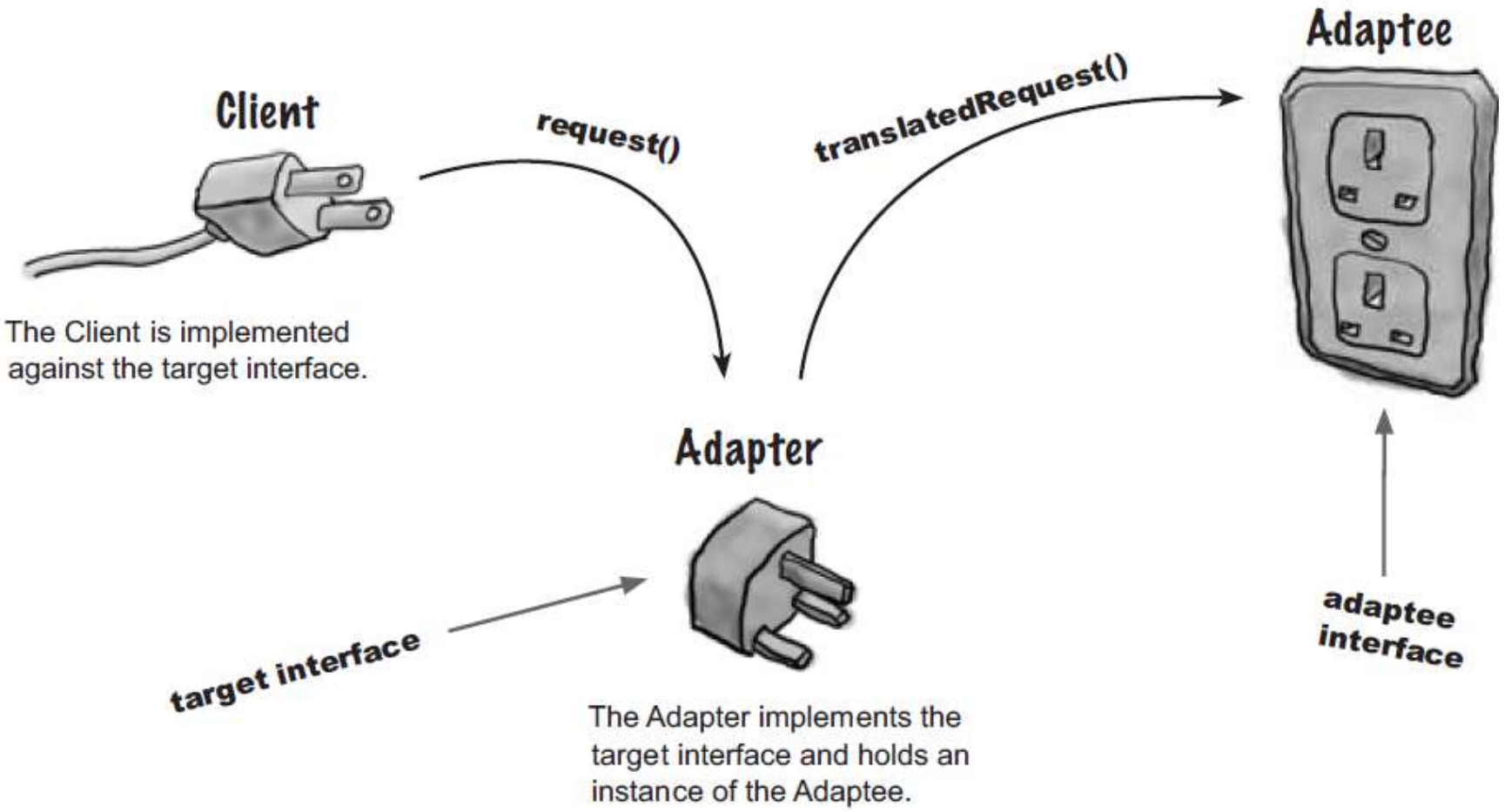
※ 提醒：在設計模式中，所謂的「**實踐一個介面**」，並不一定表示「寫一個類別，並利用 implements 關鍵字以實踐某個 Java介面」，「實踐一個介面」泛指「**實踐某個超型態（可以是類別或介面）的某個方法**」(Head First Design Patterns p.117)

GoF Design Pattern Catalog

Creational	Structural	Behavioral
1. **Factory Method** 2. Abstract Factory 3. Builder 4. Prototype 5. Singleton	1. **Adapter** 2. Bridge 3. Composite 4. Decorator 5. **Façade** 6. Flyweight 7. **Proxy**	1. Chain of Responsibility 2. **Command** 3. Interpreter 4. Iterator 5. **Mediator** 6. **Momento** 7. Observer 8. State 9. Strategy 10. Template 11. Visitor
如何有效率地產生、管理與操作物件。	如何設計物件之間 的靜態結構，如何完成物件之間的繼承、實現與依賴關係，這關乎著系統設計出來是否健壯（robust）：像是易懂、易維護、易修改、耦合度低等等議題。	物件之間的合作行為構成了程式最終的行為，物件之間若有設計良好的行為互動，不僅使得程式執行時更有效率，更可以讓物件的職責更為清晰、整個程式的動態結構（像是物件調度）更有彈性 ➔ 演算法、邏輯。

PIC

Adapter Pattern



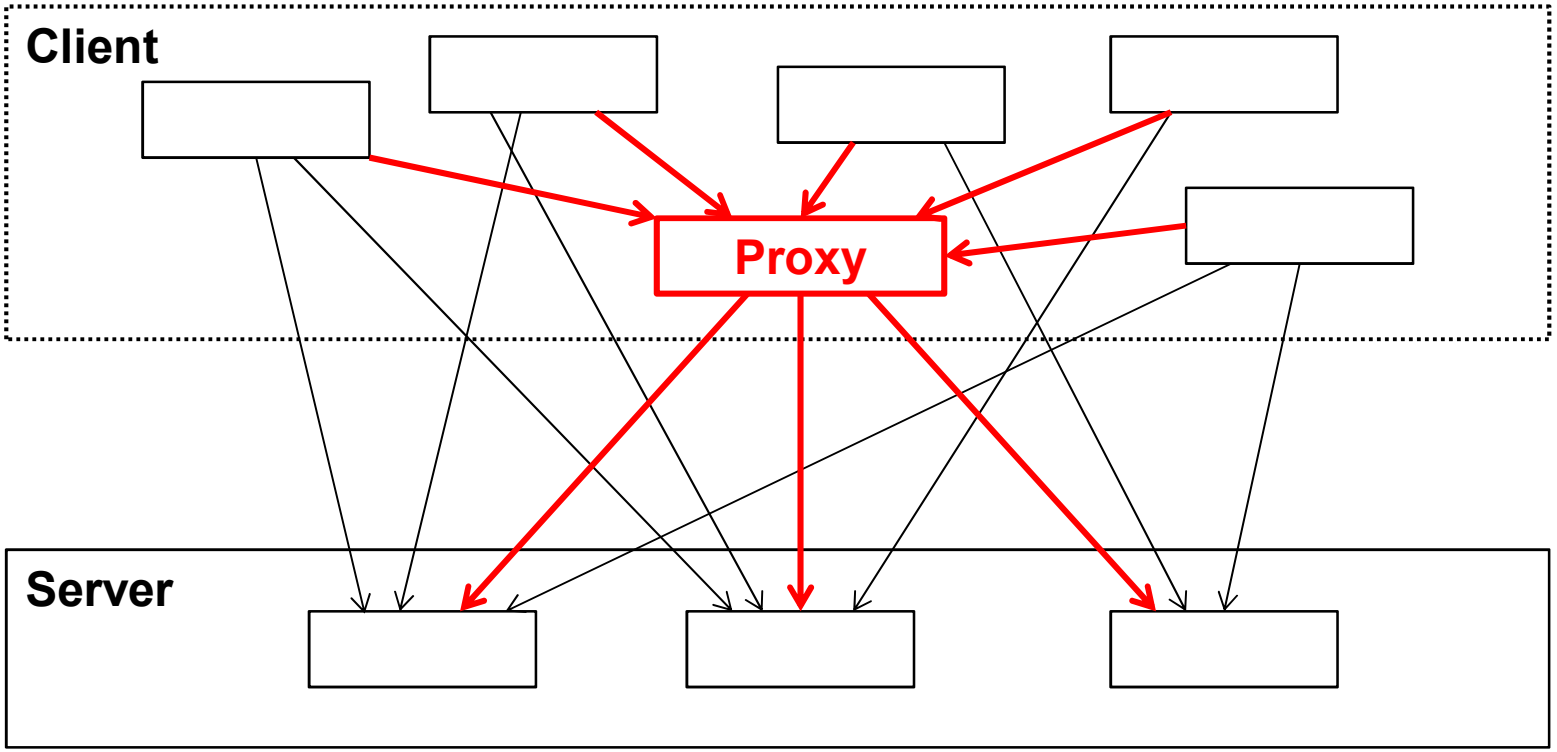
來源：「Head First Design Patterns」第 241 頁

- **Problem:**
  - Convert the interface of a class into another interface client expect.
- **Scenario:**
  - You obtain a target class which you need to use in your designs, however, the public interface of the target class (its methods) dose not match your requirements
- **Solution:**
  - Object composition

Object Adapter (Wrapper)

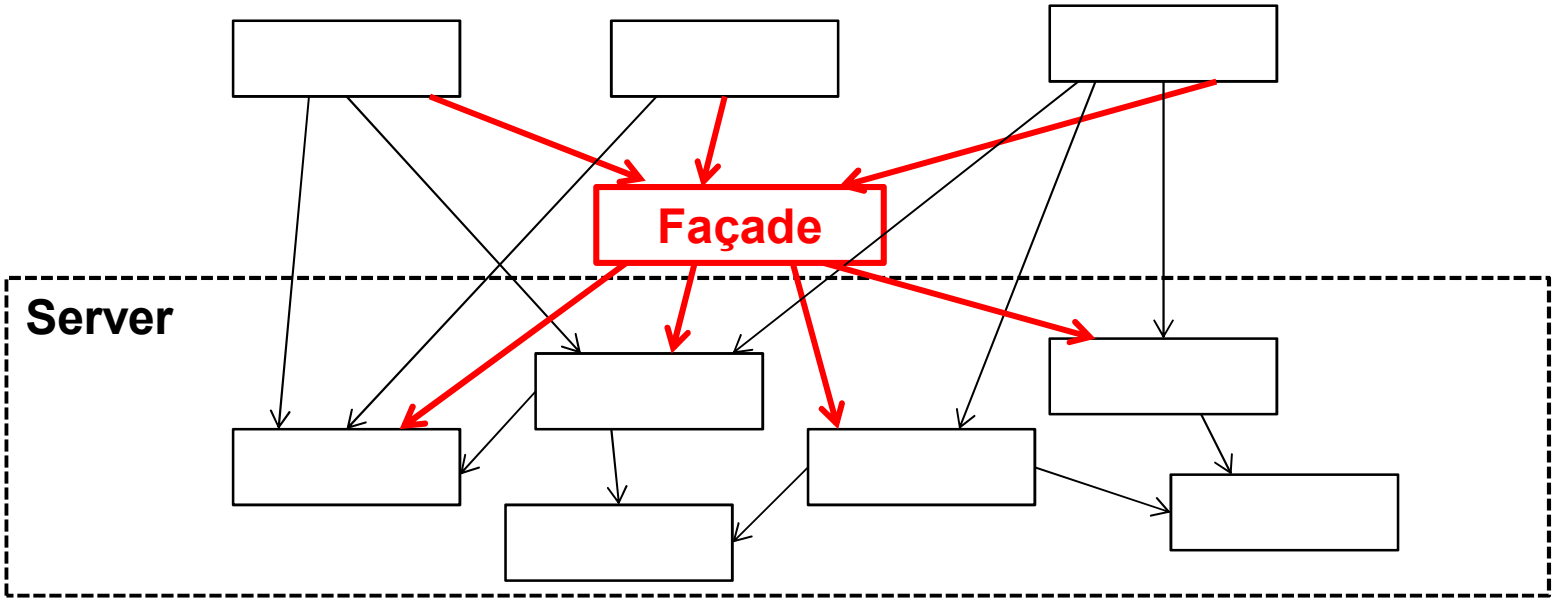
Proxy Pattern in Distributed Environment

- 代理人 (出國旅遊時，請**旅行社**代辦護照、簽證、訂機票、酒店、保險)
- Client Proxy: client representative to service.
- Isolate presentation layer from service deployment.

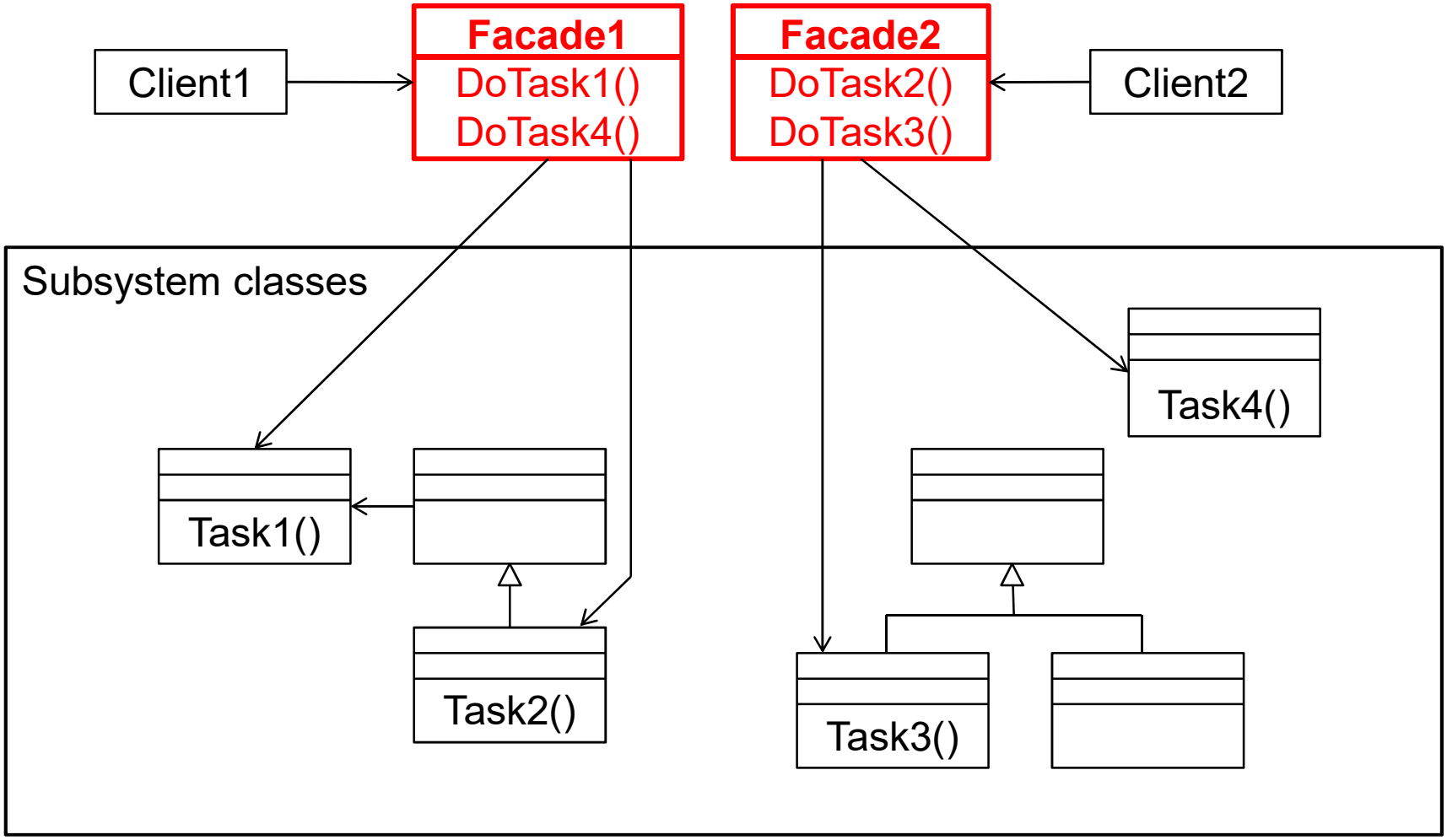


Façade Class

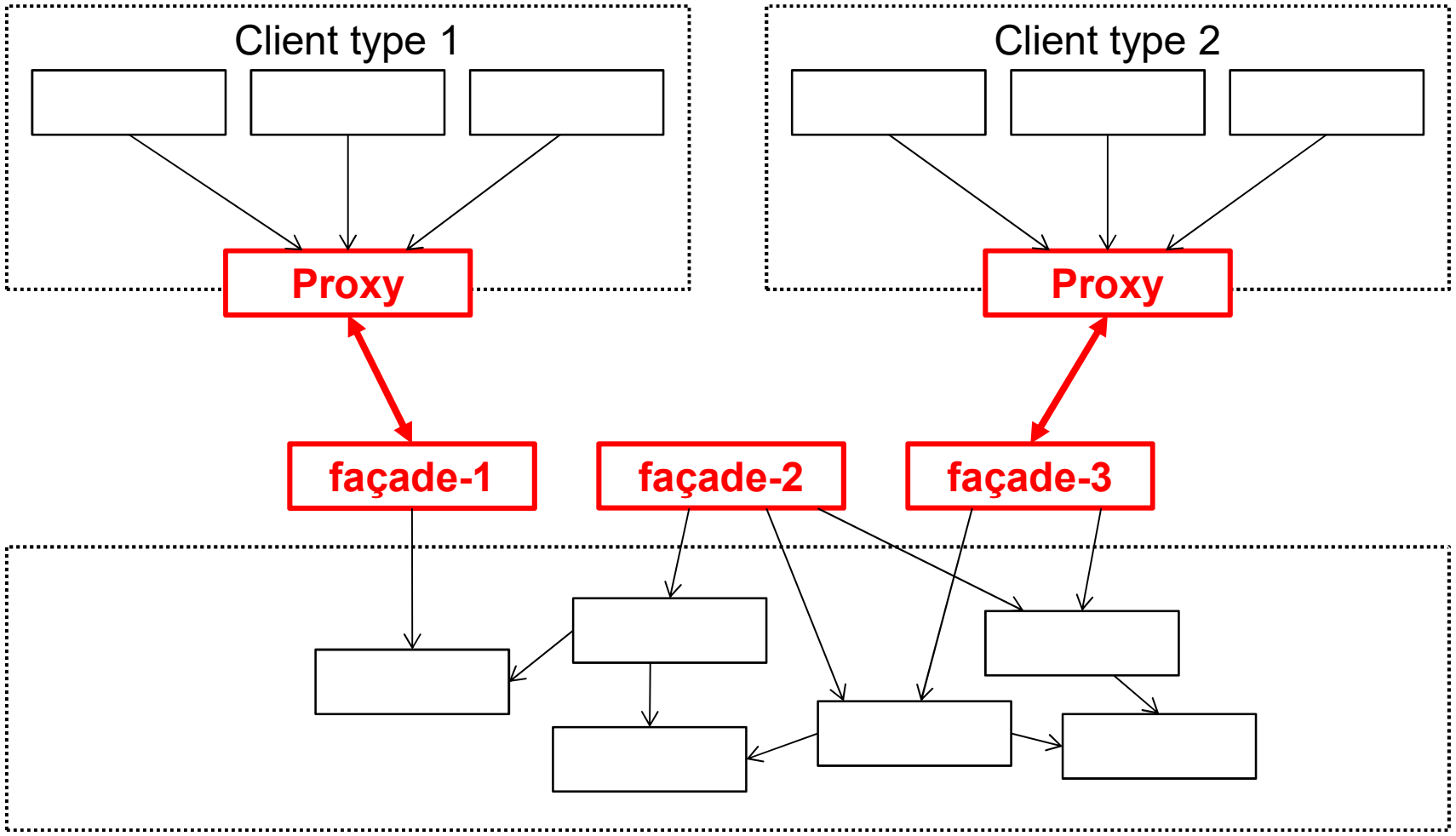
- 單一窗口，多種服務 (想像到**郵局**、**銀行**或**戶政事務所**辦事的情境)
- Provide customized access point for a specific client.
- Façade layer contains service interfaces.
- Service interface contains façade classes.



Façade Pattern

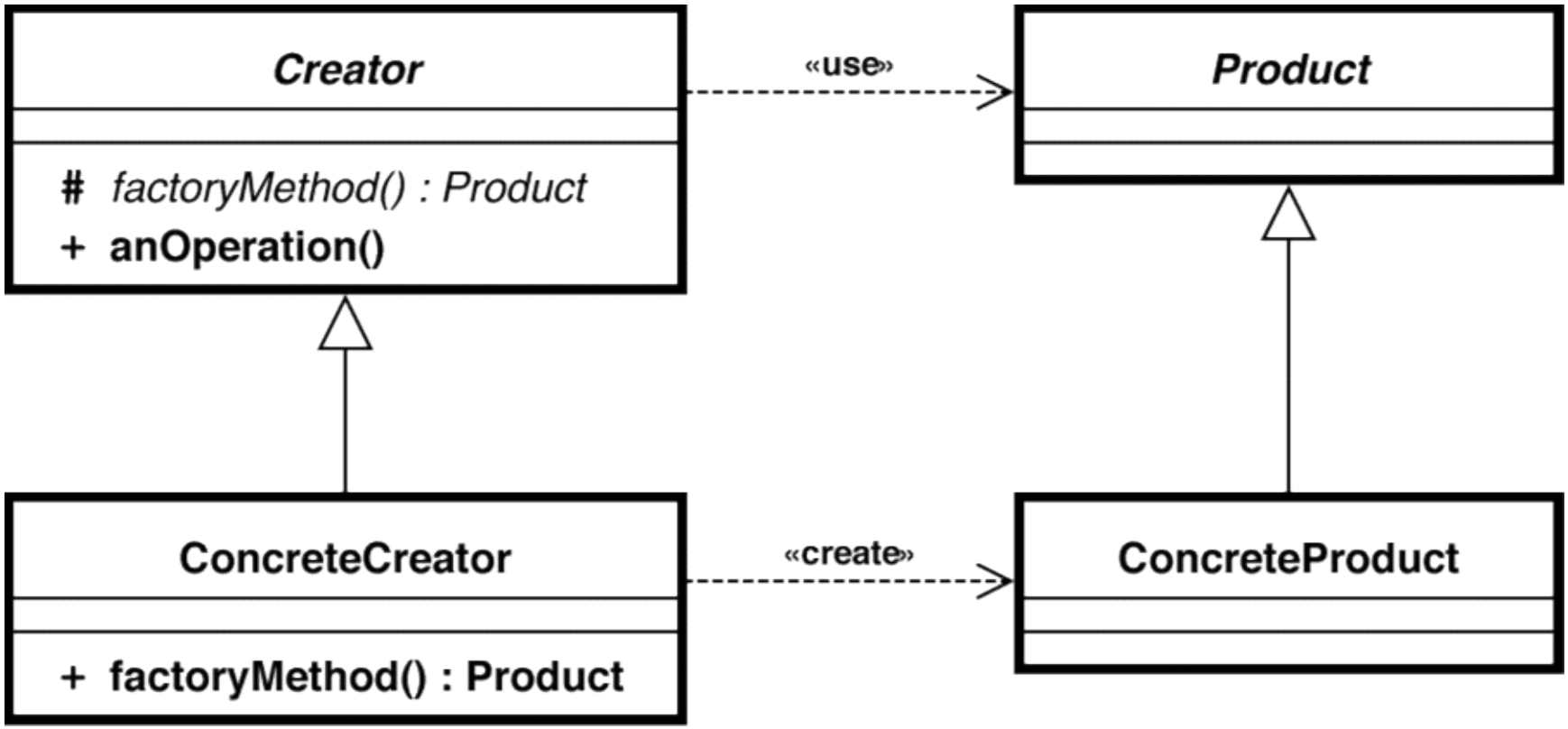


Combine Client Proxy & Façade



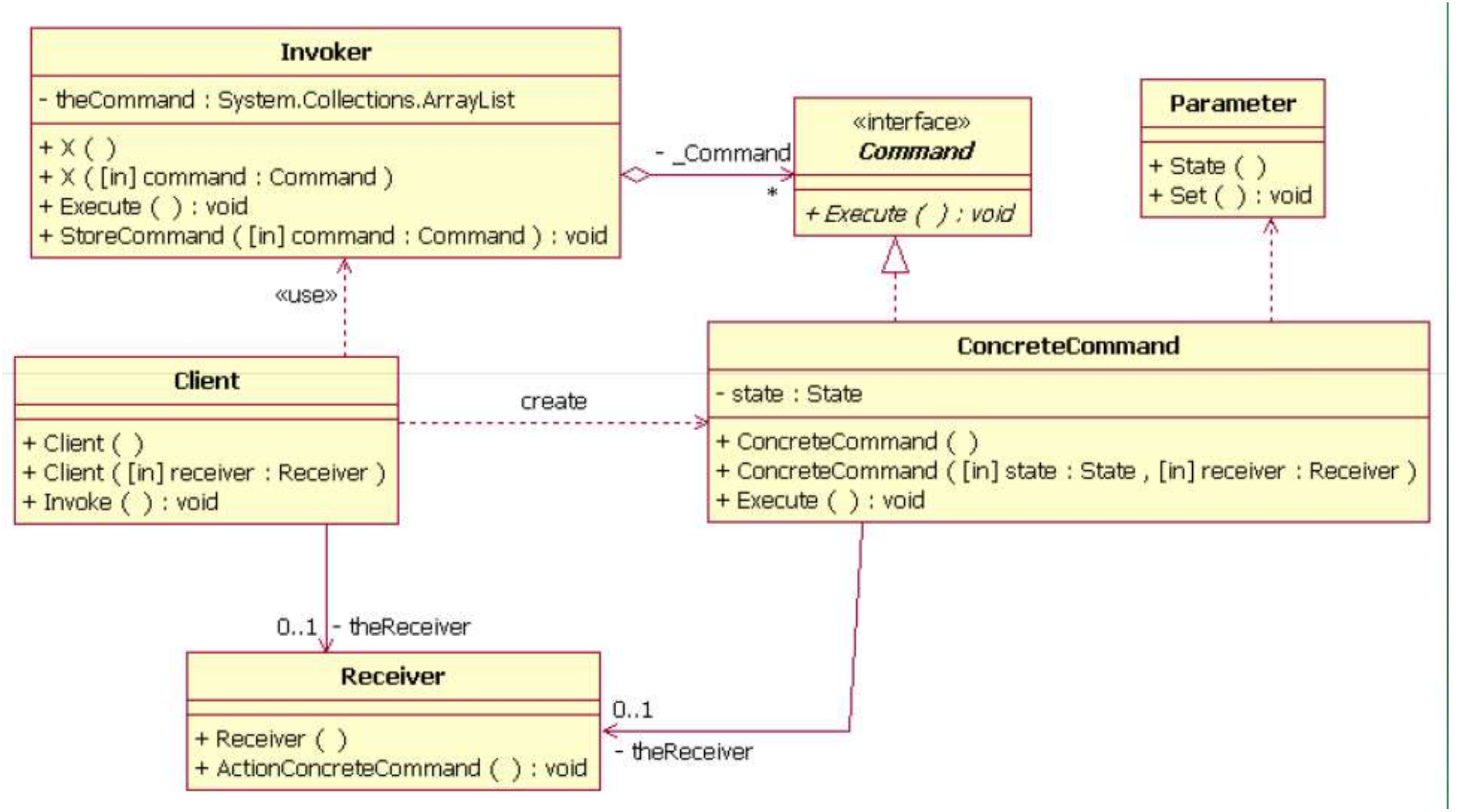
- **Objectives:**
  - Define an interface for creating an object, but let subclasses instantiate.
- **Solution:**
  - Use the **Template Method** pattern to split object creation out of a larger function and place it into a separate factory method "hook" function.
  - The specific "`ConcreteProduct`" objects which may be created by the factory methods must all be subclasses of a common "`Product`" ancestor.
  - Subclasses override the factory.
- **Consequences:**
  - Always more flexible than creating an object directly.

Factory Method Pattern - Structure



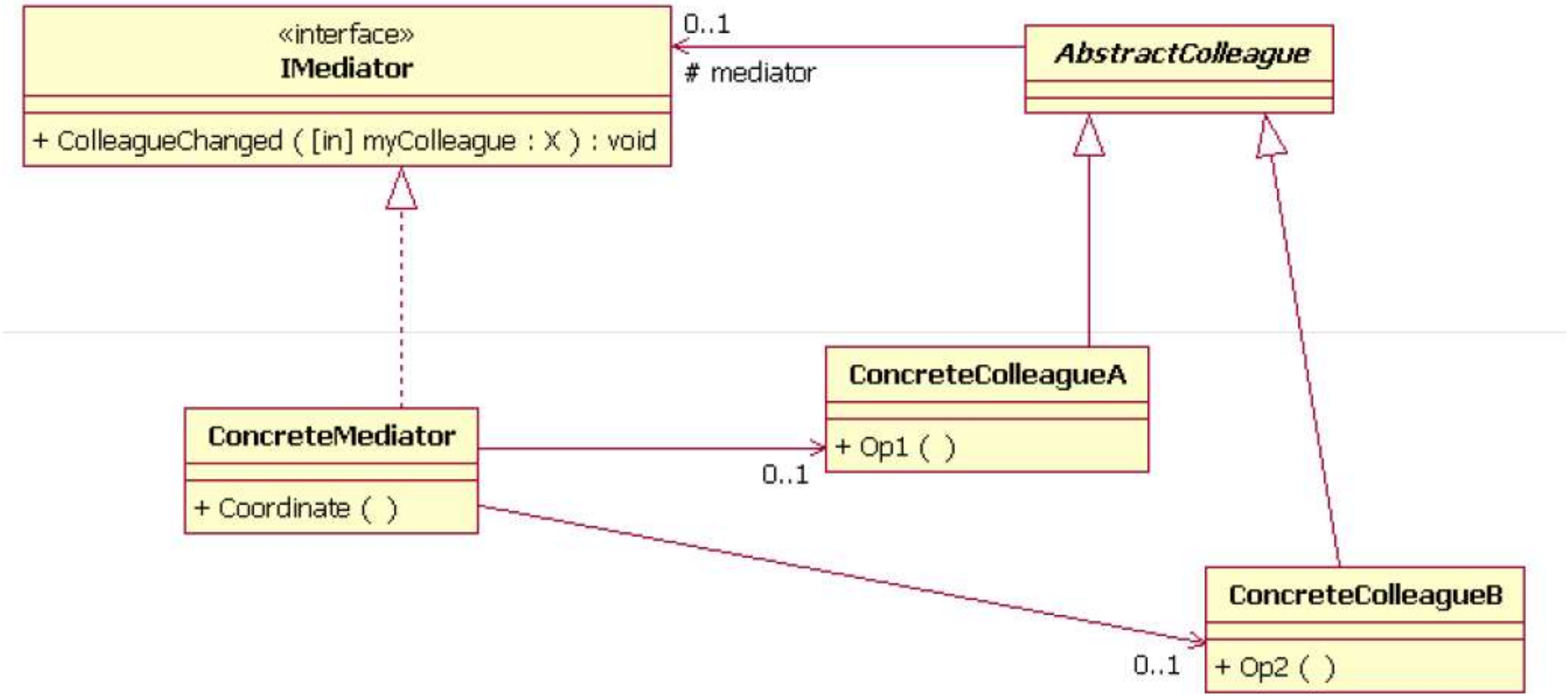
- **Objectives:**
  - Encapsulate a request as a parameterized object;
  - Allow different requests, queue or log requests and support undoable operations.
- **Solution:**
  - Client creates commands as needed, specifying the Receiver object and parameters the command will use to execute later.
  - Each Specific Command is inherited from a common abstract class which defines the basic execute interface Invoker uses to trigger commands when needed.
  - Each Specific Command knows the Reciever it works with and the parameters it needs to perform its encapsulated action by calling methods in its associated Reciever

Command Pattern - Structure



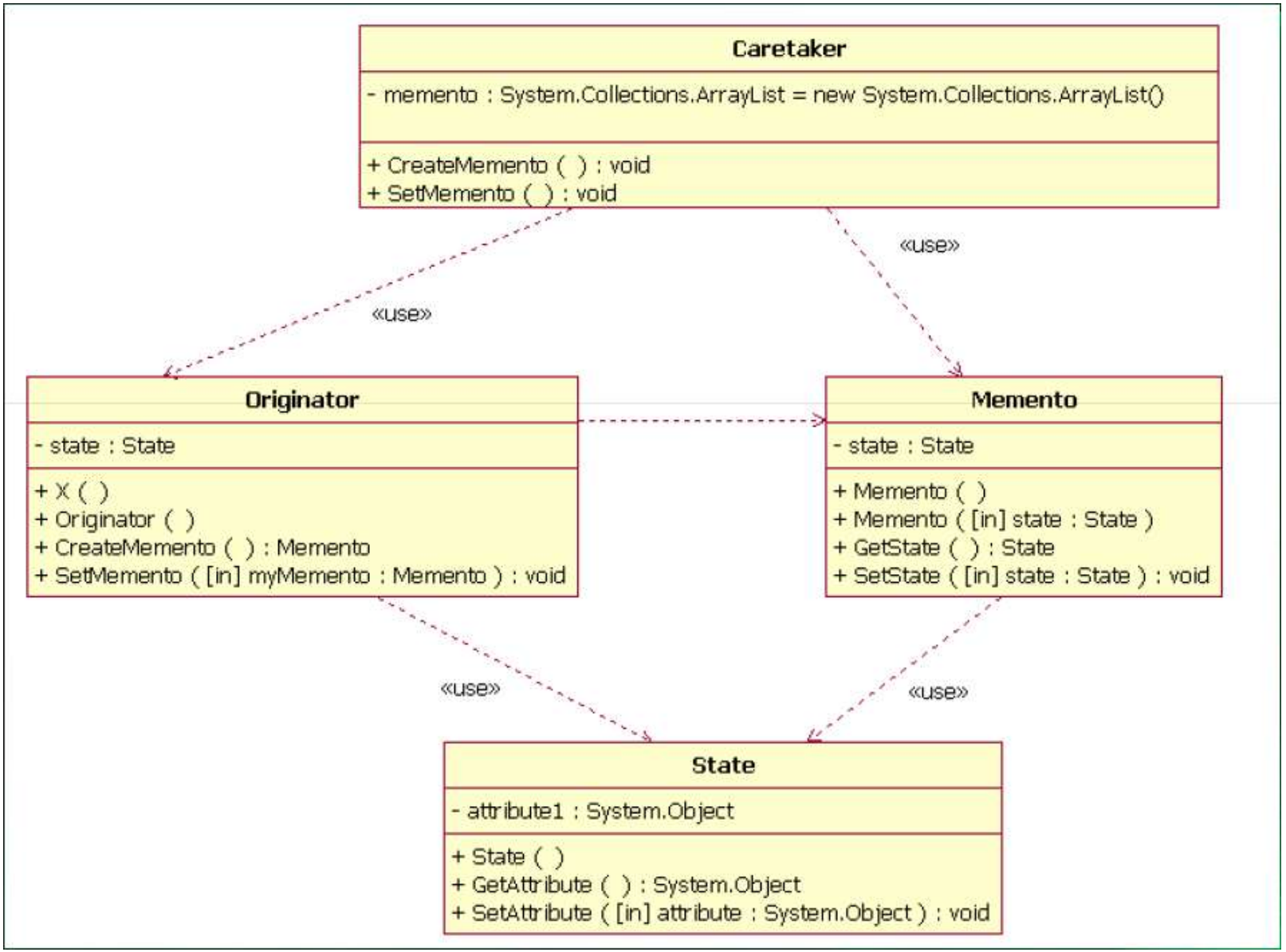
- Objectives:
  - Decouple flow coordination and logic objects
- Solutions:
  - Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently

Mediator Structure

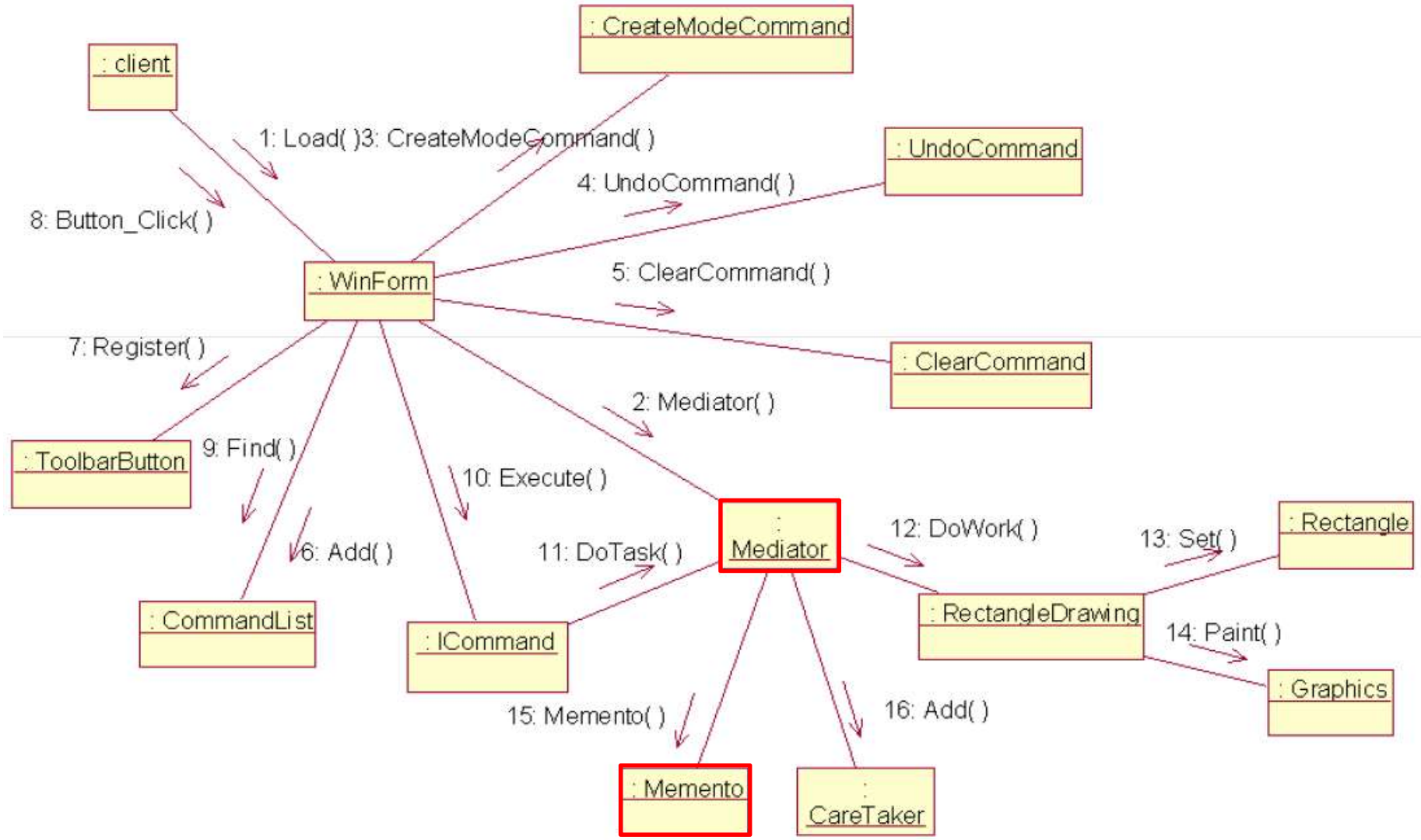


- **Objectives:**
  - Capture and externalize an object's state so that the object can be restored to that state later, without violating encapsulation
- **Solution:**
  - Use a memento object to store snapshot of an object's state so you can restore it to that state again later
  - Create a care-taker object to maintain change history

Memento Pattern Structure



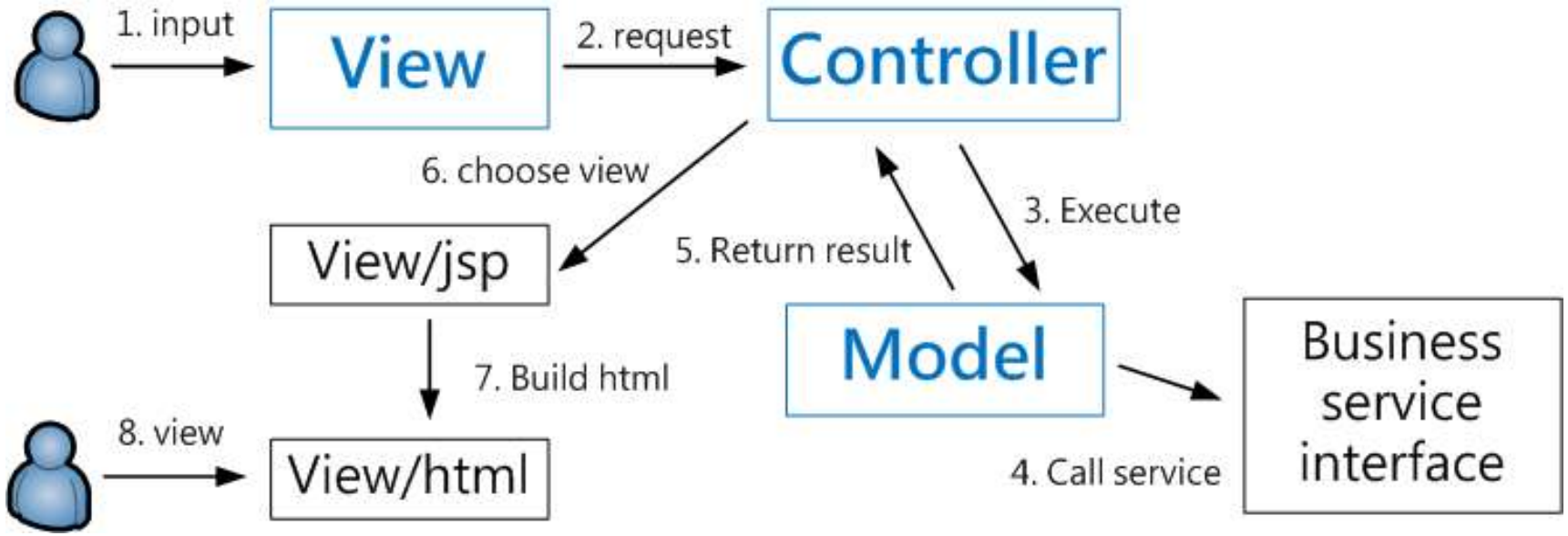
Memento & Mediator Pattern Sample



- Fundamental design pattern

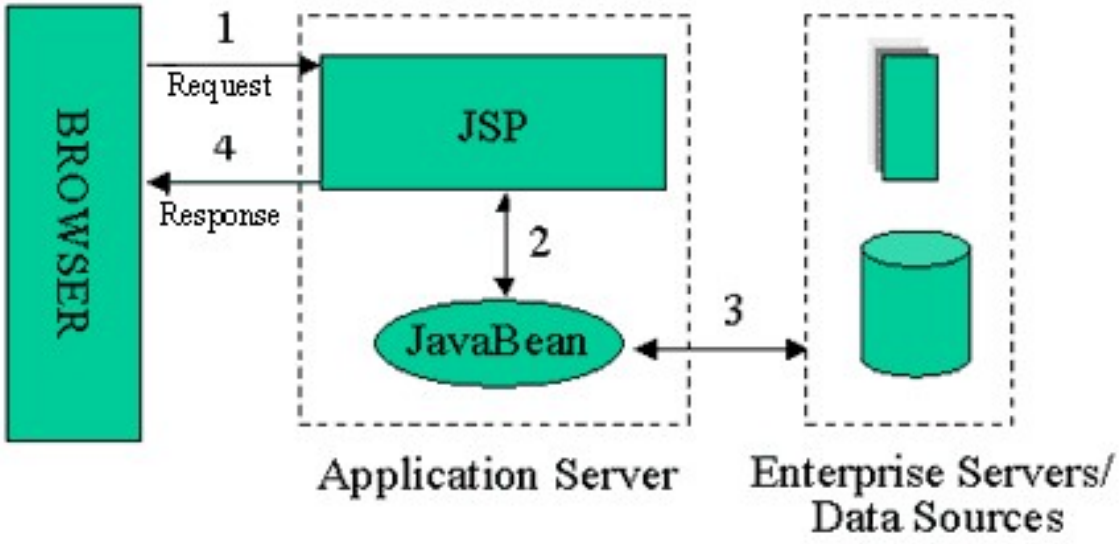
⇨ **Platform specific design pattern**

- Java implementation
  - Model: java class
  - View: jsp, html and custom tags
  - Controller: servlet

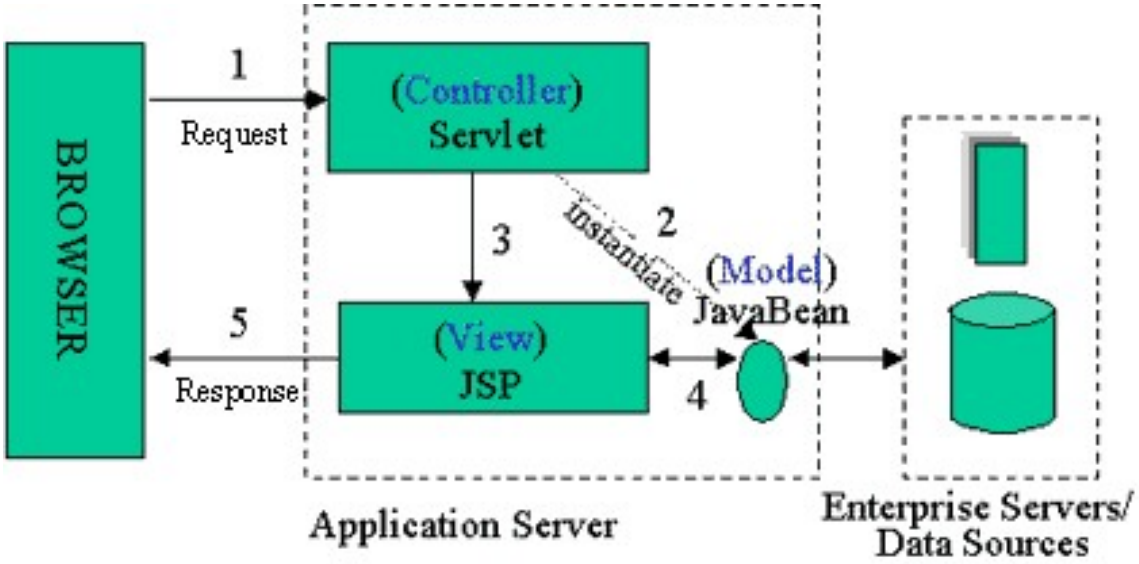


Model 1 and Model 2 (MVC) Architecture

Model 1

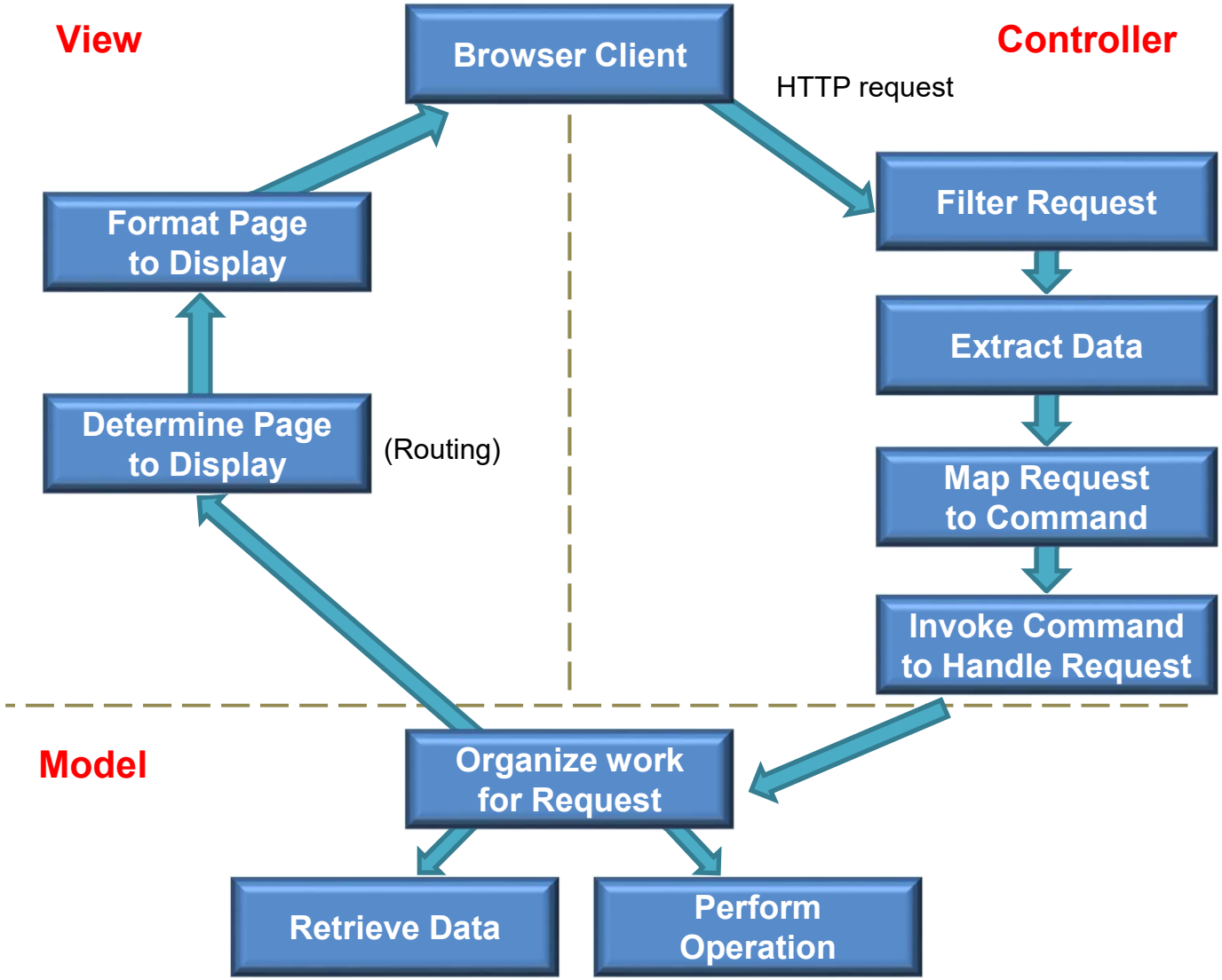


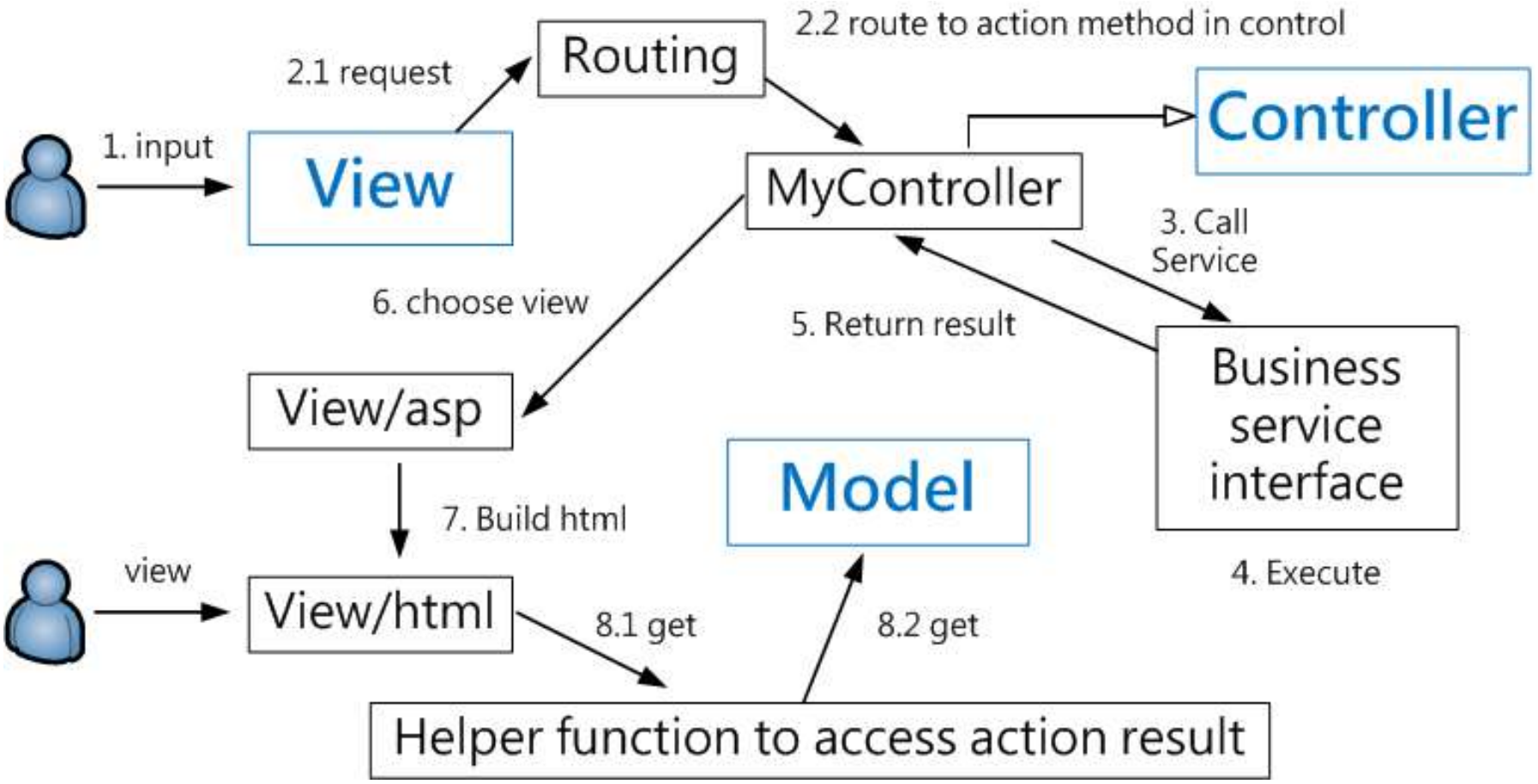
Model 2 (MVC)



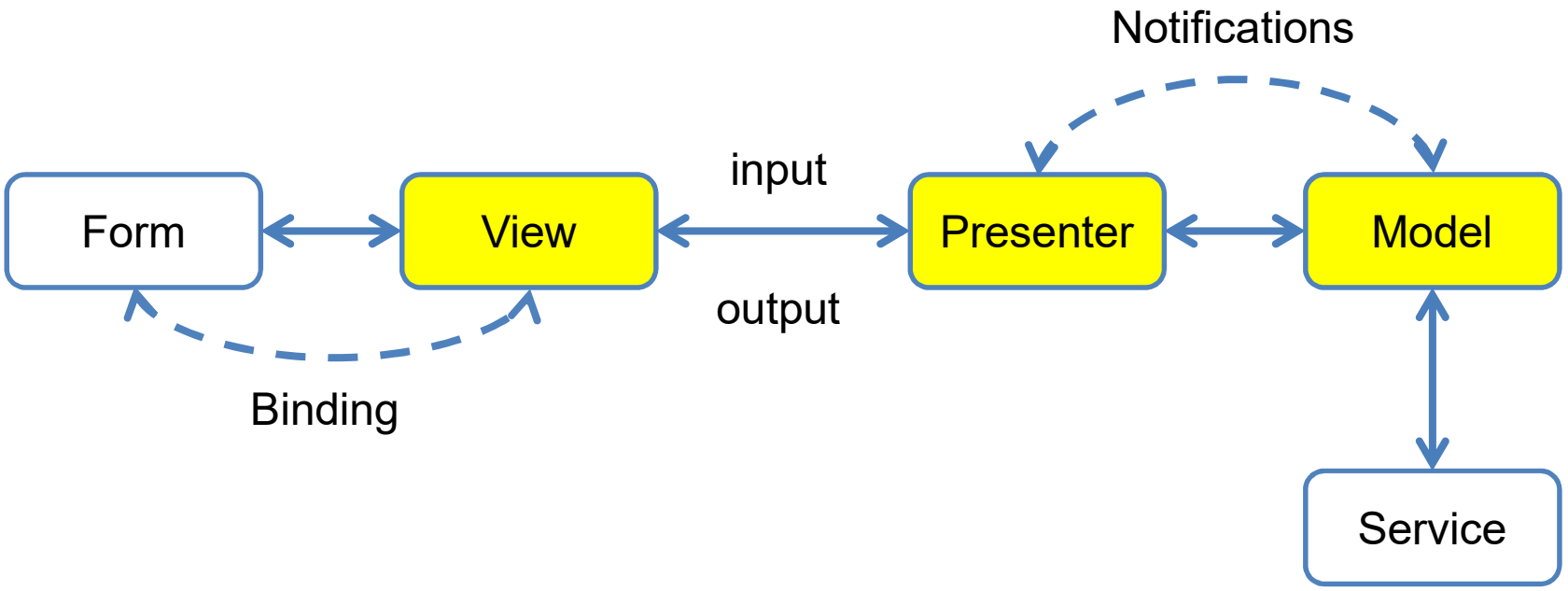
https://www.javaworld.com/article/2076557/java-web-development/understanding-javaserver-pages-model-2-architecture.html

Java Web MVC design pattern (Model 2)





- Model-View-Presenter



7. DESIGN RESILIENCE

- **Cohesion and Coupling**
- **Object-oriented design - SOLID:**
  1. **S**ingle responsibility
  2. **O**pen-closed
  3. **L**iskov substitition
  4. **I**nterface segregation
  5. **D**ependency inversion

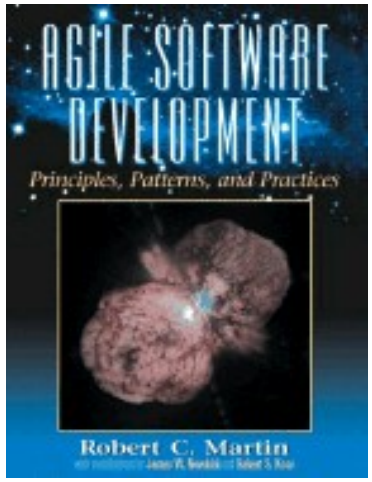
PIC

- Cohesion – How closely related are the different responsibilities of module.
- Coupling – How much one module relies on another.
- Goal - **Low Coupling and High Cohesion.**

THERE SHOULD NEVER BE MORE THAN ONE REASON FOR A CLASS TO CHANGE.

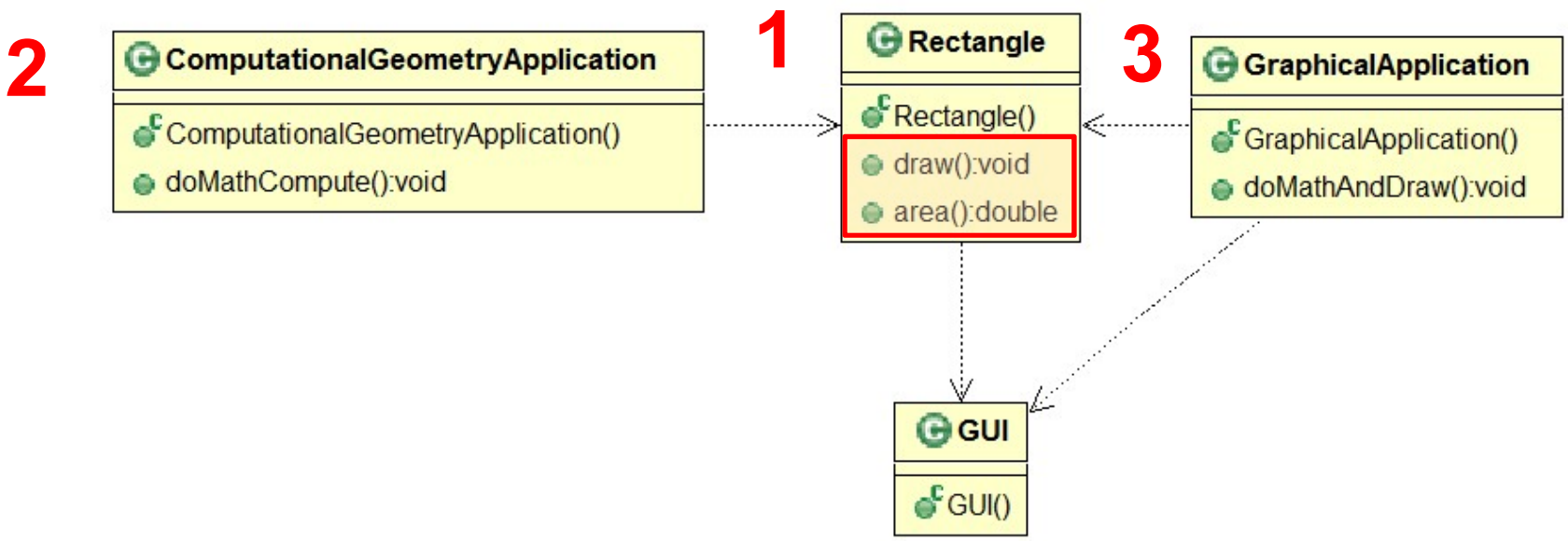
- Responsibility is what a class does.
- The more a class does, the more likely it will change.
- The more a class changes, the more likely we will introduce bugs.

Robert C. Martin



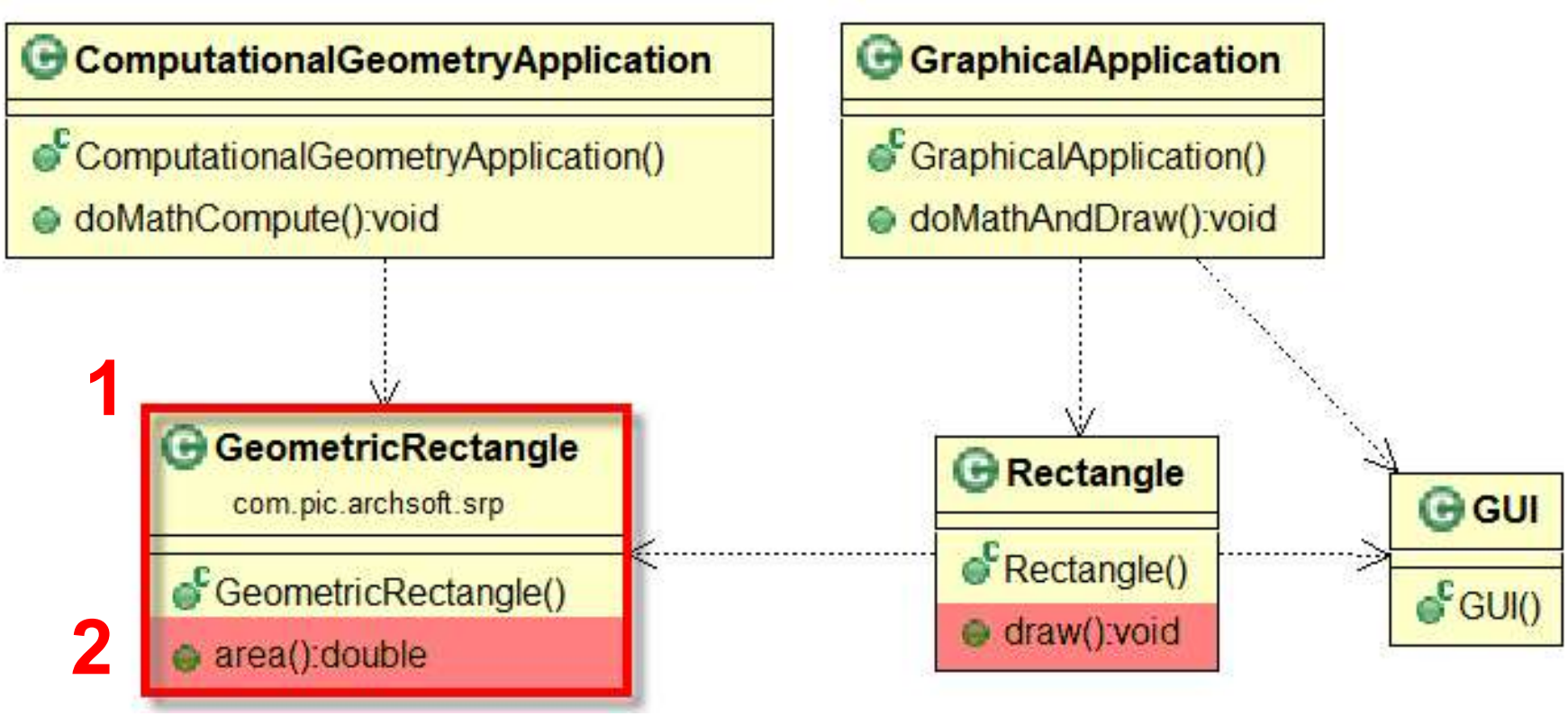
Agile Software Development: Principles, Patterns and Practices

SRP Violation – Multiple Responsibilities



1. **Rectangle**
   有一個用來繪製矩形的方法 **draw()** 以及一個用來計算面積的方法 **area()** ，且同時被兩支程式使用，其中在 **draw()** 也使用到 **GUI** 物件。
2. **ComputationalGeometryApplication**
   使用 **Rectangle** 做一些數學計算。
3. **GraphicalApplication**
   使用 Rectangle 進行數學計算並於螢幕上繪製矩形。

SRP Fix



1.新建一個 **GeometricRectangle** 的類別。

2. 將 **area()** 移到 **GeometricRectangle** 內。

3. 原 **Rectangle** 只留下 **draw()** 。

Why SRP matters

- We want the code to be reused.
- Big classes are harder to read.
- Big classes are more difficult to change.

Smaller classes and smaller methods will give you more flexibility, and you don't have to write much extra code (if any) to do it!

When violating SRP might be OK

- The violating class is **not going to be reused** and other classes don't depend on it.
- The violating class does not have **private** fields that store values that the class uses.
- Your common sense says so.
- Example: ASP .NET MVC controller classes, web services

PIC

SRP - Small Methods

- A method should have one purpose.
- Easier to read and write, which means you are less likely to write bugs.
- Write out the steps of a method using plain English method names.

Small Methods – Before

```
public void SubmitOrder(Order order) {
    // validate order
    if (order.Products.count == 0) {
        throw new InvalidOperationException("Select a product");
    }
    // calculate tax
    order.Tax = order.SubTotal * 1.0675;

    // calculate shipping
    if (order.Subtotal < 25) {
      order.ShippingCharges = 5;
    } else {
      order.ShippingCharges = 10;
    }

    // submit order
    _orderSubmissionService.SubmitOrder(order);
}
```

Small Methods – After


```
public void ValidateOrder(Order order) {
    if (Order.Products.Count == 0)
        throw new InvalidOperationException("Select a product");
}
public void CalculateTax(Order order) {
    order.Tax = order.Subtotal * 1.0675;
}
public void CalculateShipping(Order order) {
    if (order.Subtotal < 25) {
        order.ShippingCharges = 5;
    } else {
        order.ShippingCharges = 10;
    }
}
public void SendOrderToOrderSubmissionService(Order order) {
    _orderSubmissionService.SubmitOrder(order);
}
public void SubmitOrder(Order order) {
    ValidateOrder(order);
    CalculateTax(order);
    CalculateShipping(order);
    SendOrderToOrderSubmissionService(order);
}
```

PIC

Open Closed Principle (OCP)

Software entities (class, modules, methods, etc.) should be open for extension but closed for modification.

Bertrand Meyer

Object-Oriented Software Construction

OCP Violation

```
public class GetUserService
{
    public Ilist<UserSummary> FindUsers(UserSearchType type)
    {
        Ilist<User> users;
        switch (type) {
            case UserSearchType.AllUsers:
                // load the "users" variable here
                break;
            case UserSearchType.AllActiveUsers:
                // load the "users" variable here
                break;
            case UserSearchType.ActiveUsersThatCanEditQuotes:
                // load the "users" variable here
                break;
        }
        return ConvertToUserSummaries(users);
    }
}
```

OCP Fix – Strategy Pattern

```
public interface IUserQuery
{
    Ilist<Users> FilterUser(Ilist<User> allUsers);
}

public class GetUserService
{
    public Ilist<UserSummary> FindUsers(IUserQuery query)
    {
        Ilist<User> users = query.FilterUsers(GetAllUsers());
        return ConvertToUserSummaries(users);
    }
}
```

Why OCP matters

- Anytime you change code, you have the potential to break it.
- Sometimes you can't change libraries (e.g. code that isn't yours).
- May have to change code in many different places to add support for a certain type of situation.

PIC

When violating OCP is OK

- When the number of options in the `if` or `switch` statement is unlikely to change (e.g. switch on `enum`) .

```
public void UpdateFontStyle (Paragraph paragraph) {
    switch (IsBoldCheckBox.Checked) {
        case CheckState.checked:
            paragraph.IsBold = true;
            break;
        case CheckState.Unchecked:
            paragraph.IsBold = false;
            break;
        case CheckState.Indeterminate:
            break;
    }
}
```

OCP Rules of Thumb

- Use `if/switch` if the number of cases is unlikely to change
- Use **strategy pattern** when the number of cases are likely to change
- Always use common sense!

Liskov Substitution Principle (LSP)

- *Functions that use references or pointers to base classes must be able to use objects of derived classes without knowing it.*
- *People using derived classes should not encounter unexpected results when using your derived class.*

Barbara Liskov

LSP Violation – Bad Abstraction

```
class Bird {
  public void fly(){ … }
  public void eat(){ … }
}

// 烏鴉
final class Crow extends Bird {}

// 鴕鳥
final class Ostrich extends Bird {
  public void fly() {
    throw new UnsupportedOperationException();
  }
}
```

LSP Violation – Bad Abstraction


```
public BirdTest {

  public static void main(String[] args) {
    List<Bird> birdList = new ArrayList<Bird>();
    birdList.add(new Bird());
    birdList.add(new Crow());
    birdList.add(new Ostrich());
    letTheBirdsFly ( birdList );
  }

  public static void letTheBirdsFly( List<Bird> birdList ) {
    for ( Bird b : birdList ) {
      b.fly();
    }
  }

}
```

Factor out the fly feature into - FlightBird and NonFlightBird:

```
public class Bird {
  public void eat(){…}
}

public class FlightBird extends Bird {
  public void fly(){…}
}

public class NonFlightBird extends Bird {}

public final class Crow extends FlightBird {}
public final class Ostrich extends NonFlightBird {}
```

※ 盡量讓使用你元件的開發者在編譯時期就能發現錯誤。

Interface Segregation Principle (ISP)

Clients should not be forced to depend upon interfaces that they do not use.

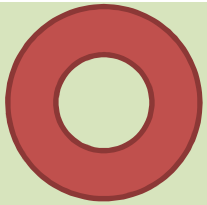
Robert C. Martin

ISP violation


```
public interface IWorker {
    public void work();
    public void eat();
}
```

```
public class HumanWorker implements IWorker {
    public void work () {
        // working....
    }
    public void eat() {
        // eating....
    }
}

public class RobotWorker implements IWorker {
    public void work() {
        // working...
    }
    public void eat() {
        throw new UnsupportedOperationException();
    }
}
```

ISP fix solution 1 – 分離成兩種介面

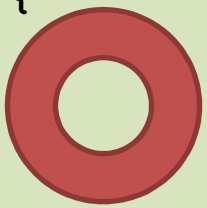


```
public interface IWorkable {
    public void work();
}

public interface IFeedable {
    public void eat();
}
```

```
public class HumanWorker implements IWorkable, IFeedable {
    public void work () {
        // working....
    }
    public void eat() {
        // eating....
    }
}

public class RobotWorker implements IWorkable {
    public void work() {
        // working...
    }
}
```

ISP fix solution 2 – 使用「Adapter」Design Pattern


```
abstract public class HumanWorkerAdpater implements IWorker {
   public void work();
   public void eat();
}

abstract public class RobotWorkerAdpater implements IWorker {
   public void work() {}
}
```



```
IWoker HumanWorker = new HumanWorkerAdpater() {
   public work() {
      // working....
   }
   public eat() {
      // eating....
   }
}

IWoker RobotWorker = new RobotWorkerAdpater() {
   public work() {
      // working....
   }
}
```

ISP Smells

- If you implement an interface or derive from a base class and you have to throw an **exception** in a method because you don't support it, the interface is probably too **FAT**.

PIC

When violating ISP is OK

- Why would you want to?
- Use common sense

Dependency Inversion Principle (DIP)

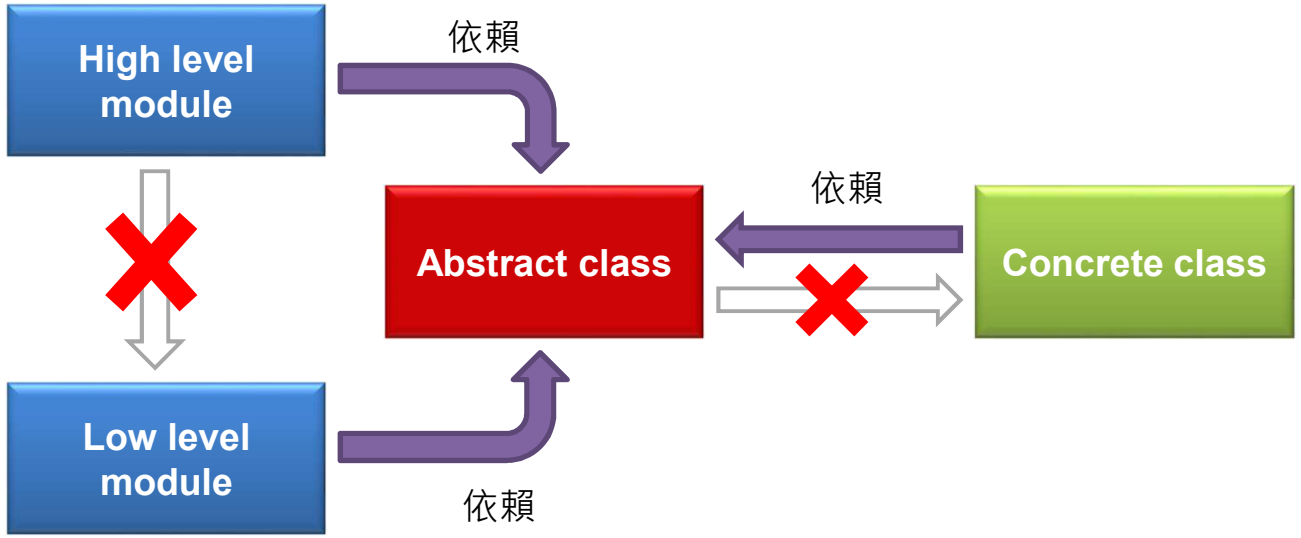
High level modules should not depend on low level modules. Both should depend on abstractions.

Abstractions should not depend on details. Details should depend on abstractions.

Robert C. Martin

Dependency Inversion Principle (DIP)

1. Modules that encapsulate high level policy should ***not*** depend upon modules that implement details. Both modules should depend on the same ***abstraction***.
2. Abstract classes should ***not*** depend upon concrete classes; concrete classes should depend upon ***abstract classes***.
3. A good example of this principle is the ***TEMPLATE METHOD*** pattern from the GOF book.



DIP violation


```
public class FloppyWriter {
    public void saveToFloppy() {
        ....
        // 實際儲存至 Floppy 的程式碼
    }
}

public class BackUp {

    private FloppyWriter writer = new FloppyWriter();

    public void save() {
        // saving...
        writer.saveToFloppy();
    }

}
```

如果今天想要將存檔改為存至 USB 硬碟呢？

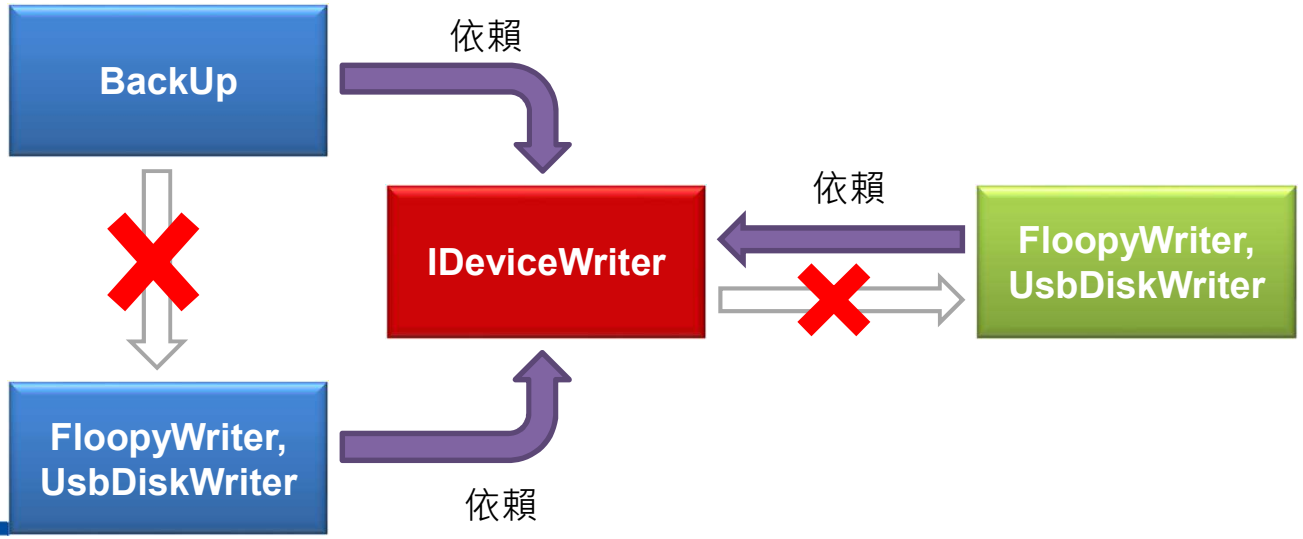
DIP fix


```
public interface IDeviceWriter {
    public void saveToDevice();
}

public class BackUp {
    private IDeviceWriter writer;

    public void setDeviceWriter(IDeviceWriter writer) {
        this.writer = writer;
    }

    public void save() {
        // saving....
        writer.saveToDevice();
    }
}
```

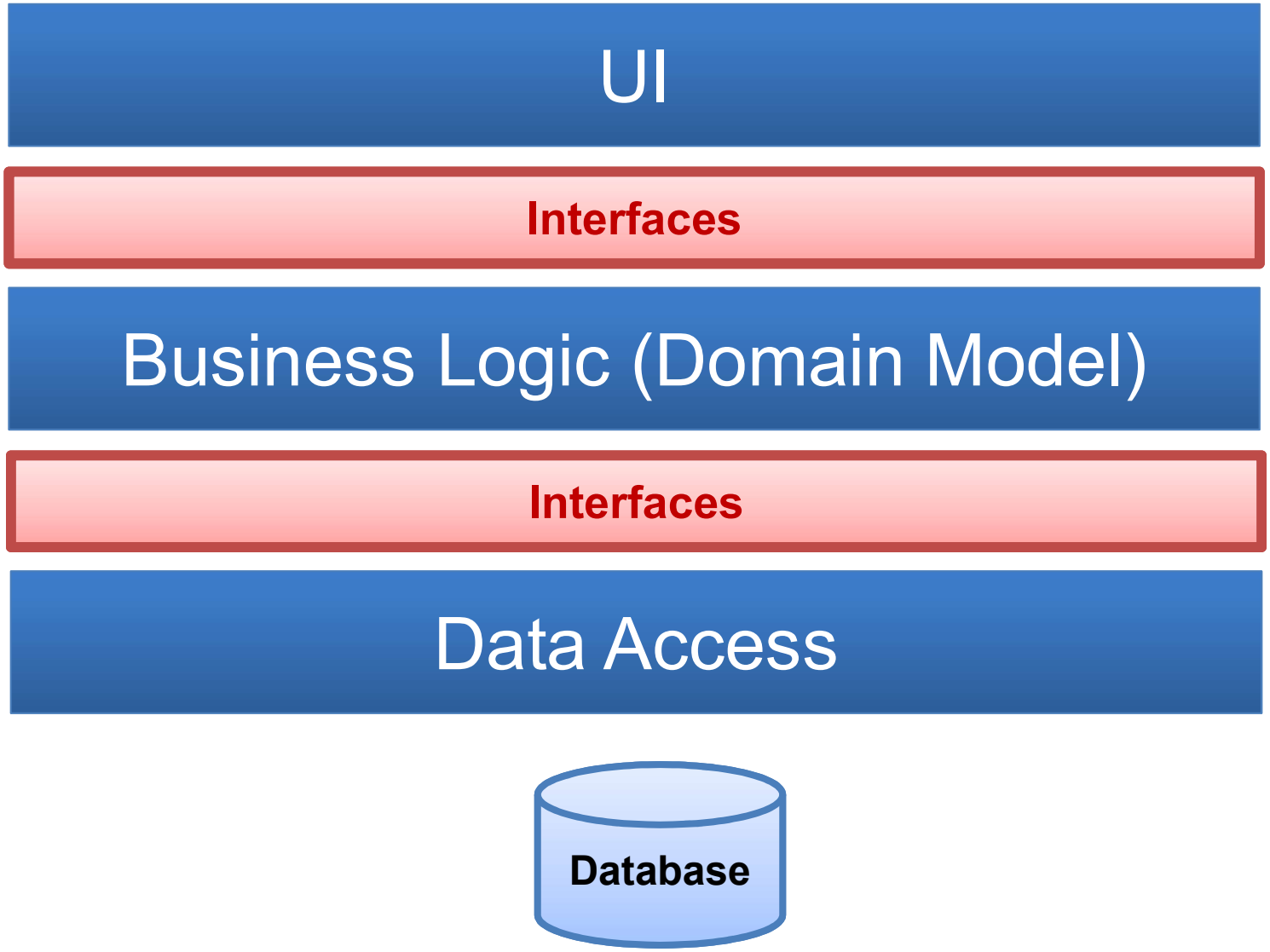
依賴注入

DIP fix


```
public class FloppyWriter implement IDeviceWriter {
    public void saveToDevice() {
        ....
        // 實際儲存至 Floppy 的程式碼
    }
}

public class UsbDiskWriter implement IDeviceWriter {
    public void saveToDevice() {
        ....
        // 實際儲存至 UsbDisk 的程式碼
    }
}
```



Tips for not violating DIP - Layers



What is DI (IoC) Container?

- **Create objects** that are ready for you to use.
- Know how to create objects and their **dependencies**.
- Knows how to **initialize** objects when they are created (if necessary).

- **.NET**
  - Autofac,
  - Unity,
  - Niject
  - StructureMap
- **Java**
  - Spring
  - Google Guice

Design Pluggable Framework

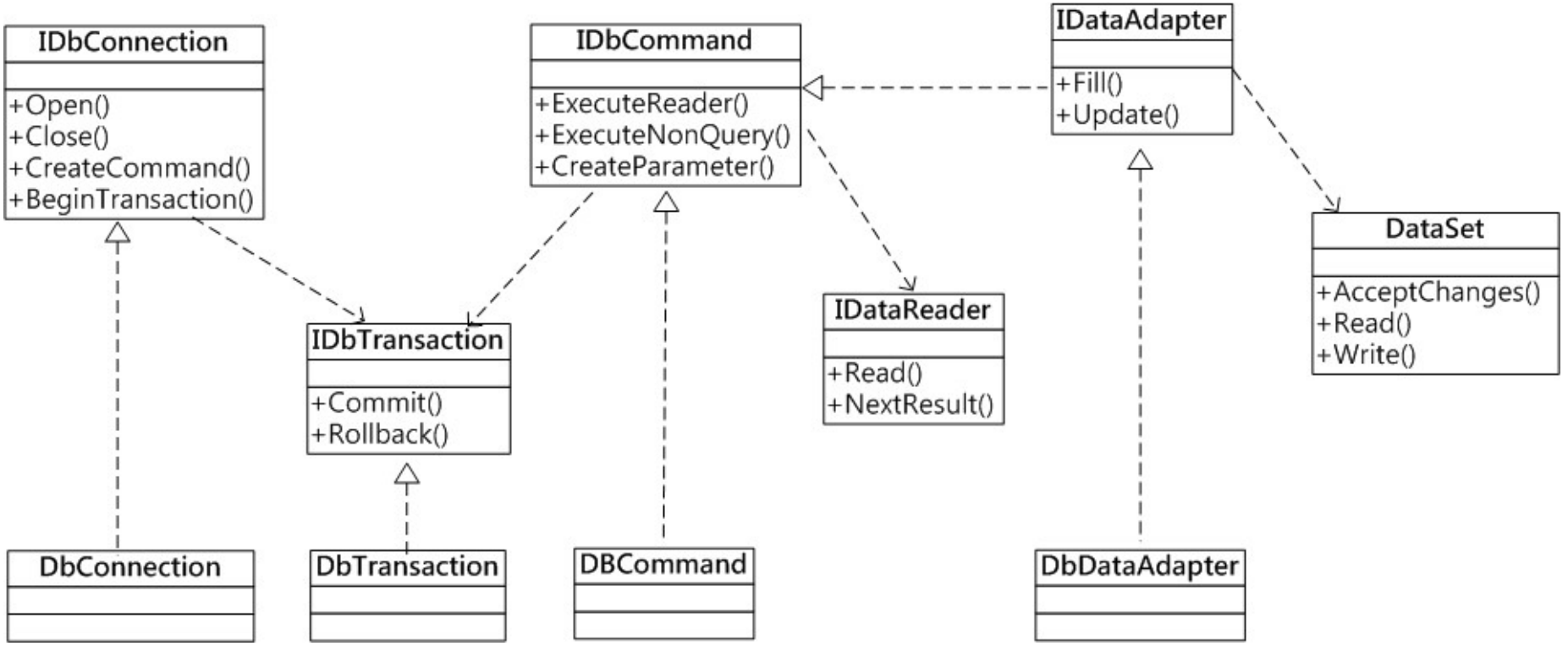
- Extensible
- Updatable
- Configurable

PIC

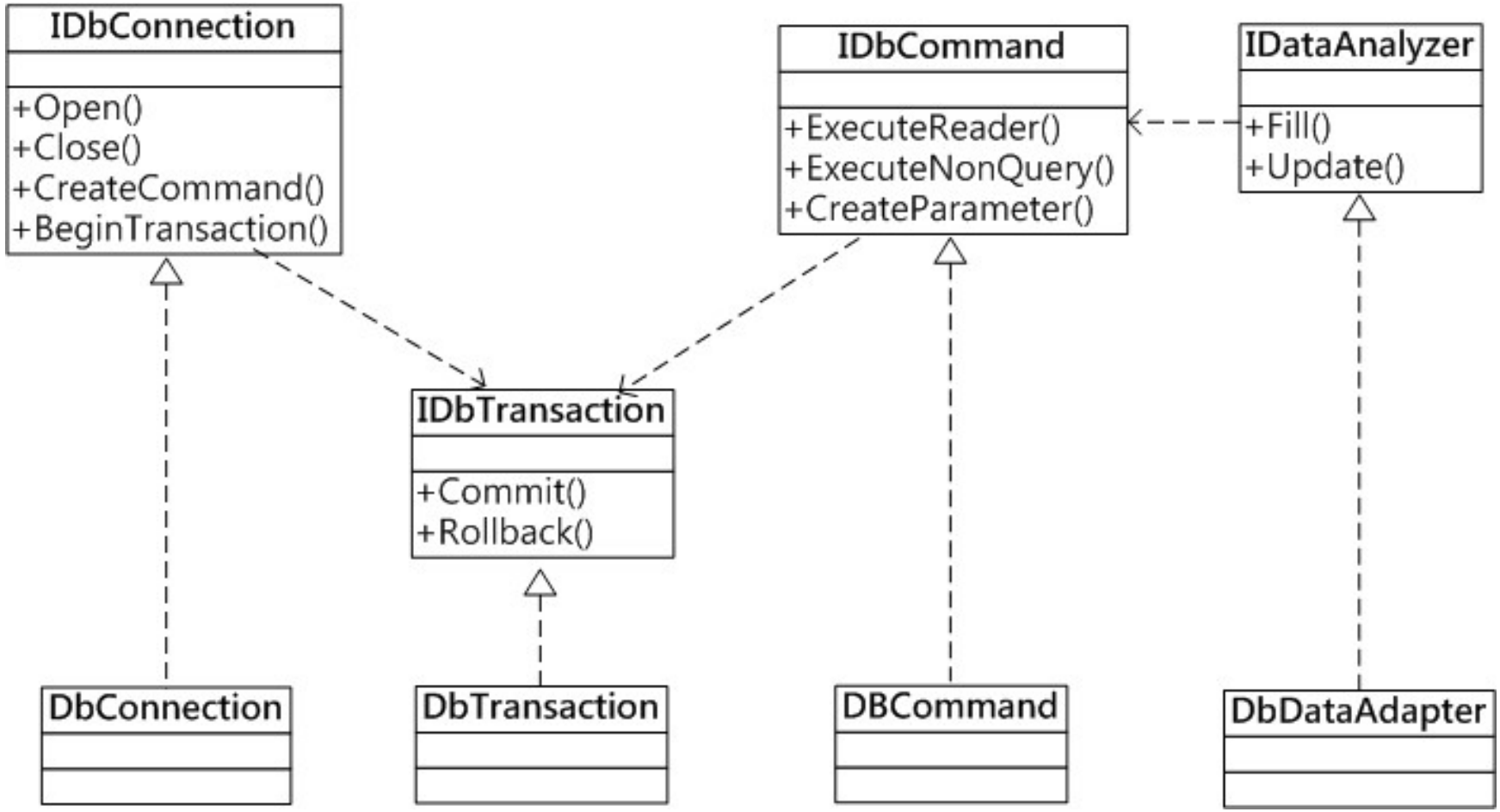
- Separate **interface** & **implementation**
- Define **stable** interfaces
- Use **Factory** design pattern
- Use **flexible** configuration system

PIC

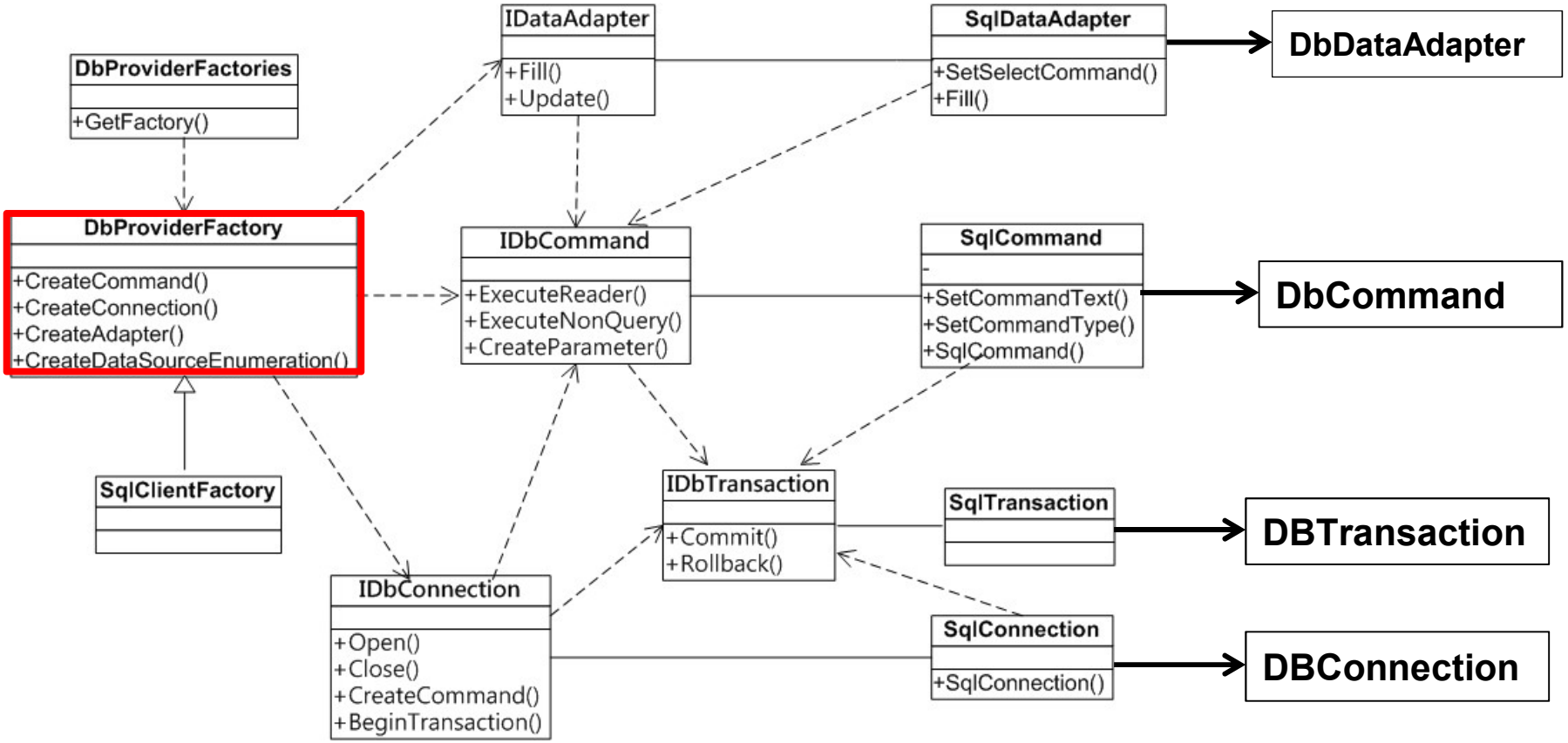
- ADO .net



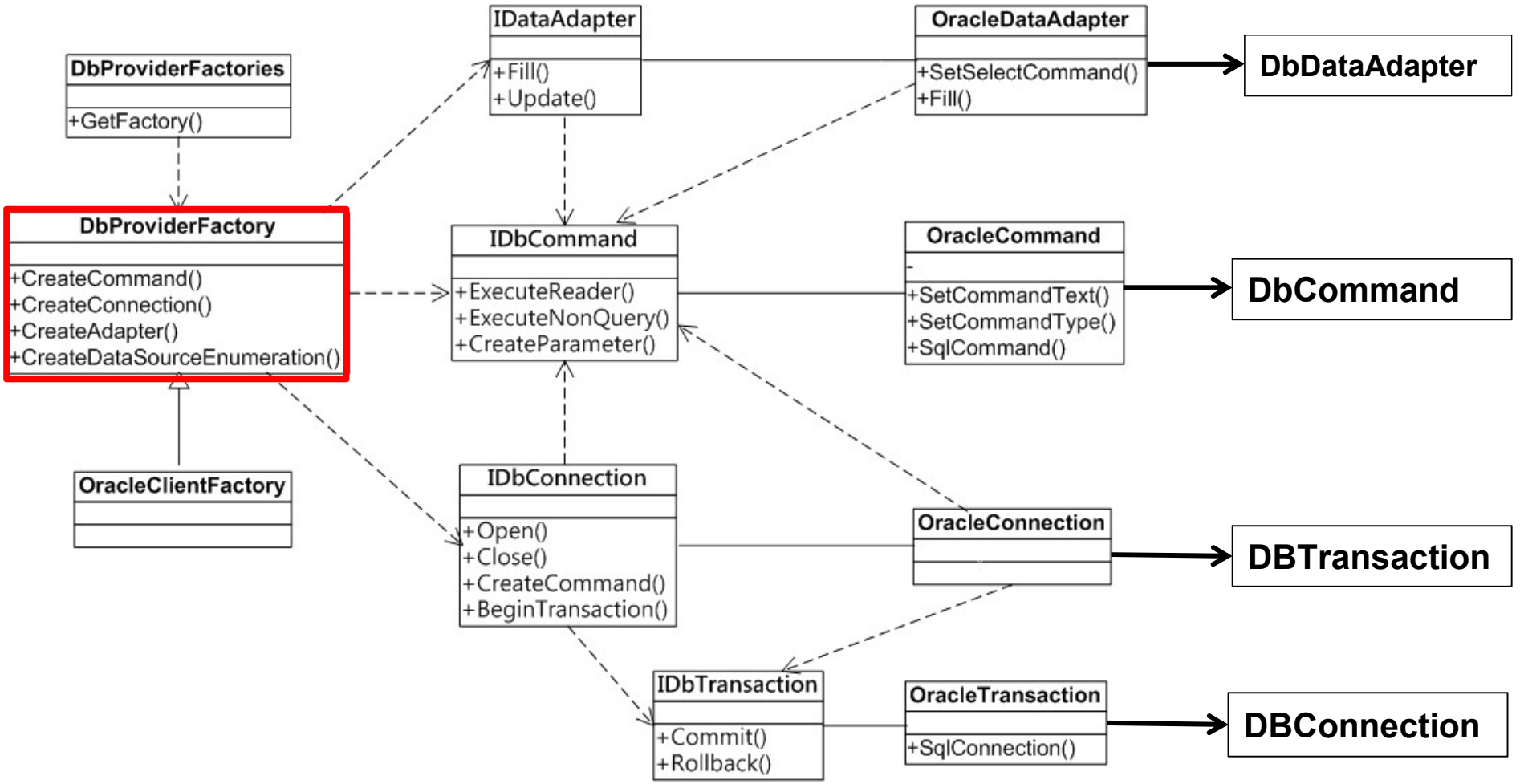
Separate Interface & Implementation



SqlServer Client Provider



OracleClient Provider



Don’t overcomplete!

- Don’t code for situation that you never need.
- Don’t create unneeded complexity.
- However, more class files != more complicated.
- Remember, this is supposed to make your lives easier! (but not easier to be lazy)

8. RUN-TIME ARCHITECTURE

- Design Process
- Case Study
- Microsoft HPC Technology

PIC

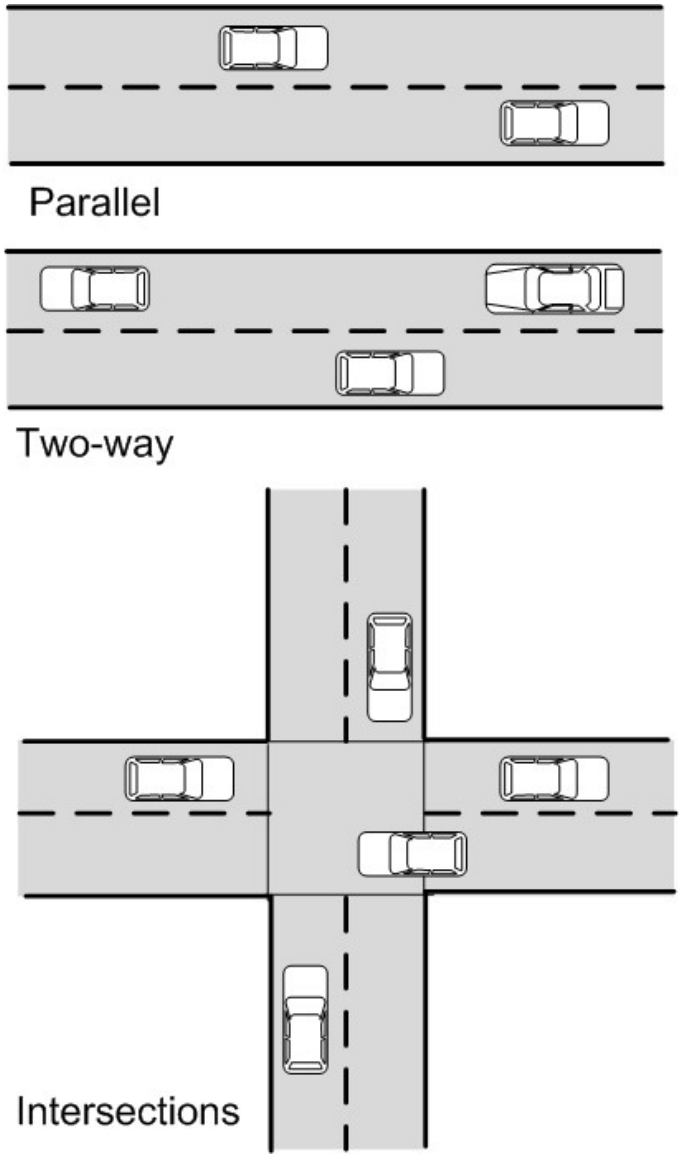
- Concurrency
- Processes and Threads
- Concurrency Control

What is Concurrency?

- The performance of two or more activities during the same time interval is greater than one by one.

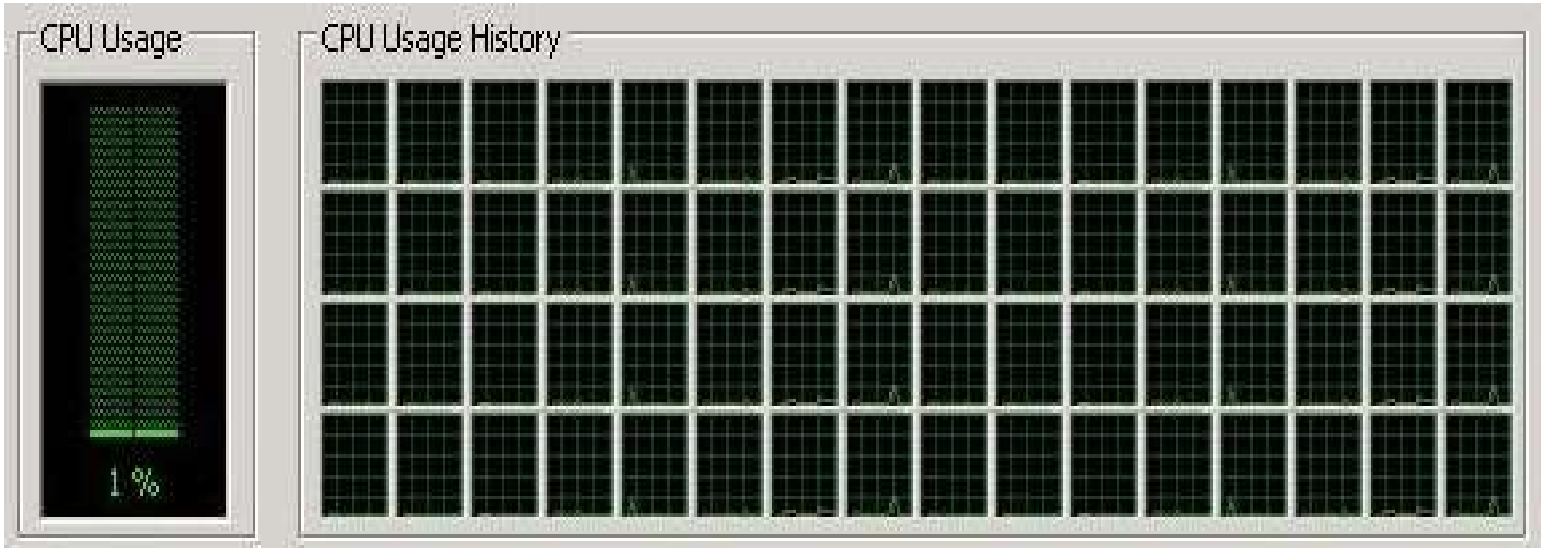
避開存取**共用資料**：

1. input parameter
2. local variables



Why Concurrency?

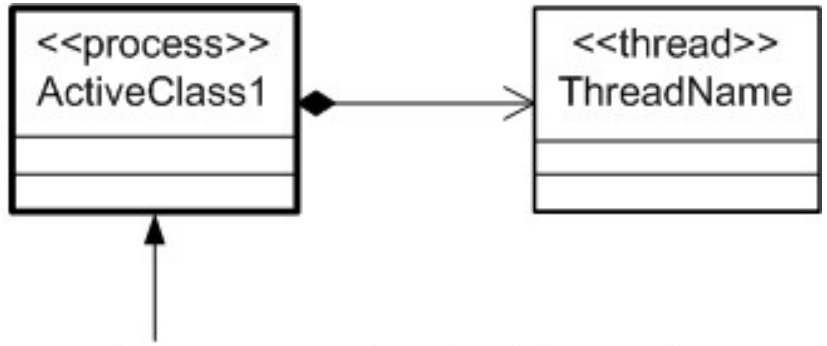
- Reactive software systems
- Optimized processing time
- Controllability of the system
- Permits a “separation of concerns” design



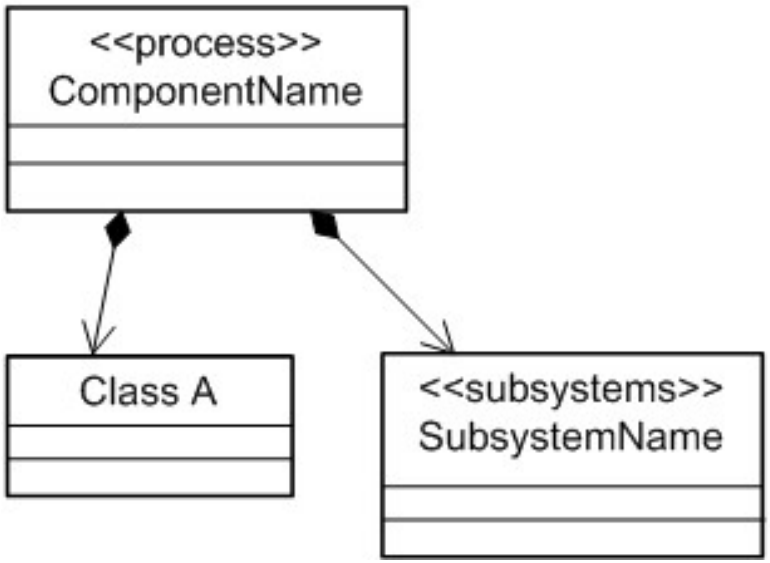
- **Multitasking**
  - The operating systems simulate concurrency on a single CPU by interleaving the execution of different tasks.
- **Multiprocessing**
  - Multiple CPUs execute concurrently.
- **Application-based solutions**
  - The application software takes responsibility for switching between different branches of code at appropriate times.

Modeling Processes

- Processes can be modeled using
  - Active classes (Class Diagrams) and Objects (Interaction Diagrams)
  - Components (Component Diagrams)
  - Stereotype <<process>>
- Process relationships can be modeled as dependencies
- Threads can be modeled using
  - Regular classes
  - Stereotypes <<thread>>
- Process to thread and process/thread to class/subsystem modeled as compositions



An active class is a class that "owns"its own thread of execution.
(this is not the standard UML2 representation)

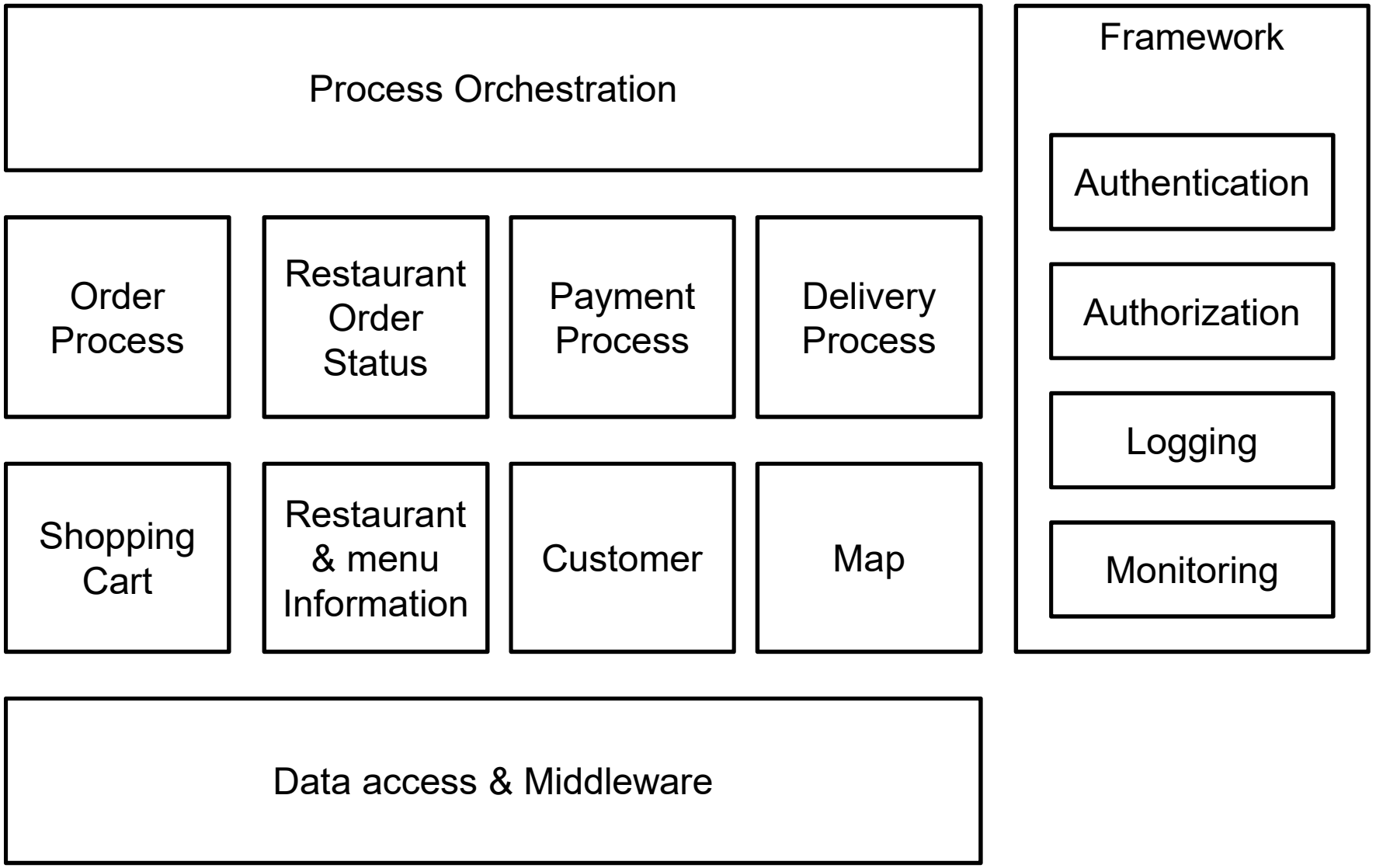


Concurrency Control

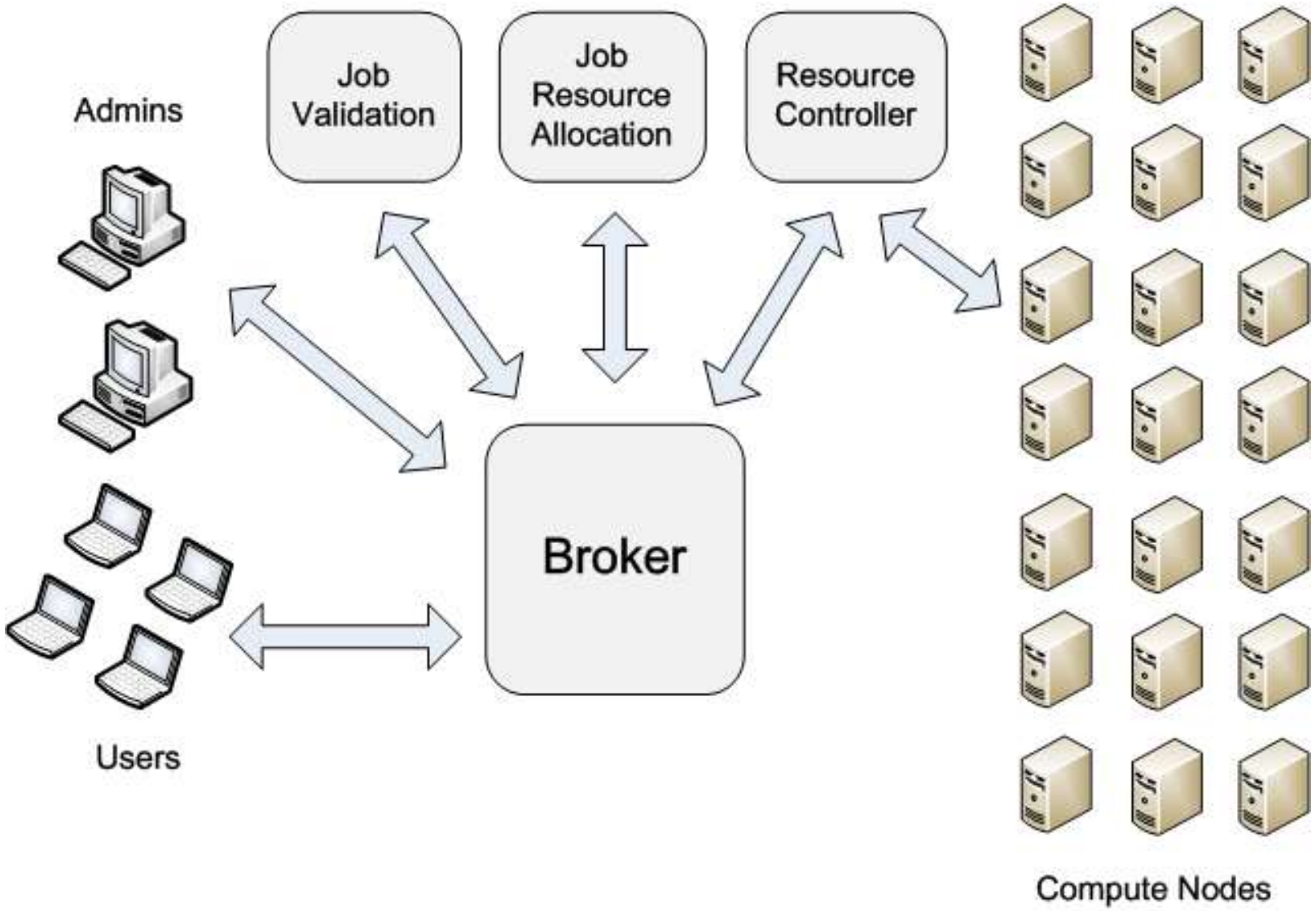
- Java provides two basic synchronization idioms:
  - Synchronized methods
  - Synchronized statements
- To make a method **synchronized**, simply add the **synchronized** keyword to its declaration.

```
public class SynchronizedCounter {
        private int c = 0;
        public synchronized void increment() {
                c++;
        }
        public synchtonized void decrement() {
                c--;
        }
        public synchronized int vlaue() {
                return c;
        }
}
```

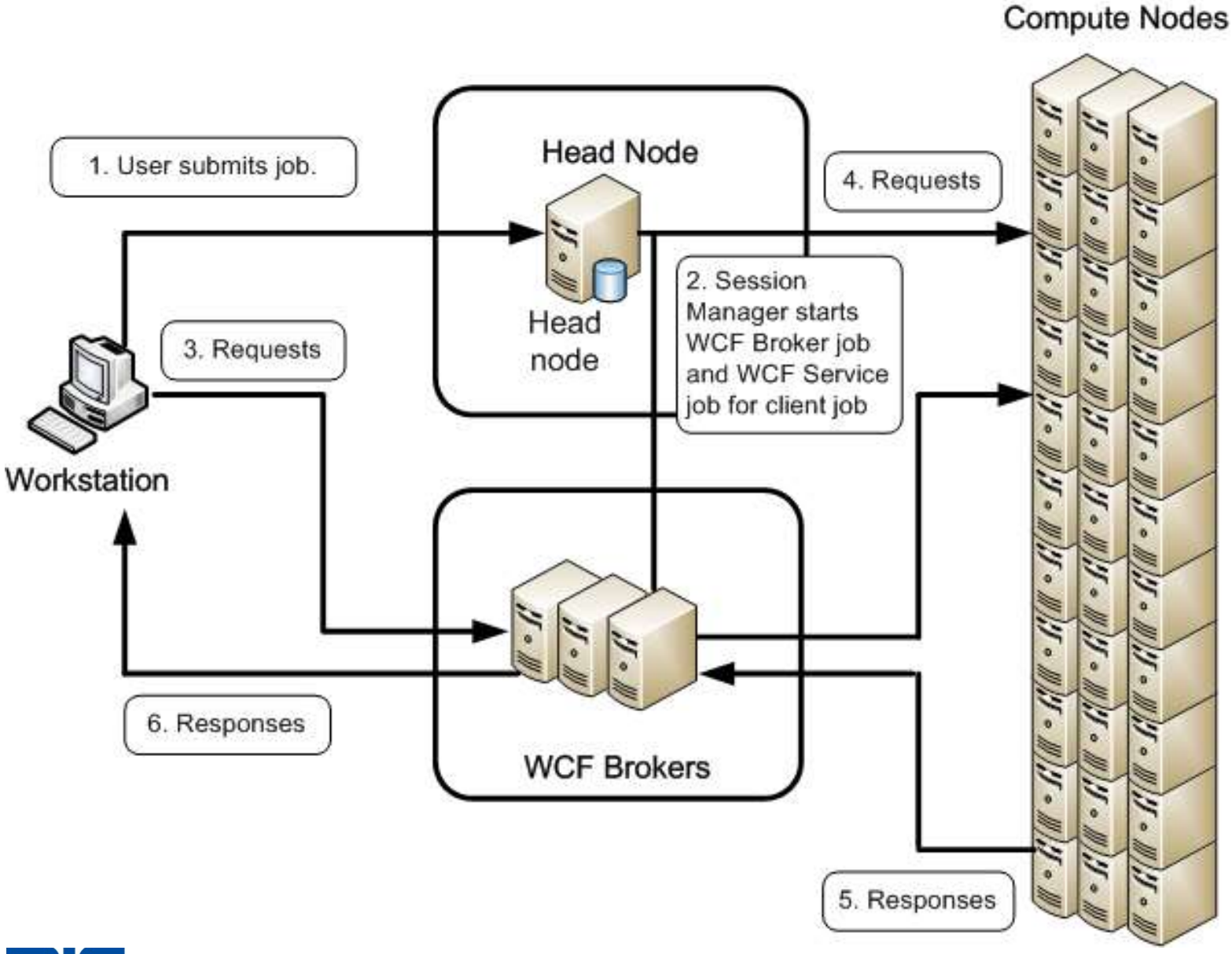
Job/Service Design



HPC - Job Scheduler Architecture



Microsoft SOA Compute Scenario

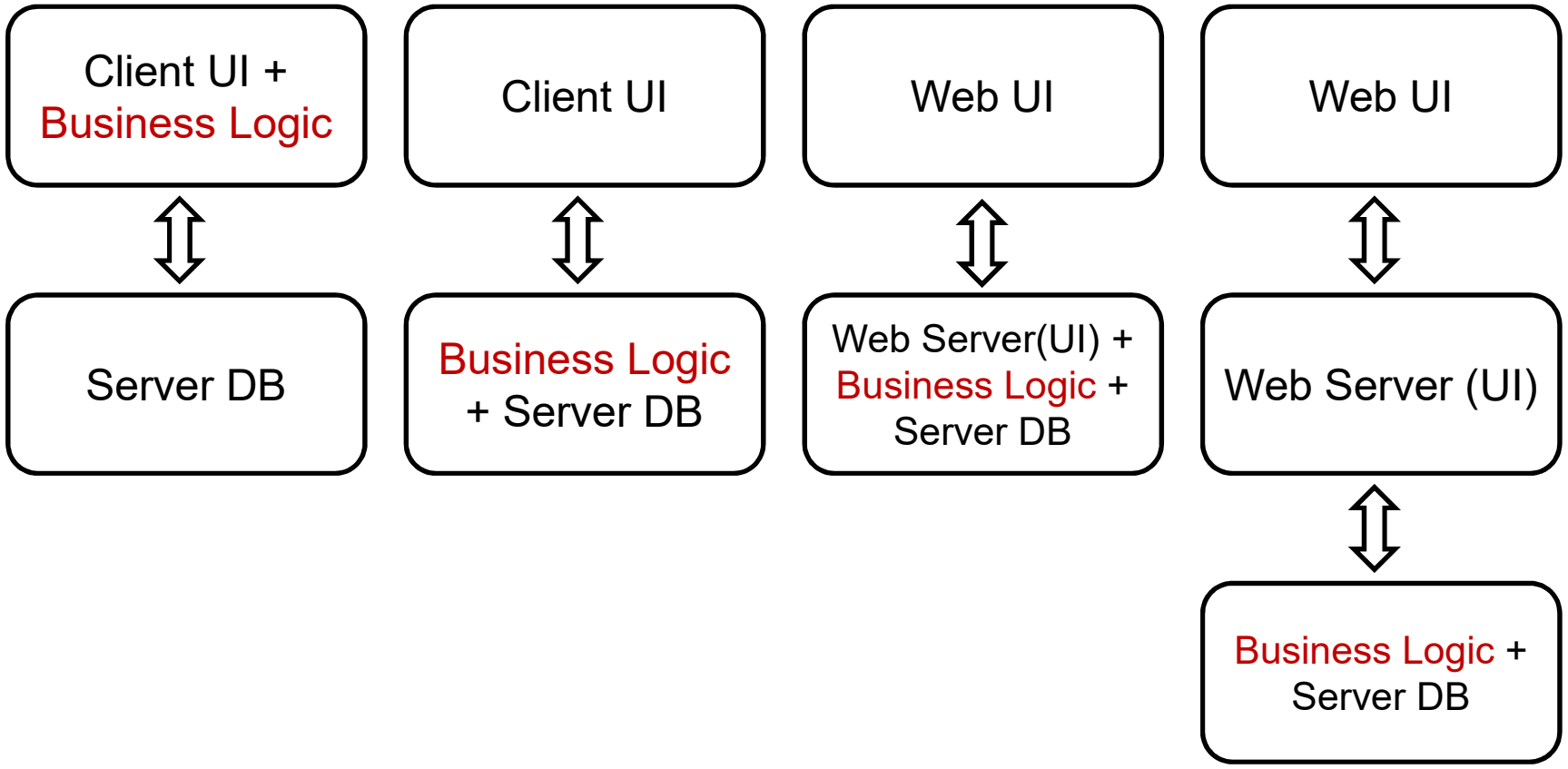


9. DESIGN DISTRIBUTED SYSTEM

- Client/Server Architecture
- Mapping Processes to Nodes
- Design Considerations

PIC

- Good: business logic separation



Process-to-Node Allocation Considerations

- Client/Server architecture
- Response time and system throughput
- Minimization of cross-network traffic
- Node capacity
- Communication medium bandwidth
- Availability of hardware and communication links
- Rerouting requirements

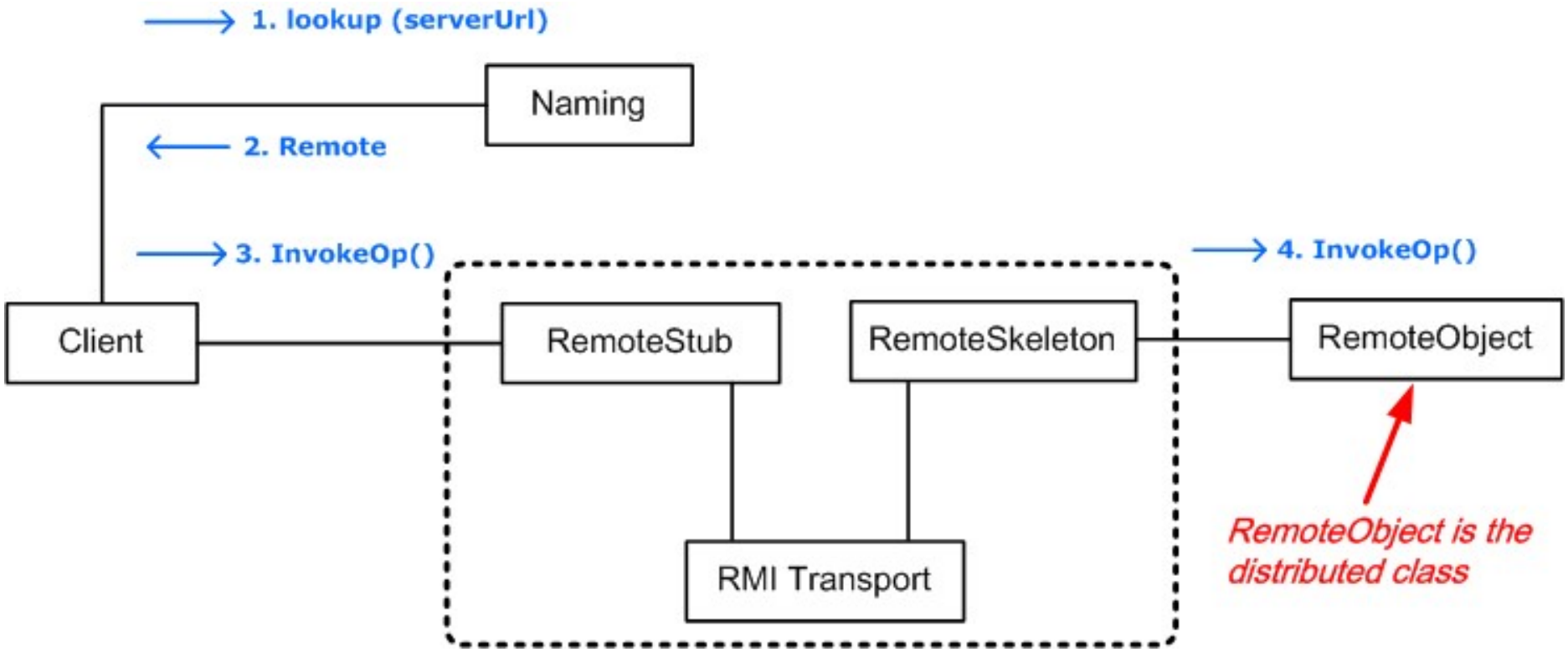
- Topology
  - Point-to-Point (點對點)
  - Broker/Middleware
  - Peer to Peer (P2P、群對群、端對端)
- Transportation
  - Client side
    - Direct communicate Server or via proxy
  - Server side
    - Receive request directly or via queue
    - Event-driven or pooling

Design Consideration

- Standard protocol
- Standard data format
- Performance
- Latency
- Synchronicity (Sync & Async)
- Message Size
- …

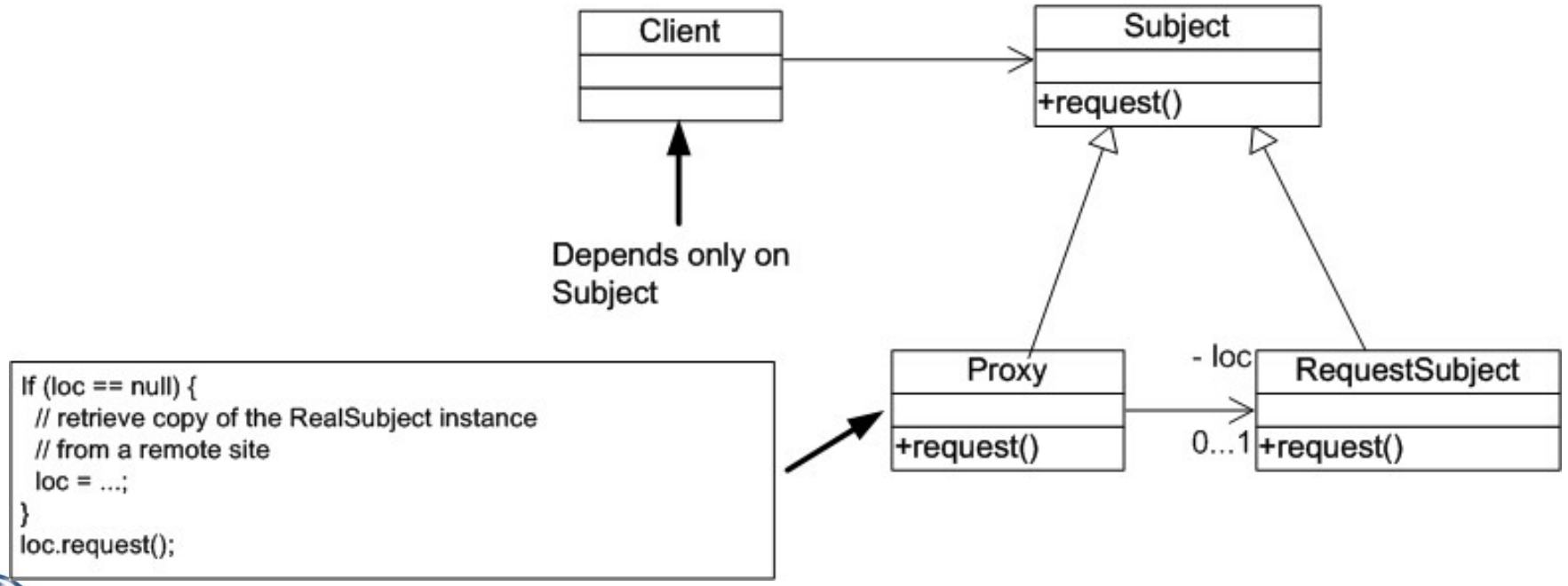
Distribution Mechanism Sample

- Java RMI: Remote Method Invocation



Proxy Design Pattern

- A proxy is a placeholder for another object to control access to it.
- Applicability (適用時機)
  - Remote proxy (our example)
  - Virtual proxy (creates “expensive” objects on demand)
- Smart Remote Proxy
  - Store messages and provide reliable transportation



Reliable Message Semantics

- Idempotent operations
- In-order delivery
- Building exactly once semantics is hard
- Can be custom-built on any communication framework

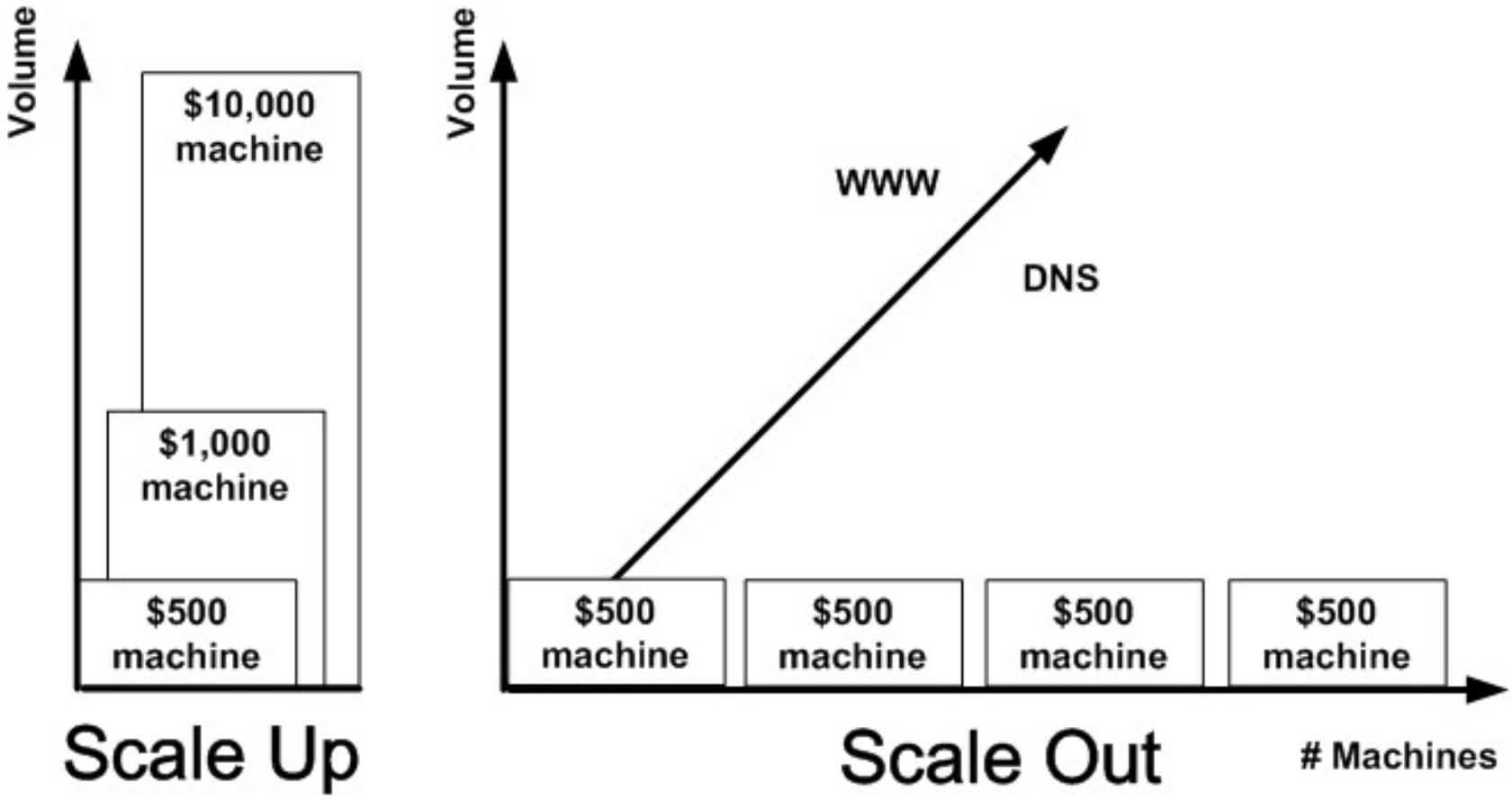
Choose a middleware which supports above features

http://xml.coverpages.org/reliableMessaging.html

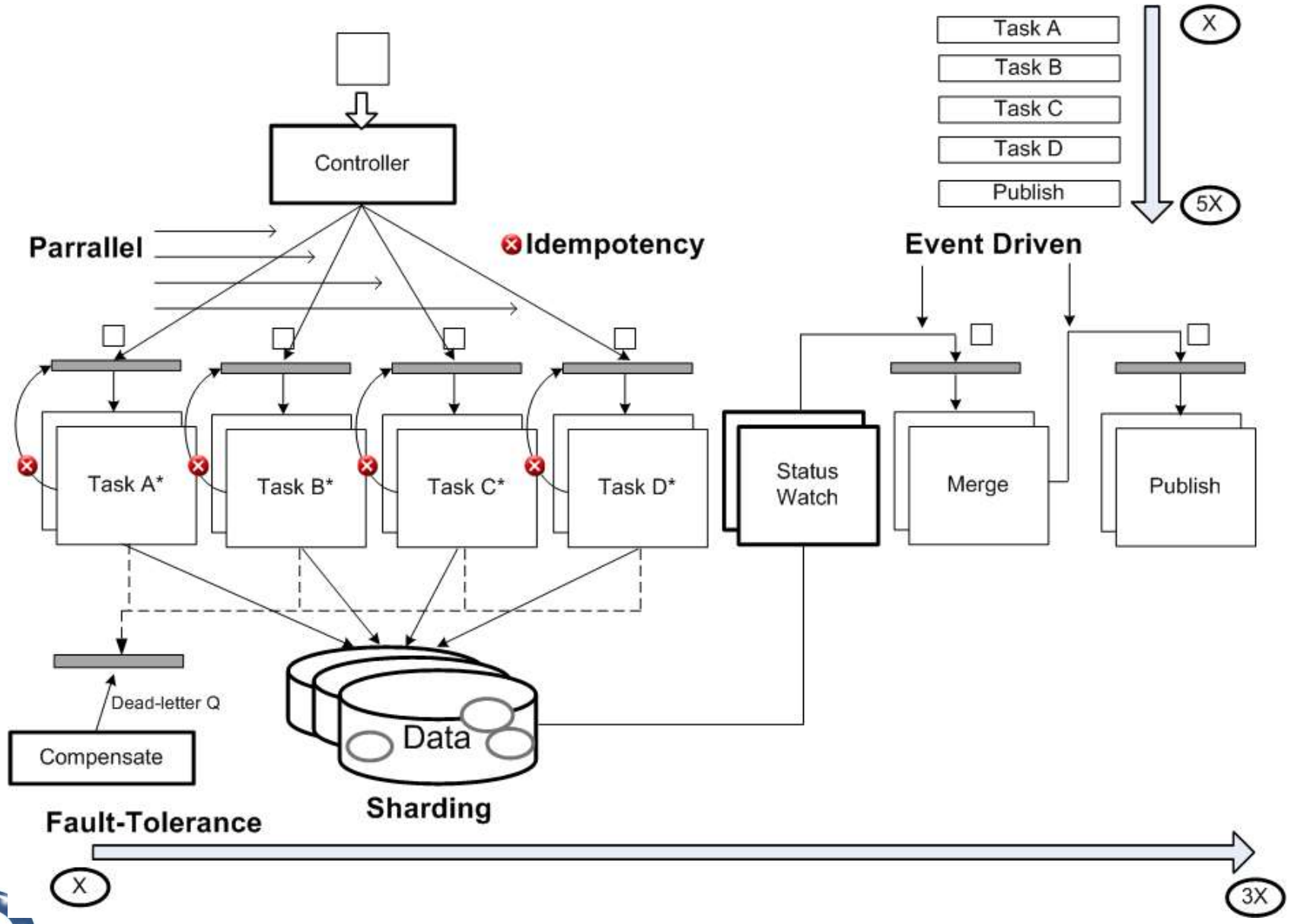
10. DESIGN SCALABILITY

- Design Principles
- Case Study

Scale-up and Scale-out



Typical Scale-out design pattern

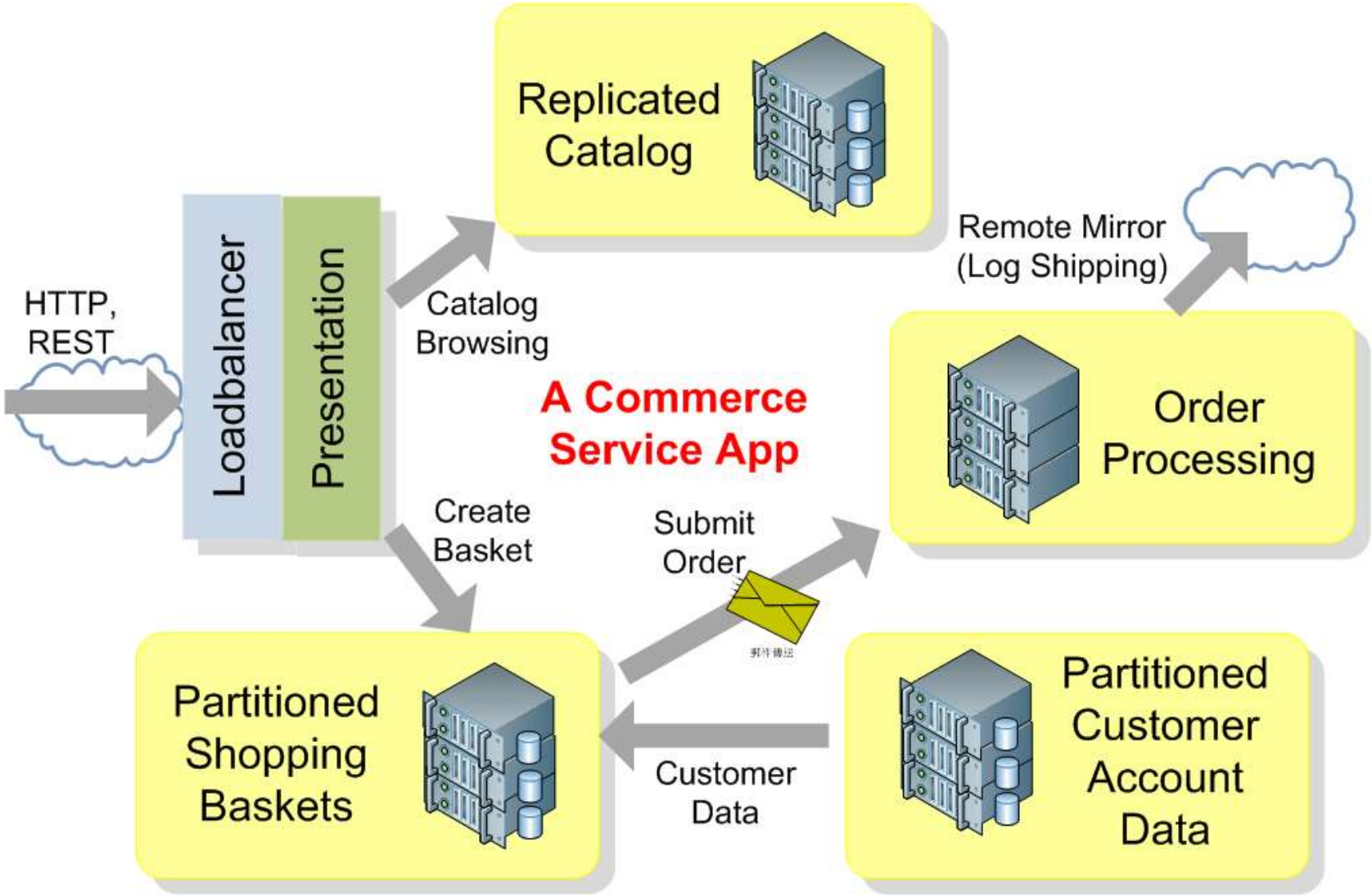


Key Principles of Scale out design

- Partitioning – functional, data, transactions
- Parallelization
  - Multi-tasking (Task pool)
- Minimize task hosts
  - Web, DB, App, Cache…
- Messaging vs. Procedure Call
  - loosely coupled components
- Asynchronous vs. Synchronous
- Stateless
- Do not use distributed DB Transactions
- Use scalable infrastructure and framework

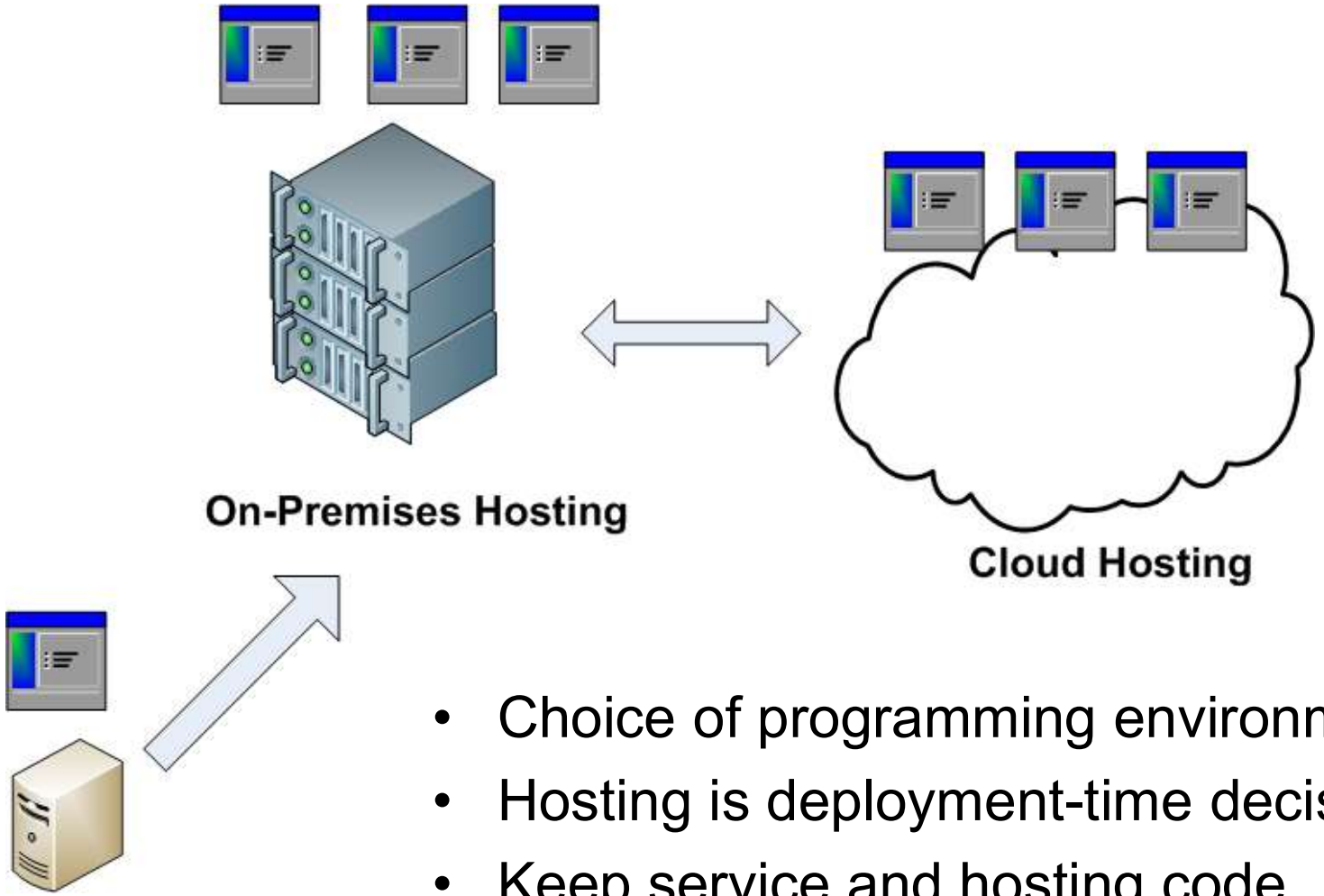
PIC

Functional Partitioning



Best Practices

- Event-driven vs. Pooling
- De-normalized, partitioned data (sharding)
- Shared nothing architecture (no 2-Phases commit)
- Use caches data
- Separate message and attachment
- Separate read/write operation & db
- Separate data: text, Blob, streaming
- Separate transient & permanent data
- Use affinity group
- Choose NoSQL over RDBMS
- Instrument code to trace execution



- Choice of programming environments.
- Hosting is deployment-time decision.
- Keep service and hosting code separate.

遺失 16 年的 Steve Jobs 電視訪談

虎嗅网：赶紧来看这段16年前遗失的、72分长的乔布斯访谈！

URL: http://www.huxiu.com/article/13992/1.html

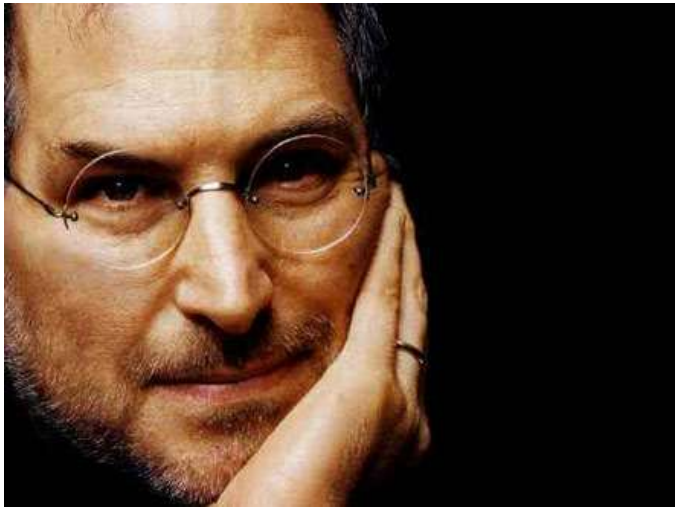
赶紧来看这段16年前遗失的、72分长的乔布斯访谈！

这个周末夜晚最有价值的视频不是《中国最强音》，是重新发布于世的、16年前遗失的长达72分钟的乔布斯访谈，原生态视频。译者团队@七印部落说，他们跟36位网友花了5个月时间，翻译了这则《乔布斯：遗失的访谈》的中英文字幕。

16年前Bob Cringley为了制作《书呆子的胜利》节目而采访了乔布斯。当时乔布斯刚刚被排挤出苹果，正在经营自己创办的NeXT。当时的节目只用了这次访谈的一小段。1990年代末，采访母带从伦敦运往美国途中遗失，这段访谈自此消失。

而几个月前，导演Paul Sen在车库里发现了一份访谈拷贝。Bob Cringley说：“乔布斯生前很少接受电视采访，如此精彩的访谈更是罕见。它记录了乔布斯的坦率、非凡的魅力和独特的视野。为了向这位奇人致敬，我们几乎一刀未剪。大部分内容是首次公布于众。”

嗅哥自己也还没看完，不过看完的同学纷纷说帅死了。



“經驗告訴我，優秀的人才是那些一心想著產品的人。雖然這些人很難管理，但我寧願和他們一起工作。”

– Steve Jobs

簡報完畢　謝謝大家

Thanks for your participation!