4 Elaboration Iteration 1 Basics
Second Week

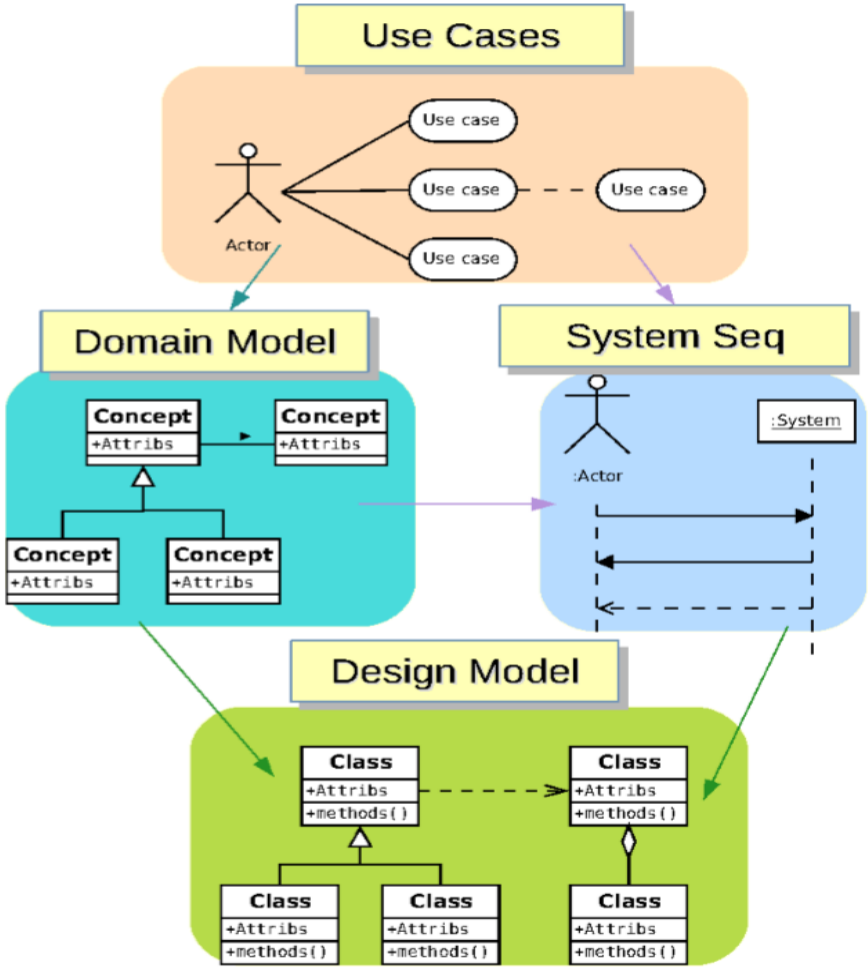
庞雄文
Tel：18620638848
Wechat: augepang
QQ: 443121909

Contents

- Part3: Elaboration Iteration 1 Basics
- 4.4.0 Operation Contracts
- 4.4 Logical Architecture and UML Package Diagrams
- 4.5 Mapping Analysis Models to Design Models
- 4.6 UML Interaction Diagrams homework
- Homework

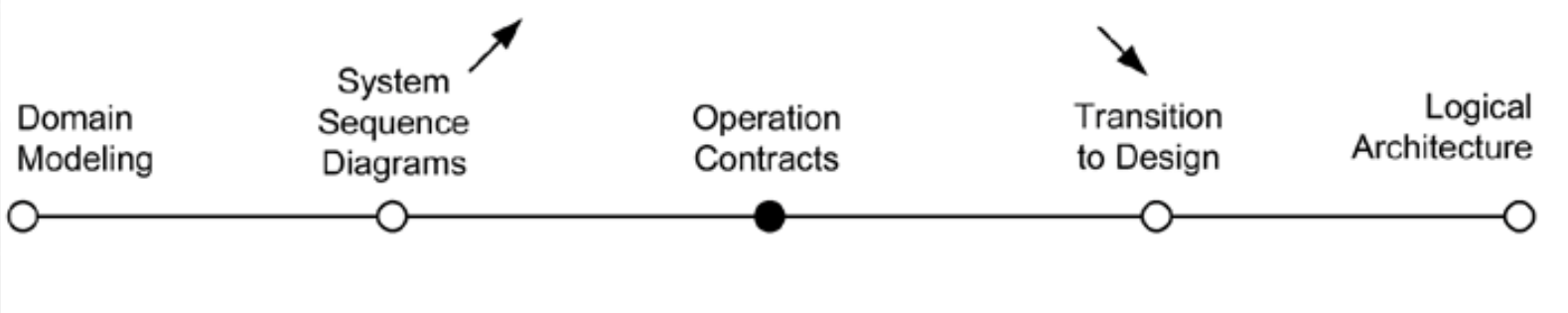
Design road

- We have described:
  - Use Cases
  - Domain Model
  - System Sequence Diagrams
- We now describe Operation Contracts
- Afterwards, we go into the Design Model



4.4.0 Operation Contracts

Where are we?



- Use cases or system features are the main ways in the UP to describe system behavior, and are usually sufficient. **But they may not be enough**
- Sometimes a more detailed or precise description of system behavior has value. **Operation contracts use a pre- and post-condition form to describe detailed changes to objects in a domain model**, as the result of a system operation

Goal of Operation Contracts

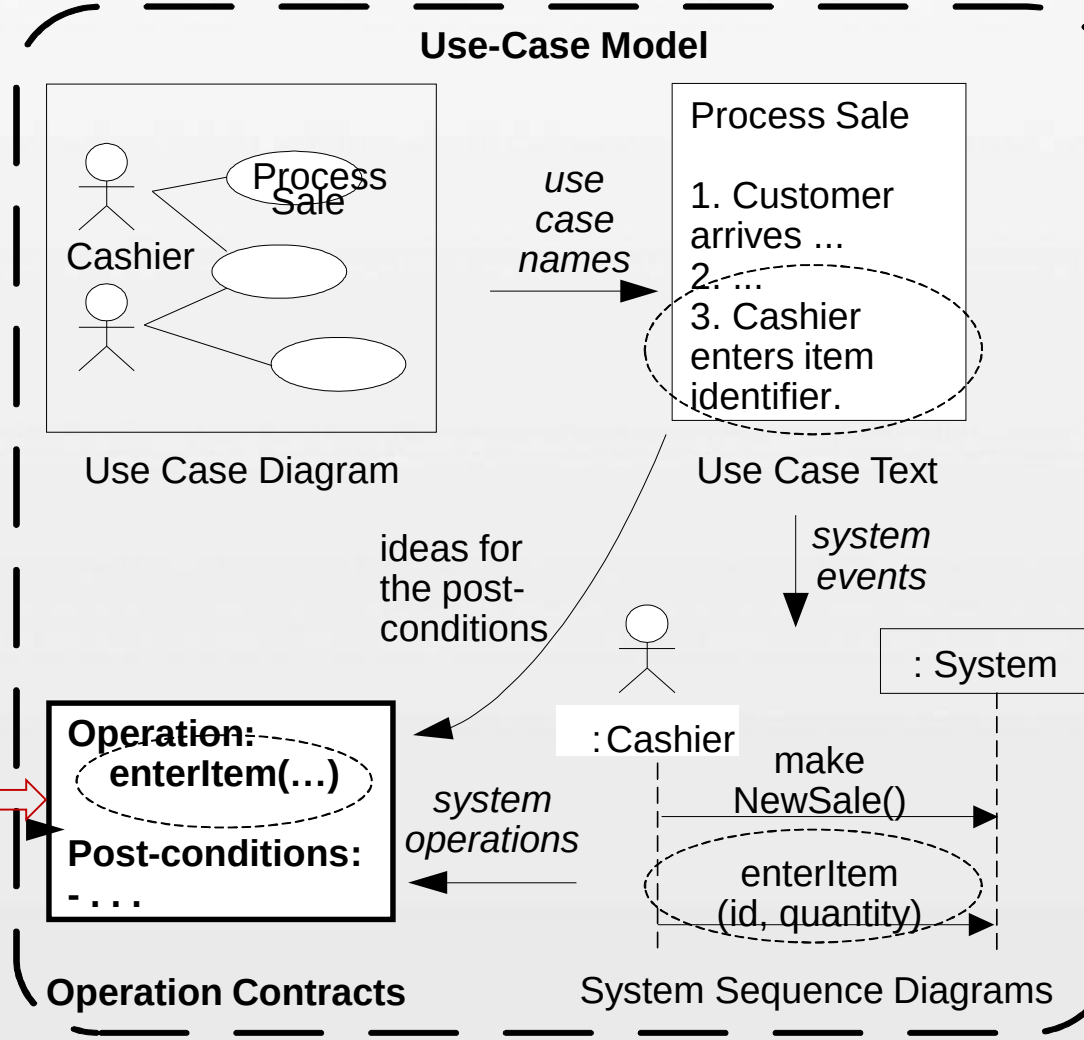
- **Define system operations.**
- **Create contracts for system operations.**

4.4.0 Operation Contracts

Example of Operation Contracts

Operation:	enterItem(itemID: ItemID, quantity: integer)
Cross References:	Use Cases: Process Sale
Preconditions:	There is a sale underway.
Postconditions:	- A SalesLineItem instance sli was created (instance creation). - sli was associated with the current Sale (association formed). - sli.quantity became quantity (attribute modification). - sli was associated with a ProductDescription, based on itemID match (association formed).

This is where we stand right now



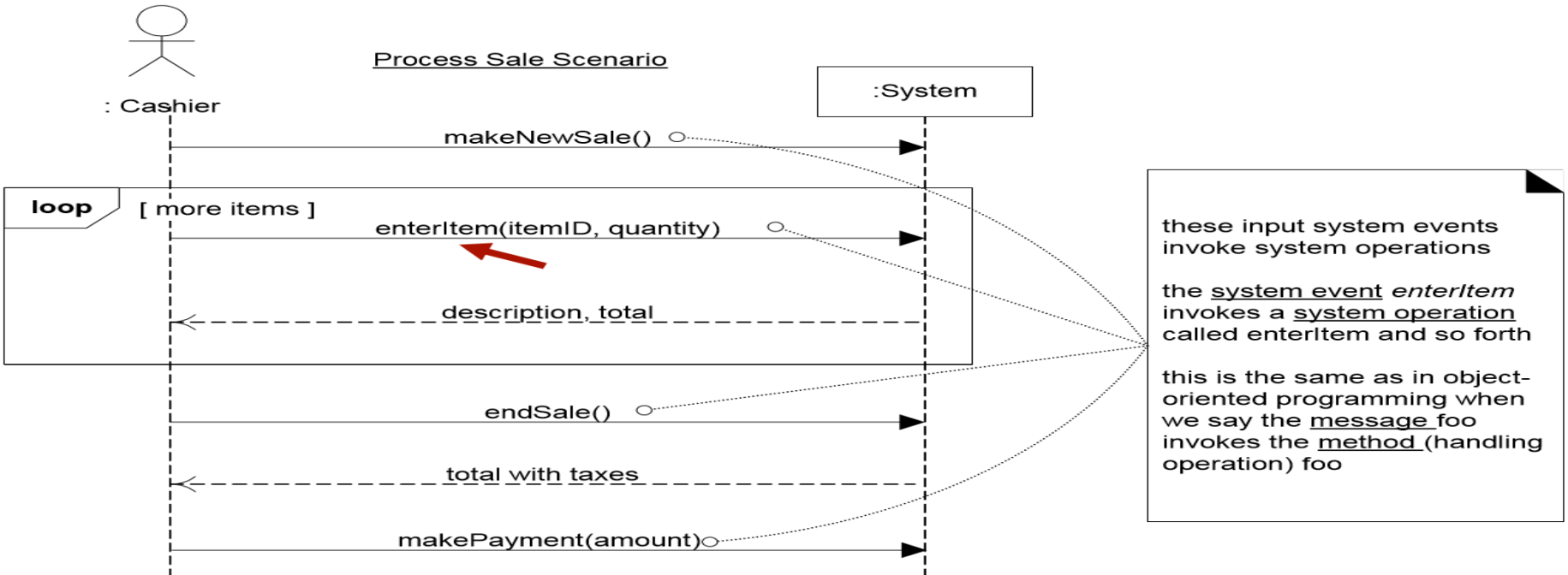
4.4.0 Operation Contracts

- **What are the Sections of a Contract?**

Operation:	Name of operation, and parameters
Cross References:	Use cases this operation can occur within
Preconditions:	Noteworthy assumptions about the state of the system or objects in the Domain Model before execution of the operation. These are non-trivial assumptions the reader should be told.
Post-conditions:	This is the most important section. The state of objects in the Domain Model after completion of the operation. Discussed in detail in a following section.

4.4.0 Operation Contracts

- **What is a System Operation?**
  - Operation Contracts are defined in terms of **system operations**
  - Operations (say, methods) that the system offers as a whole
    - The system is still a black box at this stage
  - The System Sequence Diagrams show system events
    - I.e., the SSD's messages
  - System operations handle system events



4.4.0 Operation Contracts

- **Definition: Post-conditions**
  - **Most important part of OCs**
  - The post-conditions describe changes in the state of objects in the domain model. Domain model state changes include instances created, associations formed or broken, and attributes changed.
  - Post-conditions are not actions to be performed during the operation; rather, they are observations about the domain model objects
    - **Instance creation** or **deletion.**
    - **Attribute change of value.**
    - **Associations formed** or **broken**.
  - Why Post-conditions?
    - Post-conditions aren't always necessary. Most often, the effect of a system operation is relatively clear to the developers
    - sometimes more detail and precision is useful. Contracts offer that

4.4.0 Operation Contracts

- **Guideline: How to write** post-conditions
  - **1. write the post-conditions in a declarative, passive past tense form (was …) to emphasize the observation of a change**
    - (better) A SalesLineItem was created.
    - (worse) Create a SalesLineItem.
  - 2. Write post-conditions **agile**. If the developer can accurately understand the work to be done without contract operation, there is no need to write a contract.
  - 3. if necessary, Update the Domain Model

4.4.0 Operation Contracts

- **Guideline: How to Create and Write Contracts**
  - **to create contracts:**
    - Identify system operations from the SSDs.
    - For system operations that are **complex** and perhaps subtle in their results, or which are not clear in the use case, construct a contract.
    - To **describe the post-conditions**, use the following categories:
      - instance creation and deletion
      - attribute modification
      - associations formed and broken

4.4.0 Operation Contracts

- **Example: NextGen POS Contracts**

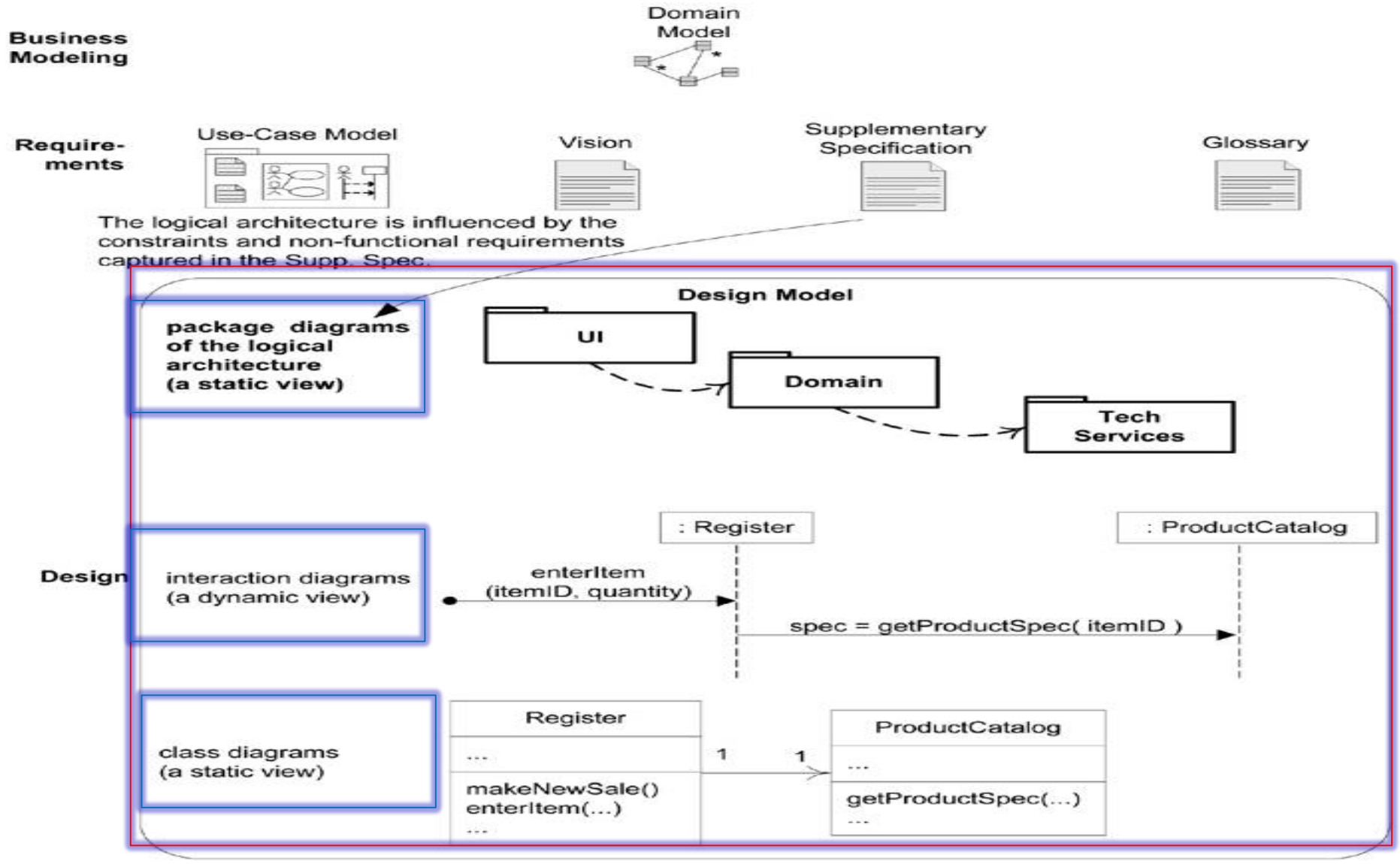
Contract CO4: makePayment	
Operation:	makePayment(amount: Money)
Cross References:	Use Cases: Process Sale
Preconditions:	There is a sale underway.
Postconditions:	- A Payment instance p was created (instance creation). - p.amountTendered became amount (attribute modification - p was associated with the current Sale (association formed). - The current Sale was associated with the Store (association formed); (to add it to the historical log of completed sales)

4.4.0 Operation Contracts

- **Applying UML: Operations, Contracts, and the OCL**
  - The pre- and post-condition format is informal natural language
  - UML use **a formal, rigorous language called Object Constraint Language (OCL)** to express constraints of UML operations
    - The OCL defines an official format for specifying pre- and post-conditions for operations
- **Phases**
  - Inception Contracts are not motivated during inception they are too detailed.
  - Elaboration If used at all, most contracts will be written during elaboration, when most use cases are written. Only write contracts for the most complex and subtle system operations.

4.4 Logical Architecture and UML Package Diagrams

- short introduction to the logical architecture



4.4 Logical Architecture and UML Package Diagrams

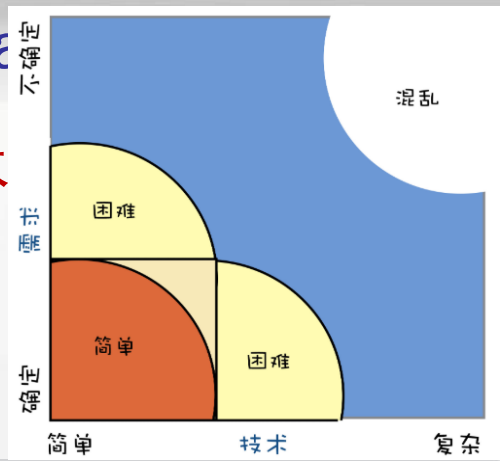
Software Architecture

- There is no standard, universally-accepted definition of the term, for software architecture is a field in its infancy, although its roots run deep in software engineering. ” https://resources.sei.cmu.edu/asset_files/FactSheet/2010_010_001_513810.pdf

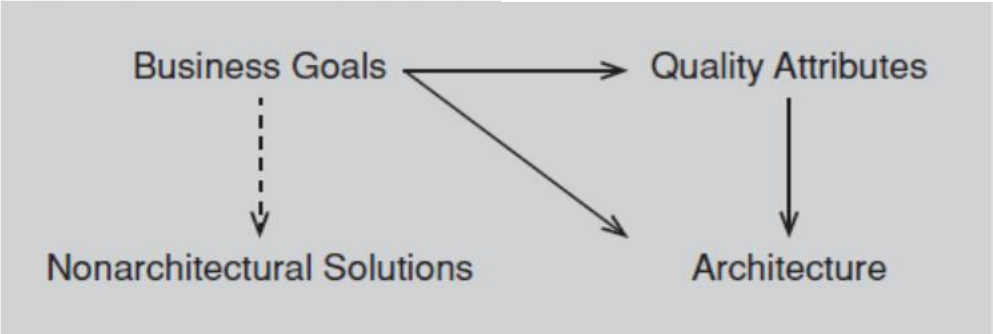
- **Definition by IEEE**
  - Software architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environments, and the principles governing its design and evolution.
  - **Architecture={component, relations, environment, principle}.**

- Architecture involves a set of strategic design decisions, rules and patterns that constrain design and construction
- .Architecture Decisions are the most fundamental decisions, and changing them will have significant effects.

4.4 Logical Architecture and UML Packa



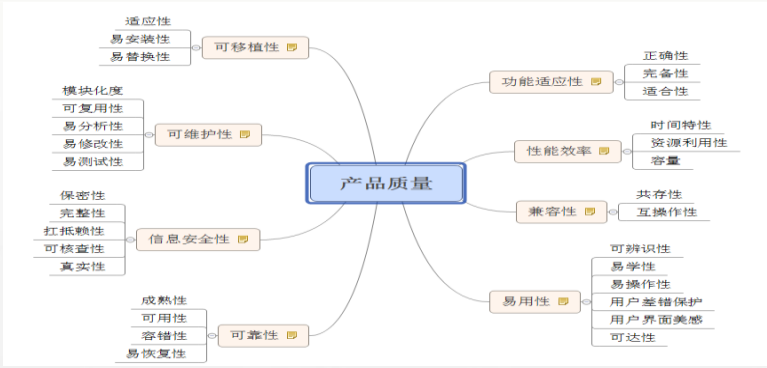
- 软件体系结构关注的是： 复杂系统(需求和技术
  - 如何将复杂的软件系统划分为模块
  - 如何规范模块的构成,
  - 如何将这些模块组织为完整的系统,
  - 以及保证系统的质量要求。



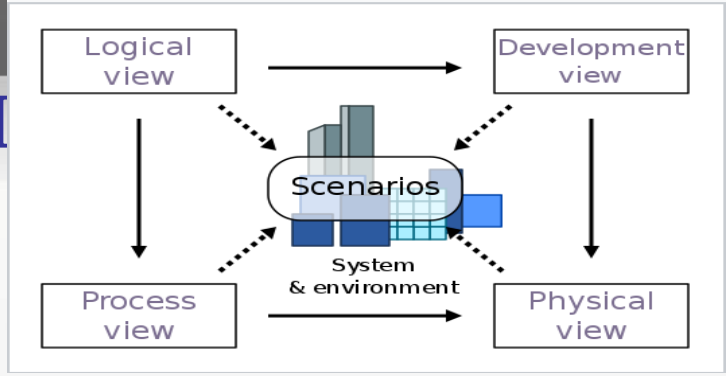
- 主要目标：
  - 建立一个一致的系统及其视图集，并表达为最终用户和软件设计者需要的结构形式，支持用户和设计者之间的交流与理解。
  - 分为两方面：
    - 外向目标：建立满足最终用户要求的系统；
    - 内向目标：建立满足系统设计者需要以及易于系统实现、维护和扩展的系统构件构成。--质量属性

4.4 Logical Architecture and UML Package Diagrams

- **Quality of software(**ISO25010**)**



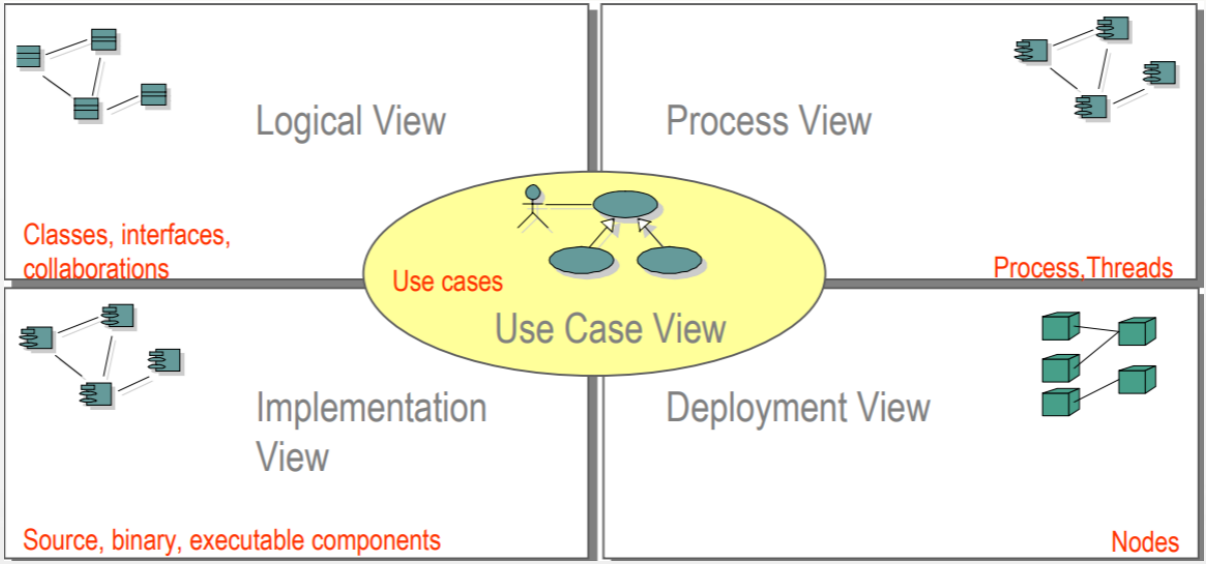
4.4 Logical Architecture and UMI



- 4+1 Views of software architecture
  - *use case view.*
    - use cases, or scenarios describe sequences of interactions between objects and between processes
  - *Logical view*:
    - The logical view is concerned with the functionality that the system provides to end-users. UML diagrams are used to represent the logical view, and include class diagrams, and state diagrams
  - *Process view*:
    - The process view deals with the dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the run time behavior of the system. The process view addresses concurrency, distribution, integrator, performance, and scalability, etc. UML diagrams to represent process view include the sequence diagram, communication diagram, activity diagram

4.4 Logical Architecture and UML Package Diagrams

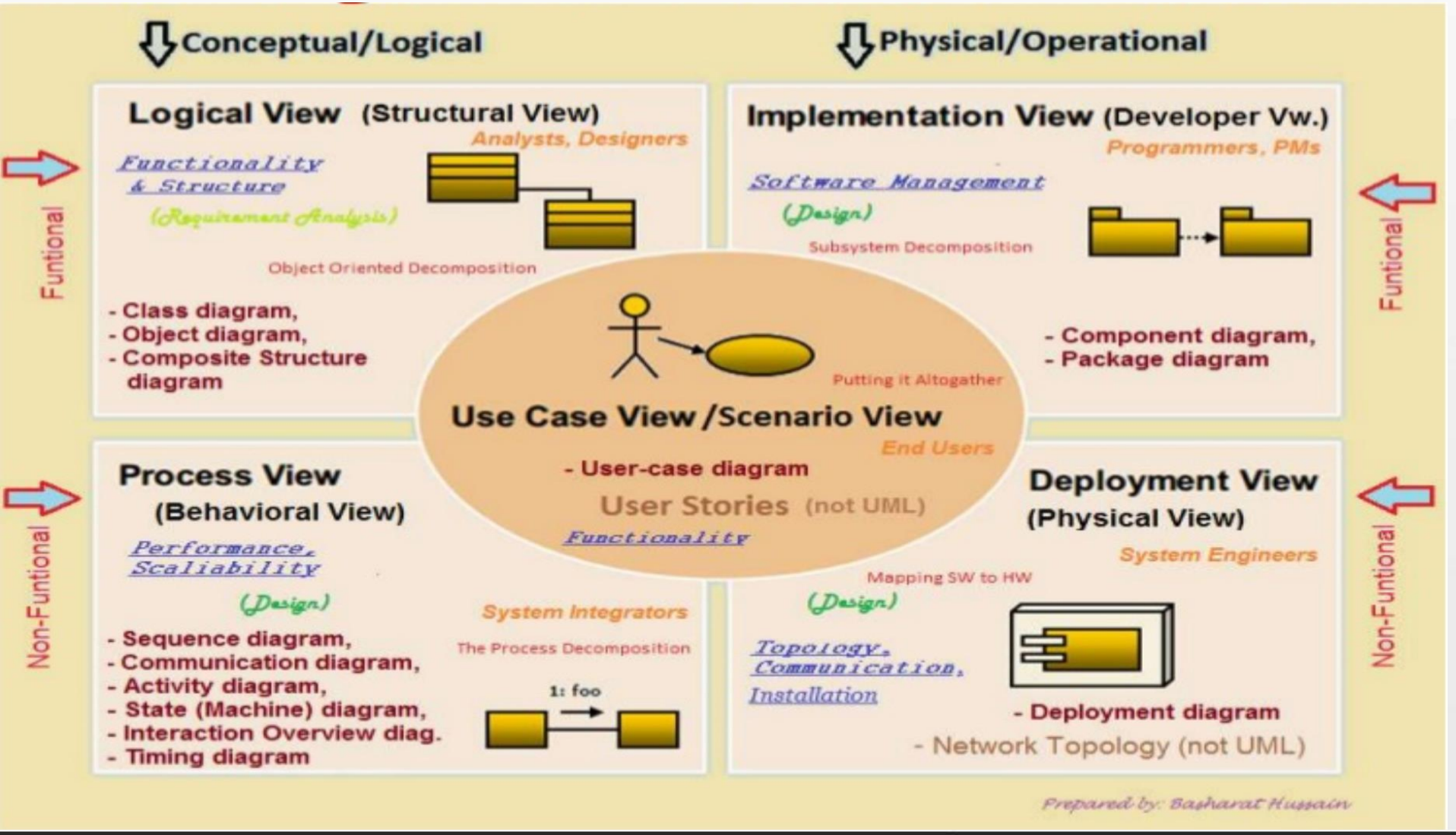
4+1 Views of software architecture



- ***Development view***:
  - The development view illustrates a system from a programmer's perspective and is concerned with software management. This view is also known as the implementation view, include the Component diagram , Package diagram.[2]
- ***Physical view***:
  - The physical view depicts the system from a system engineer‘s point of view. It is concerned with the topology of software components on the physical layer as well as the physical connections between these components. deployment diagram.[2]

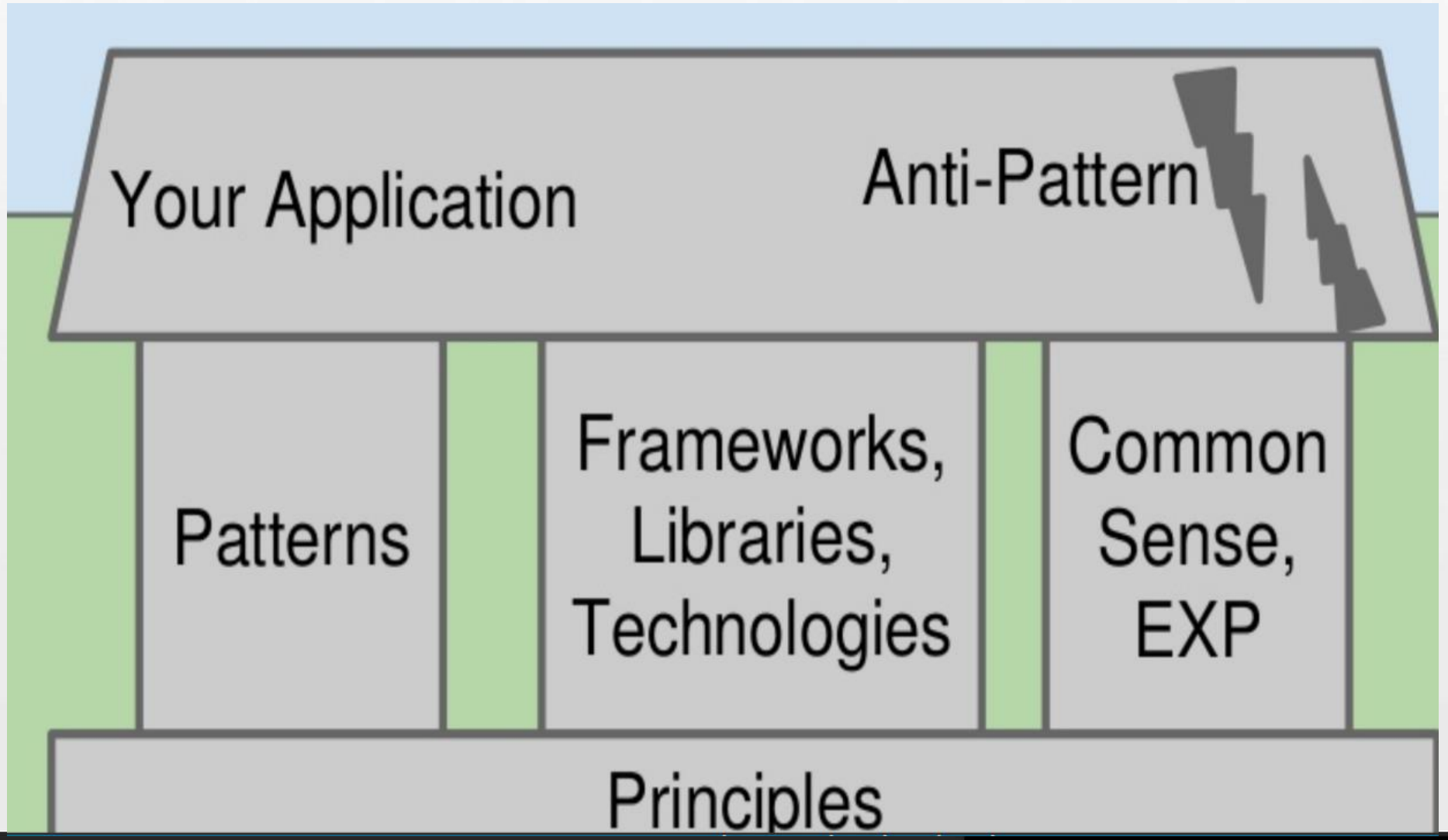
4.4 Logical Architecture and UML Package Diagrams

- 4+1 Views of software architecture



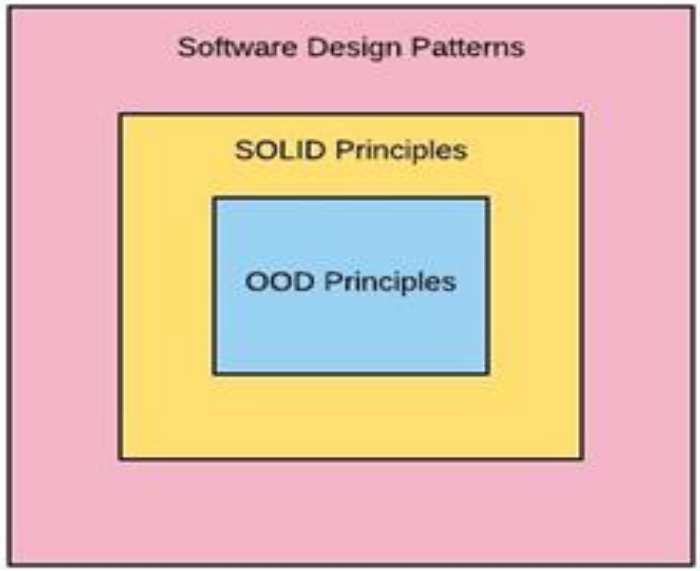
4.4 Logical Architecture and UML Package Diagrams

- How to design software architecture

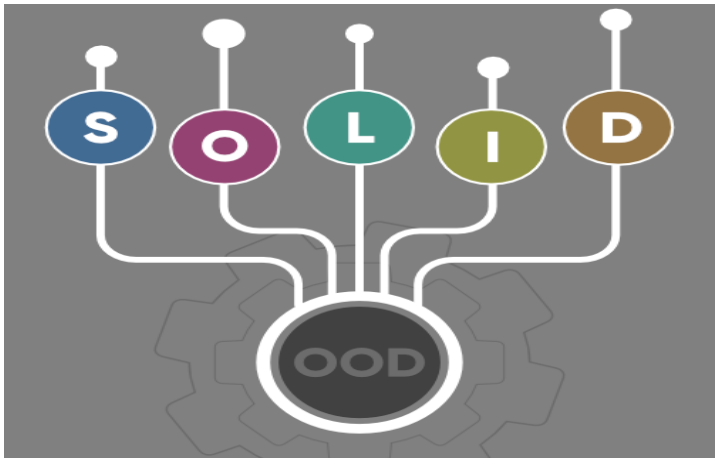


4.4 Logical Architecture and UML Package Diagrams

- **Design Principles**
  - **Decomposition**
  - **Abstraction**
  - separation of concerns (SoC)
  - **information hiding and localization**
  - **Modularity**
    - **coupling**
    - **Cohesion**



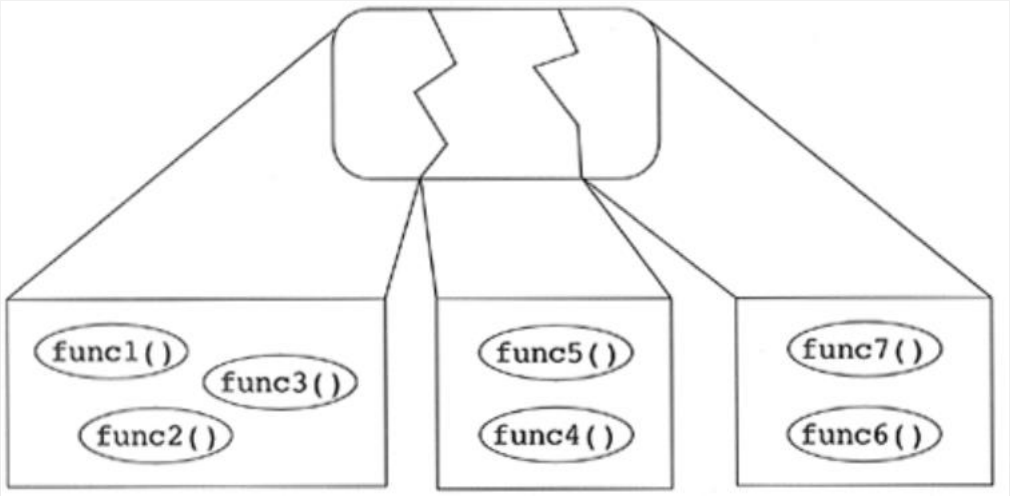
- **SOLID Principles**
  - Single Responsibility Principle(SRP)
  - Open-Closed Principle(OCP)
  - Liskov Substitution Principle(LSP)
  - Interface Segregation Principle(ISP)
  - Dependency Inversion Principle(DIP)
  - Law of Demeter:Don't talk to strangers!
  - Composite Reuse Principle, CRP

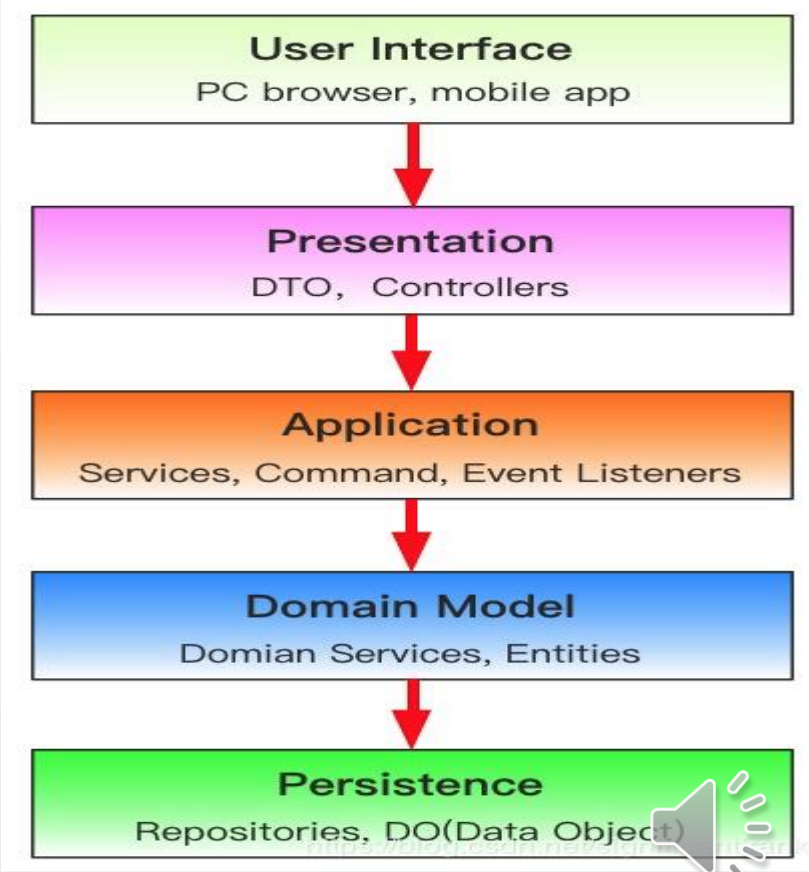


4.4 Logical Architecture and UML Package Diagrams

■ Separation of concerns (SOC)

➢ a design principle for separating a computer program into distinct sections such that each section addresses a separate concern

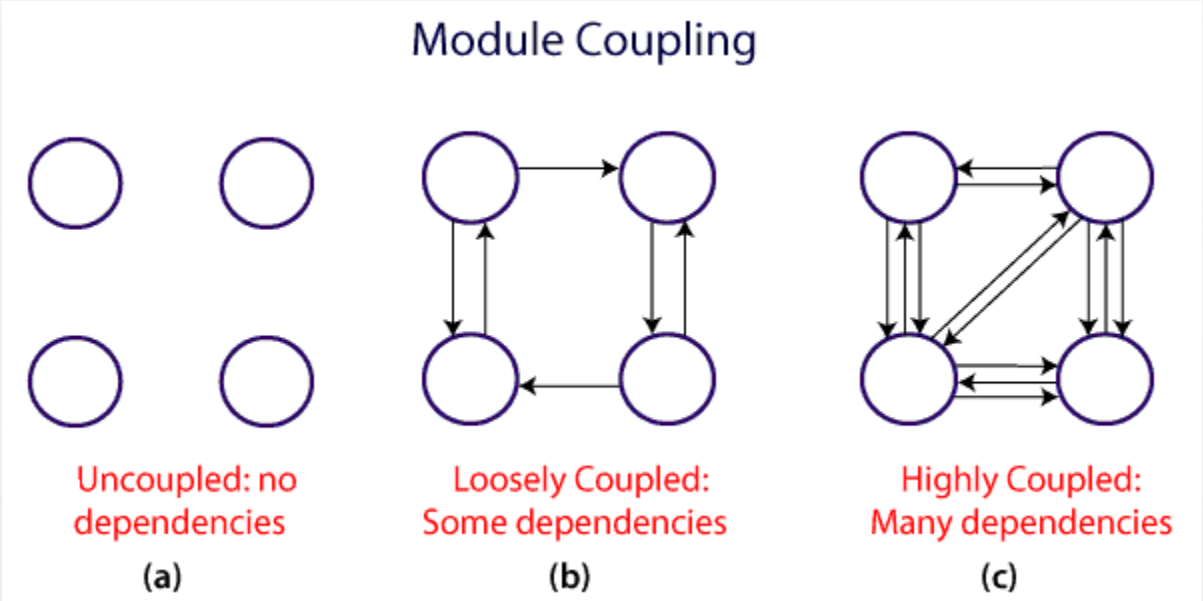




4.4 Logical Architecture and UML Package Diagrams

Coupling and Cohesion

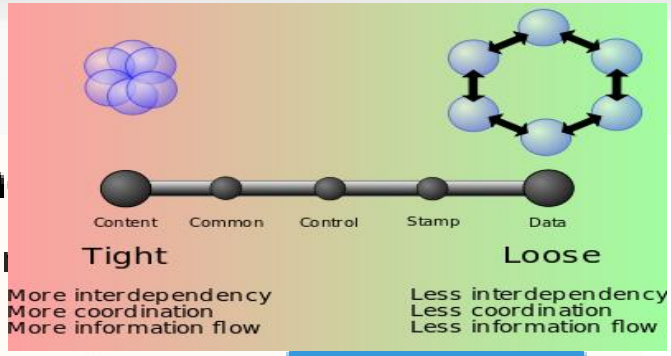
- the **coupling** is the degree of interdependence between software modules
- Coupling is measured by the number of relations between the modules.
  - Uncoupled
  - Loose Coupled
  - High Coupled



- A good design is the one that has **low(loose) coupling**
- **Why low coupling?** **Decoupling**

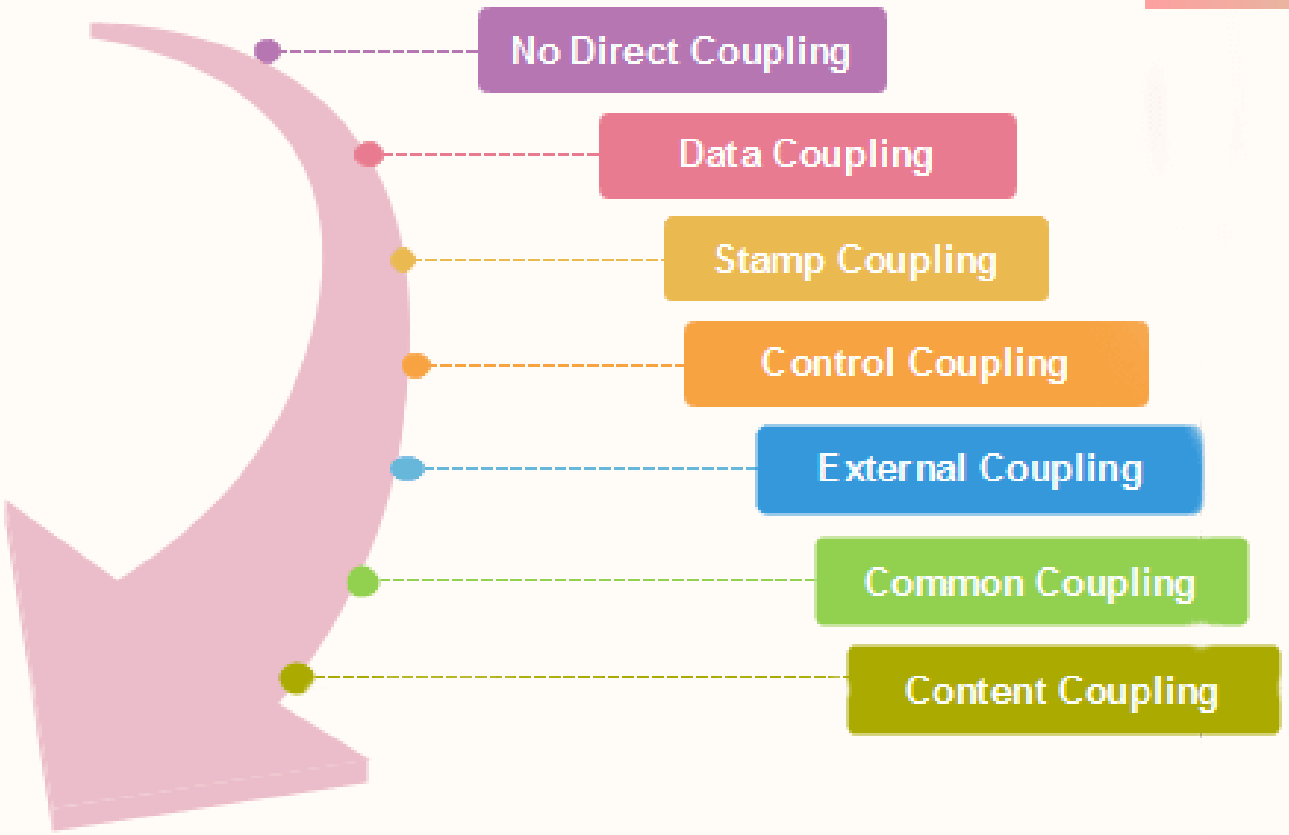
4.4 Logical Architecture and UML Package Diagrams

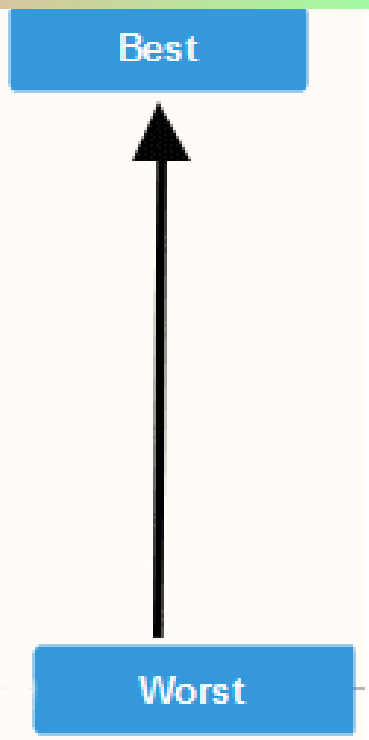
- Coupling and Cohesion



Types of Modules Couplin

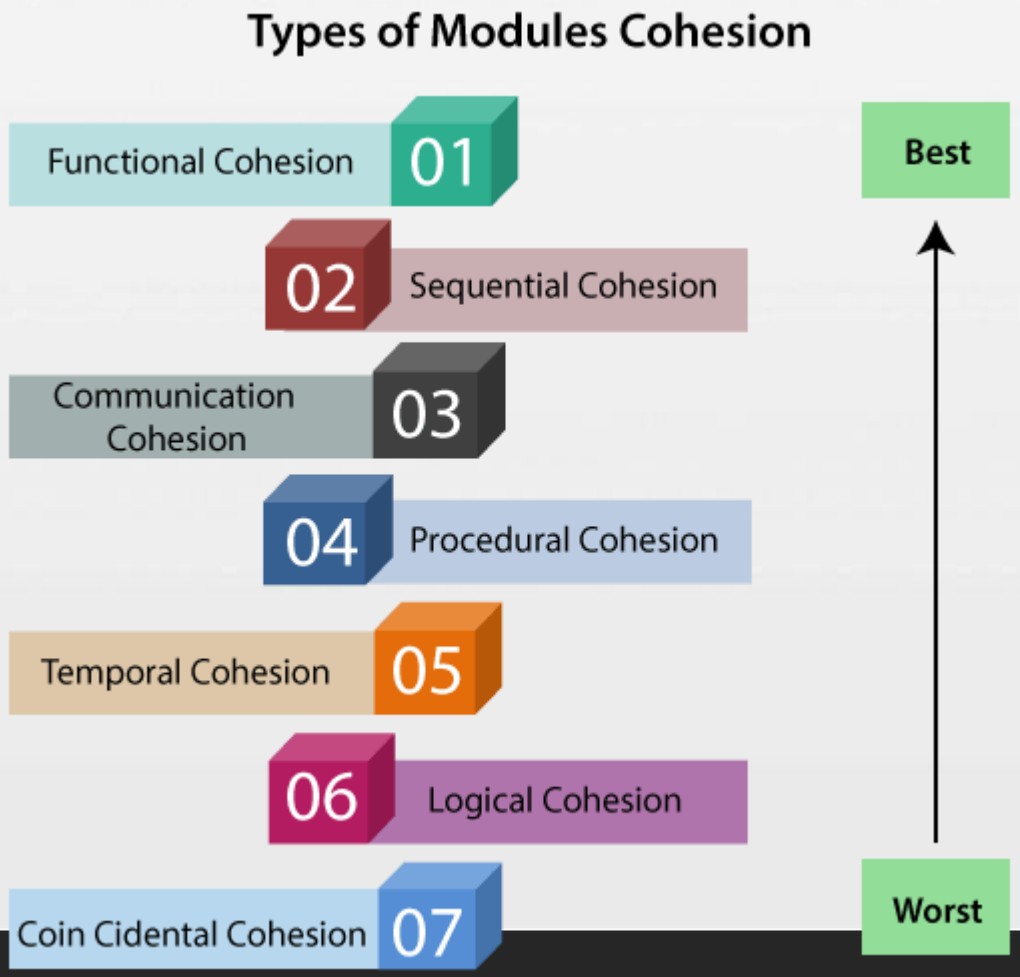
There are various types of module Coupli

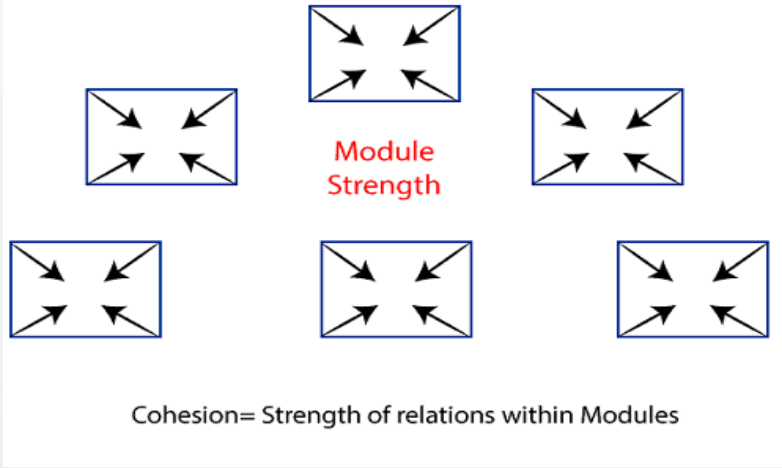




4.4 Logical Architecture and UML Package Diagrams

- Cohesion
  - cohesion measures the strength of relationships between pieces of functionality within a given module





A good design is the one that has **high cohesion**

4.4 Logical Architecture and UML Package Diagrams

- SOLID Principles

Single Responsibility Principle

Each class has a single purpose. All its methods should relate to function

Open / Closed Principle

Classes (or methods) should be open for extension and closed for modification

Liskov Substitution Principle

You should be able to replace an object with any of its derived classes.

Interface Segregation Principle

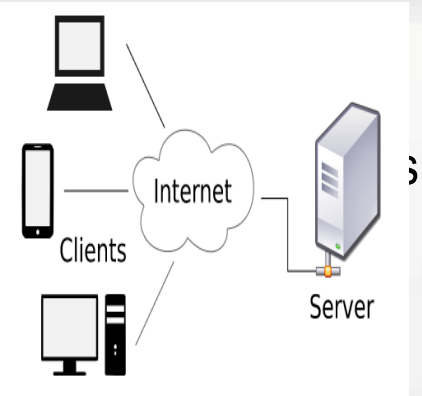
Define subsets of functionality as interfaces

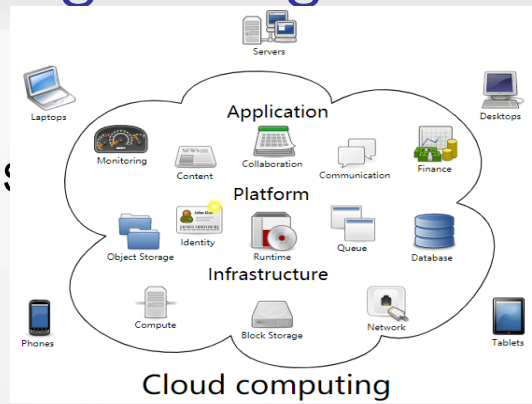
Dependency Inversion Principle

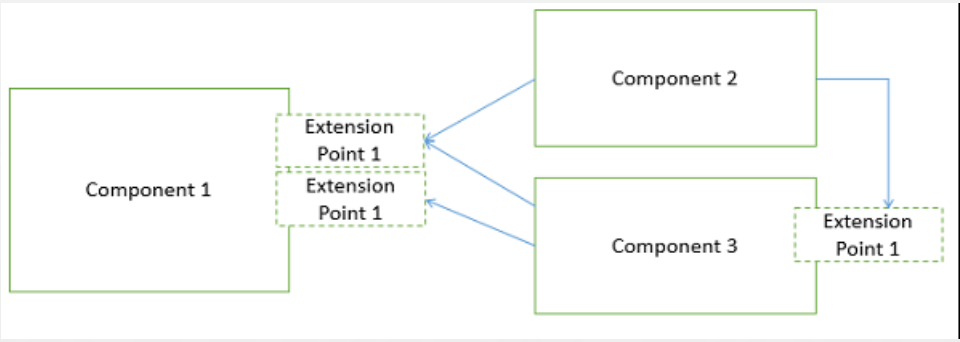
High level modules should not depend on low-level modules. Instead, both should depend on abstractions. Abstractions should not depend on details. Details

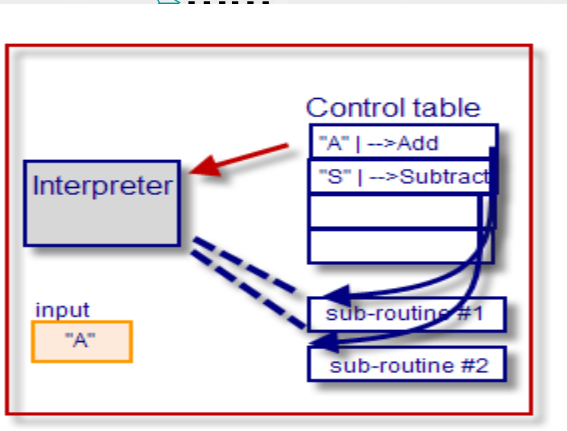
4.4 Logical Architecture and UML Package Diagrams

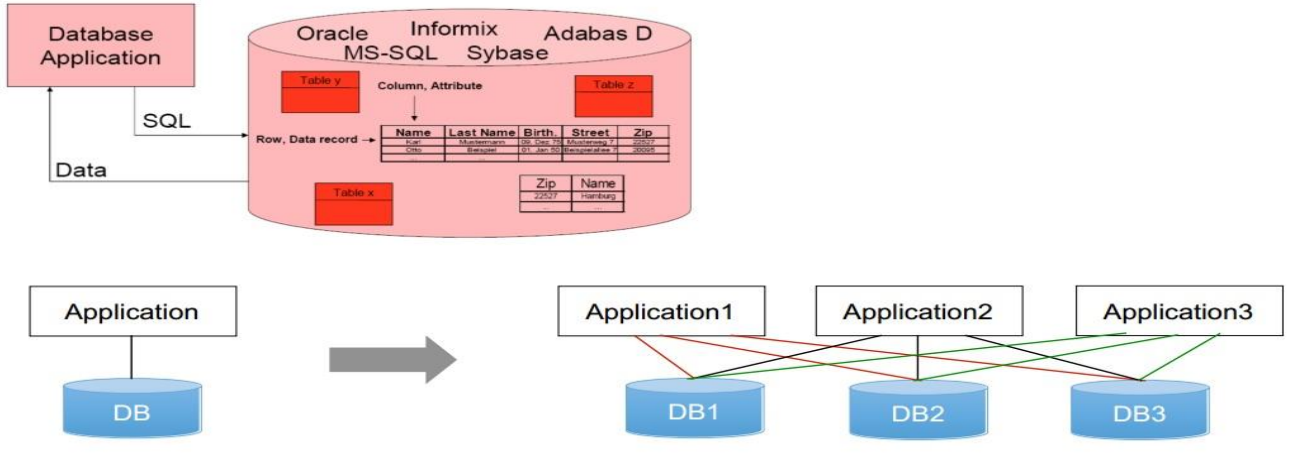
- **Software architectural patterns**
  - Client-server (2-tier, 3-tier, *n*-tier, cloud
  - Component-based
  - Data-centric
  - Event-driven (or implicit invocation)
  - Layered (or multilayered architecture)
  - Micro-services architecture
  - Peer-to-peer (P2P)
  - Pipes and filters
  - Service-oriented
  - ……





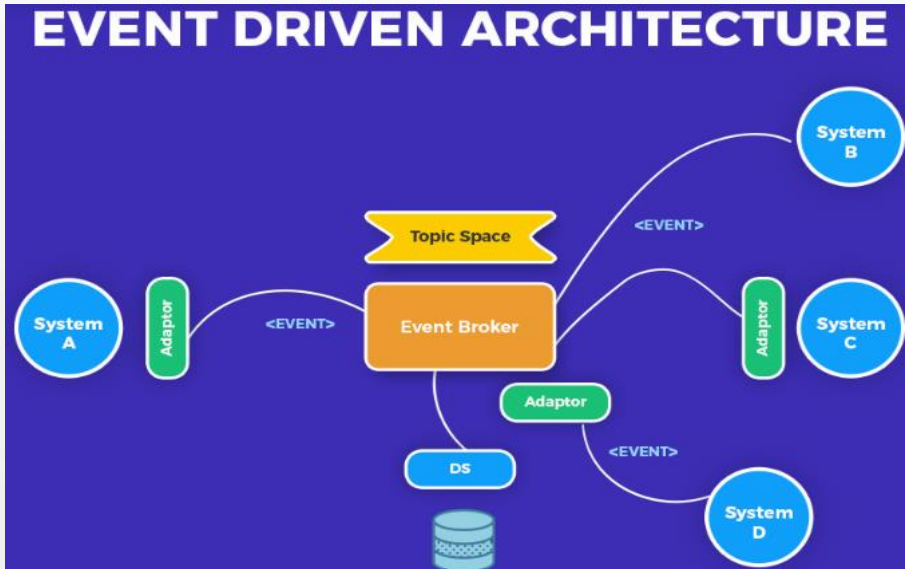


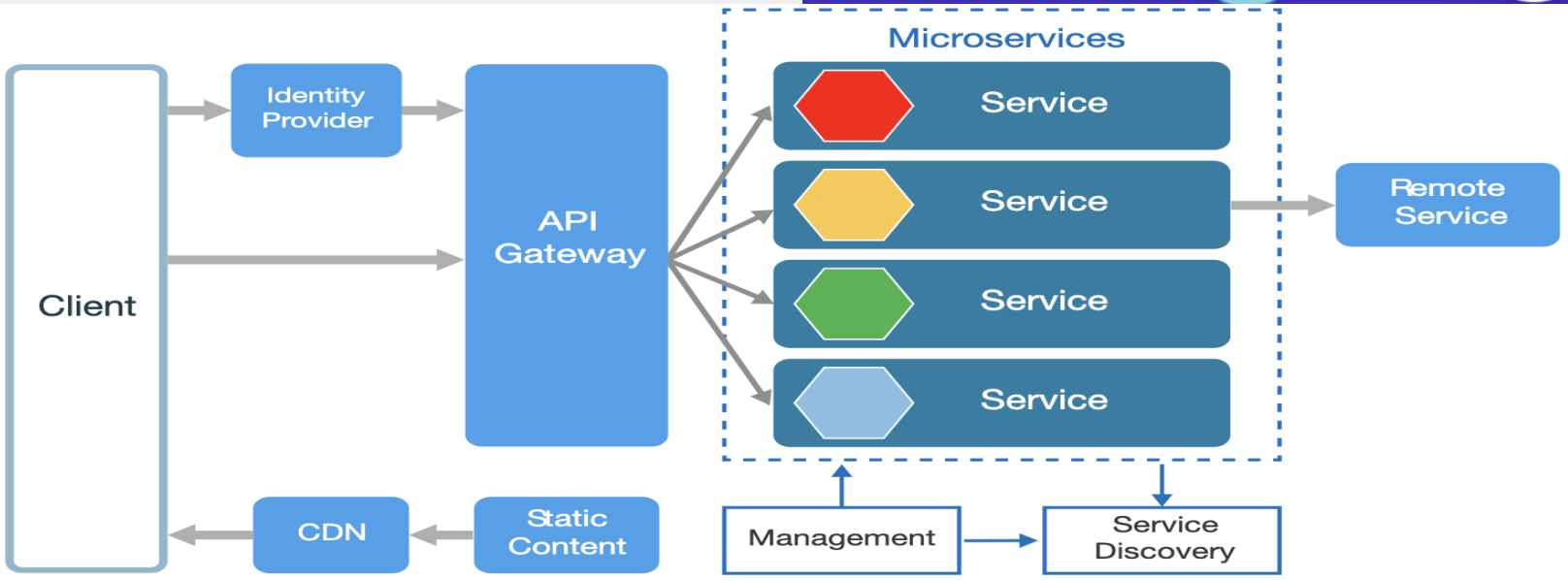




4.4 Logical Architecture and UML Package Diagrams

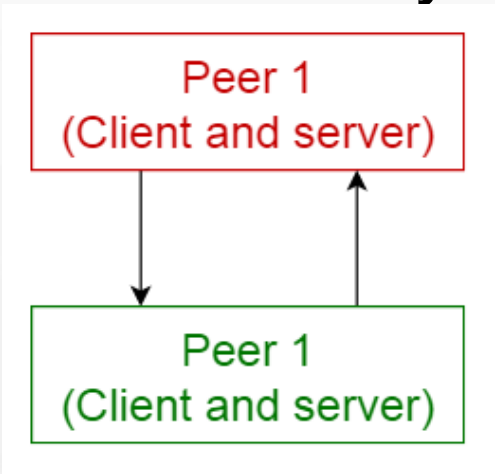
- **Software architectural patterns and styles**
  - Event-driven (or implicit invocation)
  - Layered (or multilayered architecture)
  - Micro-services architecture

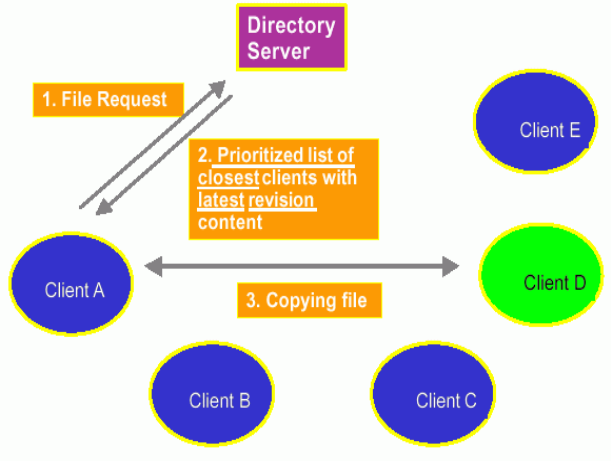


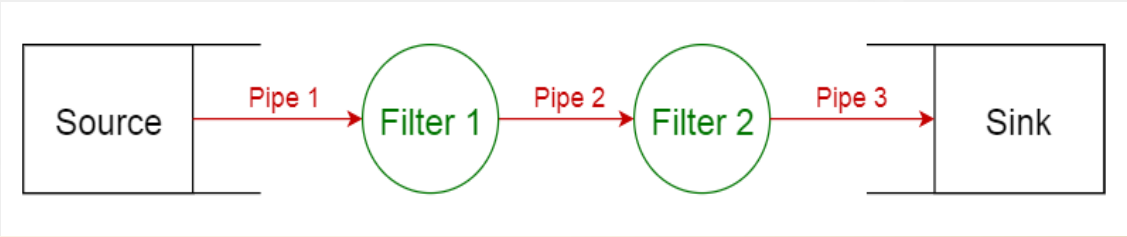


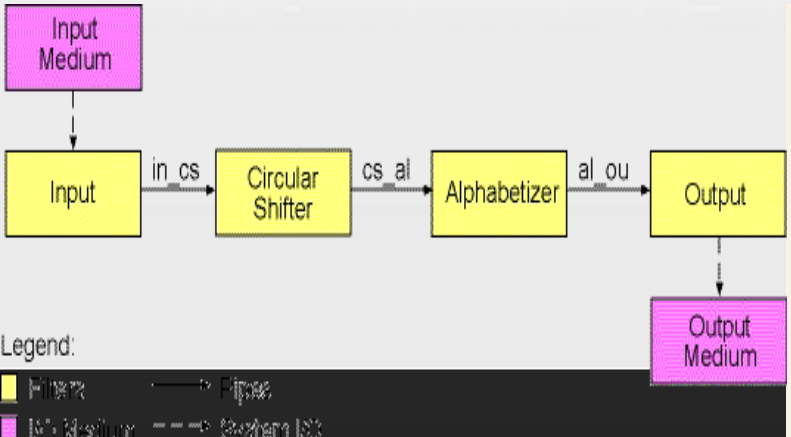
4.4 Logical Architecture and UML Package Diagrams

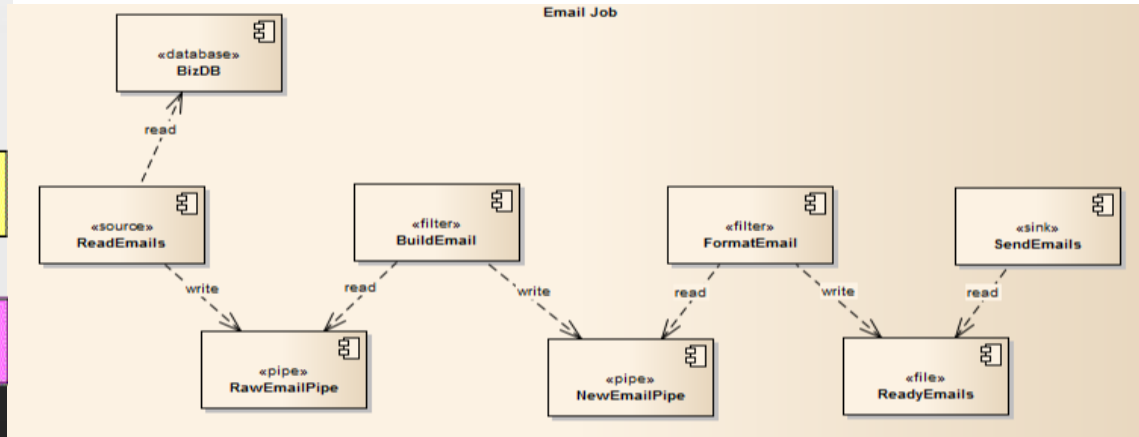
- **Software architectural patterns and styles**
  - Peer-to-peer (P2P)
  - Pipes and filters





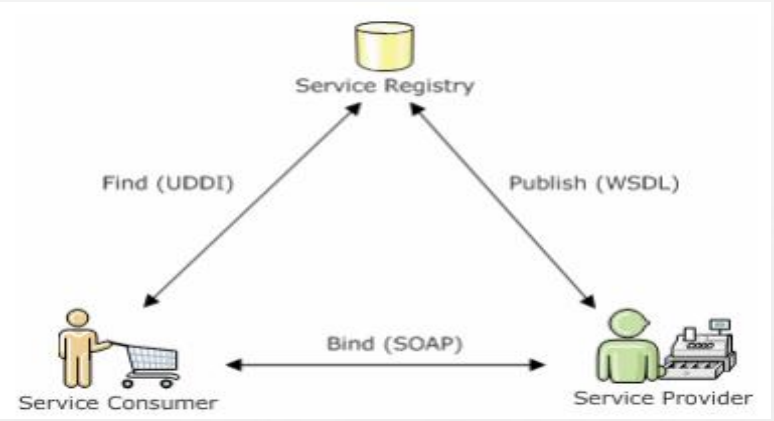


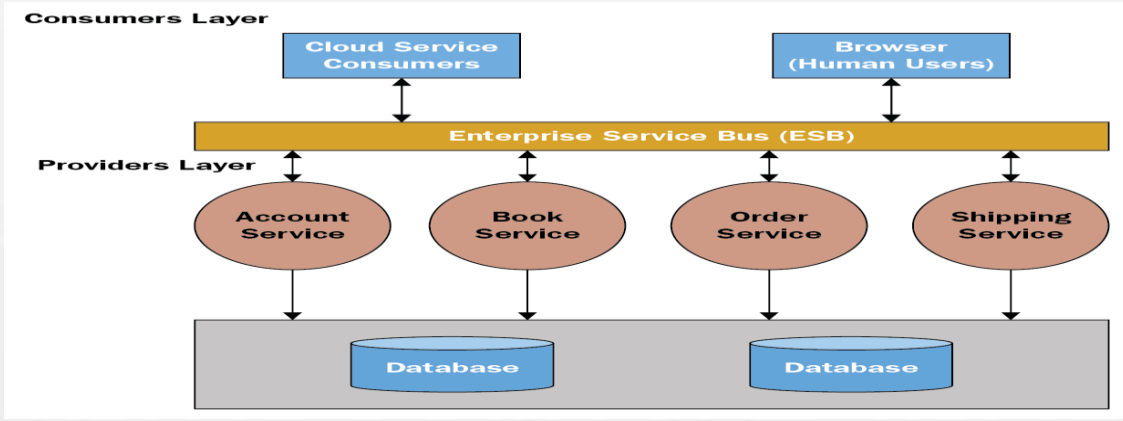


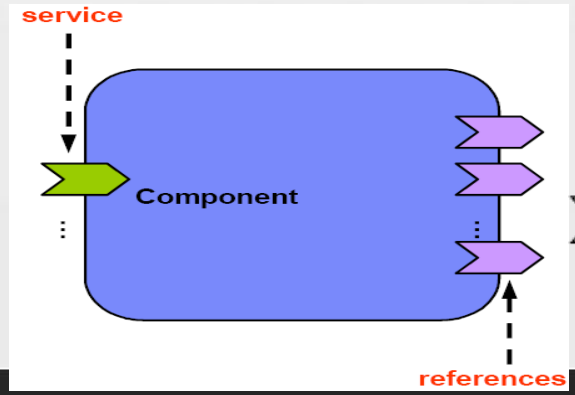


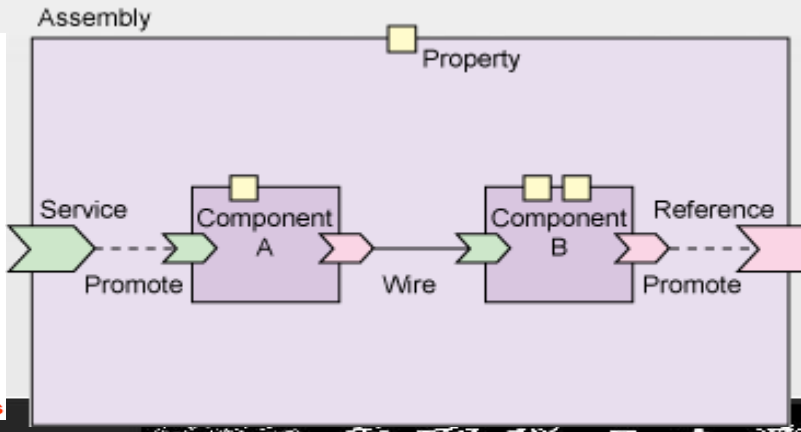
4.4 Logical Architecture and UML Package Diagrams

- **Software architectural patterns and styles**
  - Service-oriented (SOA)
    - A **service** is an application function composed of serially reusable components that can be used in a business process









4.4 Logical Architecture and UML Package Diagrams

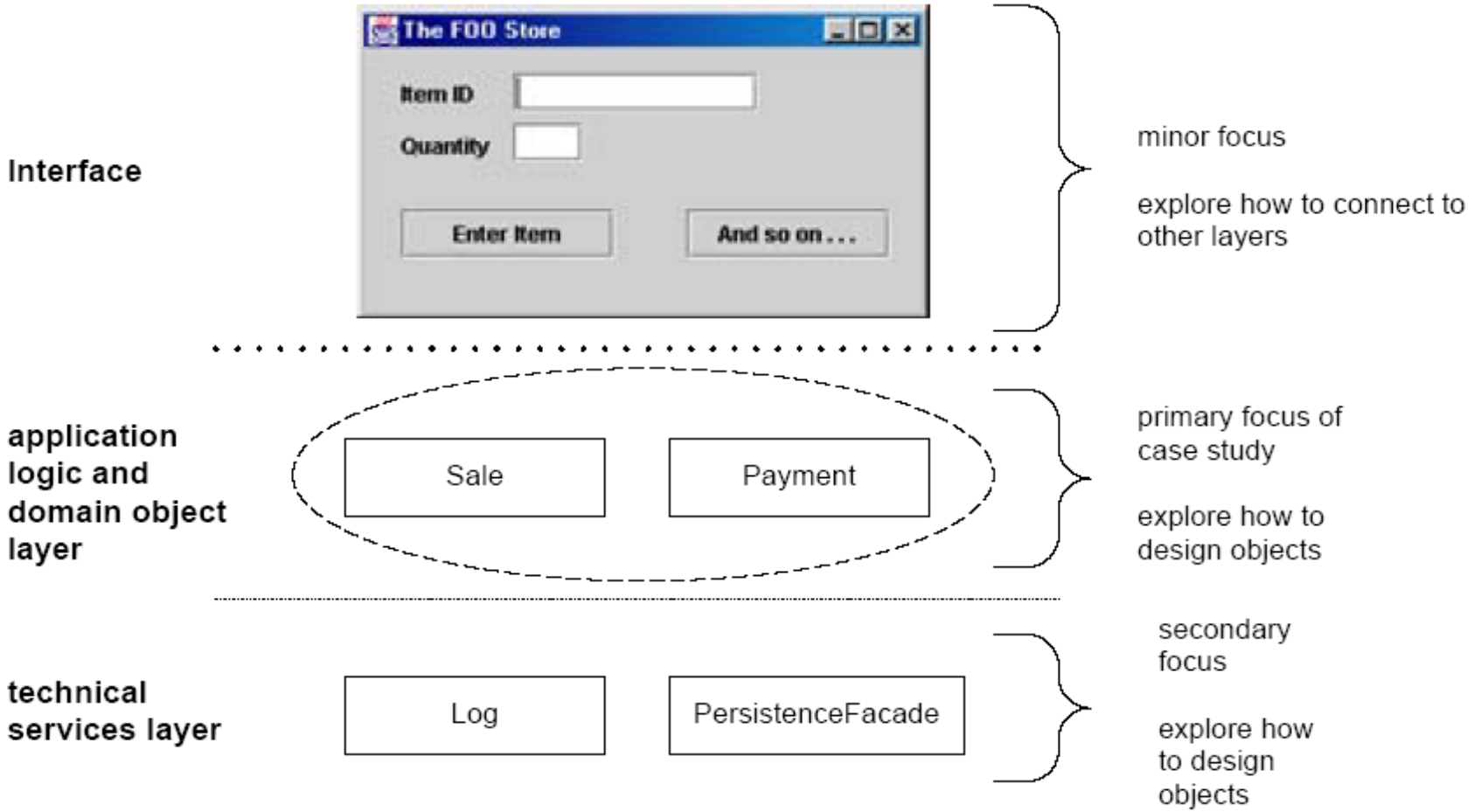
What is the Logical Architecture? And Layers?

- **The logical architecture is the large-scale organization of the software classes into packages (or namespaces), subsystems, and layers**

- **A layer** is a very coarse-grained grouping of classes, packages, or subsystems that has cohesive responsibility for a major aspect of the system. Also, layers are organized such that "higher" layers (such as the UI layer) call upon services of "lower" layers, but not vice versa
  - User Interface.
  - Application Logic and Domain Objects software objects
    - software objects that fulfill application requirements
  - Technical Services
    - Always application-independent and reusable across several systems, such as interfacing with a database or error logging

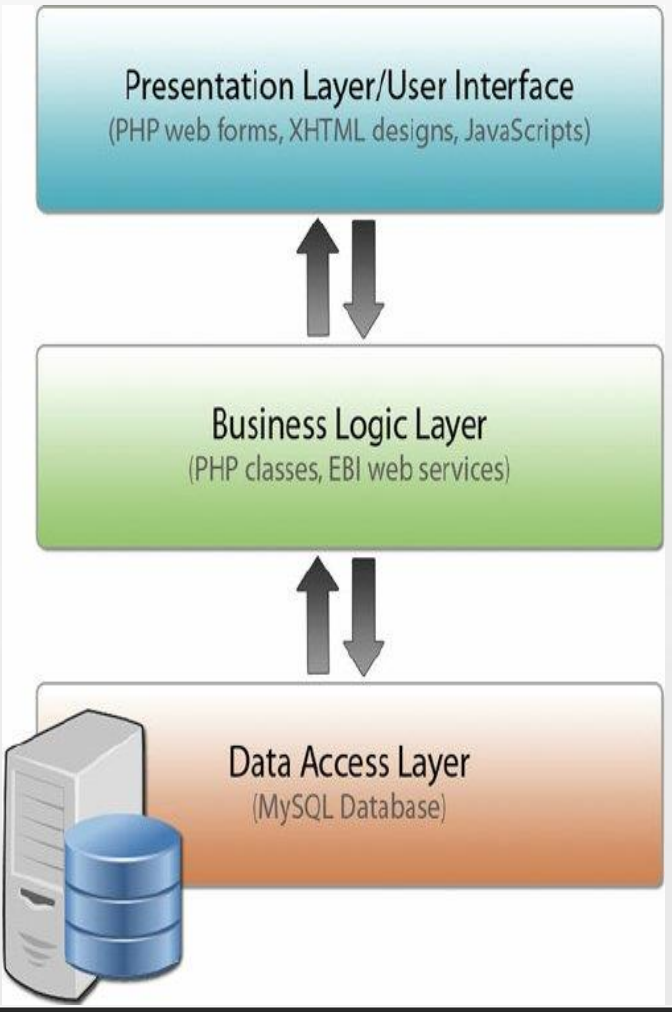
4.4 Logical Architecture and UML Package Diagrams

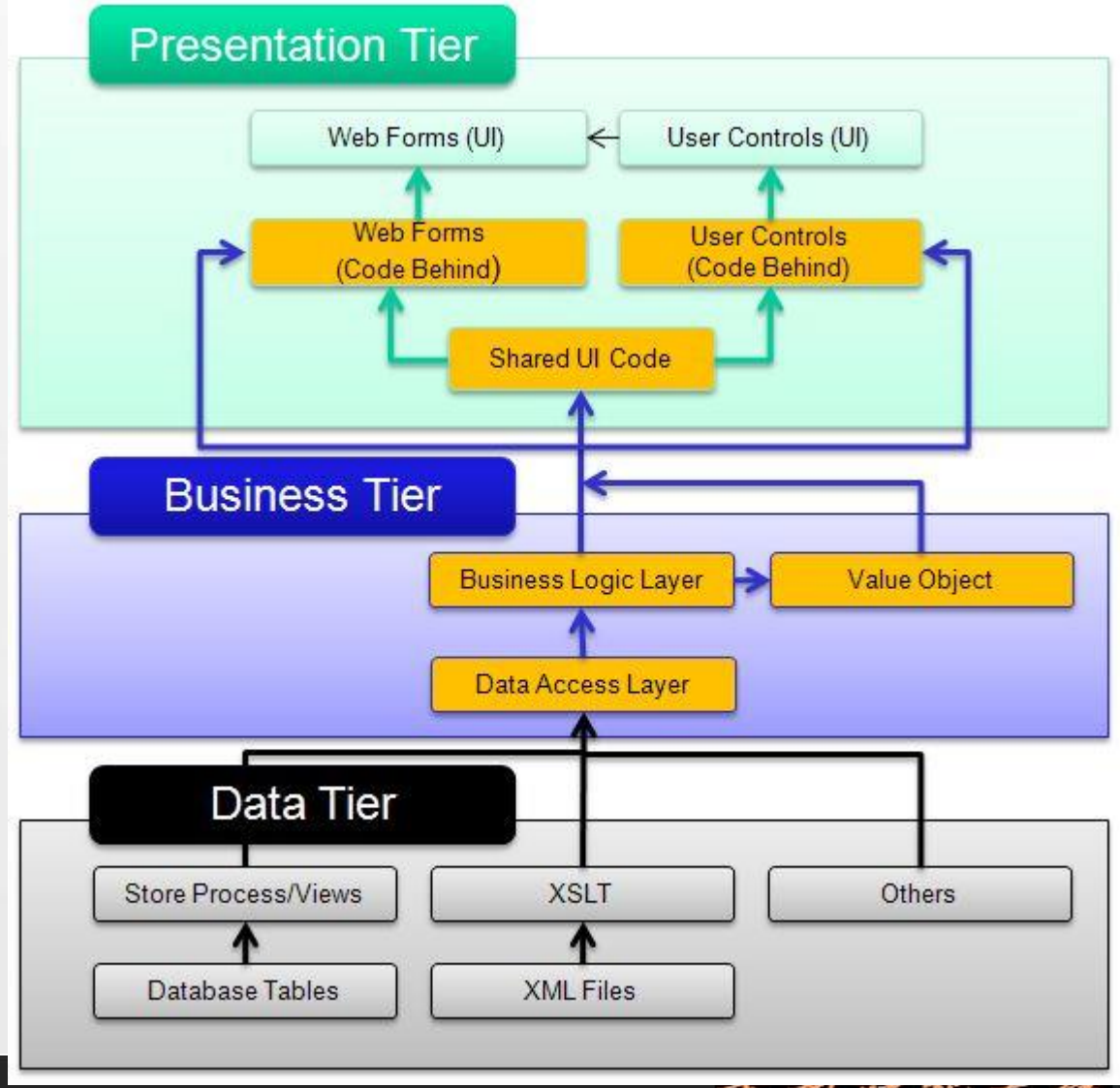
- **What is the Logical Architecture? And Layers?**
  - Architecture layers



4.4 Logical Architecture and UML Package Diagrams

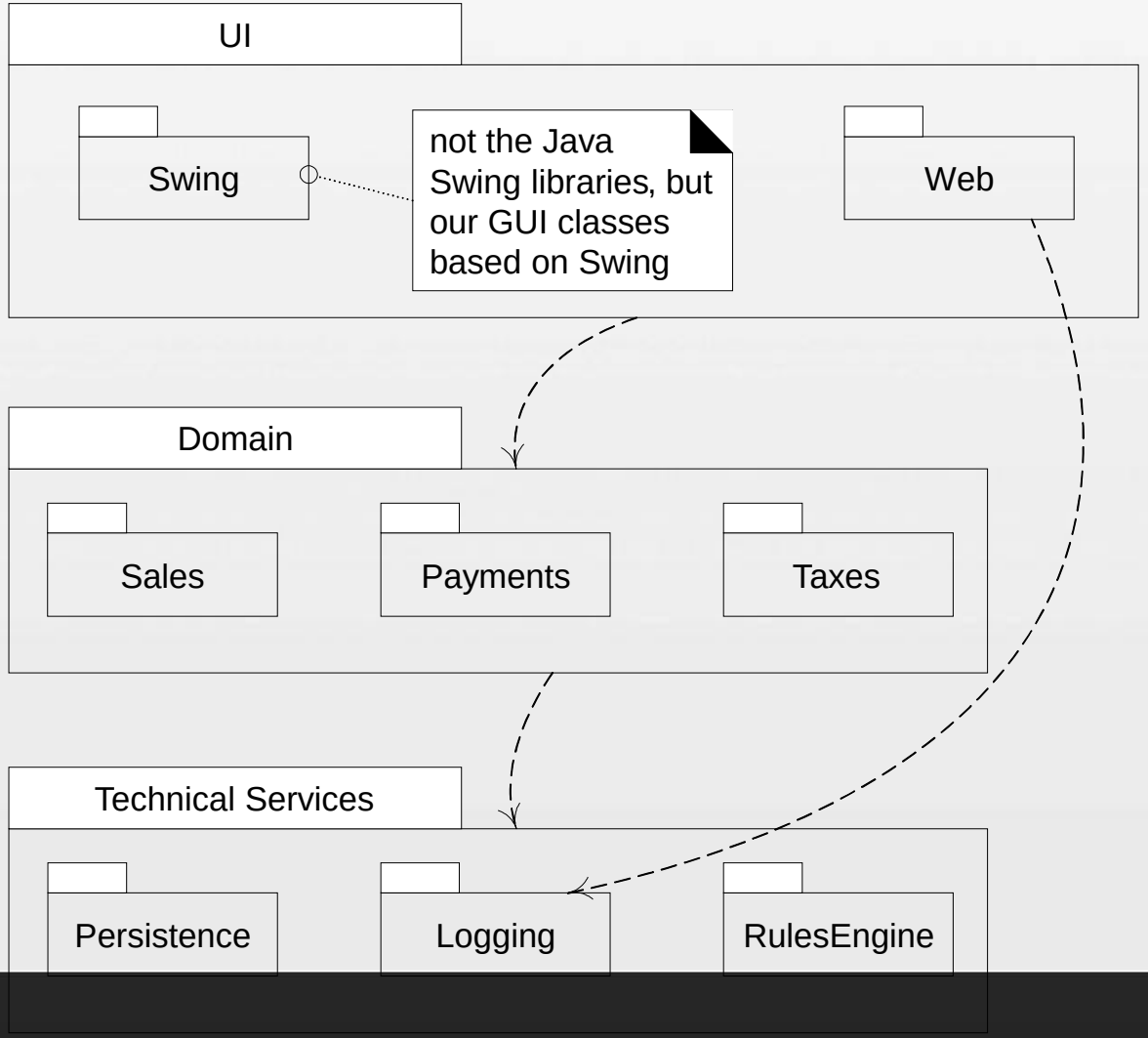
- **What is the Logical Architecture? And Layers?**
  - Architecture layers





4.4 Logical Architecture and UML Package Diagrams

- **What is the Logical Architecture? And Layers?**
  - **Layers shown with UML package diagram notation**



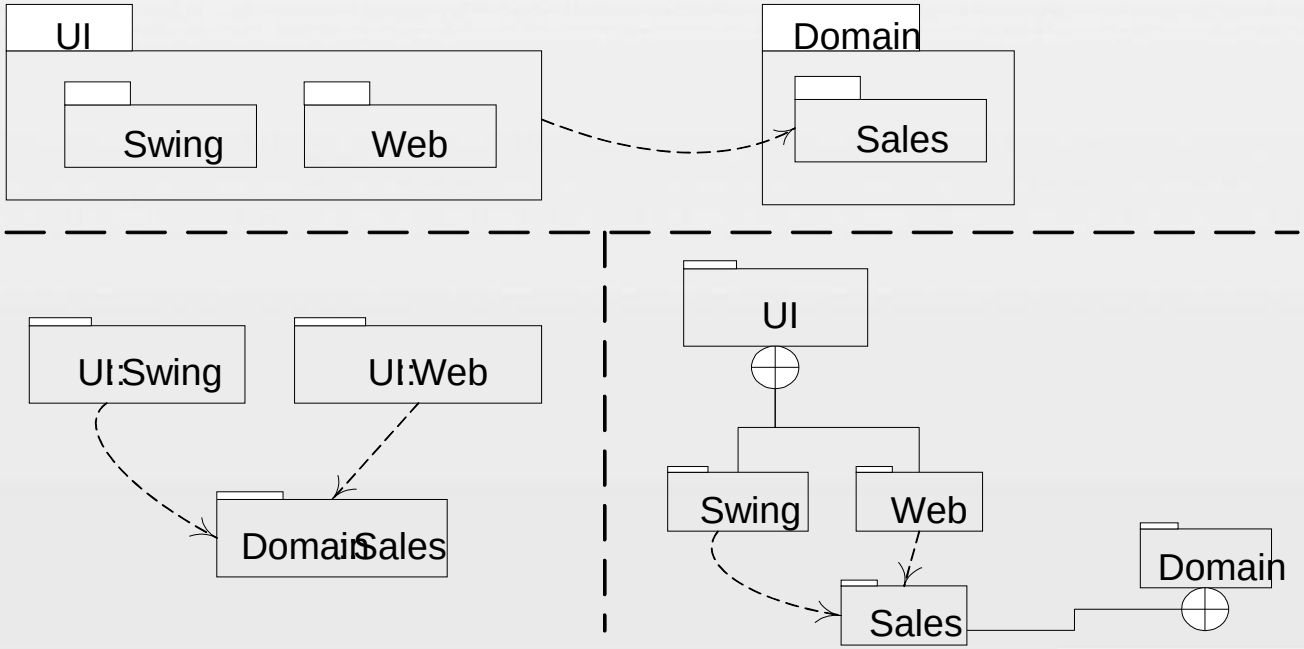
information systems, have a relaxed layered architecture, in which a higher layer calls upon several lower layers. For example, the UI layer may call upon its directly subordinate application logic layer, and also upon elements of a lower technical service layer, for logging and so forth.

A logical architecture doesn't have to be organized in layers. But it's very common

4.4 Logical Architecture and UML Package Diagrams

Applying UML: Package Diagrams

- UML package diagrams are often used to illustrate the logical architecture of a system the layers, subsystems, packages
- A UML package represents a **namespace**, can group anything: classes, other packages, use cases, and so on
- UML dependency line is used to show **dependency** between packages
- **Nesting packages** is very common



4.4 Logical Architecture and UML Package Diagrams

Design with Layers

- **essential ideas of using layers**
  - Organize the large-scale logical structure of a system into discrete layers of distinct, related responsibilities, with a clean, cohesive separation of concerns such that the "lower" layers are low-level and general services, and the higher layers are more application specific.
  - Collaboration and coupling is from higher to lower layers; lower-to-higher layer coupling is avoided.
- **Using layers helps address several problems**
  - Source code changes are rippling throughout the system
  - Application logic is intertwined with the user interface, so it cannot be reused with a different interface or distributed to another node.
  - Potentially general technical services or business logic is intertwined with more application-specific logic, so it cannot be reused
  - high coupling across different areas of concern. It is thus difficult to divide the work along clear boundaries for different developers.

4.4 Logical Architecture and UML Package Diagrams

Design with Layers

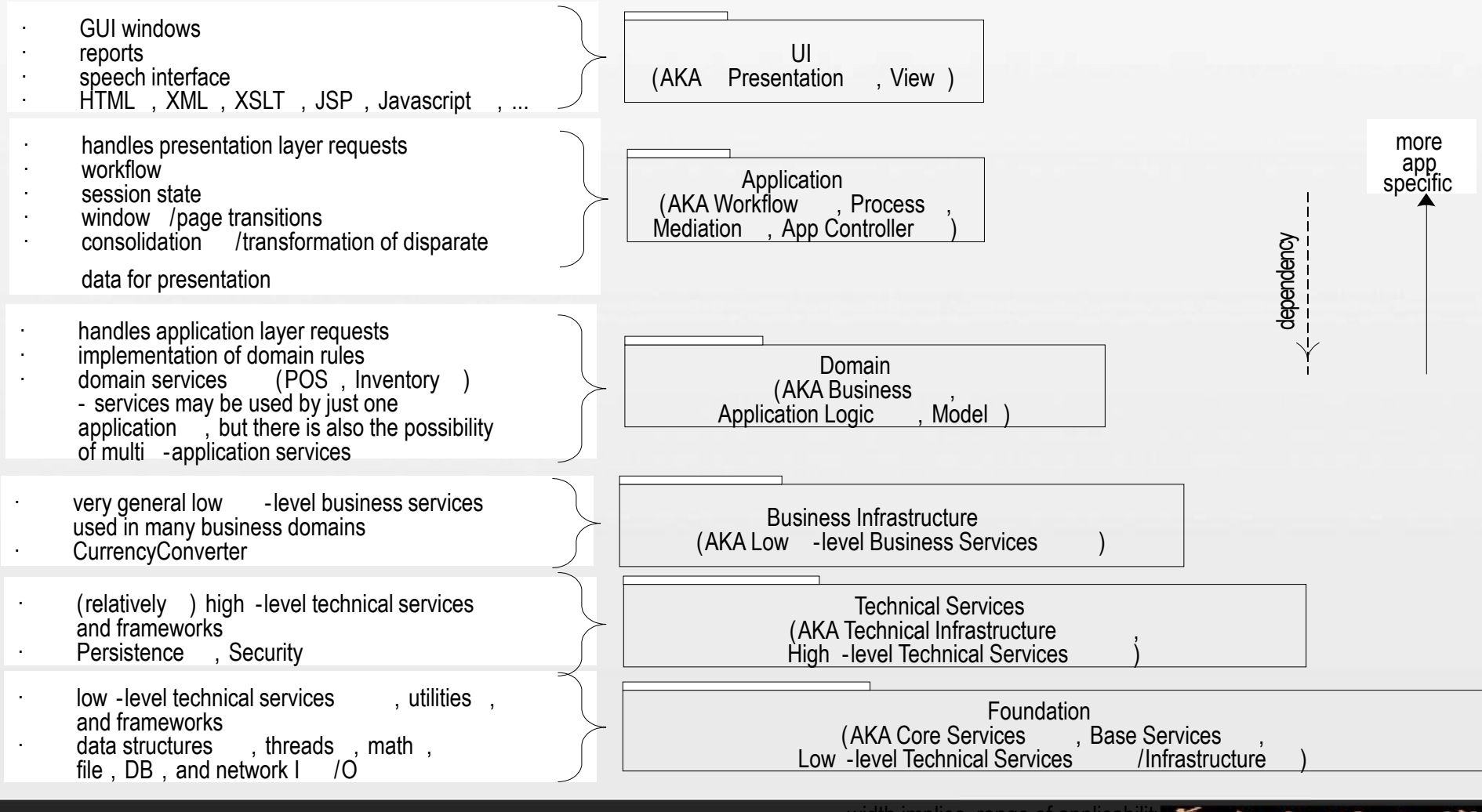
Benefits of Using Layers

- A separation of concerns, a separation of high from low-level services, and of application-specific from general services. This reduces coupling and dependencies, improves cohesion, increases reuse potential, and increases clarity.
- Related complexity is encapsulated and decomposable.
- Some layers can be replaced with new implementations.
- Lower layers contain reusable functions.
- Some layers (primarily the Domain and Technical Services) can be distributed.
- Development by teams is aided because of the logical segmentation.

4.4 Logical Architecture and UML Package Diagrams

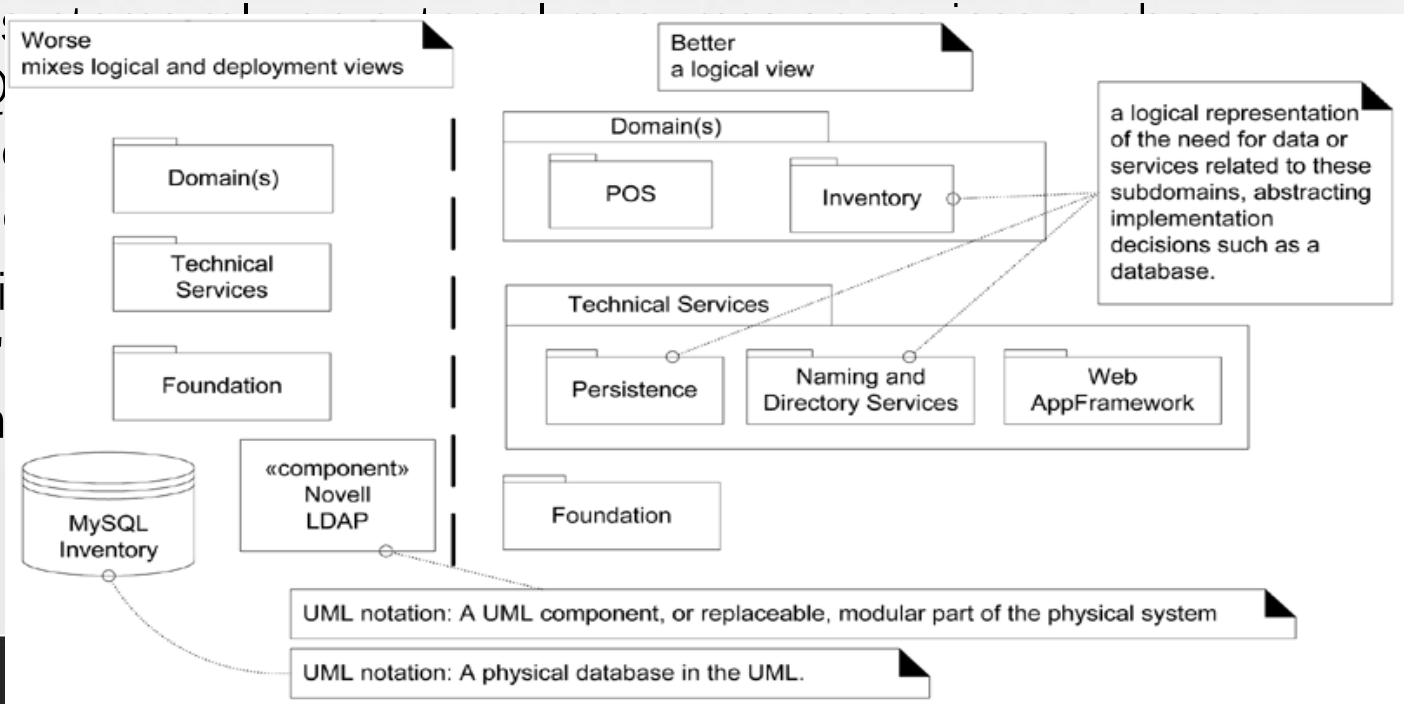
Design with Layers

Common layers in an information system logical architecture



4.4 Logical Architecture and UML Package Diagrams

- **Design with Layers**
  - **Guidelines:**
    - **1.Cohesive Responsibilities; Maintain a Separation of Concerns**
      - The responsibilities of the objects in a layer should be strongly related to each other and should not be mixed with responsibilities of other layers
    - **2.Don't Show External Resources as the Bottom Layer**
      - Most
        MySQ
        direct
        comp
      - Showi
        layer
        logica



4.4 Logical Architecture and UML Package Diagrams

Design with Layers

Guidelines:

3.The Model-View Separation Principle

- Do not connect or couple non-UI objects directly to UI objects
- Do not put application logic (such as a tax calculation) in the UI object methods
- domain classes encapsulate the information and behavior related to application logic. The window classes are relatively thin; they are responsible for input and output, and catching GUI events, but do not maintain application data or directly provide application logic
- the Observer pattern is a legitimate relaxation of this principle

4.4 Logical Architecture and UML Package Diagrams

Domain Layer vs. Application Logic Layer; Domain Objects

- **domain object is** software objects with names and information similar to the real-world domain, and assign application logic responsibilities to them.
  - represents a thing in the problem domain space, and has related application or business logic, for example, a Sale object being able to calculate its total

- **domain layer** is application logic layer that contains domain objects to handle application logic work

- **Relationship Between the Domain Layer and Domain Model?**
  - The domain layer is part of the software and the domain model is part of the conceptual-perspective analysis ,**they aren't the same thing.** But by creating a domain layer with inspiration from the domain model, we achieve a **lower representational gap, between the real-world domain, and our software design**

4.4 Logical Architecture and UML Package Diagrams

■ Domain Layer vs. Application Logic Layer; Domain Objects

➢ Relationship Between the Domain Layer and Domain Model?

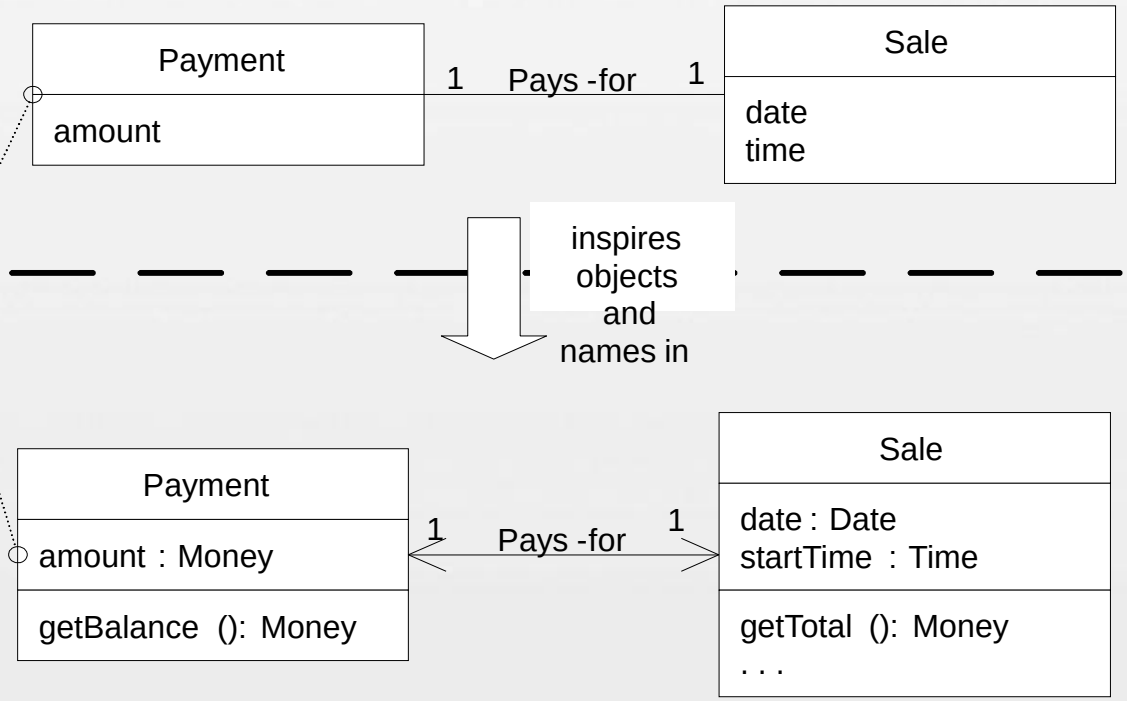
A Payment in the Domain Model is a concept , but a Payment in the Design Model is a software class . They are not the same thing , but the former inspired the naming and definition of the latter .

This reduces the representational gap .

This is one of the big ideas in object technology .

UP Domain Model

Stakeholder 's view of the noteworthy concepts in the domain .



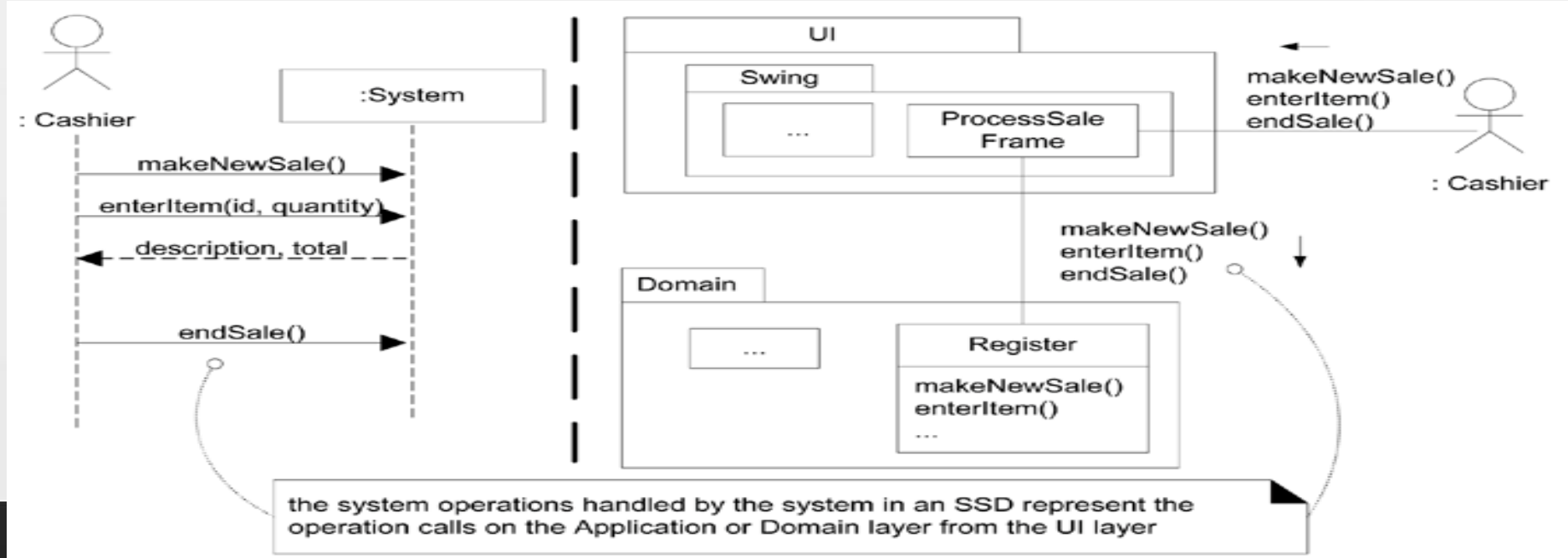
Domain layer of the architecture in the UP Design Model

The object -oriented developer has taken inspiration from the real world domain in creating software classes .

Therefore , the representational gap between how stakeholders conceive the domain , and its representation in software , has been lowered.

4.4 Logical Architecture and UML Package Diagrams

- **Connection Between SSDs, System Operations, and Layers?**
  - The SSDs illustrate these system operations, but hide the specific UI objects. normally it will be objects in the UI layer that capture these system operation requests, then forward or delegate the request from the UI layer onto the domain layer for handling
  - **Key Point:**
    - **The messages sent from the UI layer to the domain layer will be the messages illustrated on the SSDs,**



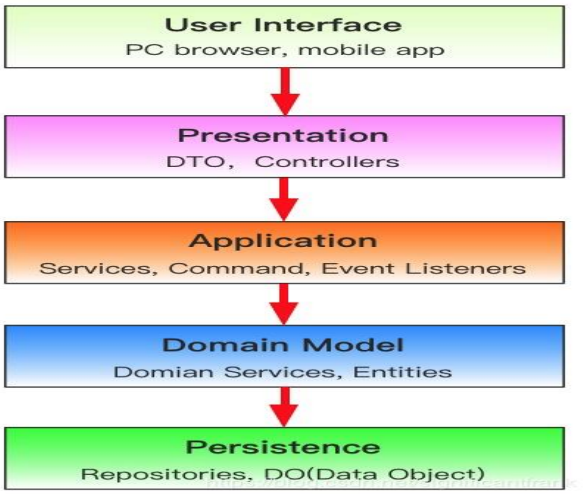
4.4 Logical Architecture and UML Package Diagrams

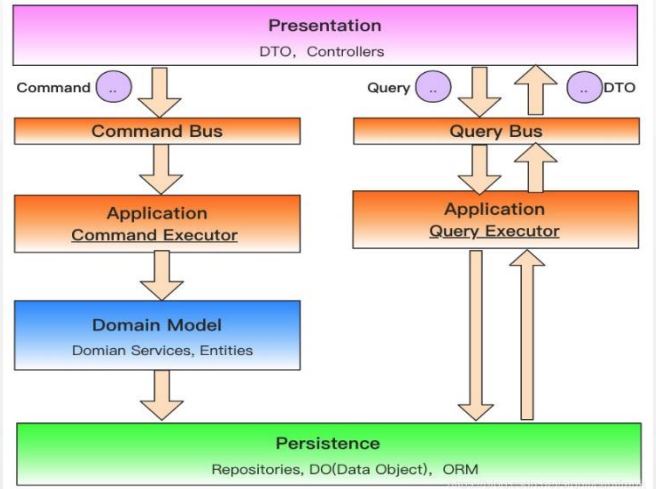
- **Example: NextGen Logical Architecture and Package Diagram**

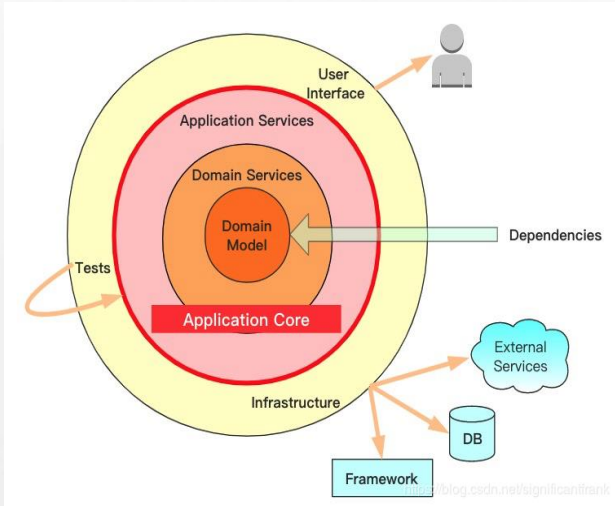
  - Figure 13.2 hints at the simple logical architecture for this iteration. Things get more interesting in later iterations; for example, see many examples of the NextGen logical architecture and package diagrams starting on p. 559.

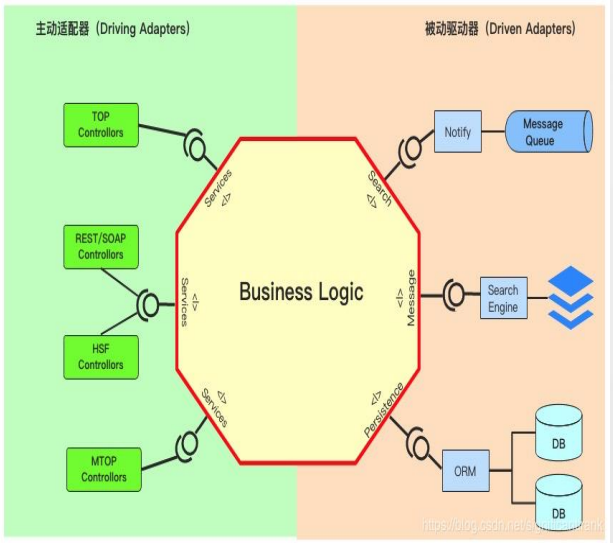
4.4 Logical Architecture and UML Package Diagrams

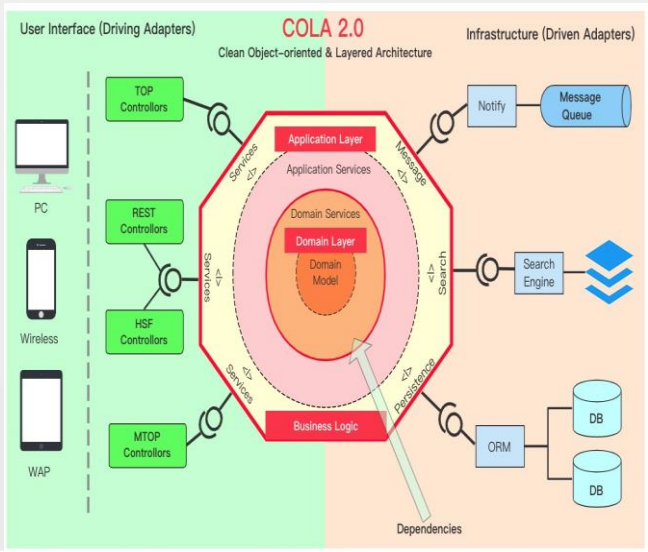
- What are the main layered models

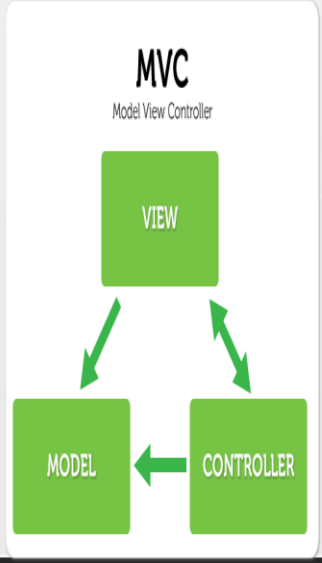


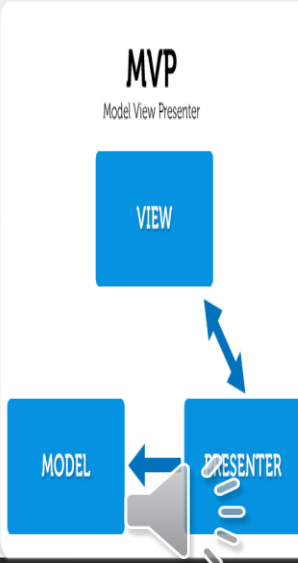




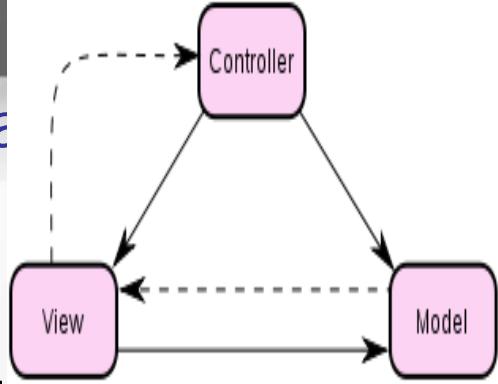








4.4 Logical Architecture and UML Packa



Model –View –Controller

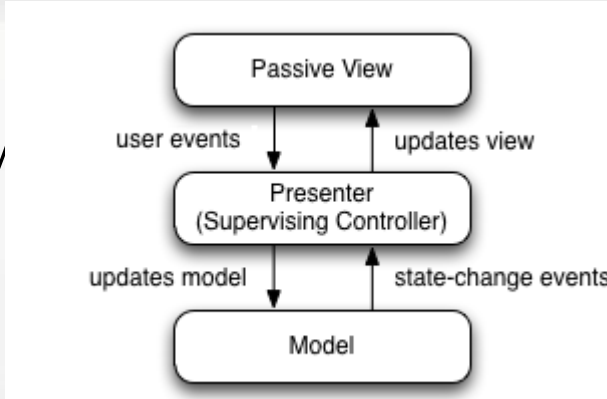
- Model is responsible for
  - **Providing the data from the database and saving the data into the data store(负责数据存取).**
  - **All the business logic are implemented in the Model(负责业务逻辑实现).**
  - **Data entered by the user through View are checked in the model before saving into the database(负责数据验证).**
- View is responsible for:
  - **Taking the input from the user, (获取用户输入)**
  - **Dispatching the request to the controller, and then (向controller发送请求)**
  - **Receiving response from the controller and displaying the result to the user.(接收来自Controller的反馈并将model的处理结果发给用户)**
- Controller is responsible for:
  - **Receiving the request from client.(接收来自客户的请求)**
  - **调用model执行)**
  - **调用View显示执行结果**

4.4 Logical Architecture and UML Package Diagrams

- **Model –View –Controller**
  - **J2EE:**
    - **Struts**
    - **Spring MVC**
  - **PHP**
    - **CakePHP**
    - **Strusts4php**
  - **C#.NET**
    - **Girders**
  - **Ruby on Rails**
  - **Go**
    - **Beego**
  - **Python**
    - Django
    - FLASK

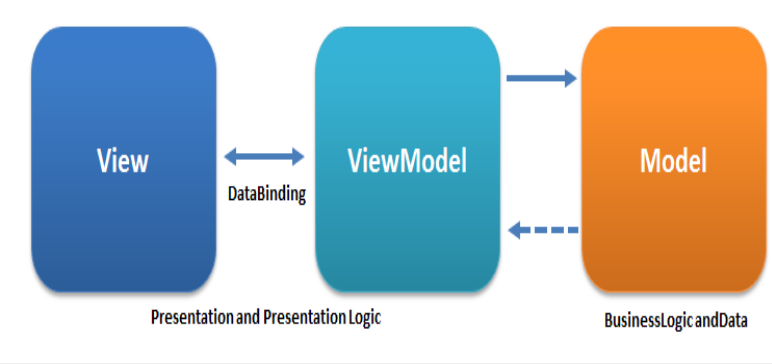
4.4 Logical Architecture and UML Package Diagrams

- **Model –View –Presenter**
  - is a derivation of the model–view–controller (MV
    and is used mostly for building user interfaces.



  - Java
    - JavaFX、Echo2、Google Web Toolkit、JFace、Swing、Vaadin、ZK
  - PHP: CodeIgniter、Laravel、Nette Framework

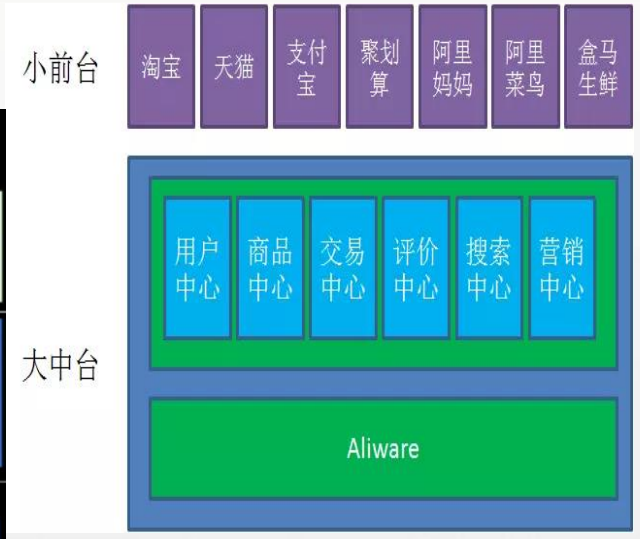
- **Model –View –Template**

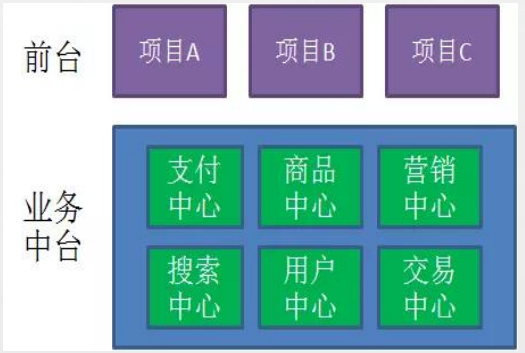


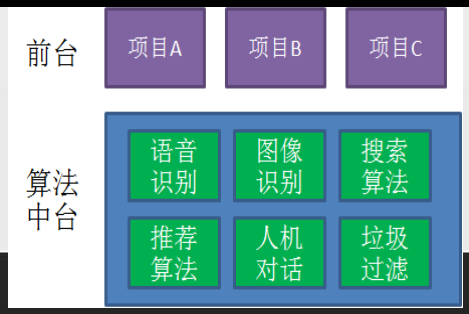
- **Model-viewmodel-view(MVVM)**
  - JavaScript frameworks
    - Angular、Aurelia、Durandal、Ember.js、Ext_JS、Knockout.js、Omi.js、Oracle JET、Svelte、Vue.js
  - .Net frameworks
    - ….

4.4 Logical Architecture and UML Package Diagrams

- 漫画前台中台后台







4.4 Logical Architecture and UML Package Diagrams

- **Logical Architecture summary**
  - **Definition**
  - **4+1 Views of software architecture**
    - UML package diagram
  - **How to design?**
    - Principle (Coupling and Cohesion)
    - Patterns (**Layered architecture**)
    - Design your architecture （推荐大家看架构设计五部曲 infoq）
      - *Logical view*: -functional requirement
      - *Development view*
      - *Physical view*
      - **Process View. Analysis Mechanisms in RUP**
        - » Persistency, Inter-process Communication, Message routing, Process control and synchronization, Transaction management, Information Exchange, Security,Redundancy, Error reporting, Format conversion
      - Data view?

4.4 Logical Architecture and UML Package Diagrams

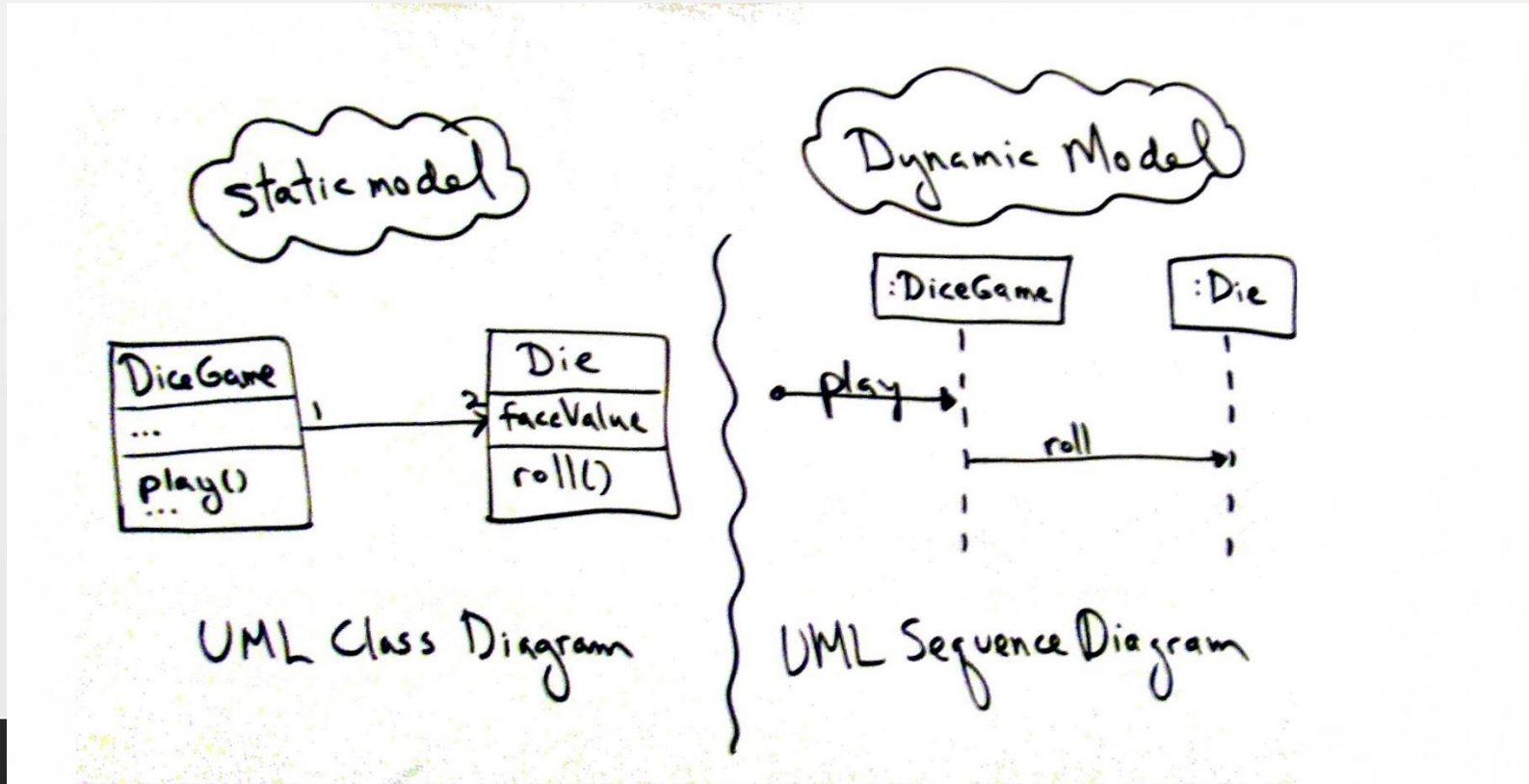
- Software Architecture Document (template example)
  - Introduction
  - Architectural Representation
  - Architectural Goals and Constraints
  - Use-Case View
  - Logical View
  - Deployment View
  - Process View
  - Implementation View
  - Data View (optional)
  - Size and Performance
  - Quality

4.5 Mapping Analysis Models to Design Models

- Goal of this section
  - Understand dynamic and static object design modeling.
  - Try agile modeling, or a UML CASE tool for drawing.

4.5 Mapping Analysis Models to Design Models

- **Designing Objects: What are Static and Dynamic Modeling?**
  - Dynamic models, such as UML interaction diagrams (sequence diagrams or communication diagrams), help design the logic, the behavior of the code or the method bodies.
  - Static models, such as UML class diagrams, help design the definition of packages, class names, attributes, and method signatures (but not method bodies).

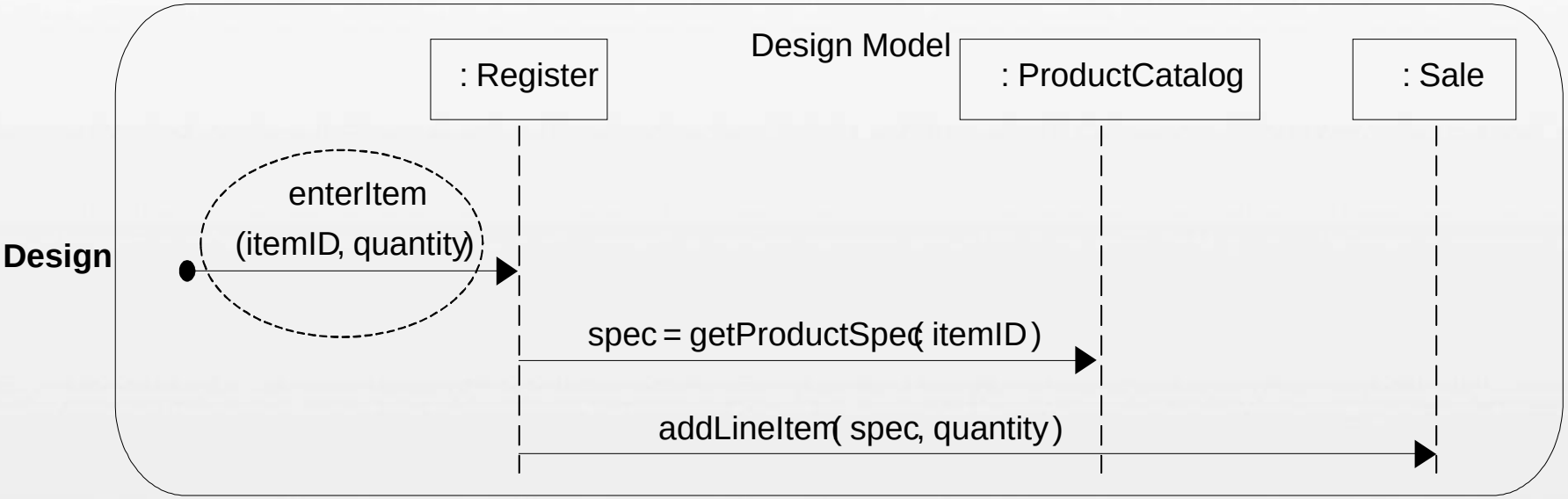


4.5 Mapping Analysis Models to Design Models

- **Designing Objects: What are Static and Dynamic Modeling?**
  - **Dynamic Object Modeling**
    - most of the challenging, interesting, useful design work happens while drawing the UML dynamic-view interaction diagrams, such as
      - what objects need to exist
      - how they collaborate via messages and methods
    - **Guideline**
      - Spend significant time doing interaction diagrams (sequence or communication diagrams), not just class diagrams.
    - **key skills in OO design**
      - responsibility-driven design and the GRASP principles
    - other dynamic tools in the UML kit, including state machine diagrams (p. 485) and activity diagrams

4.5 Mapping Analysis Models to Design Models

Designing Objects: What are Static and Dynamic Modeling?



- **Static Object Modeling**
  - most common static object modeling is with UML class diagrams
  - Domain model is concept class diagram

4.5 Mapping Analysis Models to Design Models

Agile Modeling and Lightweight UML Drawing

- aims of agile modeling are to reduce drawing overhead and model to understand and communicate
  - **UML as sketch**
  - Modeling with others.
  - Creating several models in parallel

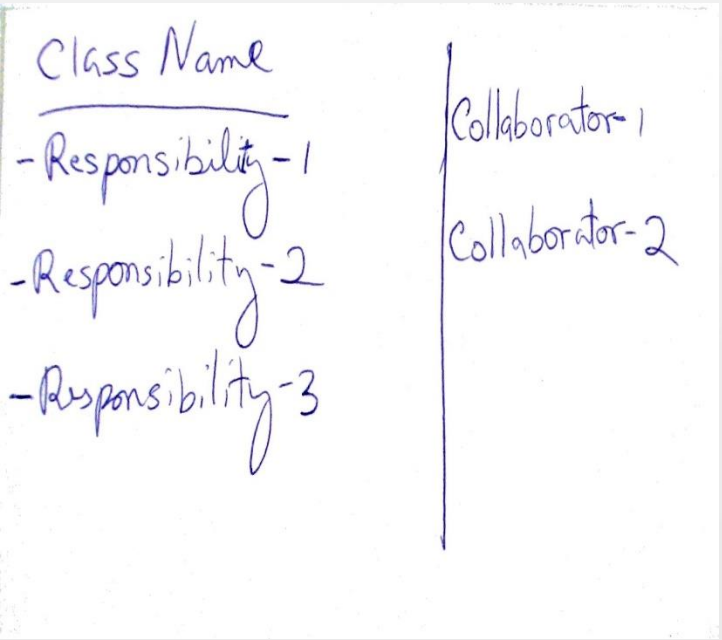
UML CASE Tools

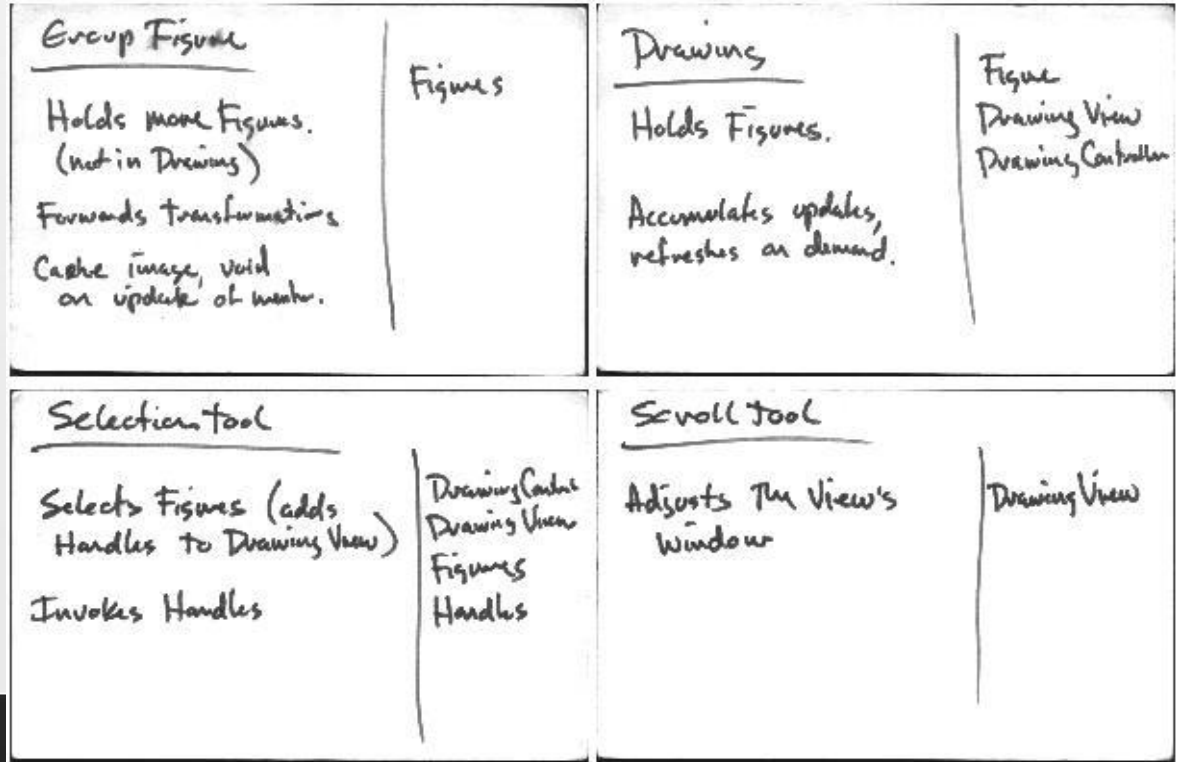
- **Guidelines**
  - Choose a UML CASE tool that integrates with popular text-strong IDEs, such as Eclipse or Visual Studio.
  - Choose a UML tool that can reverse-engineer (generate diagrams from code) not only class diagrams (common), but also interaction diagrams (more rare, but very useful to learn call-flow structure of a program).

4.5 Mapping Analysis Models to Design Models

■ Other Object Design Techniques: CRC Cards

➢**Class Responsibility Collaboration (CRC) cards** are paper index cards on which one writes the responsibilities and collaborators of classes. Each card represents one class. A CRC modeling session involves a group sitting around a table, discussing and writing on the cards as they play "what if" scenarios with the objects, considering what they must do and what other objects they must collaborate with





4.6 UML Interaction Diagrams

Sequence and Communication Diagrams

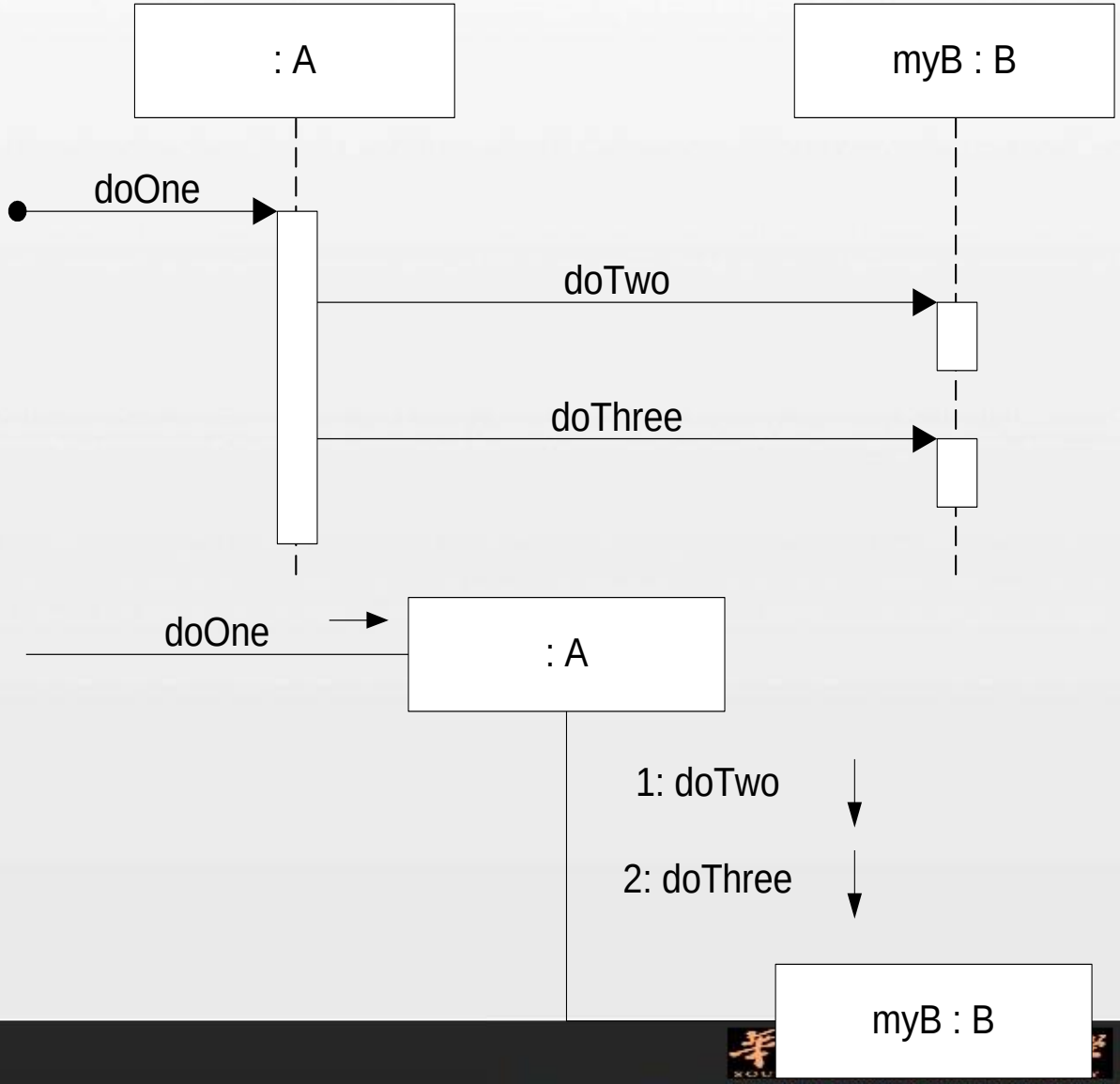
- **UML use interaction diagrams to illustrate how objects interact via messages. They are used for dynamic object modeling**

- **interaction diagram is a generalization of two more specialized UML diagram types:**
  - sequence diagrams
    - Sequence diagrams illustrate interactions in a kind of fence format,
  - communication diagrams
    - Communication diagrams illustrate object interactions in a graph or network format
  - interaction overview diagram;
    - it provides a big-picture overview of how a set of interaction diagrams are related in terms of logic and process-flow

4.6 UML Interaction Diagrams

- **Sequence and Communication Diagrams**

```
public class A {
private B myB = new B();
public void doOne()
{
        myB.doTwo();
        myB.doThree();
}
//….
}
```



4.6 UML Interaction Diagrams

- **Sequence and Communication Diagrams**
  - **Strengths and Weaknesses of Sequence vs. Communication Diagrams?**

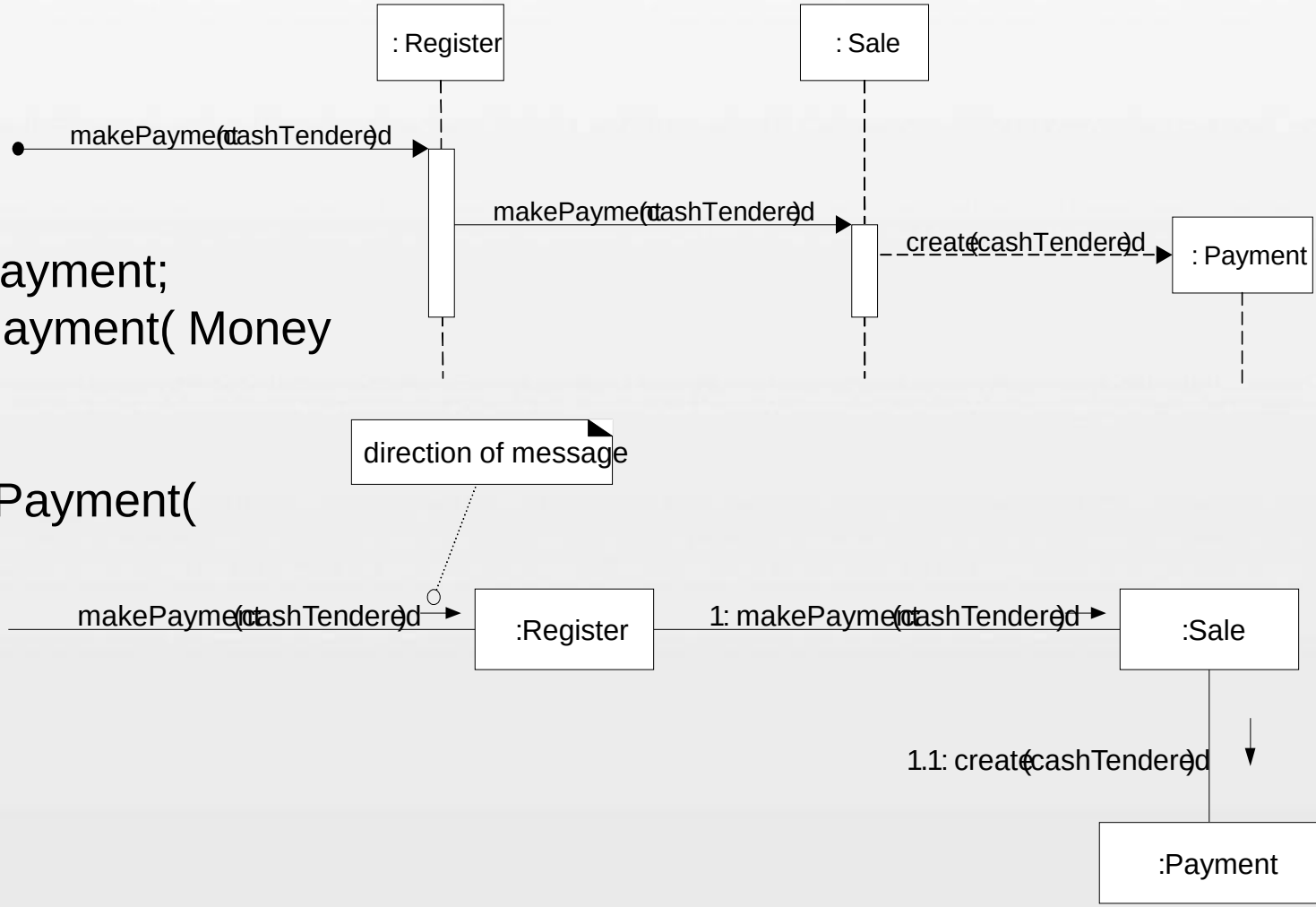
Type	Strengths	Weaknesses
sequence	clearly shows sequence or time ordering of messages large set of detailed notation options	forced to extend to the right when adding new objects; consumes horizontal space
communication	space economicalflexibility to add new objects in two dimensions	more difficult to see sequence of messages fewer notation options

  - **UML specifications are more emphasised on sequence diagrams**

4.6 UML Interaction Diagrams

- **Example: makePayment**

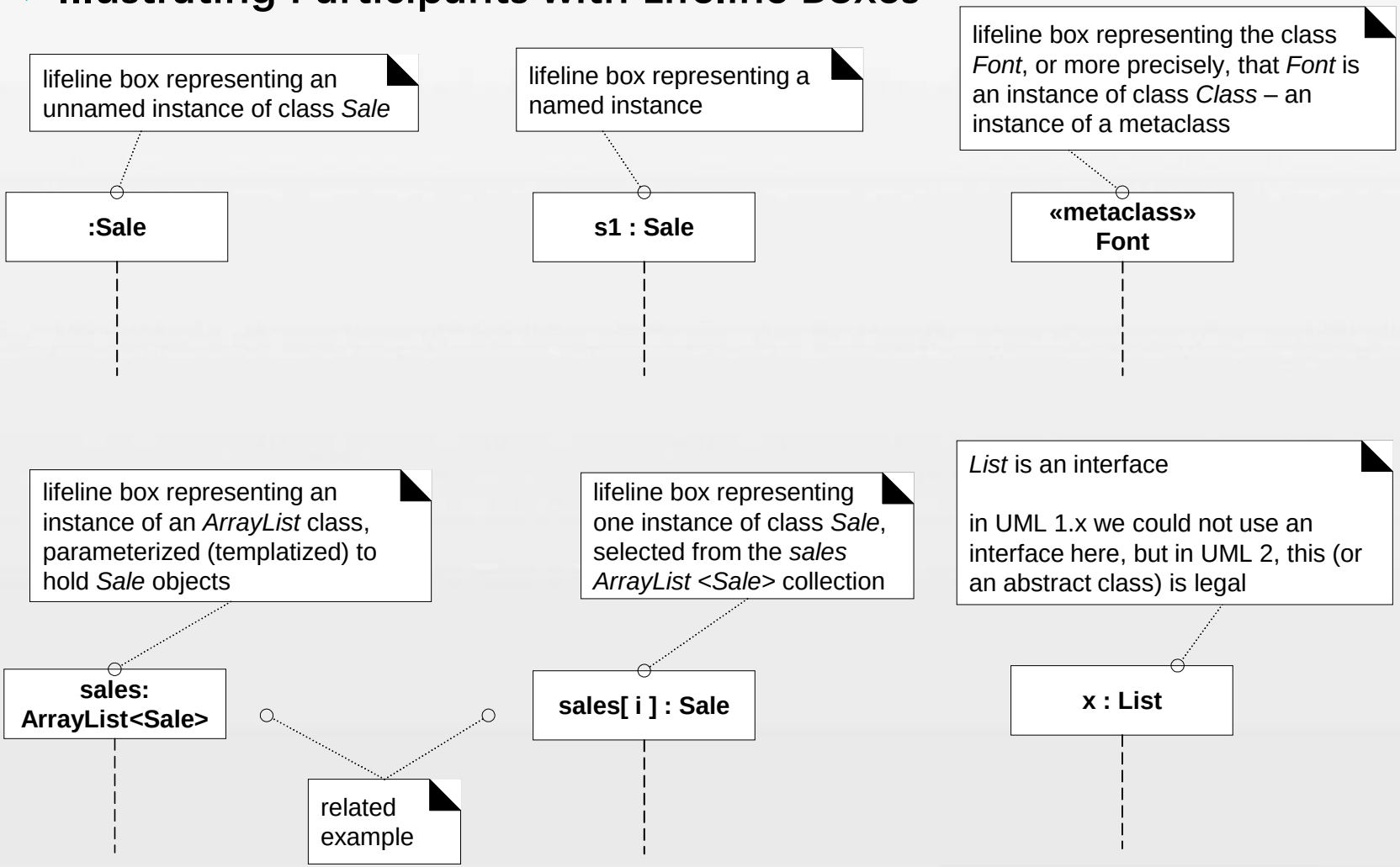
```
public class Sale
{
private Payment payment;
public void makePayment( Money
cashTendered )
{
   payment = new Payment(
cashTendered );
   //…
}
// …
}
```



4.6 UML Interaction Diagrams

Common UML Interaction Diagram Notation

Illustrating Participants with Lifeline Boxes



4.6 UML Interaction Diagrams

Common UML Interaction Diagram Notation

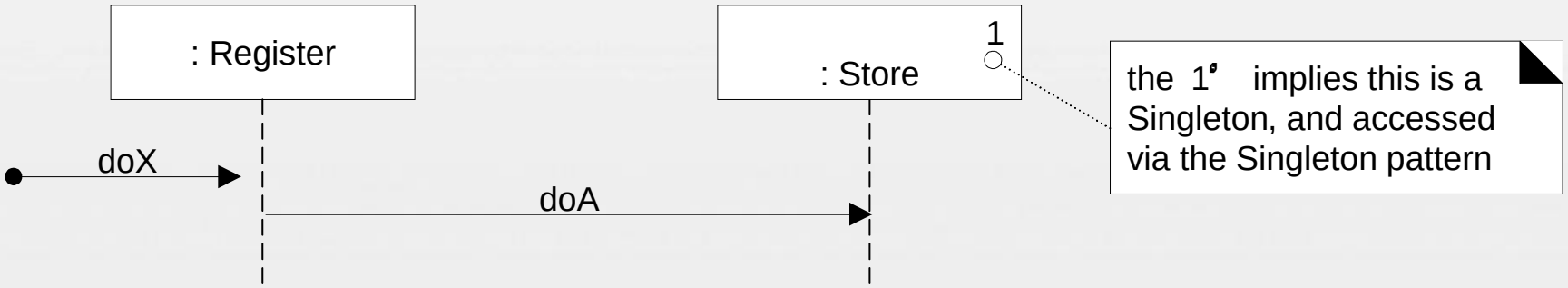
Basic Message Expression Syntax

- return = message(parameter : parameterType) : returnType

- Example
  - initialize(code)
  - Initialize
  - d = getProductDescription(id)
  - d = getProductDescription(id:ItemID)
  - d = getProductDescription(id:ItemID) : ProductDescription

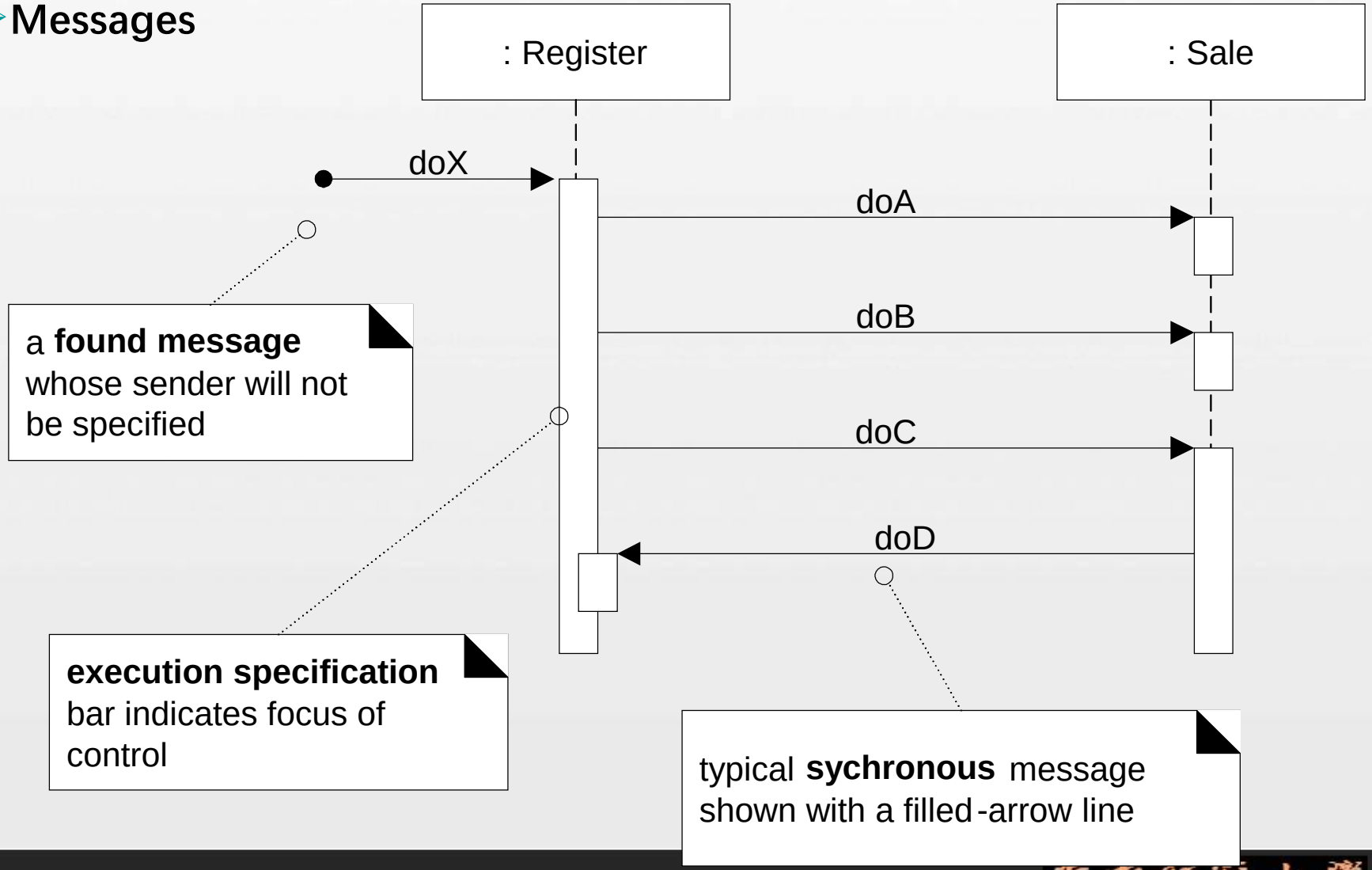
4.6 UML Interaction Diagrams

- Common UML Interaction Diagram Notation
  - Singleton Objects
    - an object is marked with a '1' in the upper right corner of the lifeline box. It implies that the Singleton pattern is used to gain visibility to the object



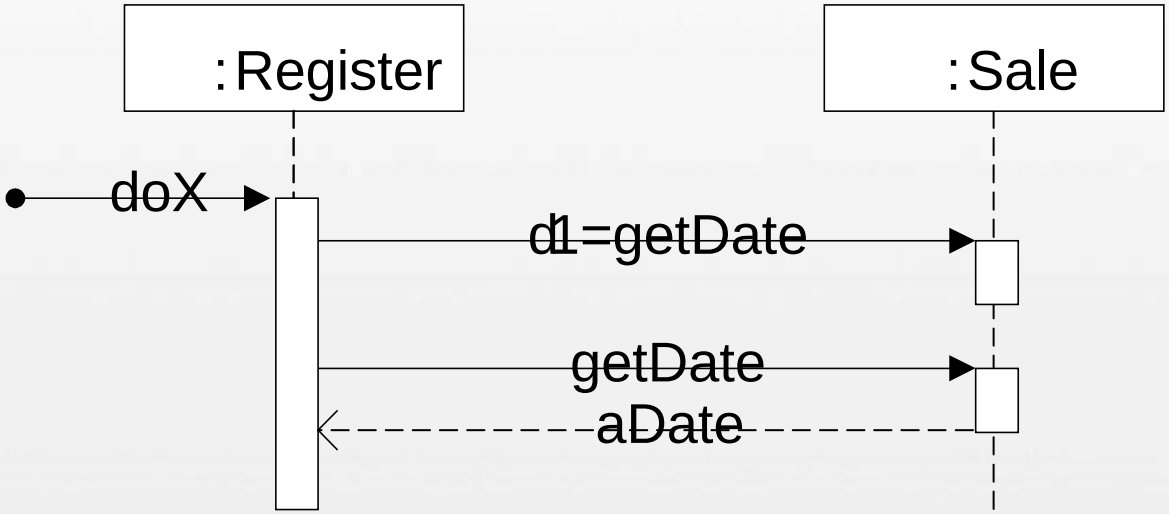
4.6 UML Interaction Diagrams

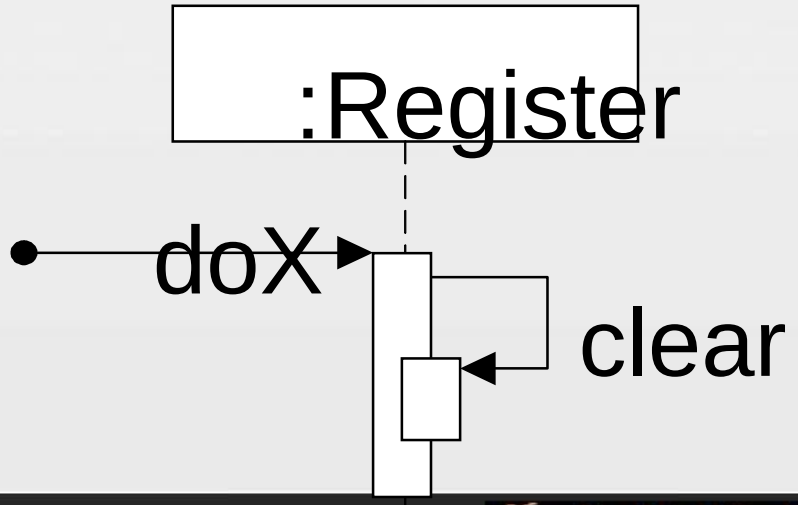
- **Basic Sequence Diagram Notation**
  - Messages



4.6 UML Interaction Diagrams

- **Basic Sequence Diagram Notation**
  - Messages

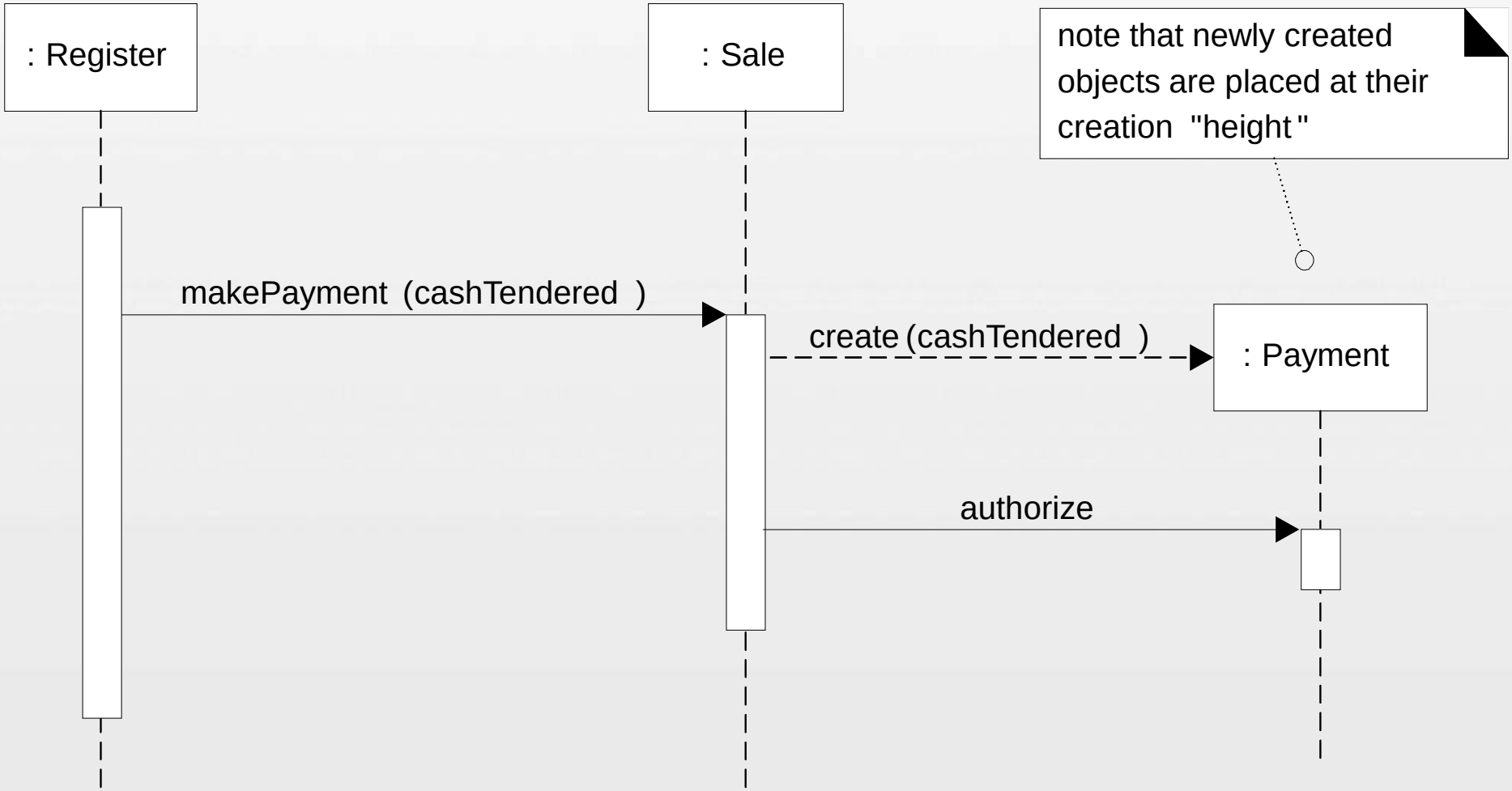




4.6 UML Interaction Diagrams

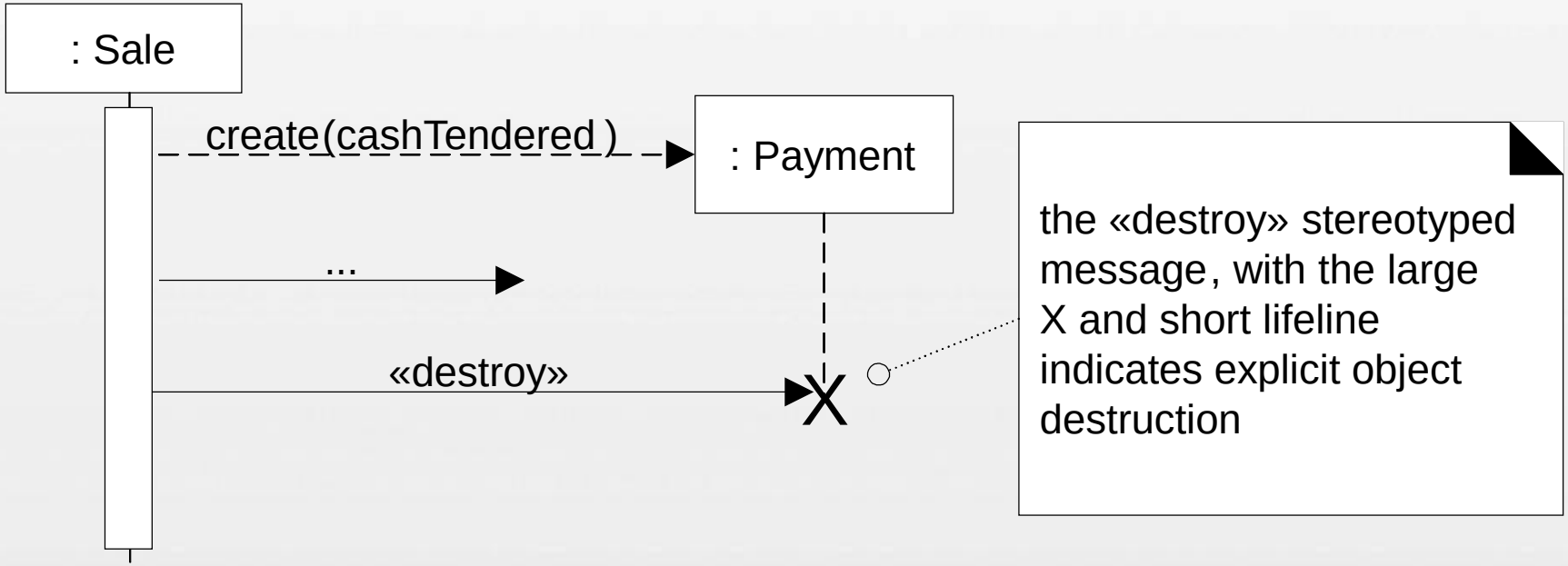
Basic Sequence Diagram Notation

Creation of Instances



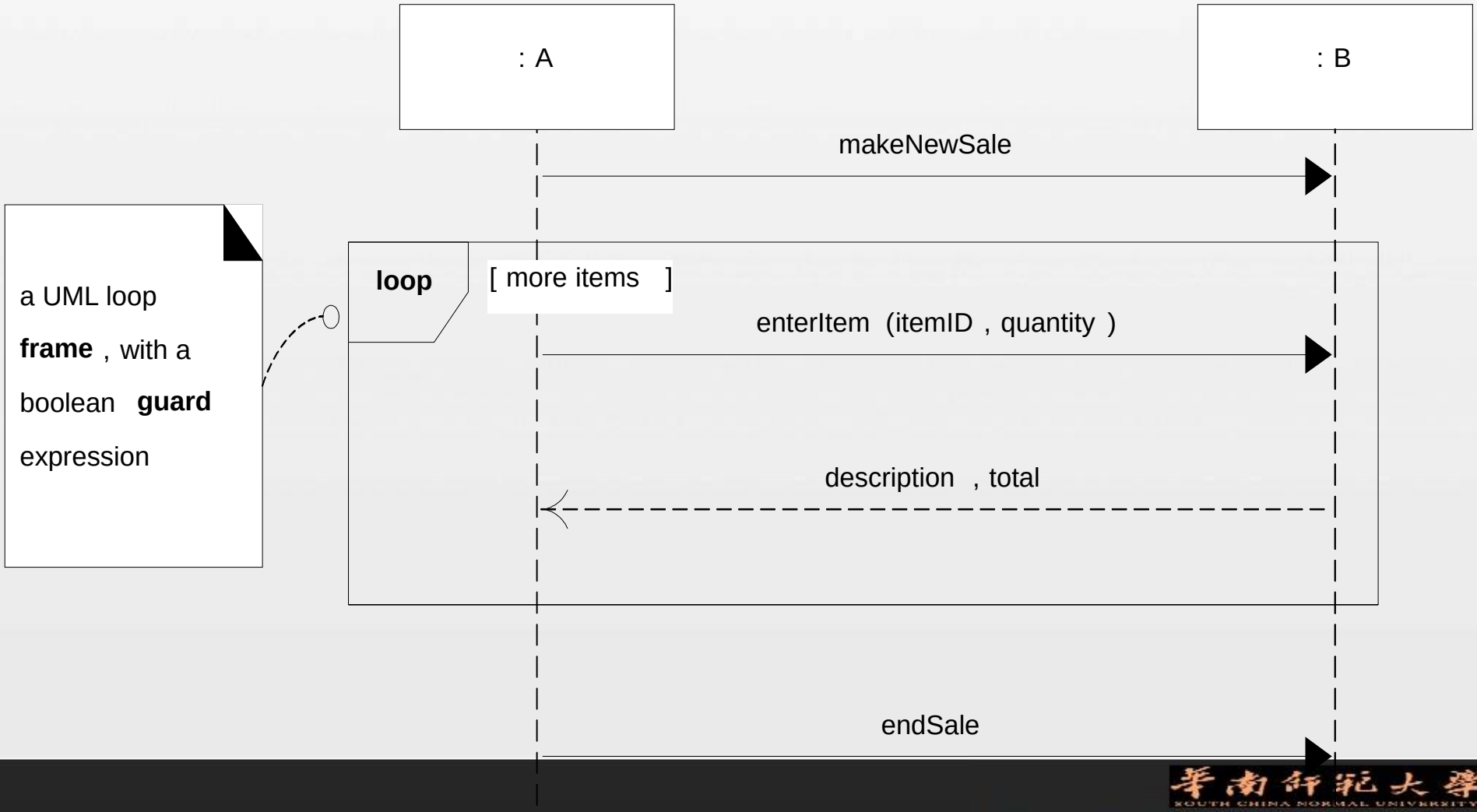
4.6 UML Interaction Diagrams

- **Basic Sequence Diagram Notation**
  - **Object Destruction**



4.6 UML Interaction Diagrams

- **Basic Sequence Diagram Notation**
  - Diagram Frames in UML Sequence Diagrams



4.6 UML Interaction Diagrams

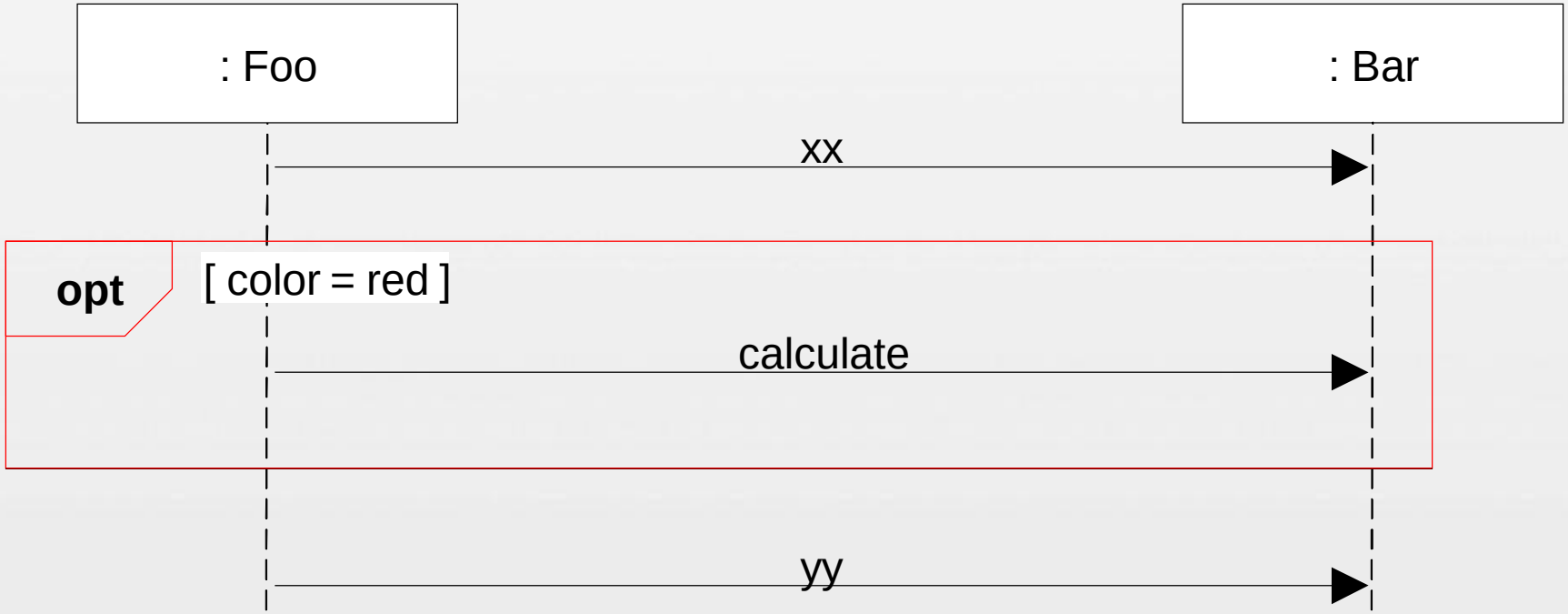
Basic Sequence Diagram Notation

Diagram Frames in UML Sequence Diagrams

Frame Operator	Meaning
alt	Alternative fragment for mutual exclusion conditional logic expressed in the guards.
loop	Loop fragment while guard is true. Can also write loop(n) to indicate looping n times. There is discussion that the specification will be enhanced to define a FOR loop, such as loop(i, 1, 10)
opt	Optional fragment that executes if guard is true.
par	Parallel fragments that execute in parallel.
region	Critical region within which only one thread can run.

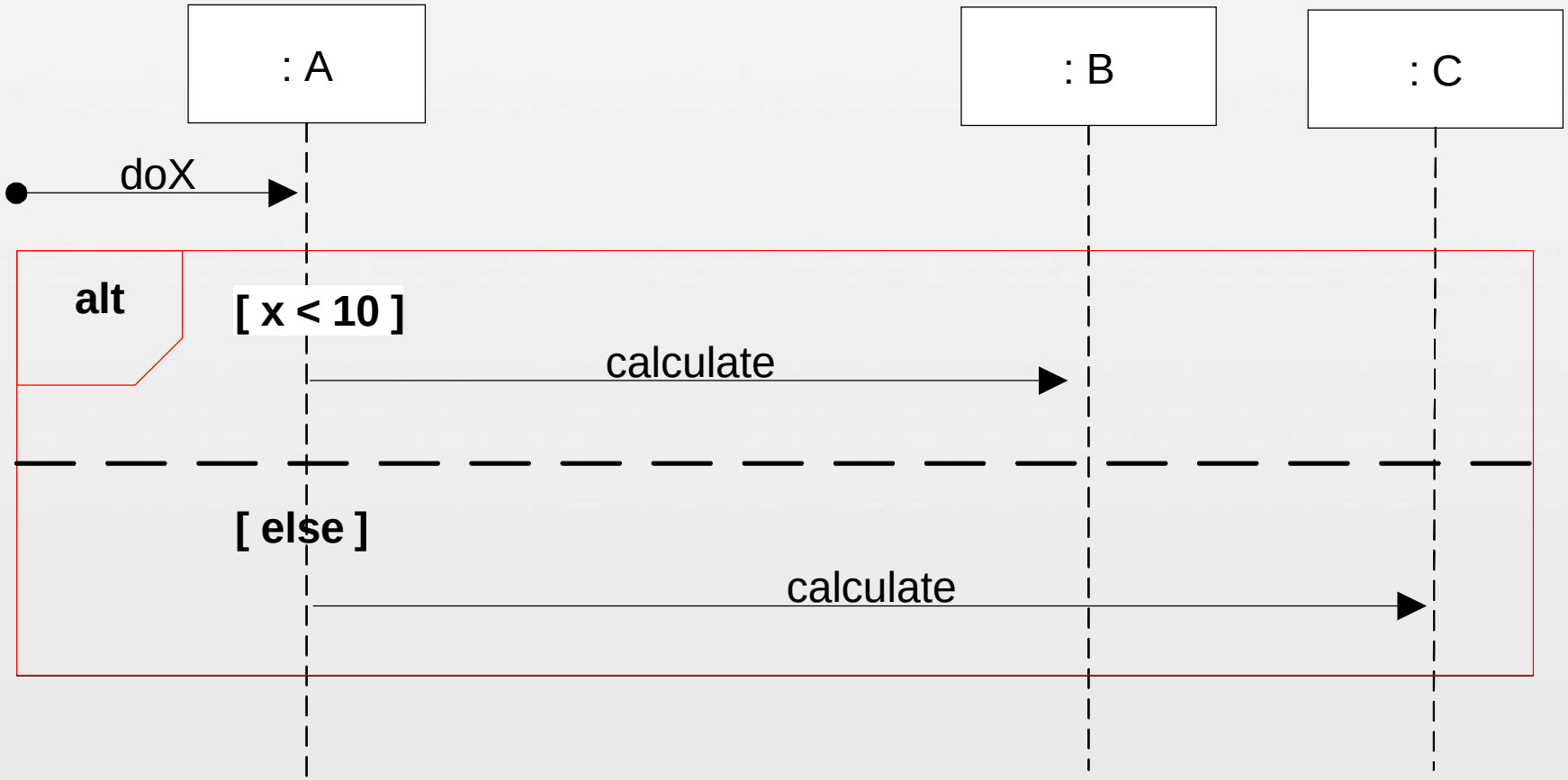
4.6 UML Interaction Diagrams

- **Basic Sequence Diagram Notation**
  - Diagram Frames in UML Sequence Diagrams
    - Conditional Messages



4.6 UML Interaction Diagrams

- **Basic Sequence Diagram Notation**
  - Diagram Frames in UML Sequence Diagrams
    - Mutually Exclusive Conditional Messages



4.6 UML Interaction Diagrams

- **Basic Sequence Diagram Notation**
  - **Diagram Frames in UML Sequence Diagrams**
    - **Iteration Over a Collection**
      - UML specification did not have an official idiom for this case

This lifeline box represents one instance from a collection of many *SalesLineItem* objects .

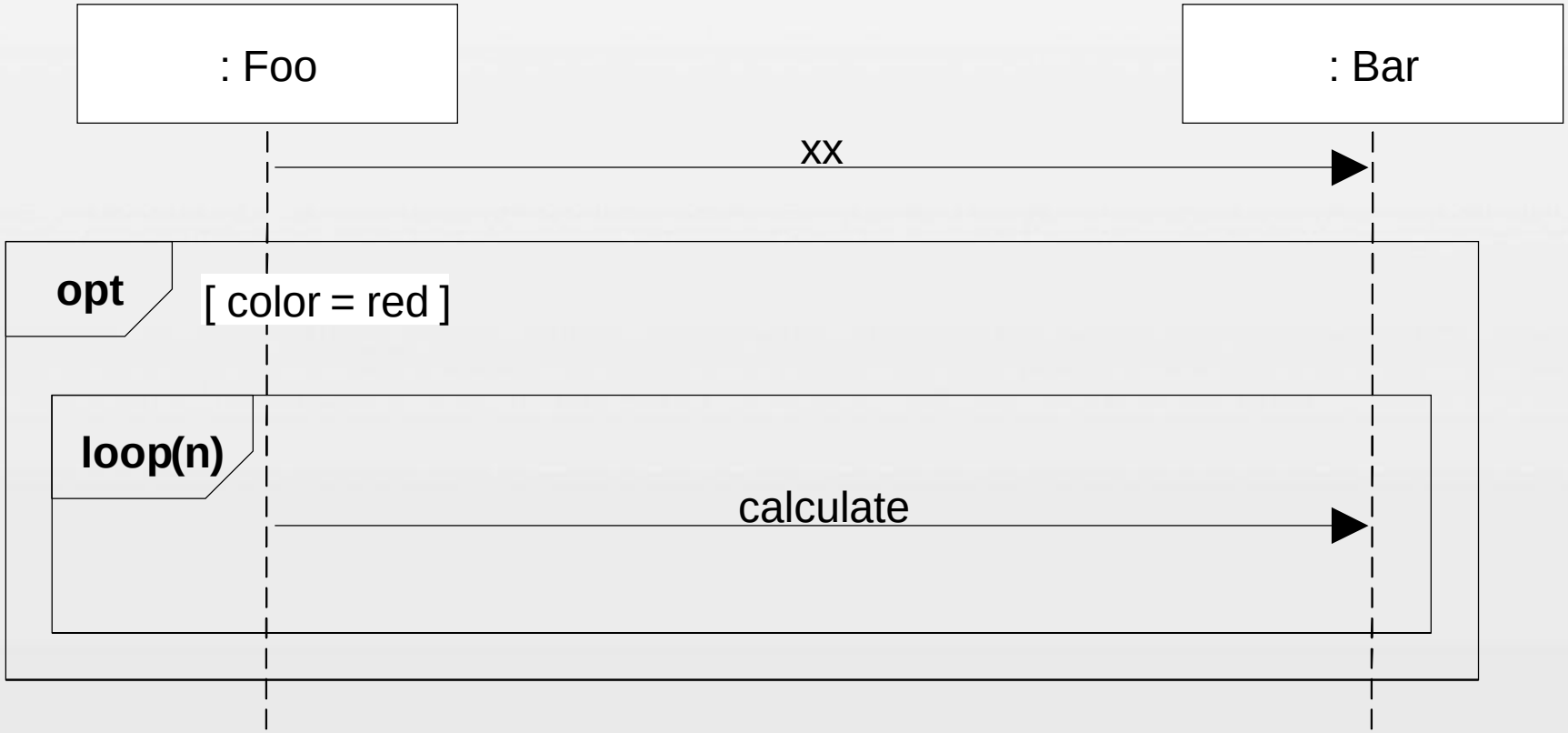
lineItems [i] is the expression to select one element from the collection of many SalesLineItems ; the "i" value refers to the same "i" in the guard in the LOOP frame

an **action box** may contain arbitrary language statements (in this case , incrementing 'i')

it is placed over the lifeline to which it applies

4.6 UML Interaction Diagrams

- **Basic Sequence Diagram Notation**
  - Diagram Frames in UML Sequence Diagrams
    - Nesting of Frames



4.6 UML Interaction Diagrams

Basic Sequence Diagram Notation

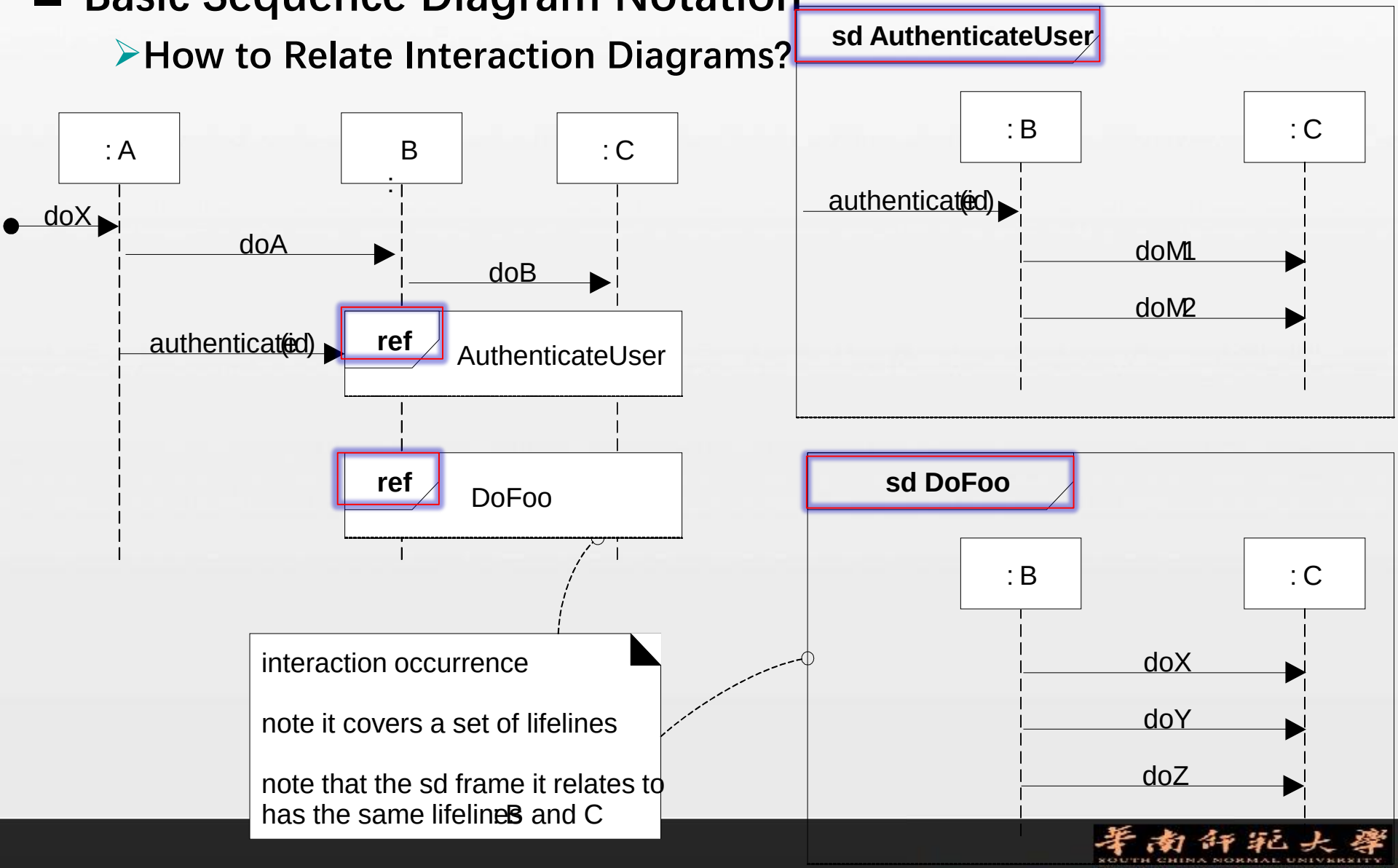
How to Relate Interaction Diagrams?

- a frame around an entire sequence diagram , labeled with the **tag sd and a name**, such as AuthenticateUser

- a frame **tagged ref**, called a reference, that refers to another named sequence diagram; it is the actual interaction occurrence

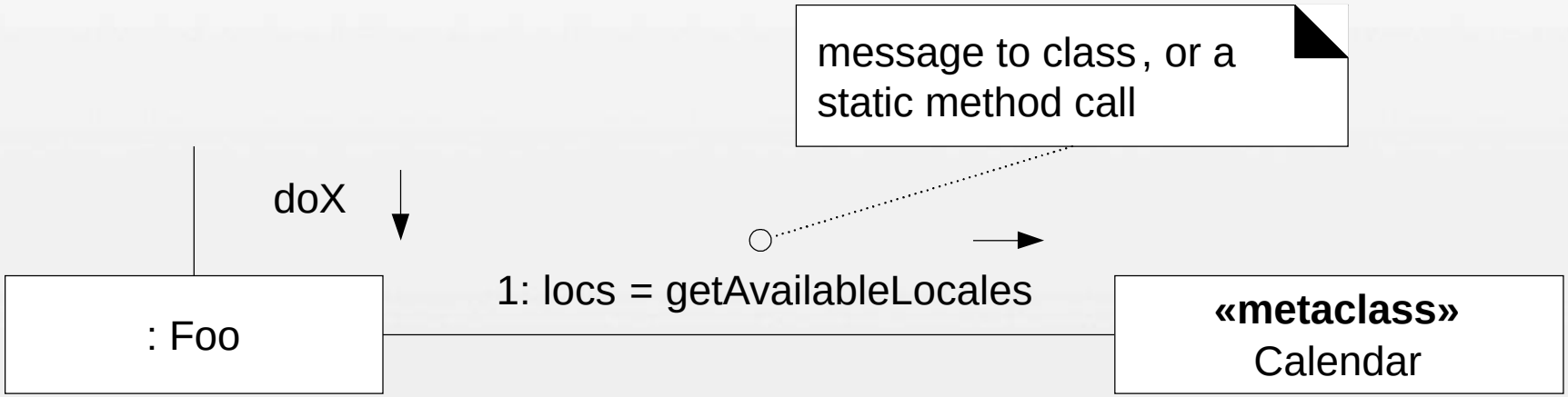
4.6 UML Interaction Diagrams

- **Basic Sequence Diagram Notation**
  - How to Relate Interaction Diagrams?



4.6 UML Interaction Diagrams

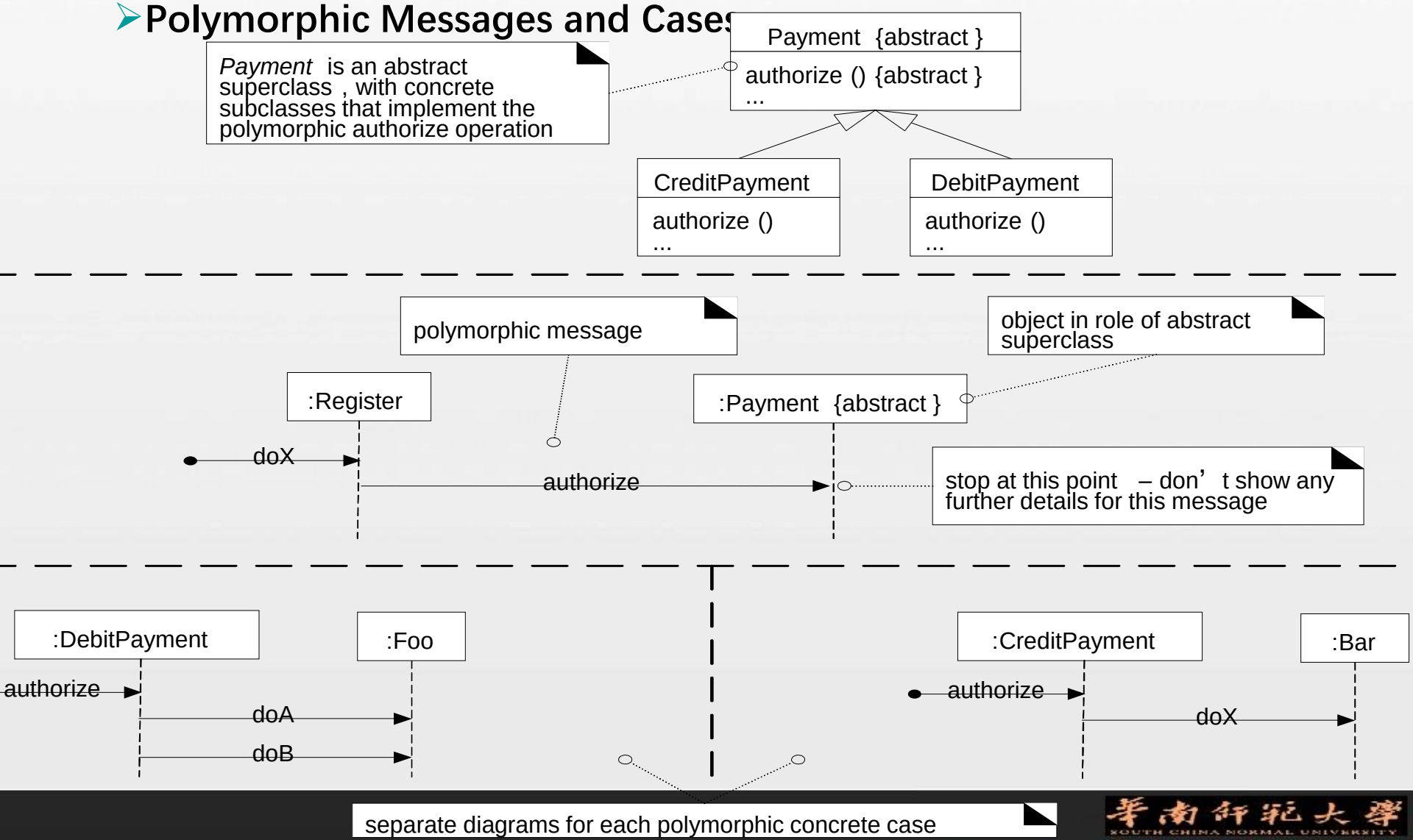
- **Basic Sequence Diagram Notation**
  - Messages to Classes to Invoke Static (or Class) Methods



4.6 UML Interaction Diagrams

Basic Sequence Diagram Notation

Polymorphic Messages and Cases



4.6 UML Interaction Diagrams

Basic Sequence Diagram Notation

Asynchronous and Synchronous Calls

a stick arrow in UML implies an asynchronous call

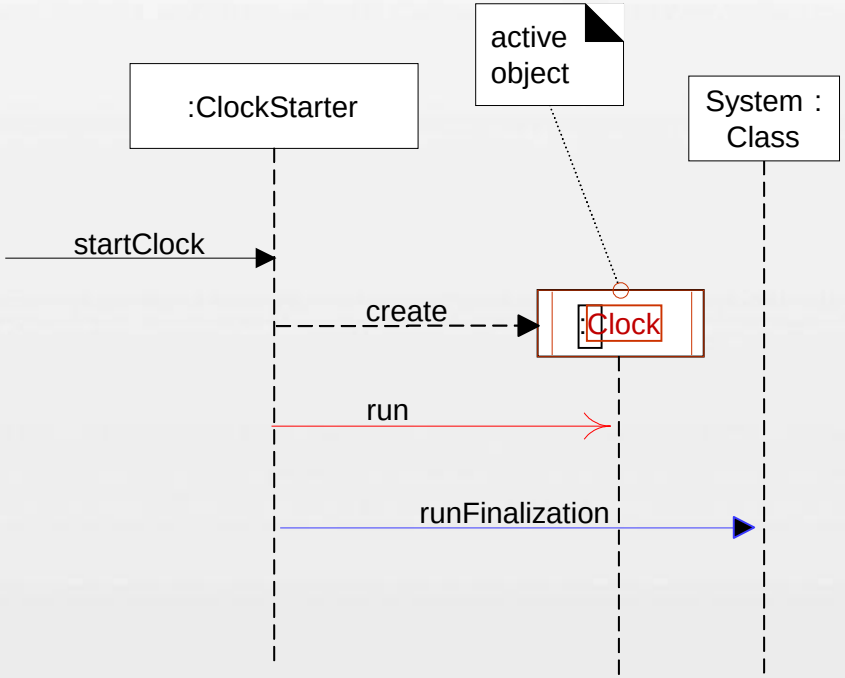
a filled arrow is the more common synchronous call

In Java, for example, an asynchronous call may occur as follows:

```
// Clock implements the Runnable interface
Thread t = new Thread( new Clock() );
t.start();
```

the asynchronous *start* call always invokes the *run* method on the *Runnable* (*Clock*) object

to simplify the UML diagram, the *Thread* object and the *start* message may be avoided (they are standard "overhead"), instead, the essential detail of the *Clock* creation and the *run* message imply the asynchronous call



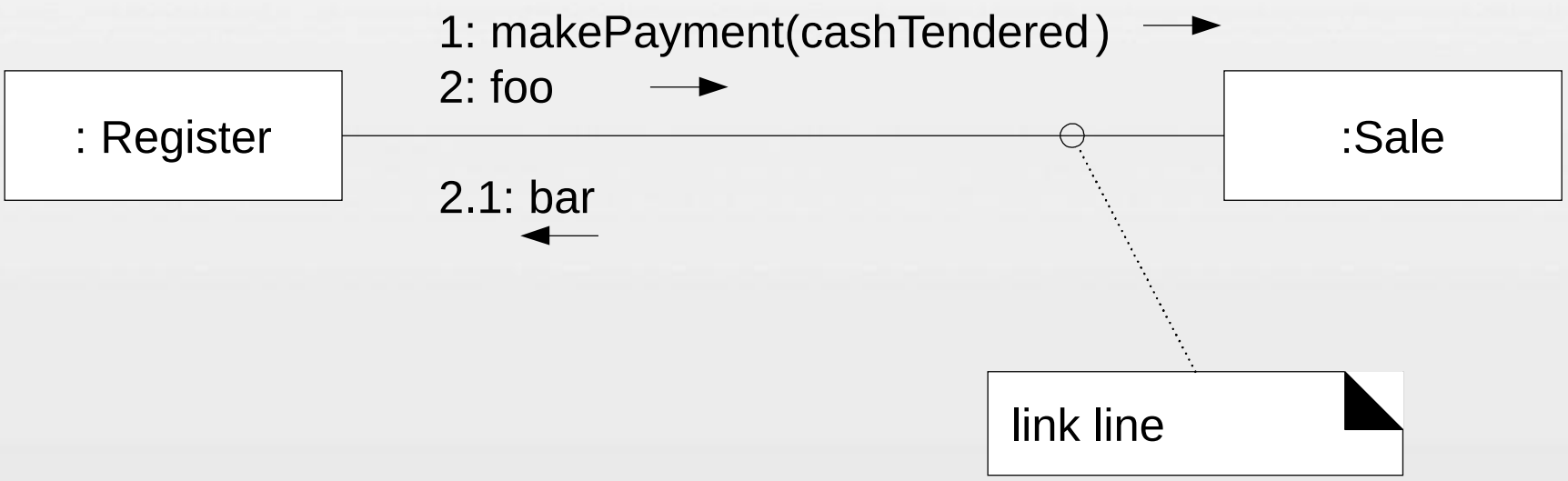
Active object?

4.6 UML Interaction Diagrams

Basic Communication Diagram Notation

Links

- A link is a connection path between two objects; it indicates some form of navigation and visibility between the objects is. More formally, a link is an instance of an association

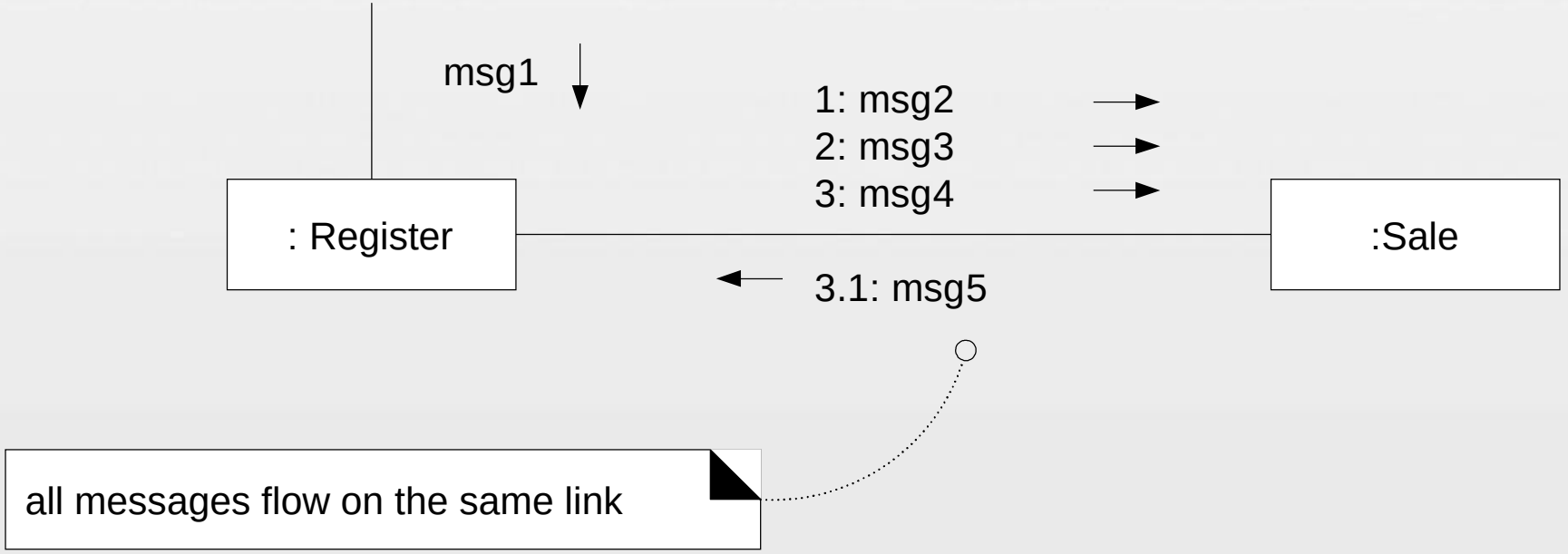


4.6 UML Interaction Diagrams

Basic Communication Diagram Notation

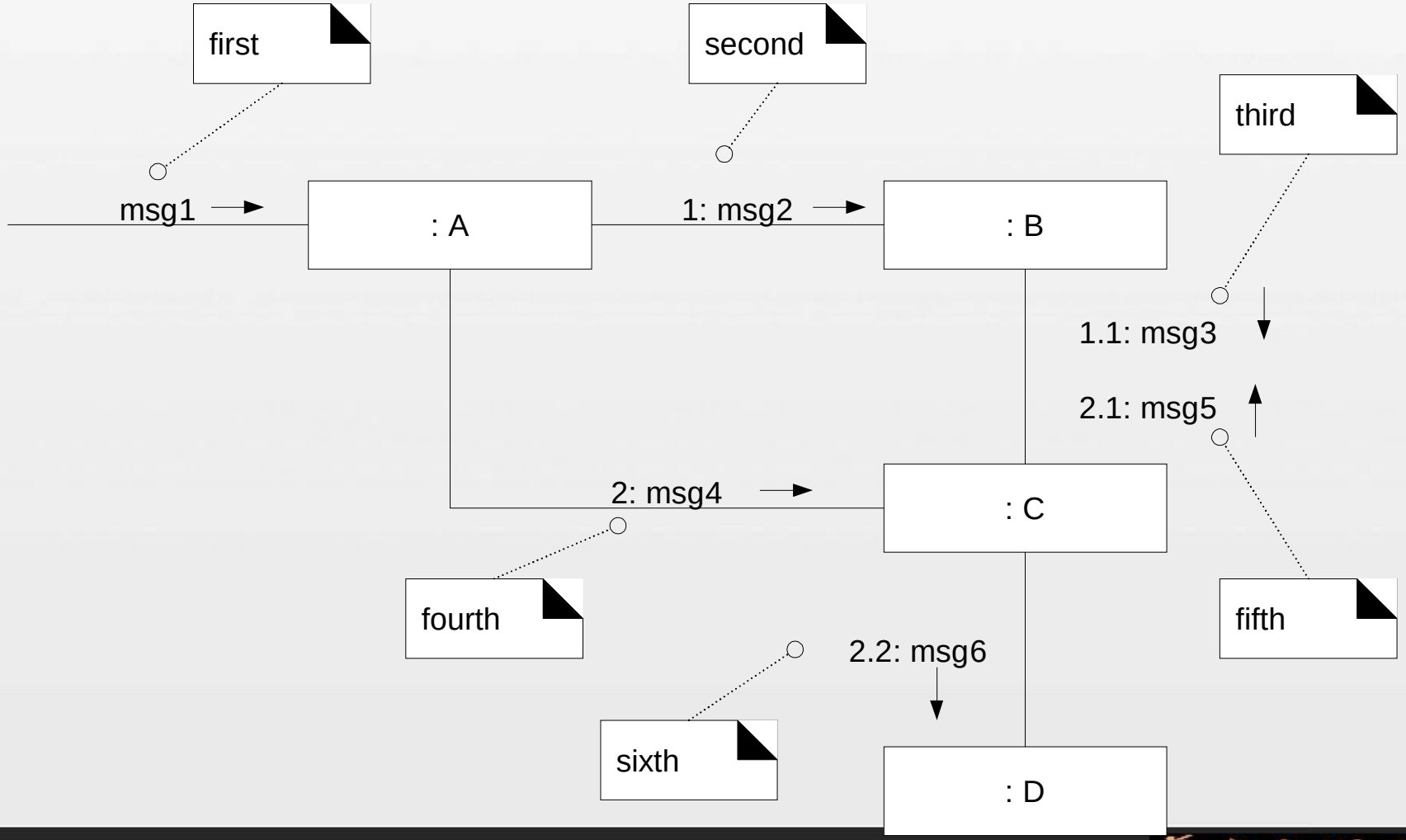
Messages

- Each message between objects is represented with a message expression and small arrow indicating the direction of the message. Many messages may flow along this link). A sequence number is added to show the sequential order of messages in the current thread of control.



4.6 UML Interaction Diagrams

- **Basic Communication Diagram Notation**
  - Message Number Sequencing



4.6 UML Interaction Diagrams

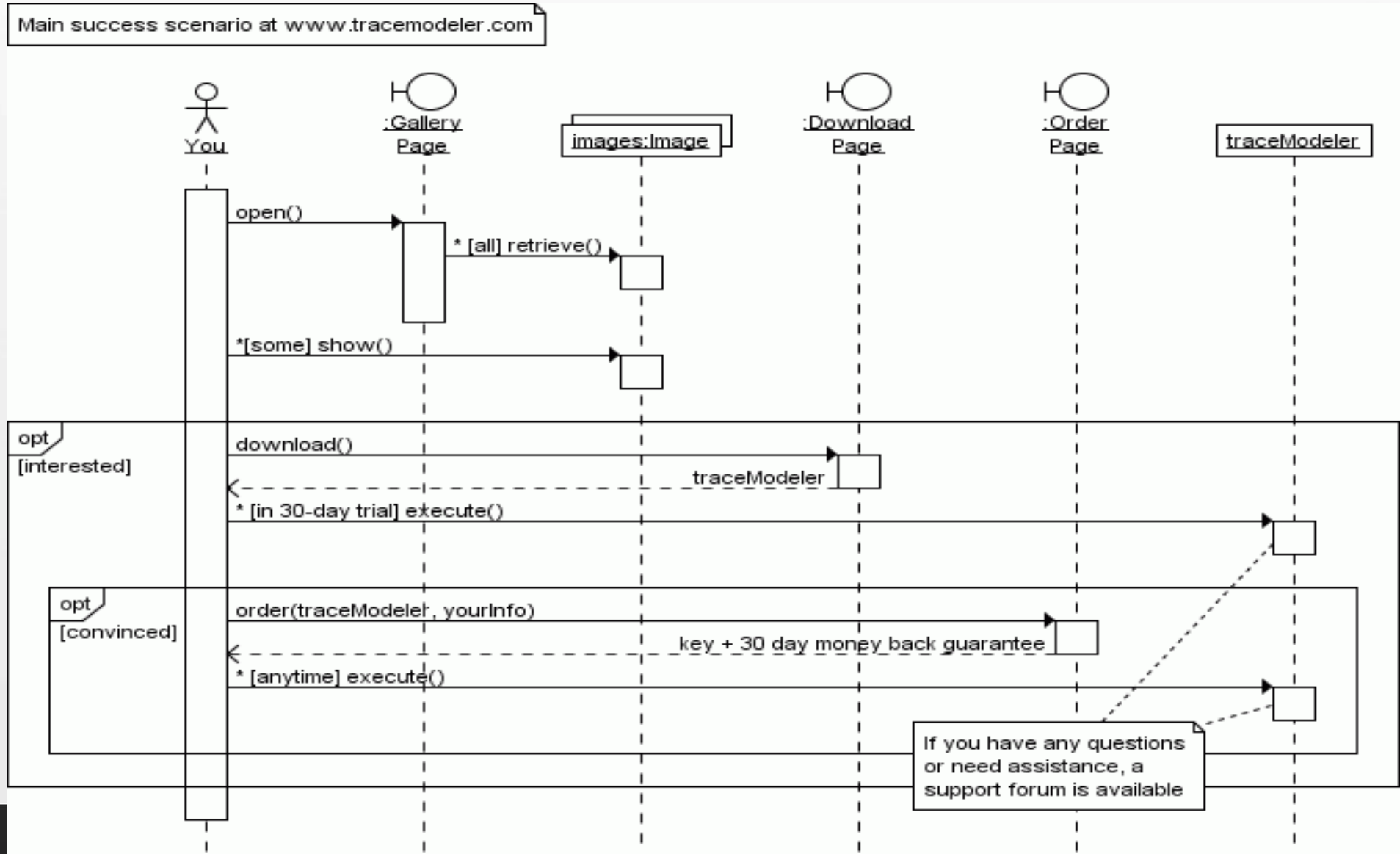
■ Communication Diagram is equal to Sequence Diagram

- Communication Diagram is equal to Sequence Diagram, a sequence diagram can equally translate to a communication diagram, Vice versa

- In general, we first create the sequence diagram, and then translate it to a equally communication diagram.

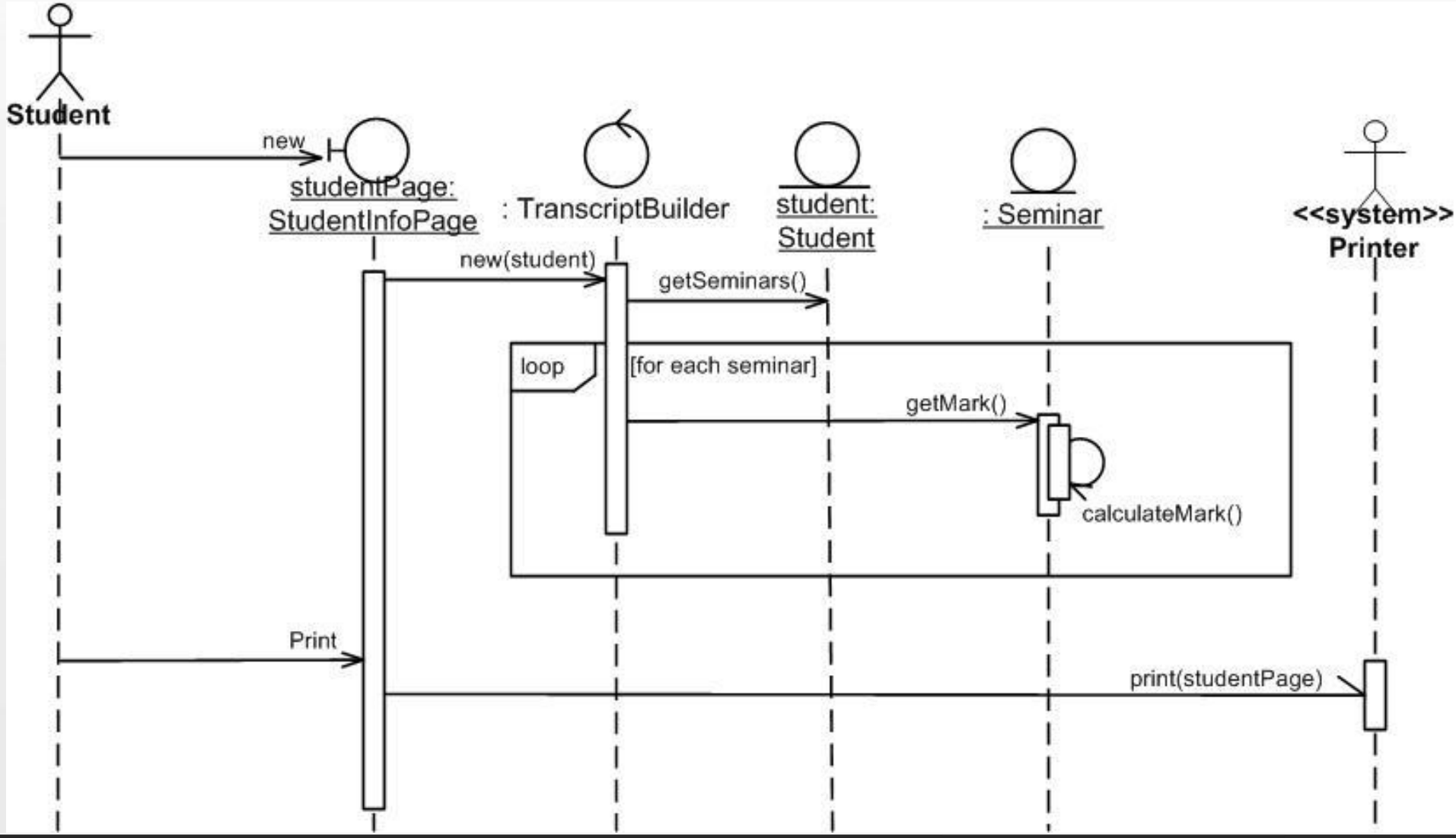
4.6 UML Interaction Diagrams

- More Samples :Trial product



4.6 UML Interaction Diagrams

- More Samples :Print Transcript



4.6 UML Interaction Diagrams

Common Mistakes

UML Sequence Diagrams that are Too Low Level

- the closer one gets to the source code level, the less useful and valuable sequence diagrams become. This is because sequence diagrams at this level tend to only give you redundant information that you could have easily obtained from just looking at the source code.

Trying to Document Too Many Scenarios Using Sequence Diagrams

- Another common mistake is attempting to create a sequence diagram for each and every possible scenario in the system, particularly when designing close to the source code level.