# Week01

***SuD***: System-under-Discussion. The actual system we are working on.
***Actor***: Something with behavior relevant to the system, such as a person, computer system or organization (e.g. a cashier).
***Scenario*** *(or* ***use case instance****)*: a specific sequence of actions and interactions between actors and the SuD.
***Use case***:
Definition: a collection of related success and failure scenarios that describe an actor using the SuD to support a goal.

- **are textual (*text stories)*** of some actor using a system to meet goals
- ***emphasize*** the user goals and perspective: "Who is using the system, what are their typical scenarios of use, and what are their goals?"
- hits a partial representation of requirements of SuD. The detail level can be brief, causal or fully dressed.
- Use case doesn't include user interface element such as "xxx button". It has to be a description of a user achieving the goal.

Main success scenario
Alternative scenario: what could also happen if something failures

***Primary actor*** has user goals fulfilled through using services of the SuD (e.g. the cashier).
– User goals drive the use case
***Supporting actor*** provides a service (e.g. information) to the SuD to clarify external interfaces and protocols (e.g. an automated payment authorization service)
– Usually be a computer system but could be an organization or person.
***Offstage actor*** has an interest in the behavior of the use case, but is not primary or supporting (e.g. a government tax agency).
– Ensure all interests identified/satisfied

***Use Case Diagrams*** provide a context of use cases and show some relationships between use cases and actors
How to draw: the big rectangle represents the SuD boundary. The circles in the rectangle are use cases. The actors at left are primary actors, the supporting actor are on the right. The third party supporting systems are written as "<actor> xxx System" in small rectangle. If the third party systems acting as primary system then put at left of the system boundary.

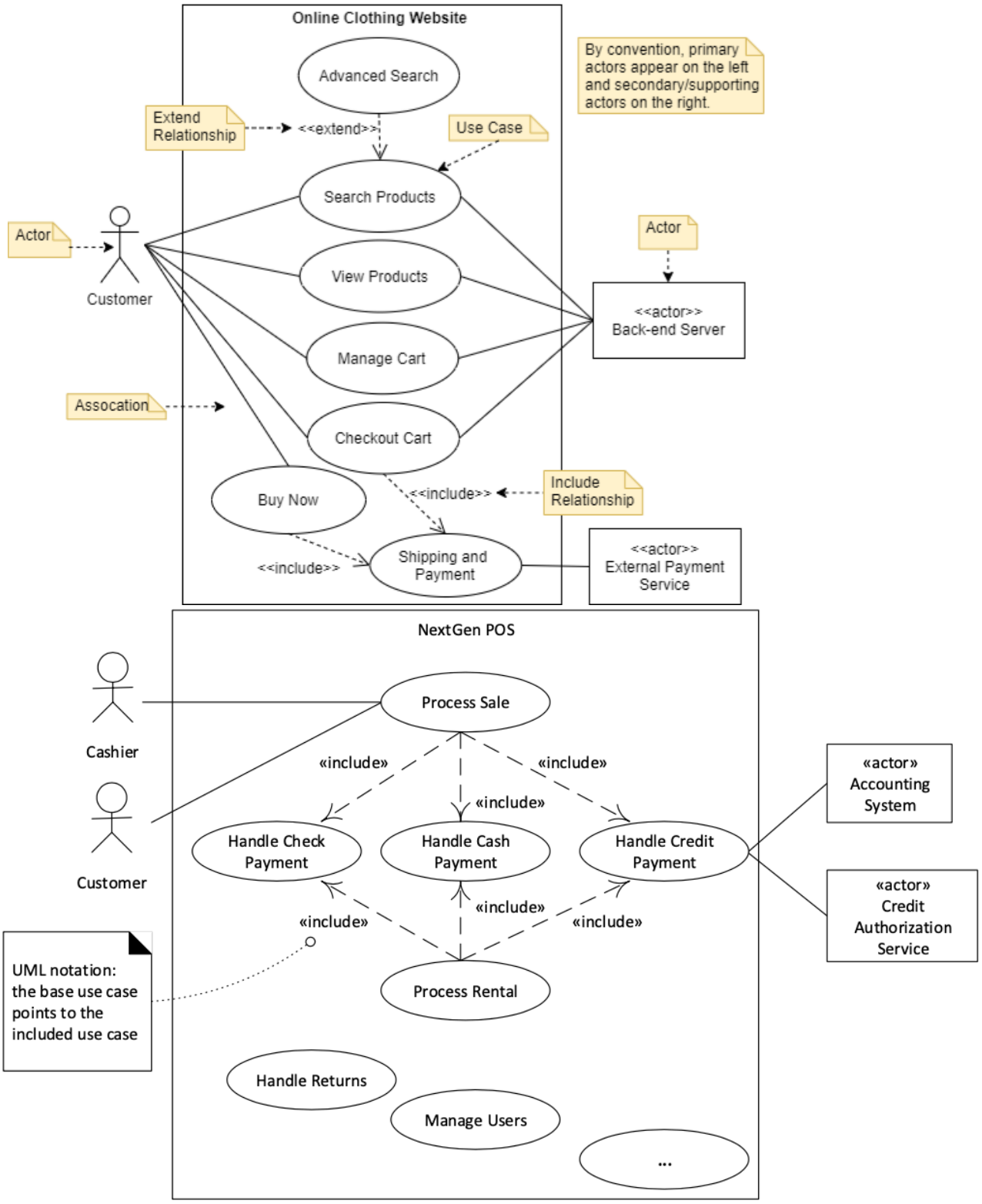
**Finding useful use cases**
Boss Test
– If your boss asks: "What have you been doing all day?" and you reply: "Logging in!", will your boss be happy?
– Is "Log In" use case useful?
Elementary Business Process (EBP) Test
– A task performed by one person in one place at one time, in response to a business event, which adds measurable business value and leaves the data in a consistent state
– Does a use case like "delete a line item" reflect EBPs? What about "Handle Returns"?
Size Test
– Is a task very seldom a single action/step; typically many steps; fully dressed often require 3–10 pages of text

Relationships should use dashed line arrow point to the included or extended use cases
***Include***: when you are repeating yourself in two or more separate use cases and you want to avoid repetition (or when a use case is too long).

- Arises when partial behavior is common across several use cases
- Refactor common part into a *sub-function* use case
- link to sub-function use case to avoid duplication
- Included case may apply in its own right, e.g. Search

Extend: when you want to add new extensions or conditional steps to a use case but can't or don't want to add to the text in the use case.

- Base case is complete without the extension
- Extension relies on the base case
- Base case doesn't know about extension



# Week02

***Object-oriented analysis*** concerns on how to describe the problem domain from the perspective of object, includes an identification of the **concepts**, **attributes**, and **associations** that are considered noteworthy in the problem domain – These can be expressed by *a domain model*

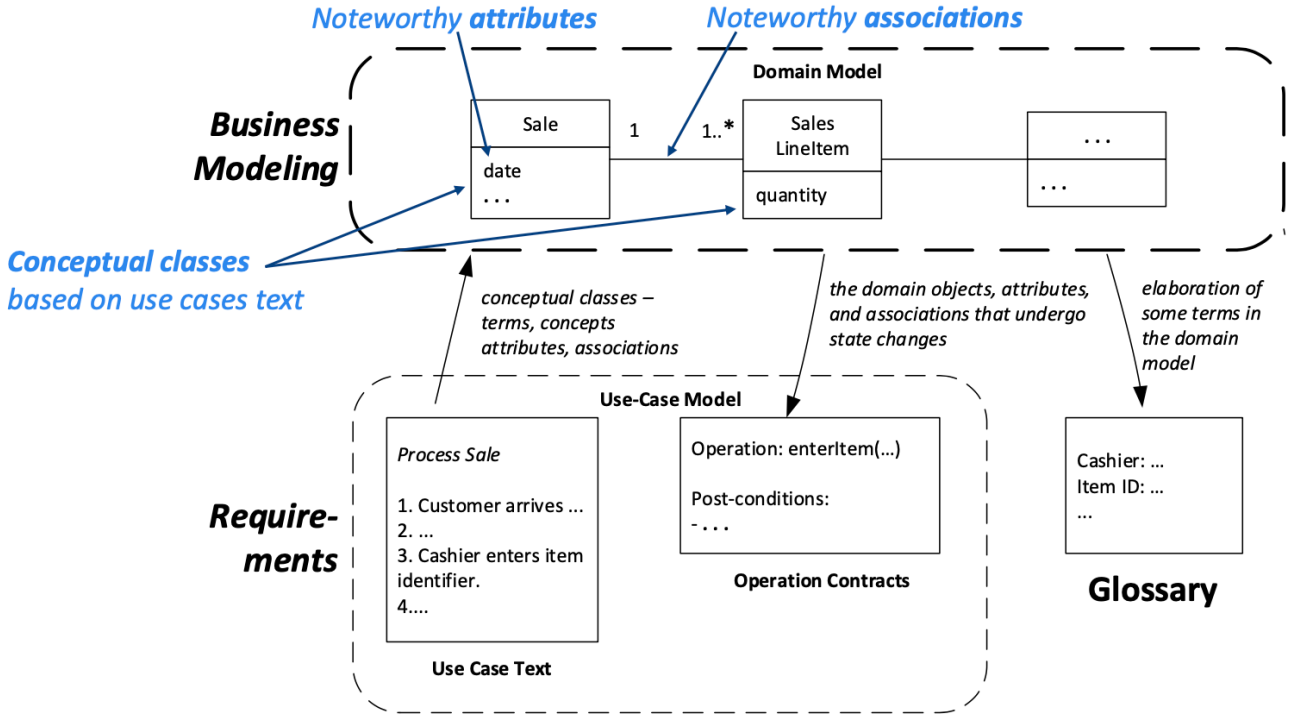
***Domain model***: A domain model is a representation of real-situation conceptual classes
- Not a software object
- Shows the noteworthy domain concepts or object
- One of OO artifacts
- It is not a set of diagrams describing software classes, or software objects with responsibilities

The UP (unify process) domain model = UP Business Object Model
- Focus on explaining things and products important to a business domain

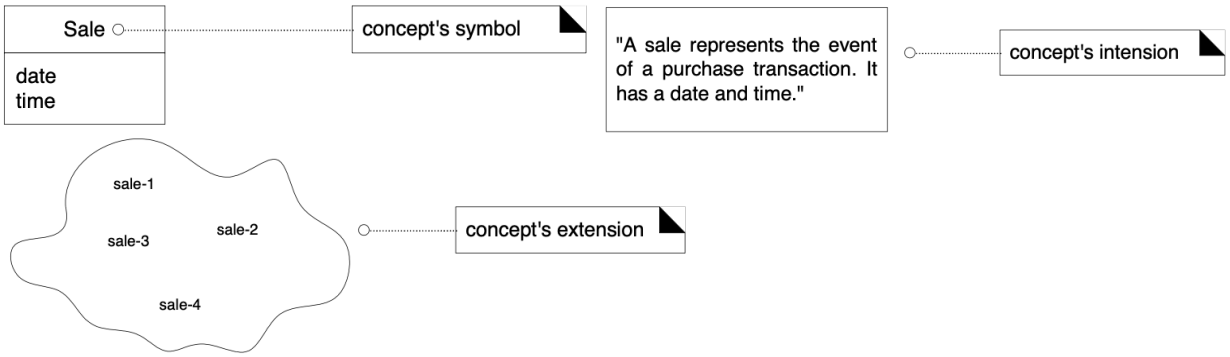
**Visual** representation: UML class diagram is used to illustrate a domain model
- A domain class diagram provides a conceptual perspective to show conceptual classes, their attributes and associations with other classes
- No operations (method signatures) are defined in the class diagram

The domain model is a *visual dictionary* of the noteworthy abstractions, domain vocabulary, and information content of the domain

***Domain Models in UP***




***Conceptual Class***: A conceptual class is an idea, thing, or object. It can be more than a data model. It can have a purely behavioral role in the domain instead of an information role.
Formal definition: A conceptual class may be considered in terms of its symbol, intension, and extension. It must be a domain space, the problem space. Not the solution or software space,
**No method included** !
- Symbol: Words or images representing a conceptual class.
- Intension: The definition of a conceptual class.
- Extension: The set of examples to which the conceptual class applies.



Find a **conceptual class from use cases**

- Identify based on the existing models for common domain problems, reuse and modify them e.g., standardized/modified domain model, organizational domain model, etc.
- Use a category list ->
- Identify noun phrases
  - Care must be applied as words in natural languages are ambiguous
  - Do not use as a mechanical noun-to-class mapping

| Conceptual Class Category | Examples |
|---|---|
| Business transactions<br><br>Guideline: Critical (involve money) | Sale, Payment<br><br>Reservation |
| Physical objects<br><br>Guideline: esp. for device-controllers or simulations | Item, Register<br>Board, Piece, Die<br>Aeroplane |
| Containers of things (physical or information) | Store, Bin<br>Board<br>Aeroplane |
| Where are transactions recorded? | Register, Ledger |
| Guideline: important | FlightManifest, MaintenanceLog |

**Example: POS**

Main Success Scenario (or Basic Flow):

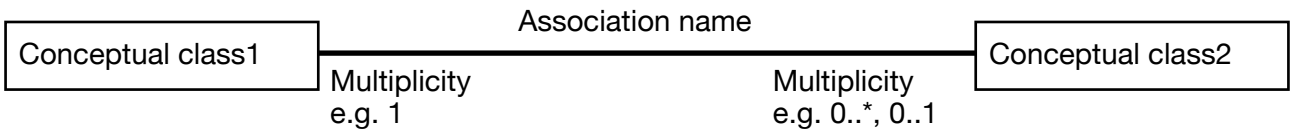
1. **Customer** arrives at **a POS checkout** with **goods and/or services** to purchase.
2. **Cashier** starts a new **sale.**
3. Cashier enters item identifier.
4. System records **sale line item** and presents **item description**, price, and running total. Price calculated from a set of price rules.

**Identify conceptual classes based on a category list**

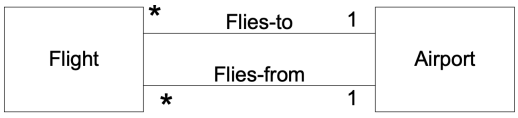
- Role of people or organization: – Customer, Cashier
- Product or services: – Goods and/or services (rename to item)
- Business transaction: – Sale
- Transaction line items: – Sale line item
- Description of things: – Item description
- Where is the transaction recorded? – A POS checkout (Register)

An ***association*** **i**s a relationship between classes that indicate some meaningful and interesting connection

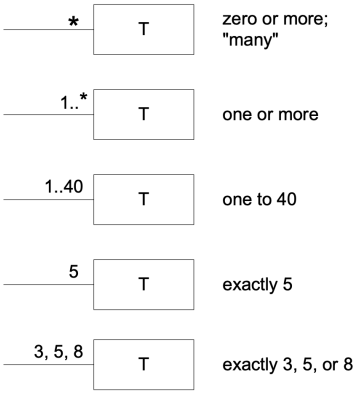
- Significant in the domain
- Knowledge of the relationship needs to be preserved
- Derived from the Common Associations List




Multiple association E.g.



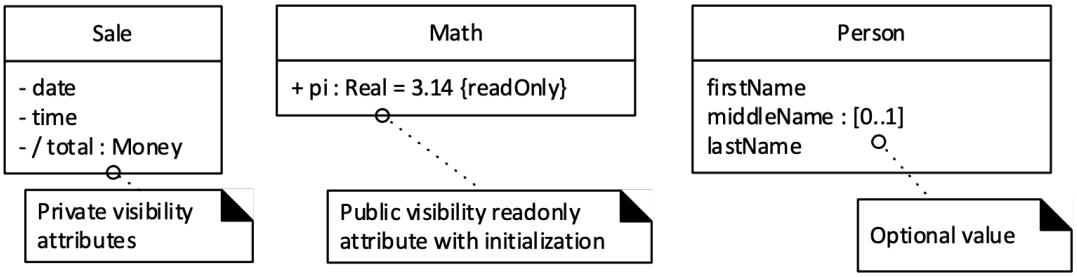

Multiplicity values




Use a Category list: Make a list of candidate associations in the domain, based on commonly occurring categories

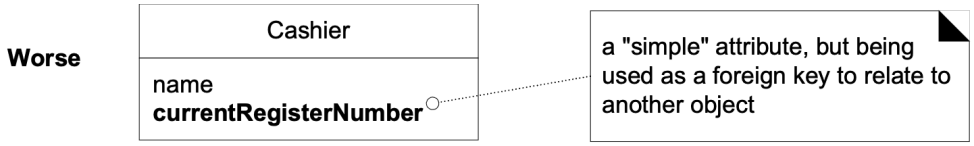
| *Association Category* | *Examples* |
|---|---|
| **A is a member of B** | *Cashier—Store*<br>*Player—MonopolyGame*<br>*Pilot—Airline* |
| **A is a role related to a transaction B** | *Customer—Payment*<br>*Passenger—Ticket* |
| **A is physically or logically contained in/on B** | *Register—Store, Item—Shelf*<br>*Square—Board*<br>*Passenger—Airplane* |
| **A is known/logged/recorded/reported/captured in B** | *Sale—Register*<br>*Piece—Square*<br>*Reservation—FlightManifest* |

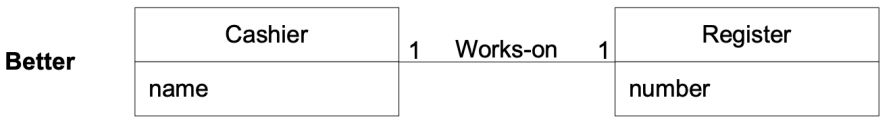
An ***attribute*** is a logical data value of an object. Only contain very important information Attributes should be included into conceptual classes when the requirements (e.g., use cases) suggest or imply ***a need to remember information***
*Attribute vs class:* If X not considered a number or text in the real world, X is probably a conceptual class, not an attribute.



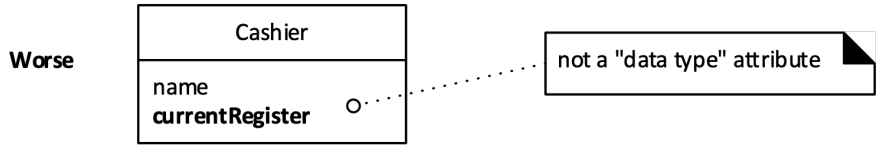

UML Attribute Syntax: visibility derived name : type multiplicity = default {property-string}

Don't Use Attributes as **Foreign Keys**



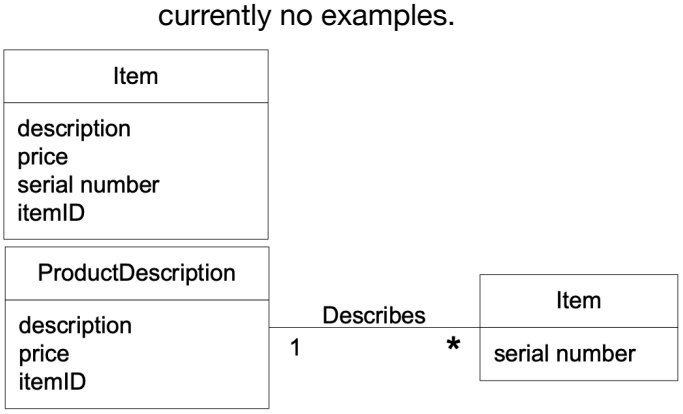



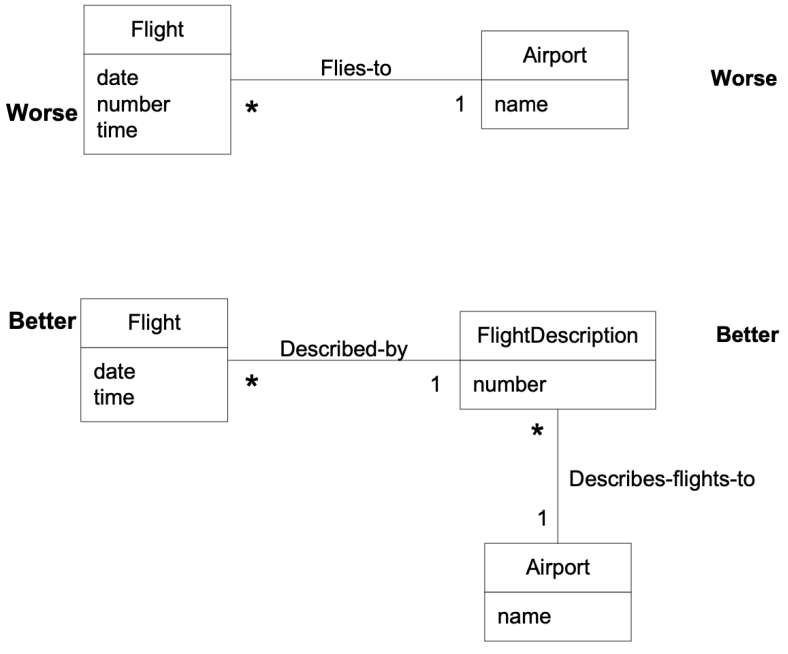

**Associations**, not Attributes



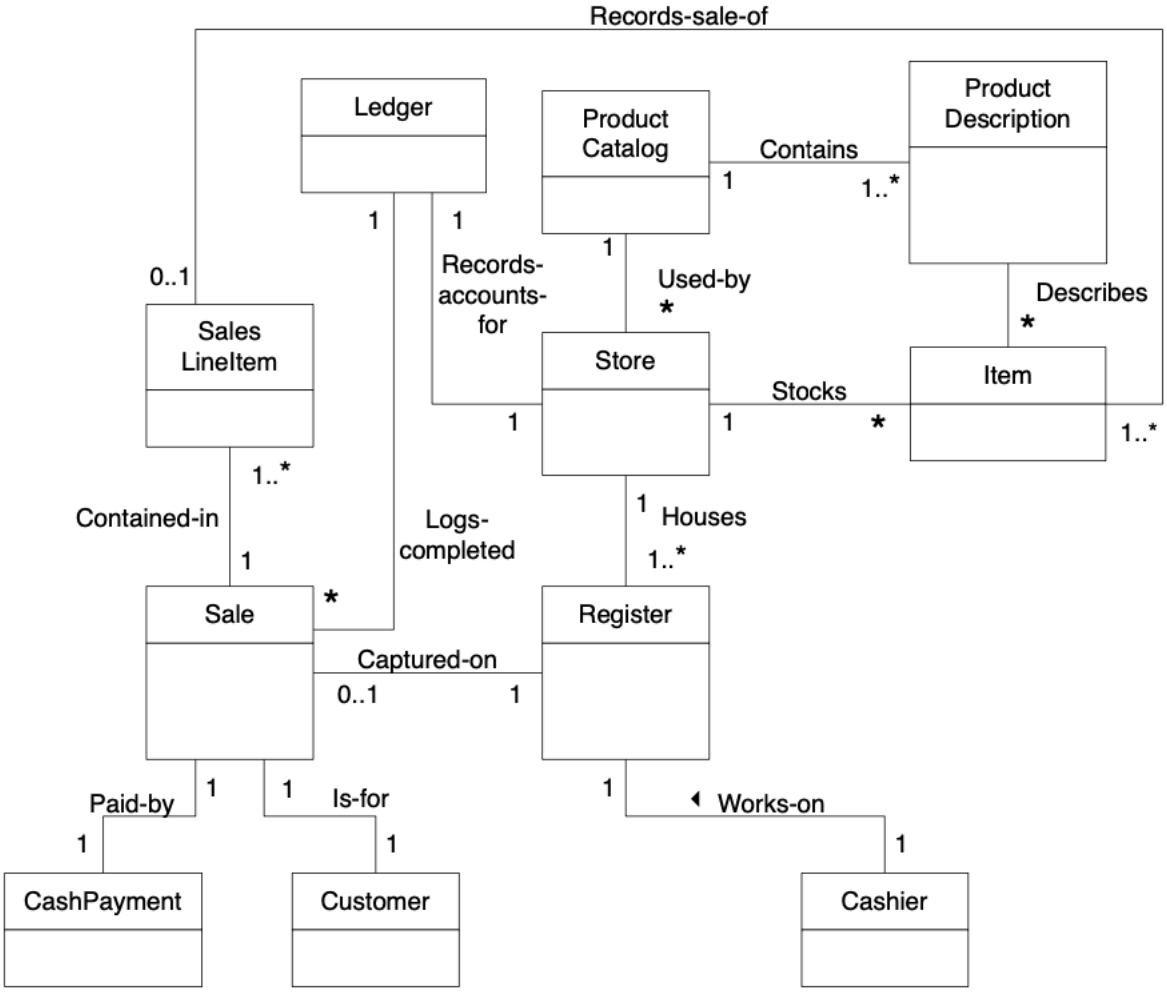


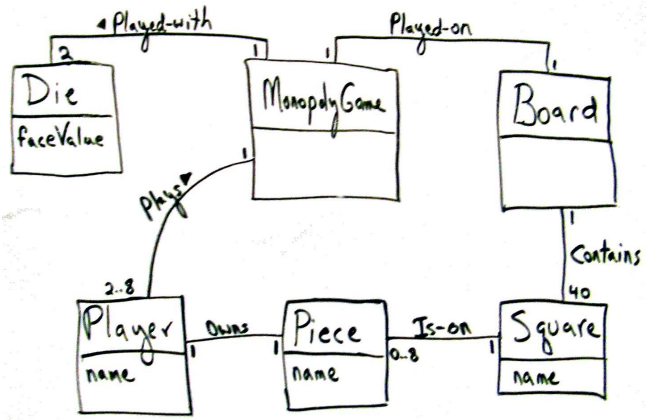
A **description** *class* contains information that describes something else; it should be used when:
- Groups of items share the same description
- Items need to be described even when there are currently no examples.



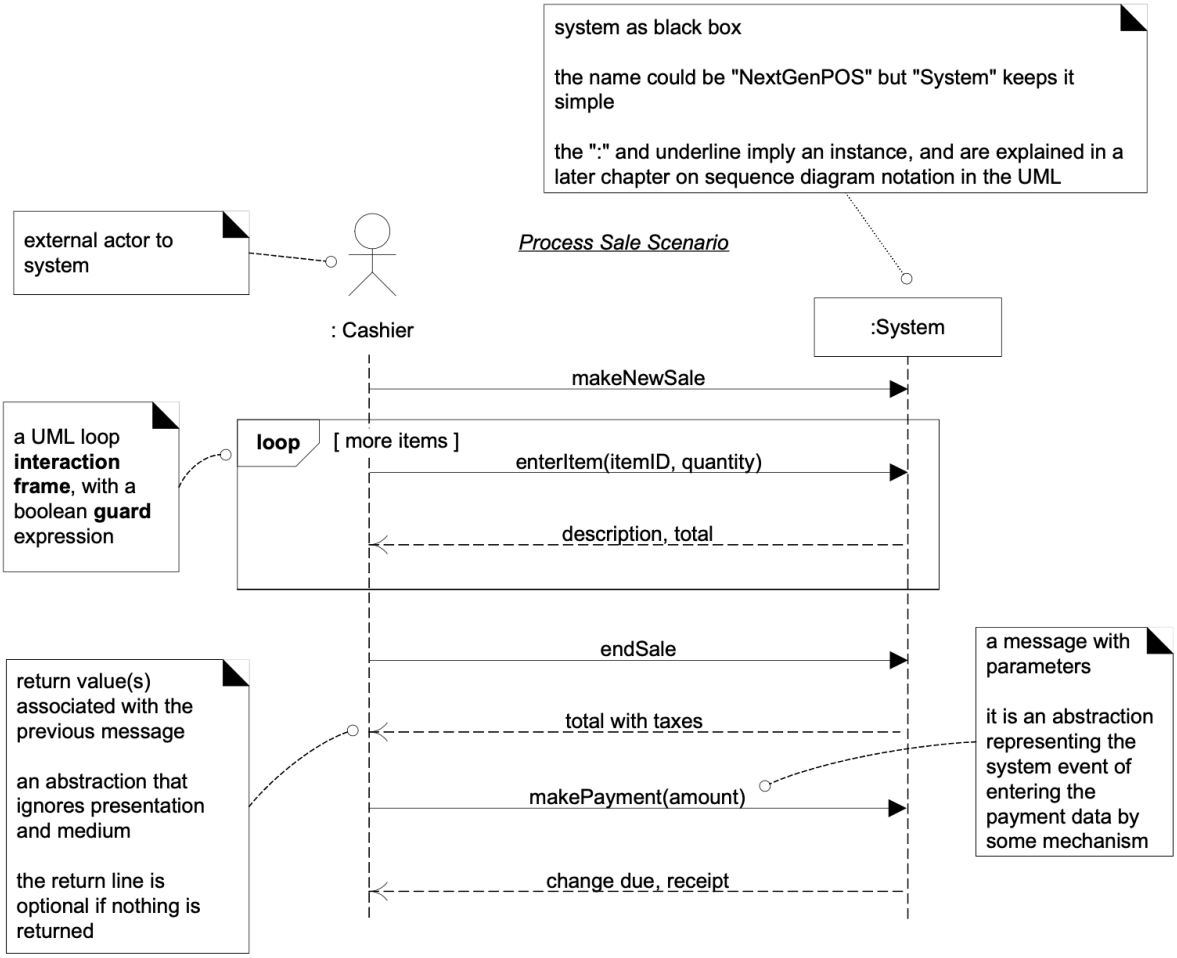
**Partial Domain model:**



**System Sequence Diagrams (SSDs)**:
A visualization of the *system events* that external actors generate, the *order* of the events, and possible inter-system events

- One SSD is for one particular scenario of a use case
- Captures dynamic context for system
- Derived from uses cases; show one scenario
- All external actors (human, non-human) for scenario are included
- Helps to identify the external input events to the system (the system events)
- Events should remain abstract: intent, not means
- Indicate events which design needs to handle
- Treats system as a 'black box' to describe what the system does without explaining how it does it.
- A good abstract name is important: use 'enterItem' rather than 'scan'



# Week03

Object-oriented Domain Models
- ***Analysis***: An investigation of the problem & requirements.
- ***Object-oriented software analysis*** emphases finding and describing objects and concepts in the problem *domain*.

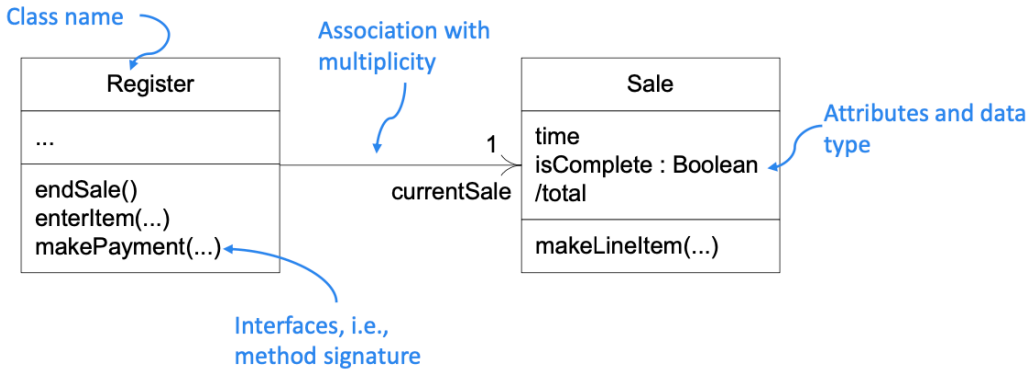
*Object-oriented Design Models*
- ***Design***: A *conceptual solution* to a problem that meets the requirements.
- Object-oriented software design emphasises defining software objects and their collaboration.

Object-oriented Implementation
- ***Implementation***: A *concrete solution* to a problem that meets the requirements.
- ***Object-oriented software implementation***: implementation in object-oriented languages and technologies.

**Object-oriented software design** is a process of creating a conceptual solution by defining software objects and their collaboration.
- Structure and connectedness
  - Levels, including Architecture
- Interfaces: methods, data types, and protocols
  - External and internal
- Assignment of responsibilities
  - Principle and patterns

**Modeling** the conceptual solution
- Static models: design class diagram
- Dynamic models: design sequence diagram

**Static Design Models**
**Definition**: A static design model is a representation of software objects which define class names, attributes, and method signatures (but not method bodies)
– Use UML Class Diagram to visualize the model
Design class diagram
- illustrates class names, interfaces, and the associations of software objects




**Domain Models vs Design Models**
- Both models use the same UML diagram, but their focuses are different
  – Domain model focuses on a conceptual perspective of the problem domain
    ‣What are the noteworthy concepts, attributes, and associations in the problem domain
  – Design model focuses on an implementation perspective of software
    ‣ What are the roles and collaborations of software objects
- Domain models **inspire** the design of software objects
  - To **reduce the representational gap** between how stakeholders conceive the domain and its representation in software

**Responsibility-Driven Design (RDD)**

- RDD focuses on assigning *responsibility* to software objects
  – Responsibility: The obligations or behaviors of an object in terms of its role

**Knowing** responsibilities include:

- knowing about private encapsulated data
- knowing about related objects
- knowing about things it can derive or calculate

**Doing** responsibilities include:

- doing something itself, such as creating an object or doing a calculation
- initiating action in other objects controlling and coordinating activities in other objects

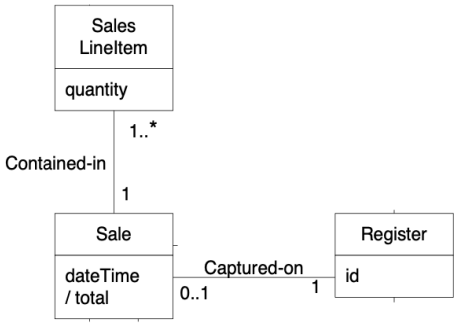
RDD also includes the idea of *collaboration*

– Responsibilities can be implemented by means of methods that either act alone or collaborate with other methods and objects

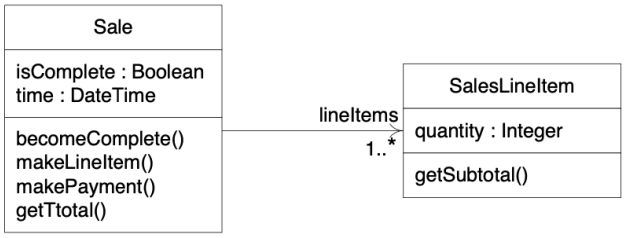
- A responsibility in RDD is not the same thing as a method—it's an abstraction—but methods fulfill responsibilities
- The domain model often inspires responsibilities related to "knowing", achieving ***low representational gap***

# Example: POS

- In the domain model, Sale has the total attribute
  – Let's declare: A sale class is responsible for *knowing* its total (knowing)
  → A Sale class will have 'getTotal' method
- Based on the domain model and use case text, Sales Line Items will be contained in Sale and will be created when a product is entered
  – Let's declare: A sale class is responsible for *creating* a Sale Line Items (doing)
  → A Sale class will have 'makeLineItem' method
- The sale total is derived from the sub total of sale line item
  – 'getTotal' of Sale is *collaborating* with 'getSubtotal' of SaleLineItem




Partial domain model



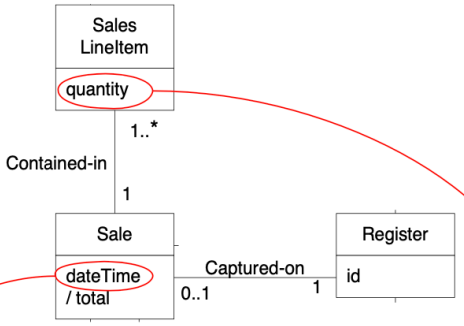

Partial design model



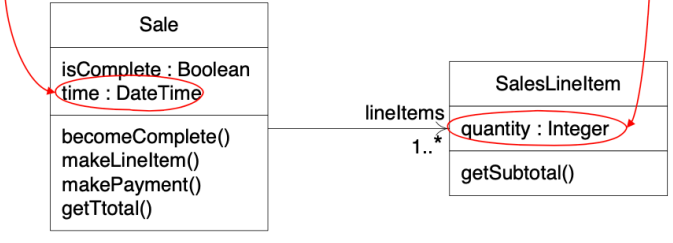
# Example: POS

Some Remarks:

- A responsibility in RDD is not the same thing as a method—it's an abstraction—but methods fulfill responsibilities
- The domain model often inspires responsibilities related to "knowing", achieving ***low representational gap***
  – Ex: the domain model Sale class has a *time* attribute
  → a software Sale class knows *time*
  – Ex: the domain SaleLineItem class has a *quantity* attribute
  → a software SaleLineItem class knows *quantity*




Partial domain model

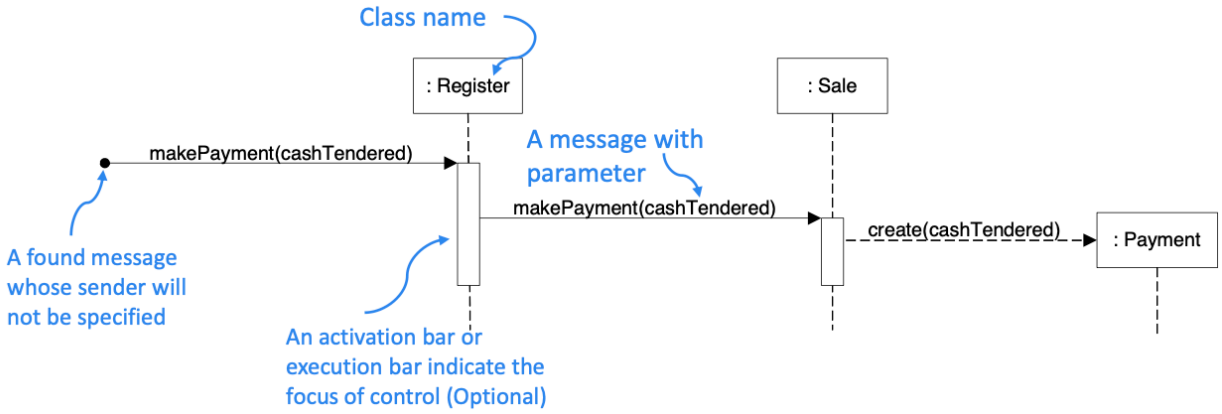



Partial design model

***Design class diagram to code:***

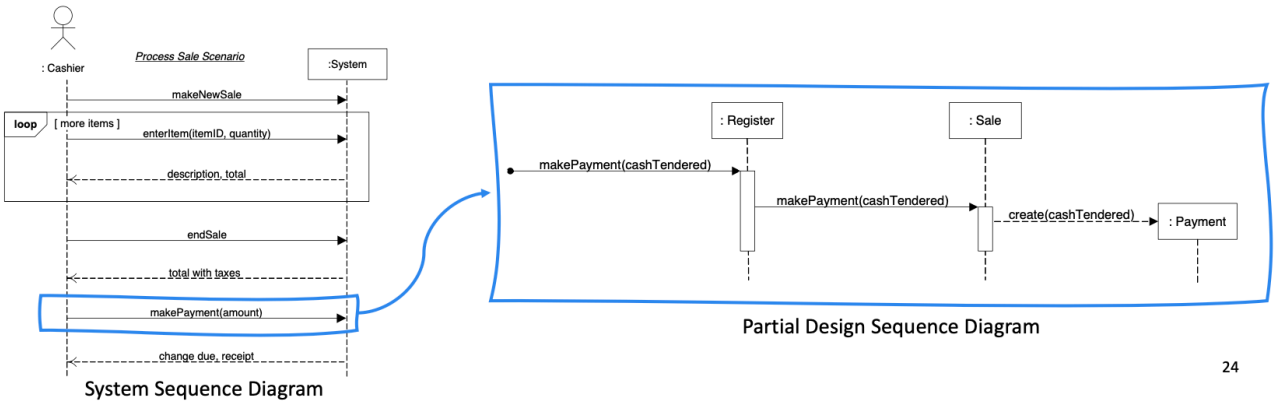
- OO Designs provide information necessary to generate the code
- OO Design can be mapped to OO Programming
- Attributes in Design models = Java fields
- Method signatures in Design models = Java methods
- The Java constructor is derived from the create responsibility

- **Dynamic Design Models**: is a representation of how software objects interact via messages
  - UML Sequence and communication* diagrams are commonly used to visualize the models
- **Design Sequence diagram**
  - Illustrates sequence or time ordering of messages sent between software objects




**System Sequence Diagram (SSD) *vs* Design Sequence Diagram** ⬆️

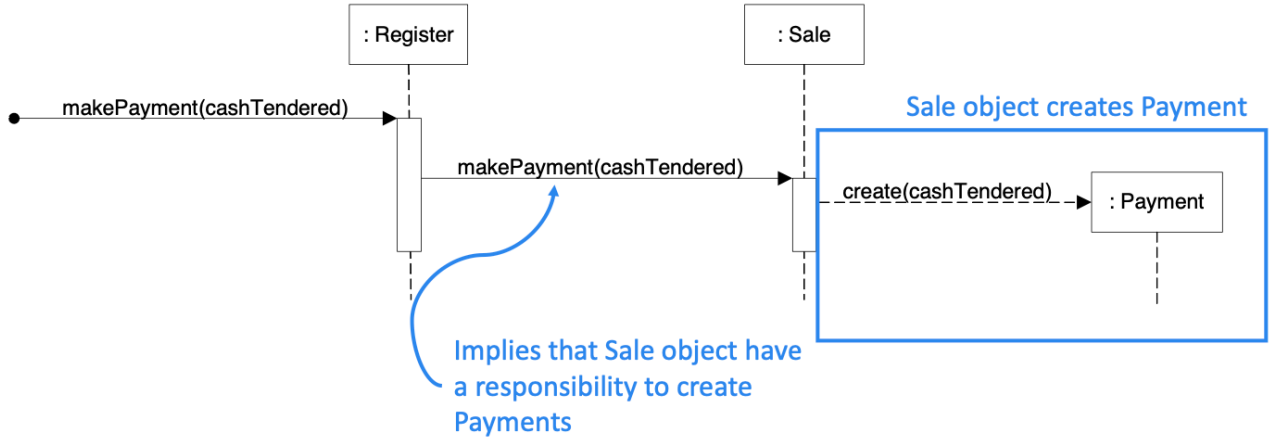
- System Sequence Diagram treats the system as a black box, focusing on the interaction between actors and the system
- Design Sequence Diagram illustrates the behaviors within the system, focusing on the interaction between software objects



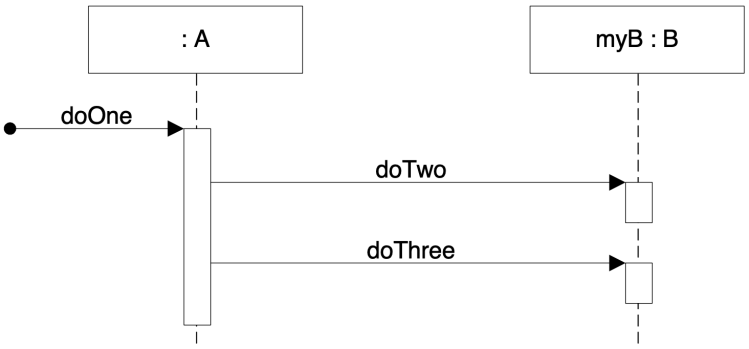



**RDD & Design Sequence Diagram**

- Design Sequence Diagram helps in better realizing responsibilities of software objects
  – Concerning the responsibility assignment when drawing the diagram

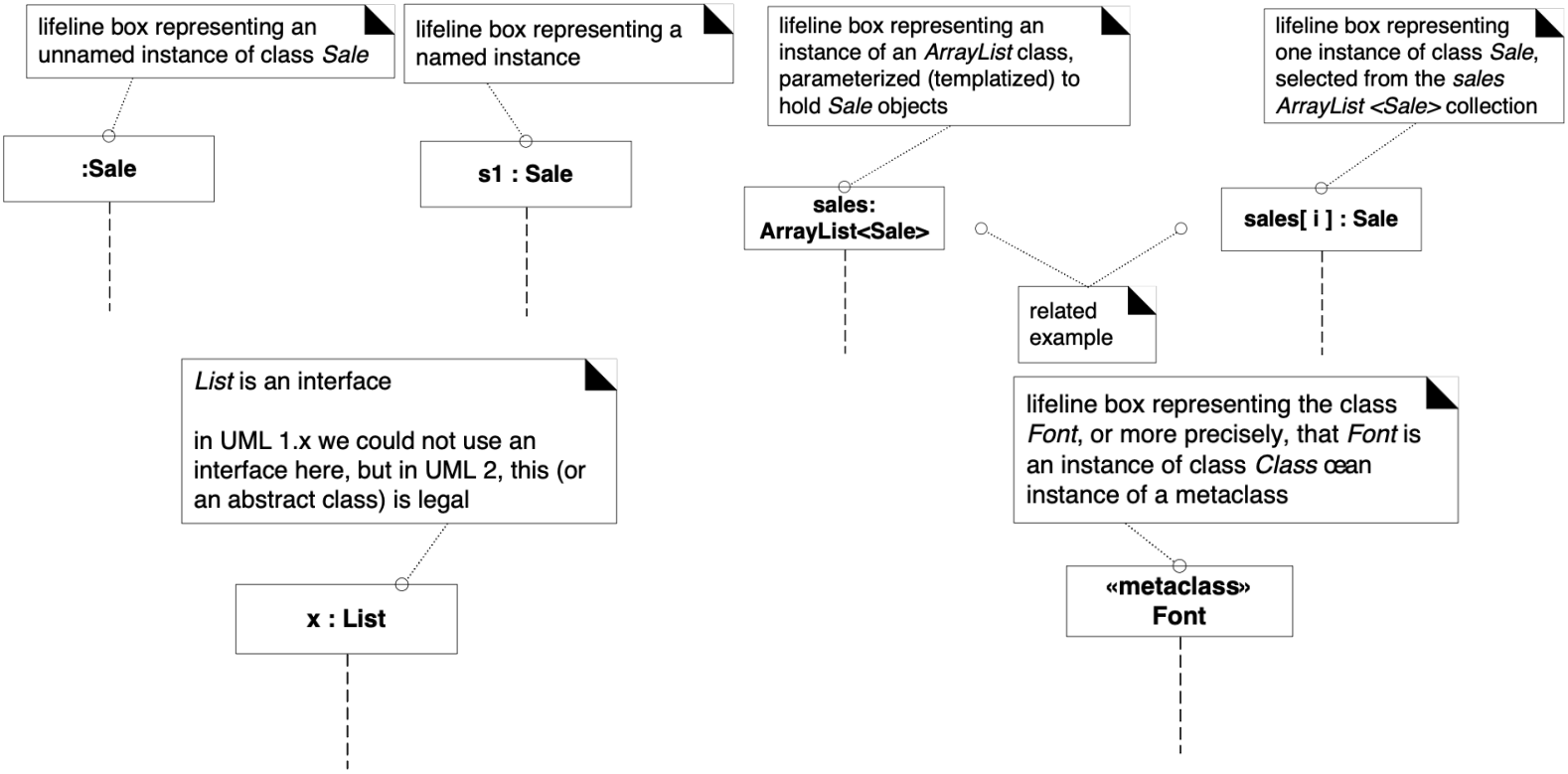


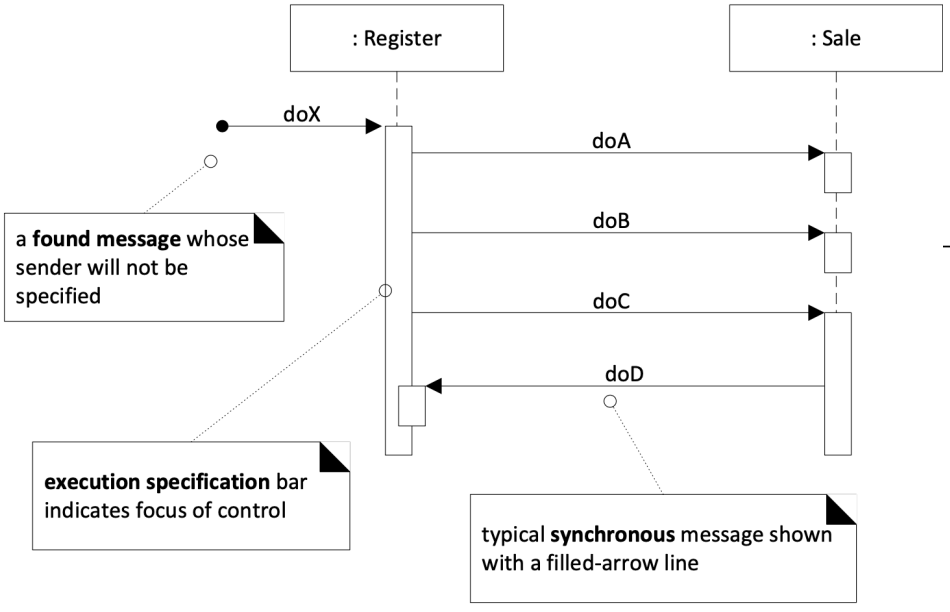
## Sequence diagram

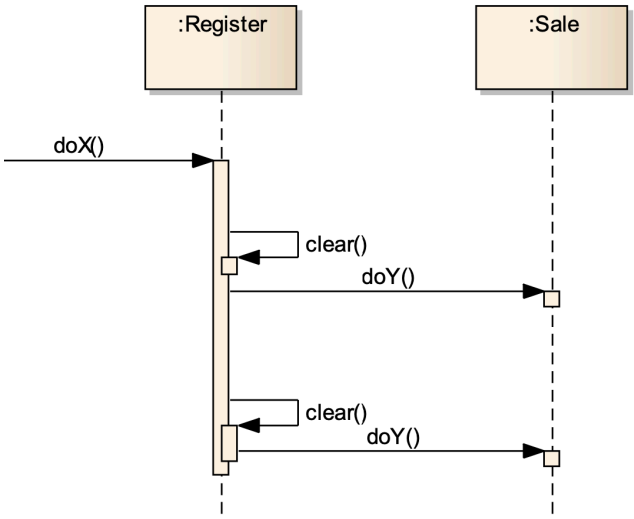



: A means a class in system called A
myB : B means a class named B which identified with myB
there's dot line going down from classes: lifeline, this is the life of the class
The messages to the class will go into the lifeline, and the messages from the class will go out from lifeline
Found message: the only message without origin, in this case is doOne, it goes early so it's at top
The messages are drew from early to late

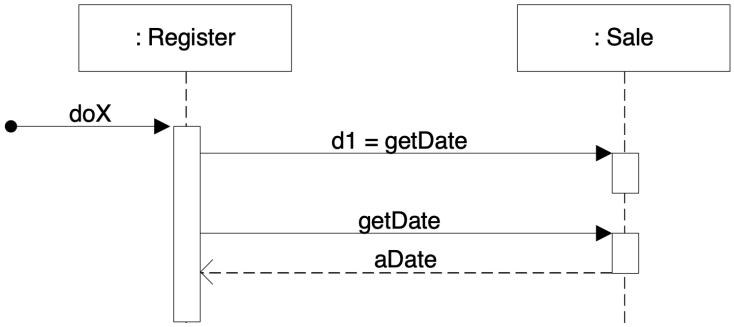
The bar on the lifeline is called execution specification bar, this means which class have the control of the program at any time



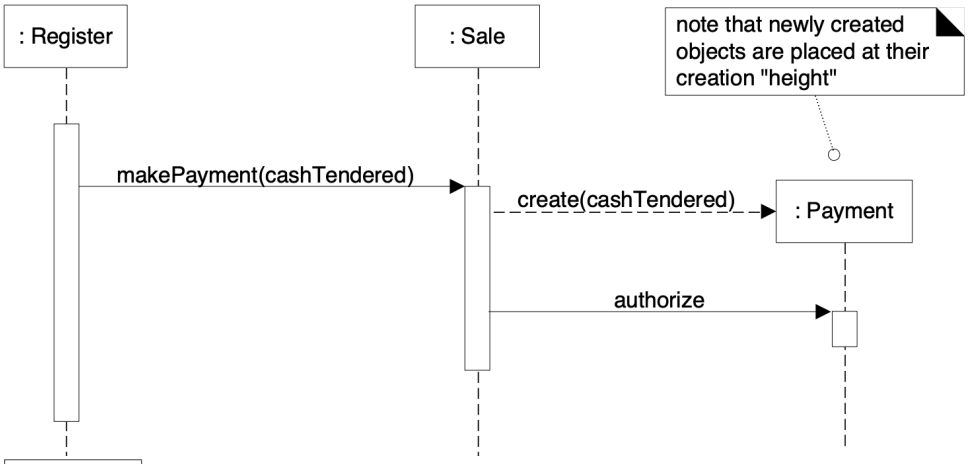



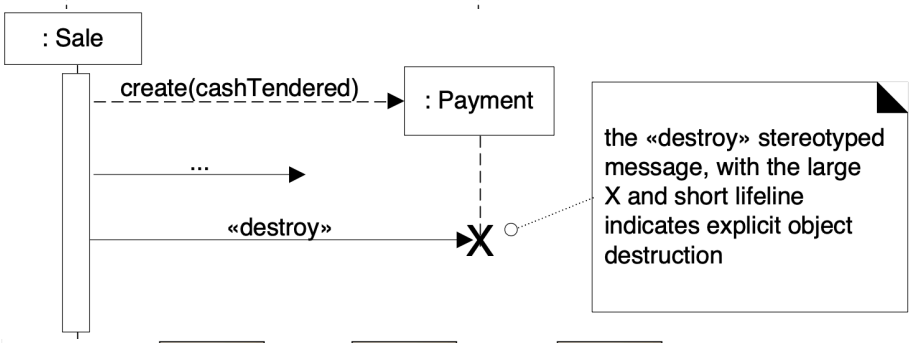



**The messages can send to self**

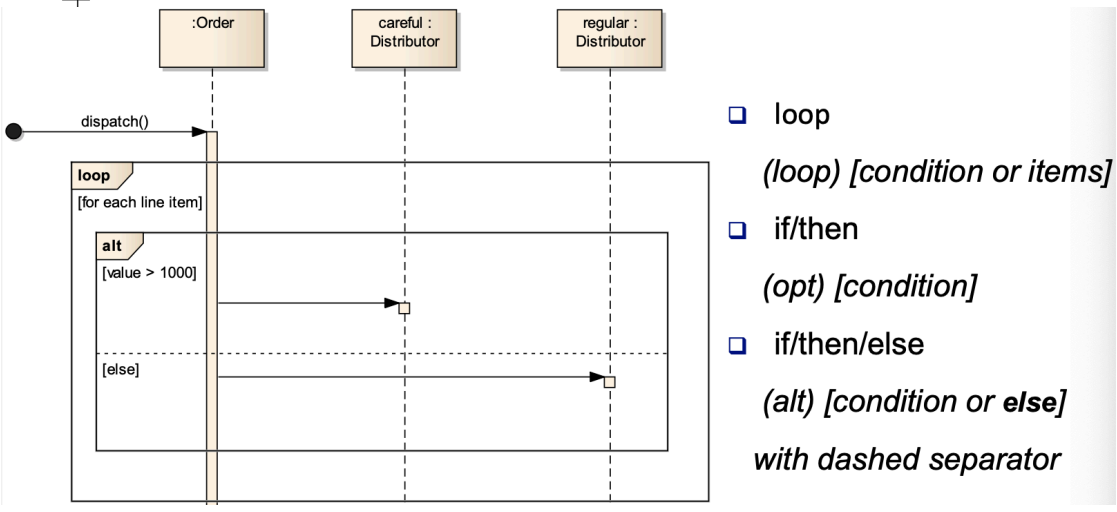



•Use dotted line to mark the returned data (aDate in this case)
•The dotted line can also use to create new class (often with 'create' word)
•Make sure it's lower down as it does not exist at start of program




If destroy object, use a <<destroy>> to mark, and at the end of the terminated object, use a large X to mark it is destroyed




- loop

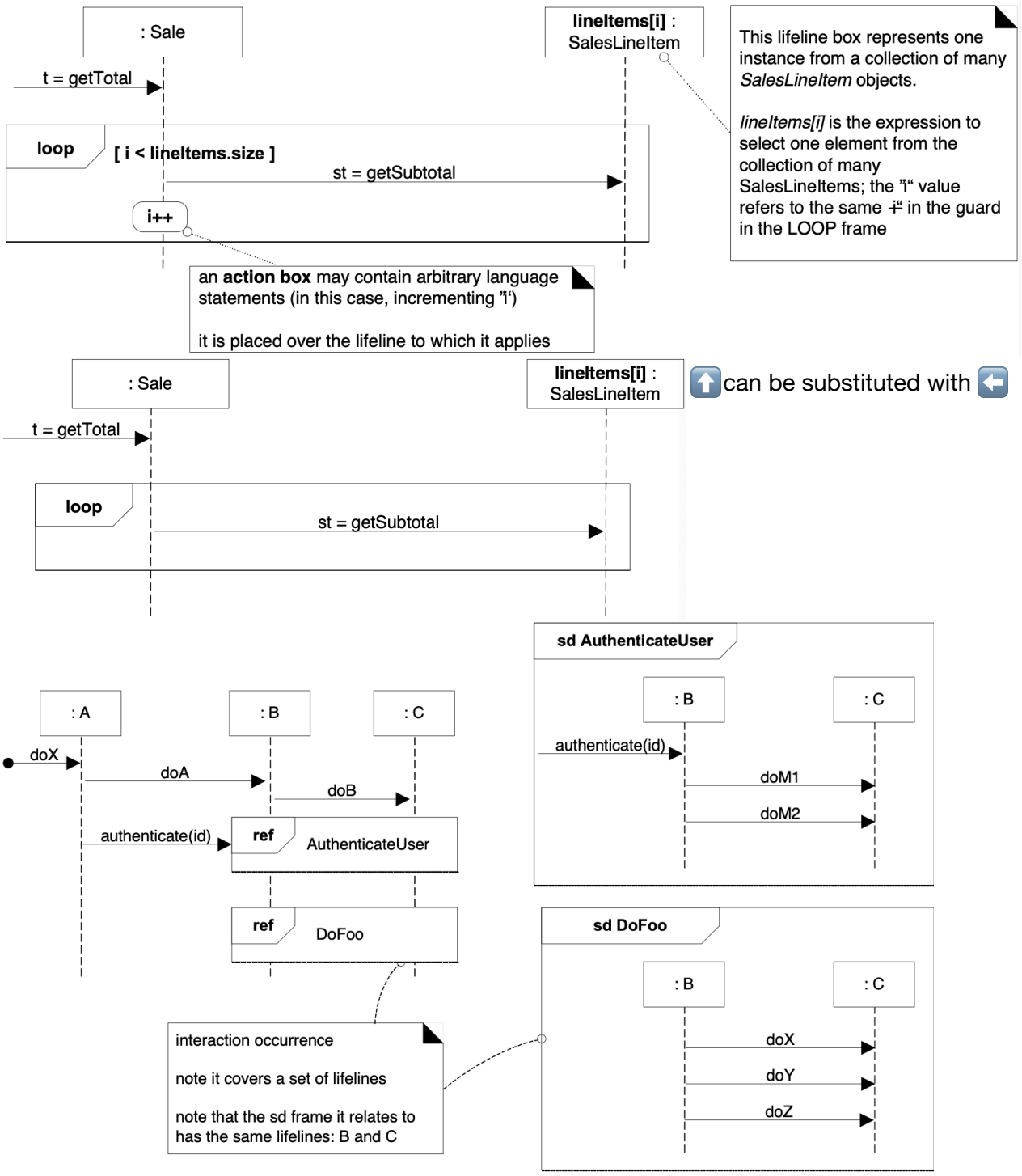
  *(loop) [condition or items]*
- if/then

  *(opt) [condition]*
- if/then/else

  *(alt) [condition or **else**]*

  *with dashed separator*

If there is a loop, mark as loop and add condition of the loop
Alt is like a if statement, use dashed separator in middle and mark the true and false results



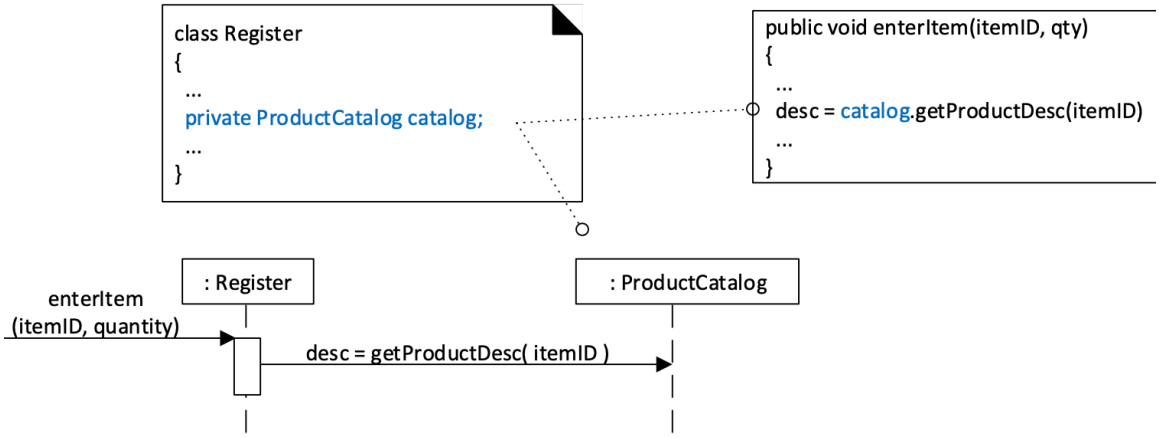

Use of reference

**What is Visibility?**

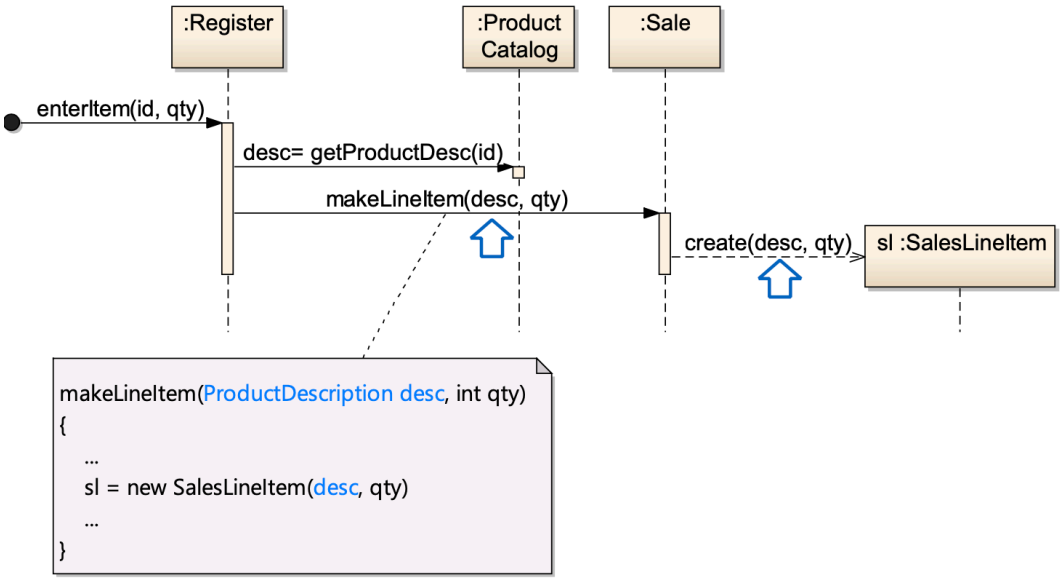
Ability of an object to "see" or refer to another object
For A to send a message to B, B must be visible to A
Is it in scope? Four common ways:

- B is an attribute of A.
- B is a parameter of a method of A.
- B is a (non-parameter) local object in a method of A.
- B has in some way global visibility.

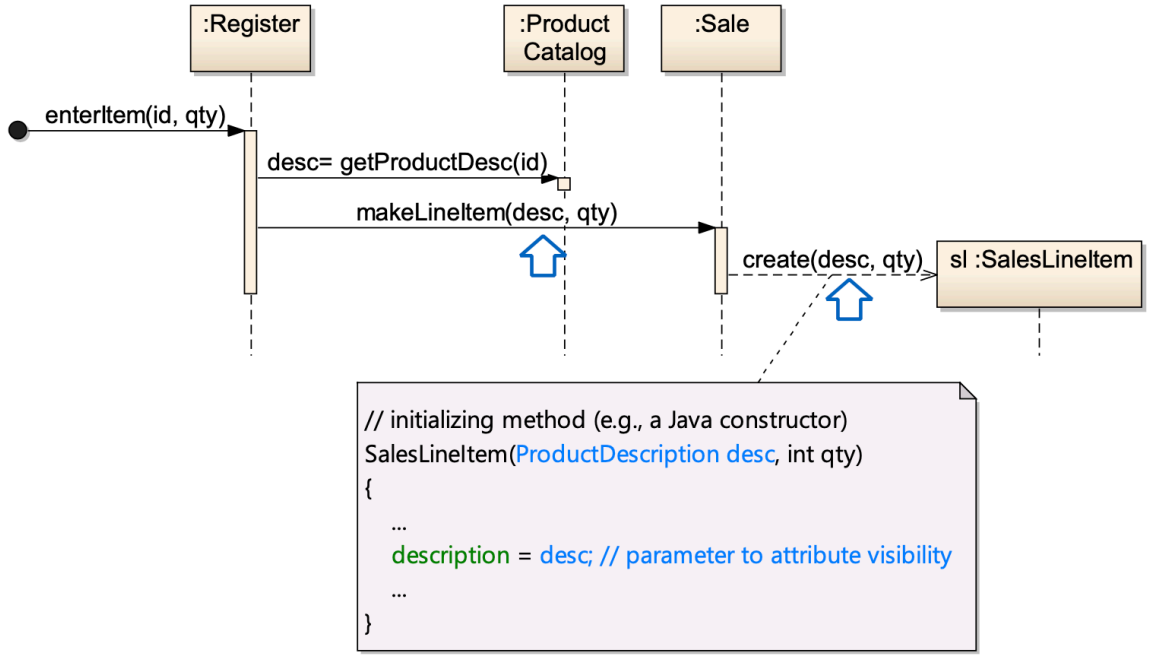
## Attribute Visibility




## Parameter Visibility

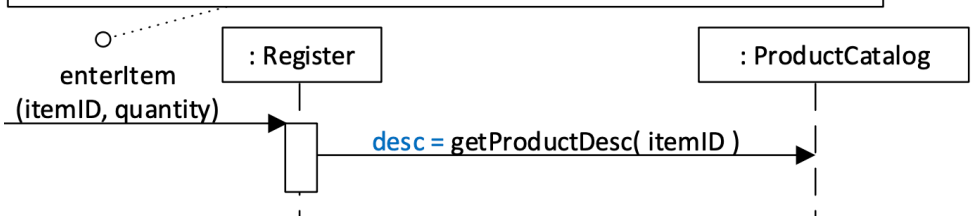



## Parameter to Attribute Visibility



## Local Visibility

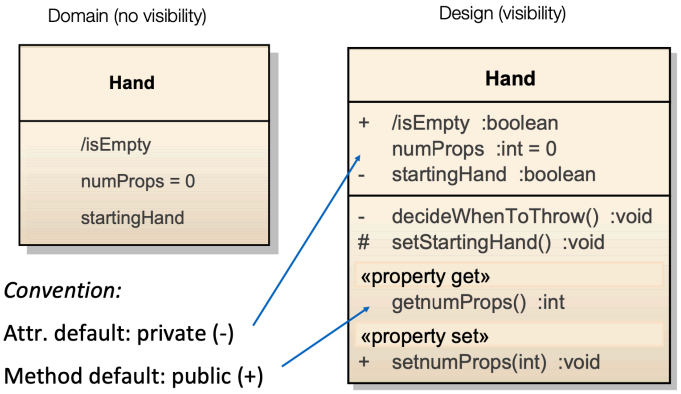
```
enterItem(id, qty)
{
...
// local visibility of ProductDescription via assignment of returning object
ProductDescription desc = catalog.getProductDesc(id);
...
}
```



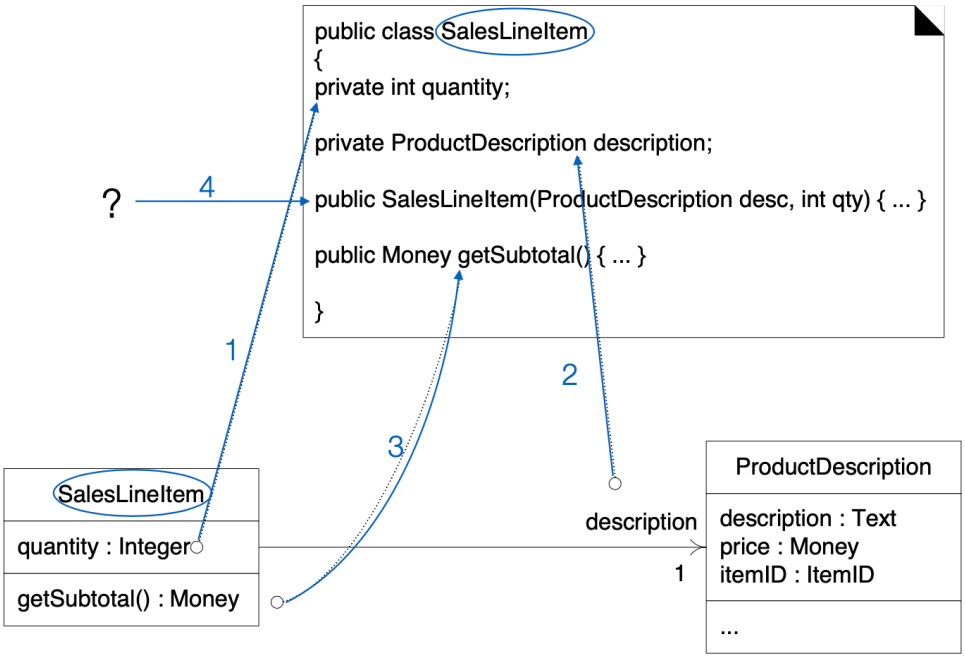

## Global Visibility

| | C++ | Java |
|---|---|---|
| **Declaration** | int v = 1; | public class Global {<br>public static int v = 1;<br>} |
| **Usage** | … main(…) {<br>int n1 = v;<br>int n2 = ::v;<br>} | … main(…) {<br>Global g = new Global();<br>int n1 = g.v;<br>int n2 = Global.v;<br>} |

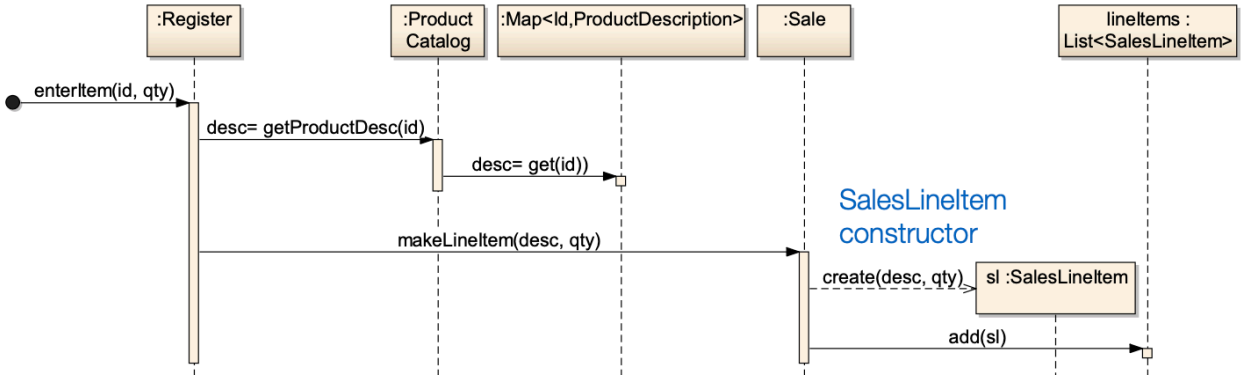
## Visibility marks



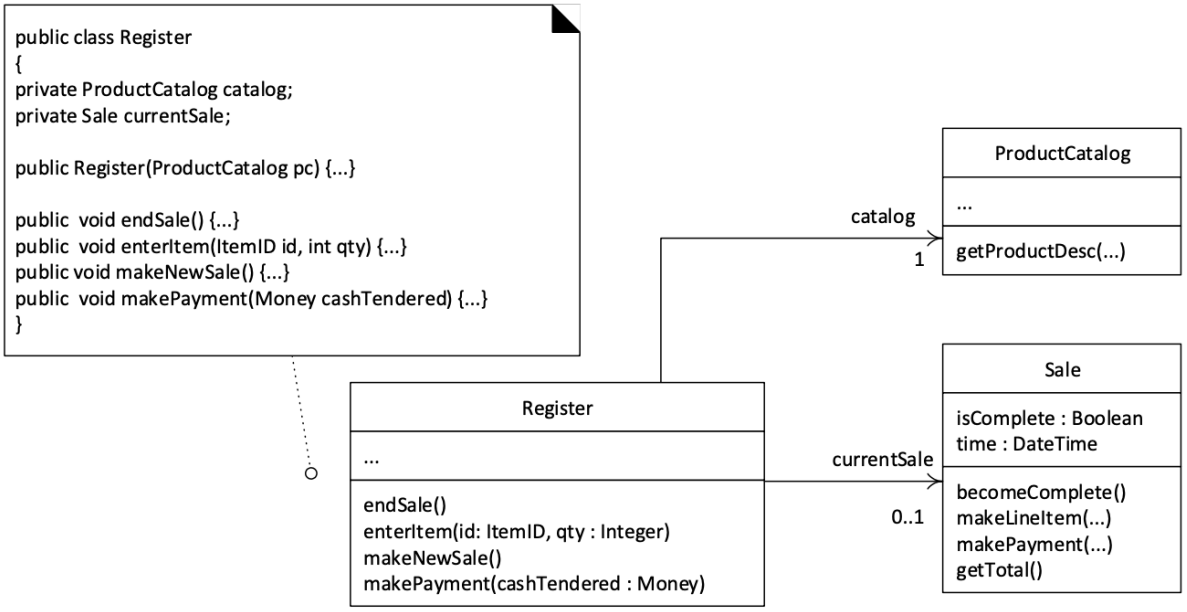

## Creating Class Definitions from DCDs (design class diagram)



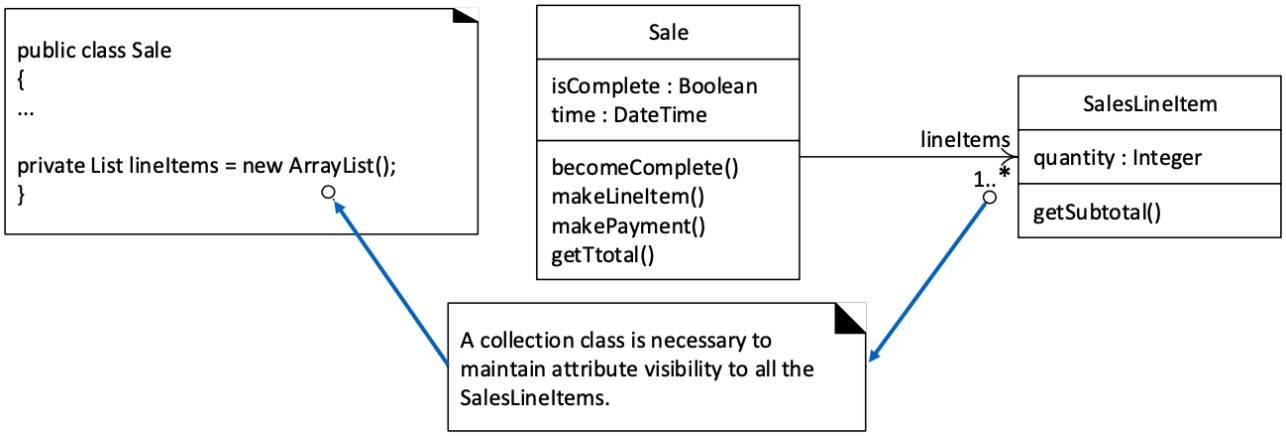
## Create methods from sequence Diagrams




## Multiplicity Adding a collection:

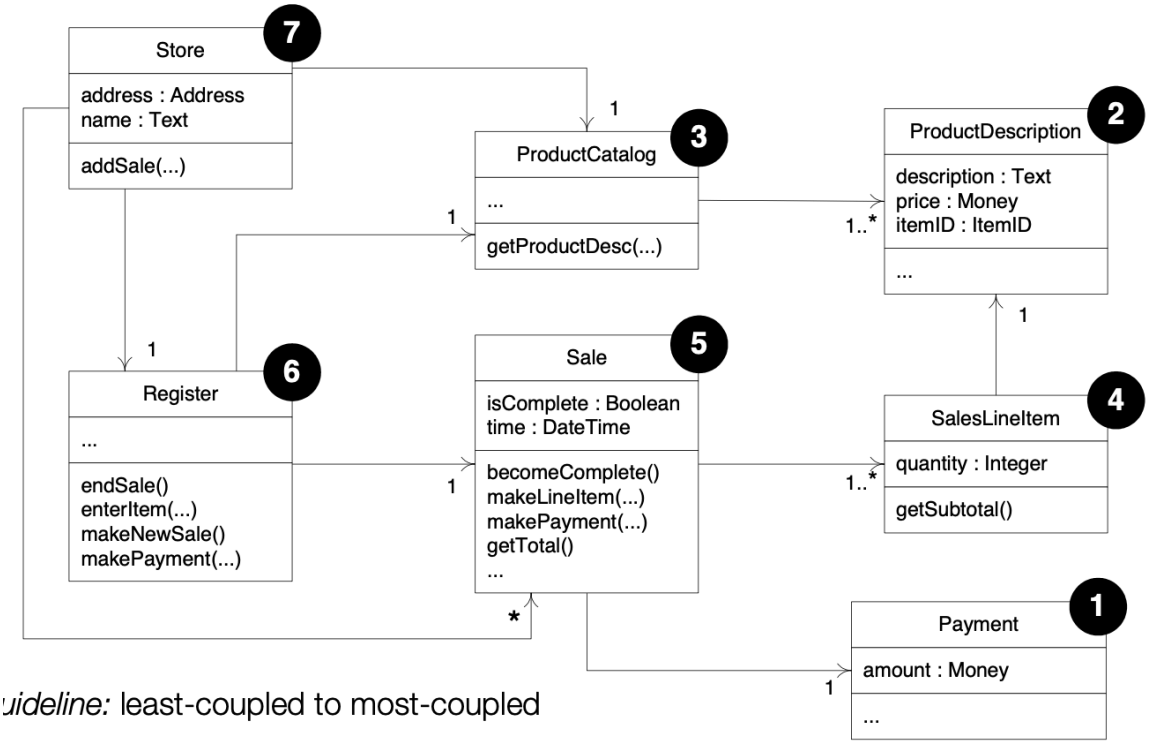



## Multiplicity: Adding a collection




Guidelines:
Chose collection class supporting required operations
E.g. Key-based lookup -> *Map*; Growing ordered list -> *List*
If it implements an interface, declare in terms of the interface
E.g. *Map<String, Integer> m = new HashMap<String, Integer>();*

## Possible order: implement/test




*uideline:* least-coupled to most-coupled

# Week04

**GRASP**: General Responsibility Assignment Software Patterns/Principles
- Definition: A set of patterns (or principles) of assigning responsibilities into an object.
  - Given a specific category of problem, GRASP guides the assignment of responsibilities to objects
- Useful to know and support Responsibility-Driven Design (RDD)
- Pattern – Definition:
  - A named and well-known problem/solution pair that can be applied in new contexts
  - A recurring successful application of expertise in a particular domain

**Advantages of Patterns**

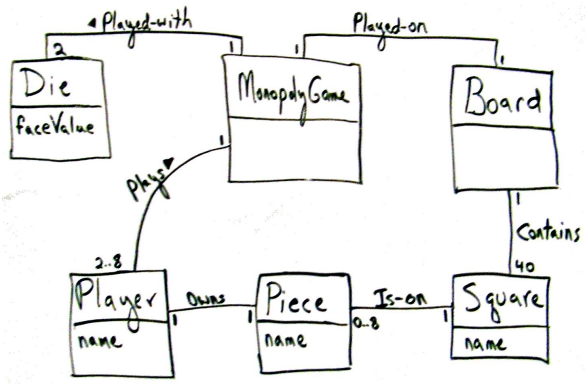
"To become a master, one must study the successful results of other masters!"
- Capture expertise and make it accessible to non-experts in a standard form
- Facilitate successful applications of expertise is easily re-used
- Improve understandability
- Facilitate communication among practitioners by providing a common language
- A new solution can be generated based on a modified version of existing patterns

**Creator**
- Problem: Who should be responsible for creating a new instance of class A? (Doing Responsibility)
  - Creation of objects is one of the most common OO activities
  - Useful for lower maintenance and higher opportunities for reuse (Low Coupling)
- Solution: Assign class B responsibility to create instances of class A if one of these is true (the more the better):
  - B "contains" or compositely aggregates A
  - B records A.
  - B closely uses A.
  - B has the initializing data for A.

To achieve **low representational gap**, we use the domain model to inspire the design classes

**Contraindications of creator**
- If creation of objects has significant complexity, e.g.,
  - Using recycled instances for performance
  - Conditionally creating an instance from one of a family of similar classes based upon some external property value or other context information.
- It is advisable to delegate creation to a helper class called a Concrete Factory or an Abstract Factory Rather than use the class suggested by Creator

**Example: Monopoly**




**Problem**: Who should be responsible for creating 'Square' object?

**Solution:**

– Who "contains" or compositely aggregates Square objects -> Board

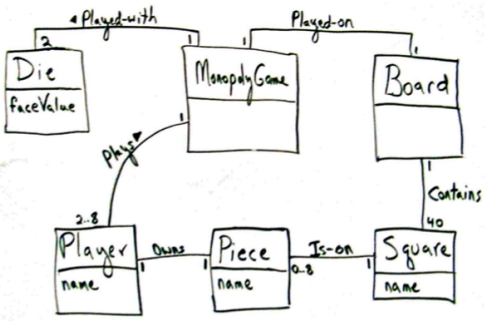
– Who records Square objects -> N/A

– Who closely uses Square objects -> Board & Piece

– Who has the initializing data for Square objects -> N/A

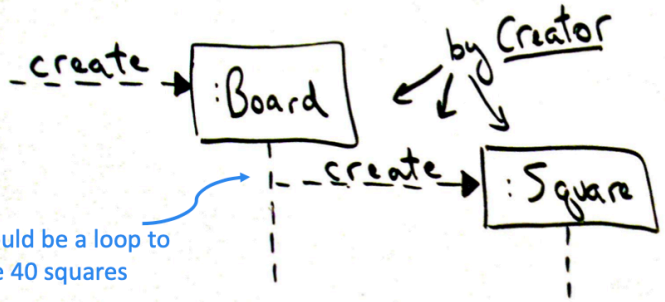
**Board class should be responsible for creating Square object**

# Example: Monopoly

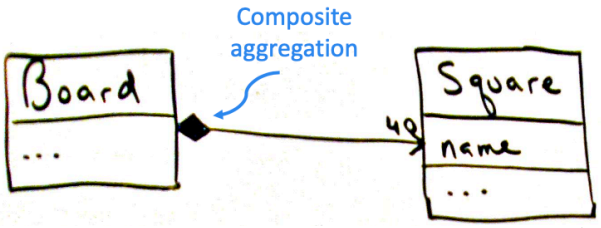



Monopoly Game Simulation Domain Model

***Board* class should be responsible for creating *Square* object**

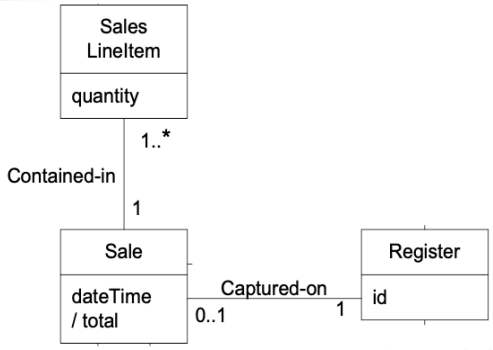



Design Sequence Diagram

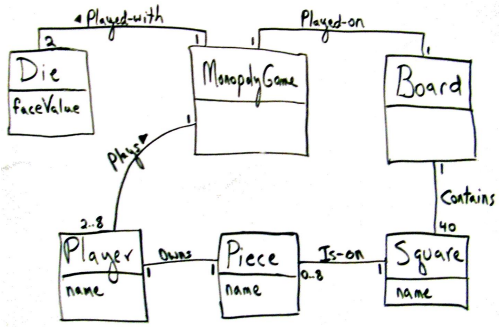



Design Class Diagram

**Who is responsible for creating "SaleLineItem" object?**
→ Since a Sale class contains/aggregates many SaleLineItem objects, Sale should create SaleLineItem objects




**Who is responsible for creating "Player" object?**
→ Since a MonopolyGame class has the initializing data for "Player" object (i.e., the Player's name received from the user input), MonopolyGame should create 'Player' object



**Information Expert (or Expert)**

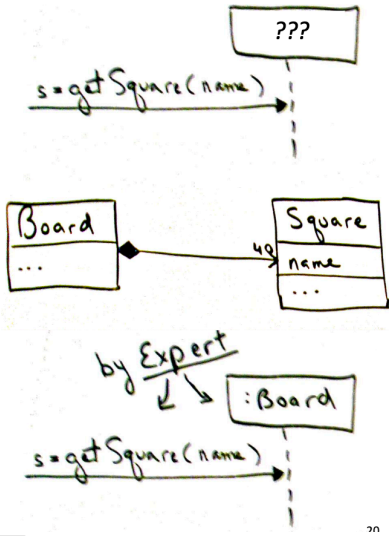
- Problem: What is a general principle of assigning responsibilities to objects?
  - A design model may have 100s (or 1,000s) software classes with 100s (or 1,000s) of responsibilities to be fulfilled
  - Given object X, which responsibilities (e.g., doing something or knowing something) can be assigned to X?
- Solution: Assign the responsibility to object X if X has the information to fulfill that responsibility
  - Information Expert is useful for understandability, maintainability, and extendibility
- Where to look?:
  - If there is relevant class in the Design model, look there first
  - Otherwise, look in the Domain model to inspire creation of design classes

**Contraindications of Information Expert**

- In some situations, a solution suggested by Expert is undesirable, usually because of problems in high dependencies
- Example: Who should be responsible for storing Sale transaction into the database?
  - By Expert, Sale should be responsible
  - BUT, database handling (e.g., SQL) is not related to the logic of the application
  - Put the database logic and application logic in one class will cause high dependencies -> poor maintainability
  - Database logic should be done by other helper classes

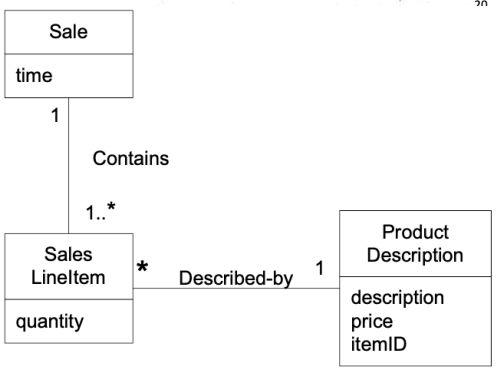
**Example: Monopoly**

Suppose there are objects that need to be able to reference a particular Square, given its name.
Who should be responsible for knowing a Square, given a key?
Approach:
→ Who has information about a Square? Who can access Square objects?
→ A software Board class aggregate all the Square objects (based on the previous example).
Solution: Board has information about 'Square' (i.e., an information expert). Then, Board should be responsible for knowing a Square, given a key.




**Another Example: POS**

Who should be responsible for calculating Subtotal?
Approach:
→ What information do we need for 'Subtotal'?
→ Subtotal = quantity * price
→ Who has (or can access) information about quantity and price?
→ SaleLineItem knows quantity, ProductDescription knows price (also information expert)
→ SaleLineItem associates with ProductDescription, then SaleLineItem can also access price
Solution: By Expert, SaleLineItem should be responsible for calculating Subtotal



**Low Coupling**

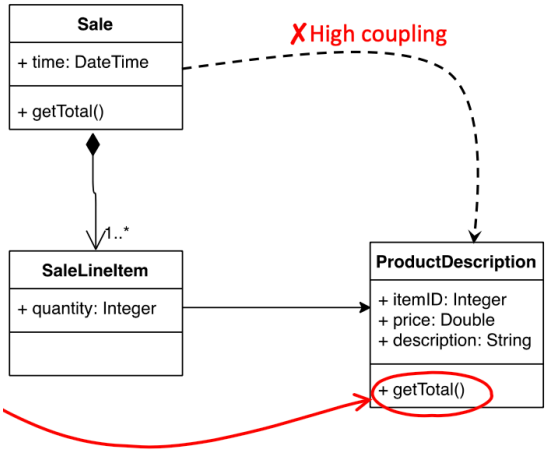
- Problem: How to support low dependency, low change impact, and increased reuse?
  - Coupling = how strongly one element is connected to (has knowledge of, or relies on) other elements
  - Low Coupling = An element is not dependent on too many other elements
  - High Coupling = Hard to understand in isolation, hard to reuse, one change impacts many classes (high impact change)
- Solution: Assign the responsibility to object X that coupling remain low.
  - Use this principle to evaluate alternatives
  - Low Coupling minimize the dependency, hence making system maintainable, efficient, and code reusable
- Low Coupling is an **evaluative principle** that should be kept in mind during all design decisions
- Low Coupling supports the design of classes that are more independent, reducing the impact of change
- Low Coupling help you avoid increasing unnecessary dependencies/coupling that could lead to a negative result

**Contraindications**

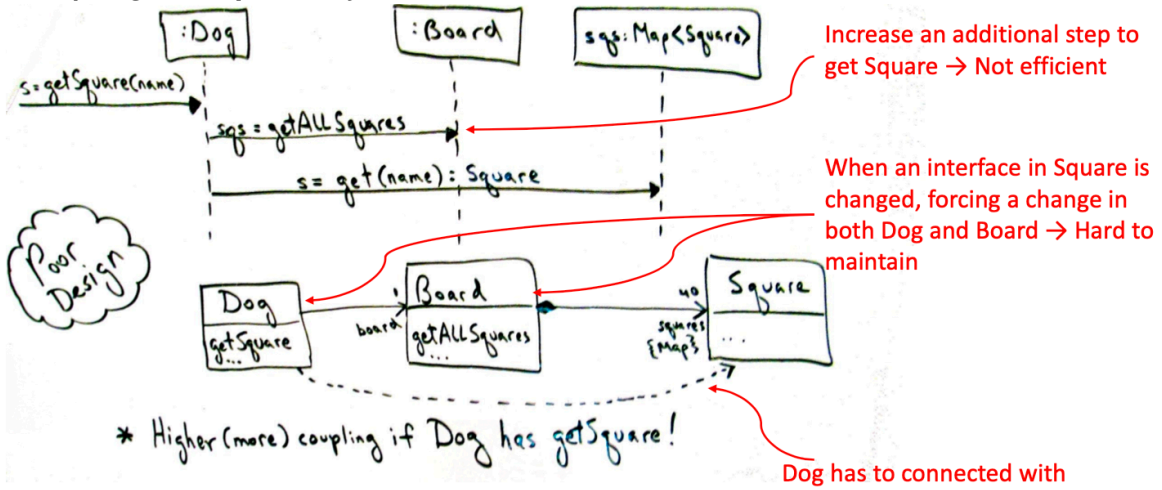
- Highly coupled with stable elements (e.g., Standard Java Libraries) should not be a problem, since these stable elements are unlikely to be changed often

**Who should be responsible for calculating SubTotal?**
SaleLineItem knows quantity,
ProductDescription knows price

**An alternative solution:**
By Expert, ProductDescription can also be responsible to calculate SubTotal
→ Yes. But when Sale object will sum the SubTotal of all SaleLineItem objects, Sale object needs to access ProductDescription
→ It is not suggested by Low Coupling principle




**Example monopoly**
**Why can't we have other class (like Dog; an arbitrary class) to get a Square object instead of Board?**



**High Cohesion**

- Problem: How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?
  - (Functional) Cohesion = How strongly related and focused the responsibilities of an element are
  - Low cohesion = a class has many unrelated things or does too much work, resulting code to be hard to comprehend, reuse, maintain
- Solution: Assign the responsibility to object X that make cohesion remains high.
  - Use this principle to evaluate alternatives
  - High cohesion will ease of comprehension of the design, simplify maintainability and enhancement
- Similar to Low Coupling, High Cohesion is an evaluative principle that should be kept in mind during all design decisions
- Both Low Coupling and High Cohesion must be considered together when designing software objects

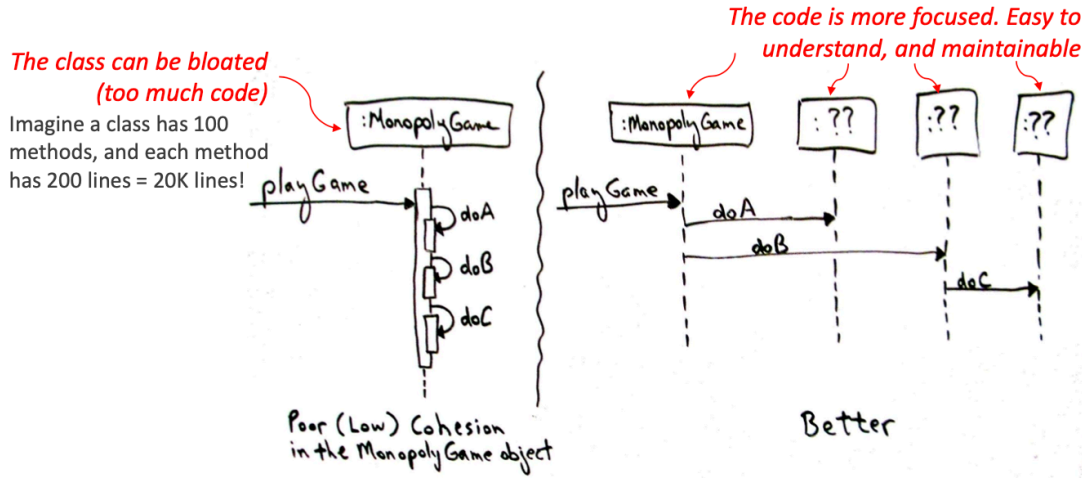
**Contraindications**

- Low cohesion (=many responsibilities in one class) sometimes is necessary for the design to meet strict non-functional requirements
  - Ex: Larger and less cohesive classes are used to reduce processing overheads (e.g., transmission, storage) to meet performance requirements

**Example: Monopoly**

**When there is an input of playGame to MonopolyGame object, what MonopolyGame should do next?**

-- Should MonopolyGame object be responsible for doing all the tasks, or delegate some tasks to other object?




Another Example: POS

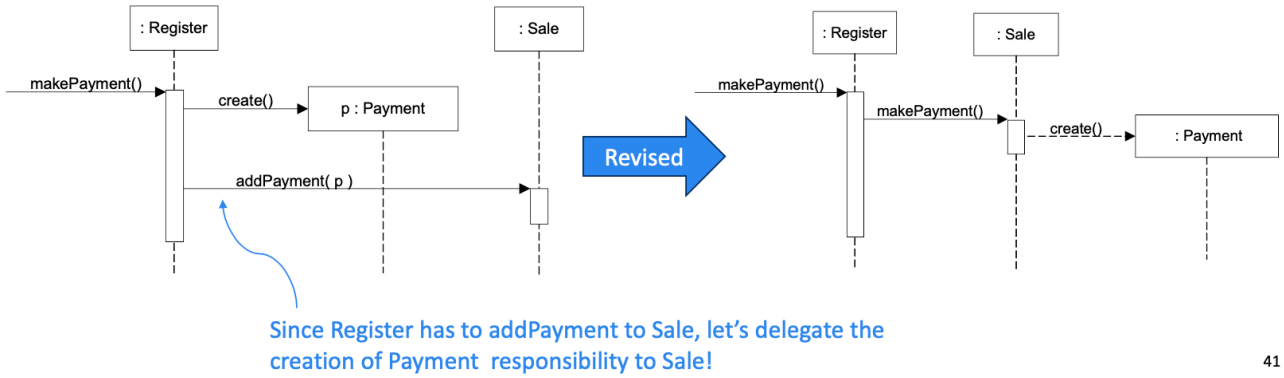
**Why Sale is responsible for creating Payment? Why not Register?**

In the real-world domain, Register records the Payment. By Expert, Register should be responsible for creating Payment.

→ Yes. It is acceptable.

→ BUT, in the real-world domain, Register involves most of the related system operations

→ Register is potentially bloated



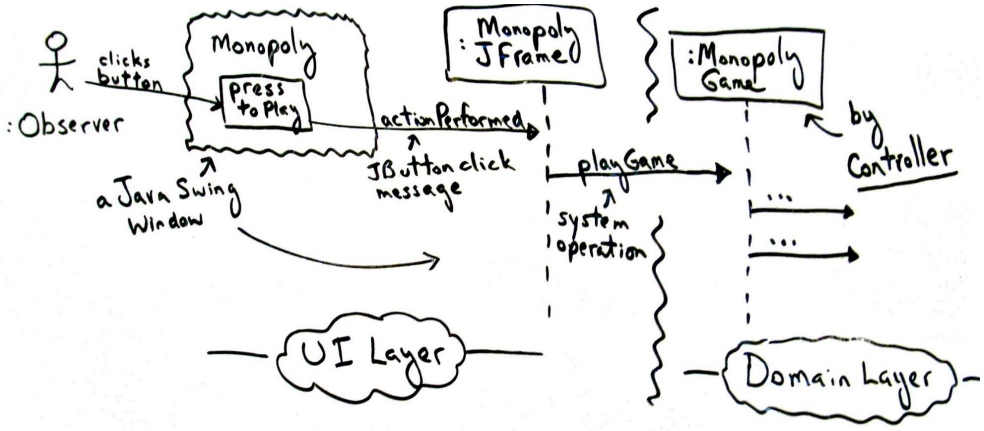
**Controller**

- **Problem**
  - What first object beyond the UI layer receives and coordinates (“controls”) a system operation?
- System operations:
  - major input events
  - appear on System Sequence Diagrams (SSDs)
- **Solution**
  - Assign responsibility to a class representing one of:
    - the overall “system”, a “root” object, a device the software is running within, or a major subsystem
      - façade controller
  - a use case scenario that deals with the event, e.g.
    - <UseCaseName>(Handler|Coordinator|Session)
      - use case or session controller

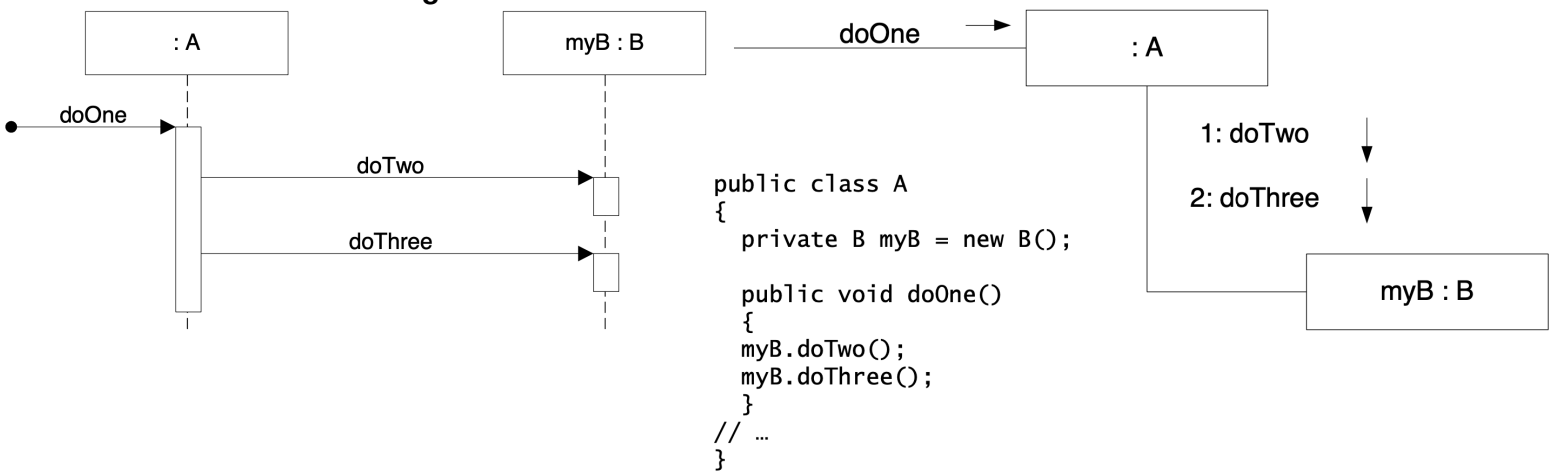
**Contraindications of controller**

- Bloated controller: too many responsibilities, low cohesion, unfocussed
  - Single controller class for all system events
    - Solution: add more controllers (façade → use case)
  - Controller too complicated, contains too much or duplicated information: violates Information Expert and High Cohesion
    - Solution: delegate!

**Controller Pattern**

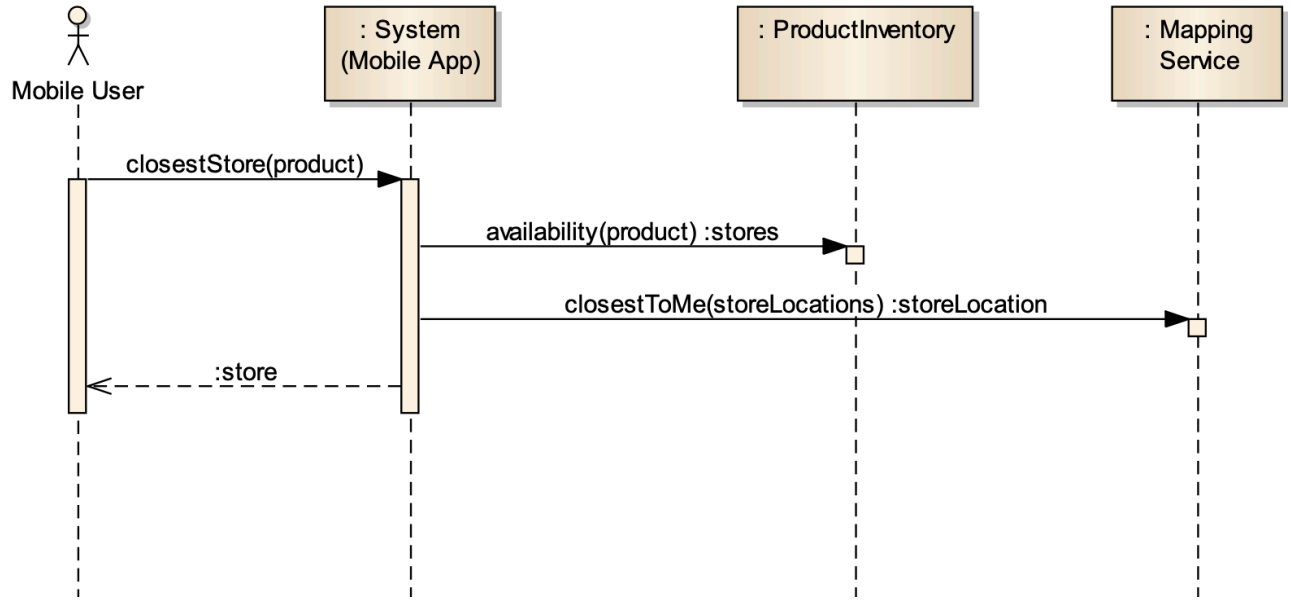


## Communication Diagram

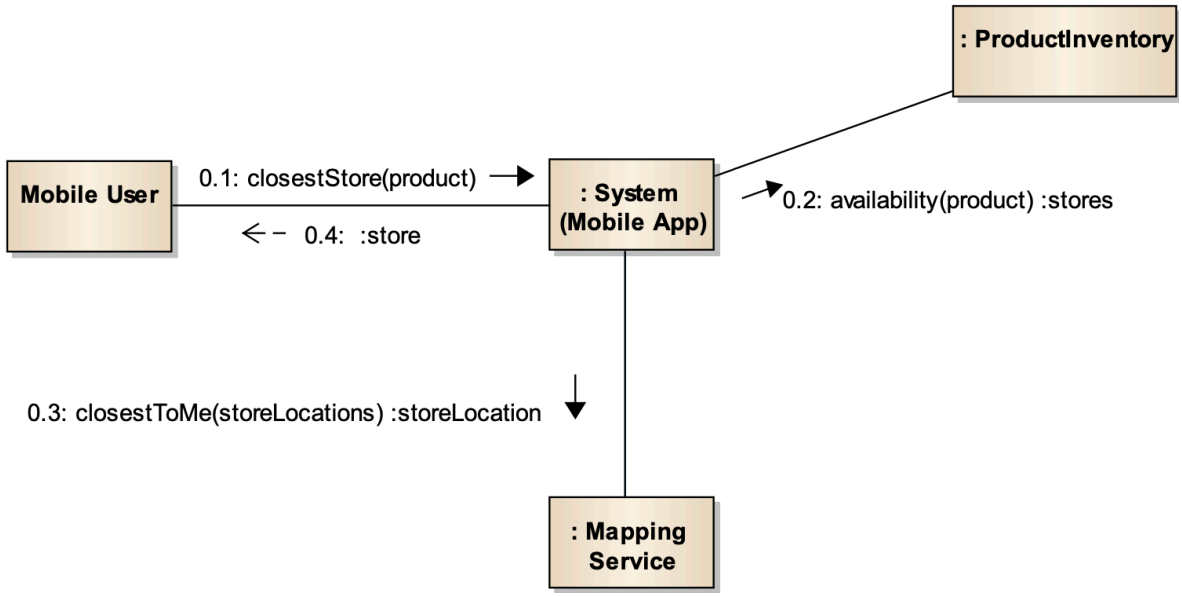



```
public class A
{
  private B myB = new B();

  public void doOne()
  {
  myB.doTwo();
  myB.doThree();
  }
// …
}
```

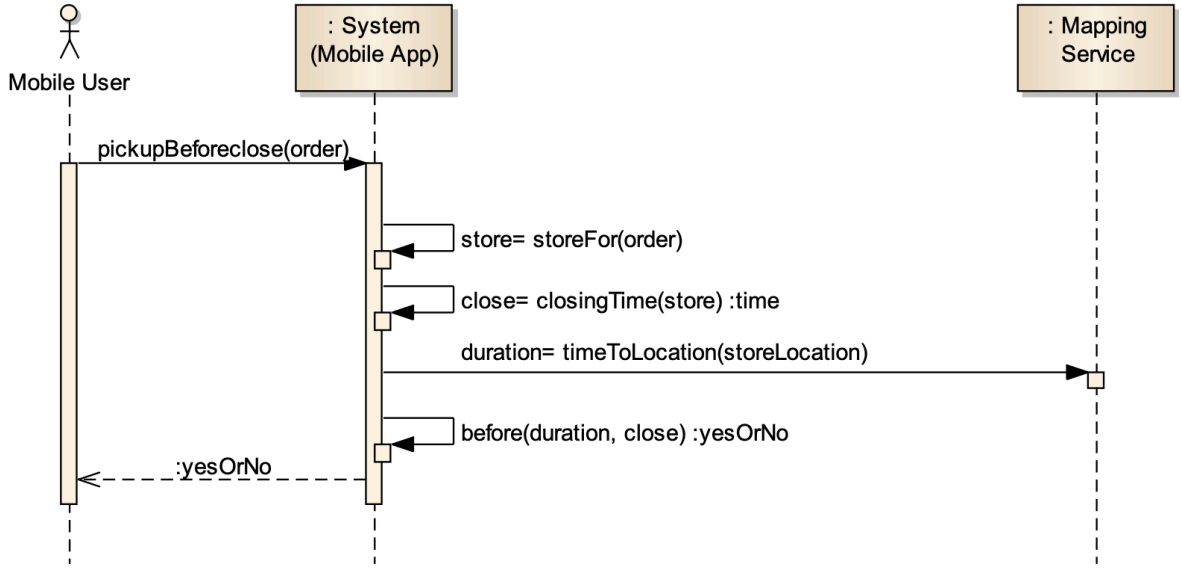
## Mobile App: SSD closestStore



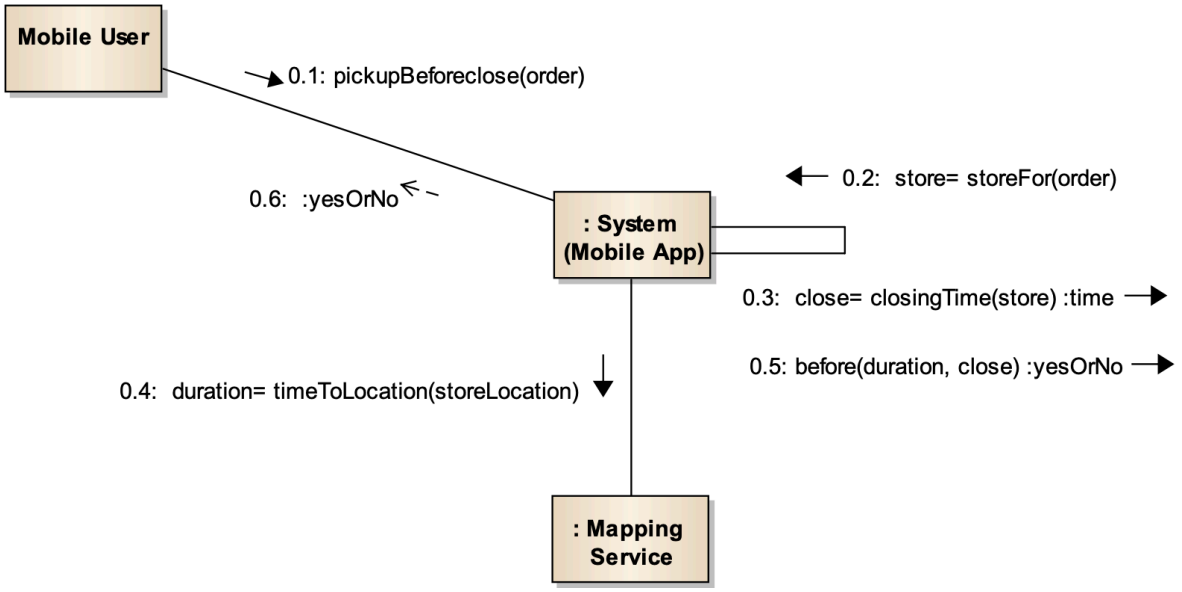
## Mobile App: SCD closestStore



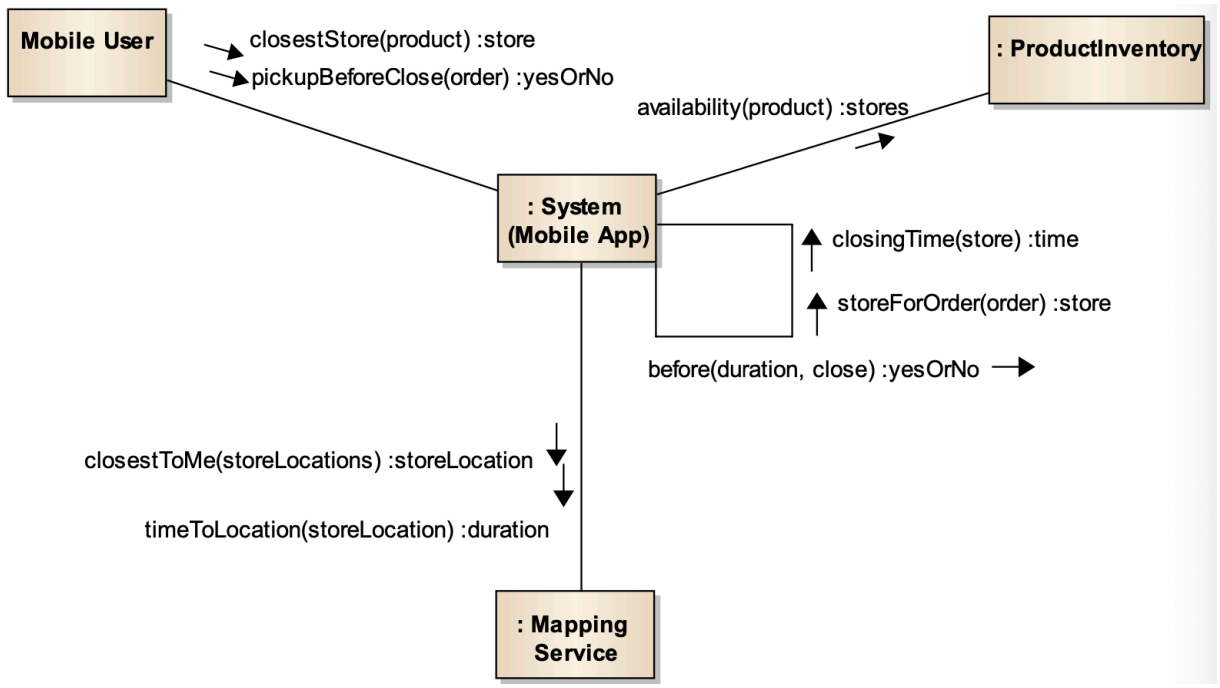

## Mobile App: SSD pickupBeforeclose



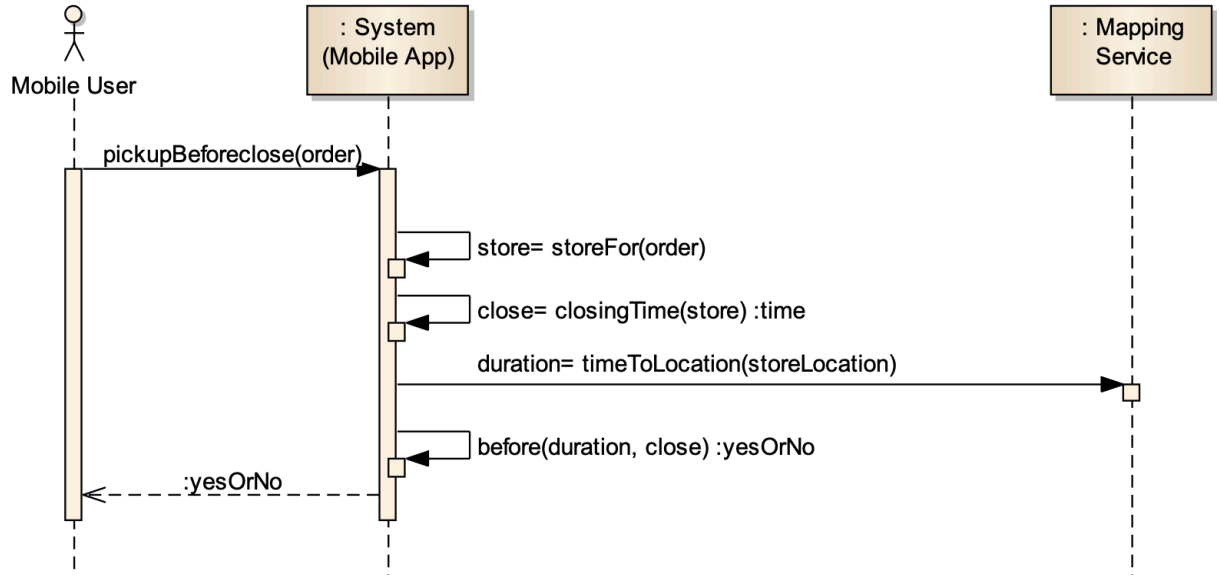

## Mobile App: SCD pickupBeforeclose



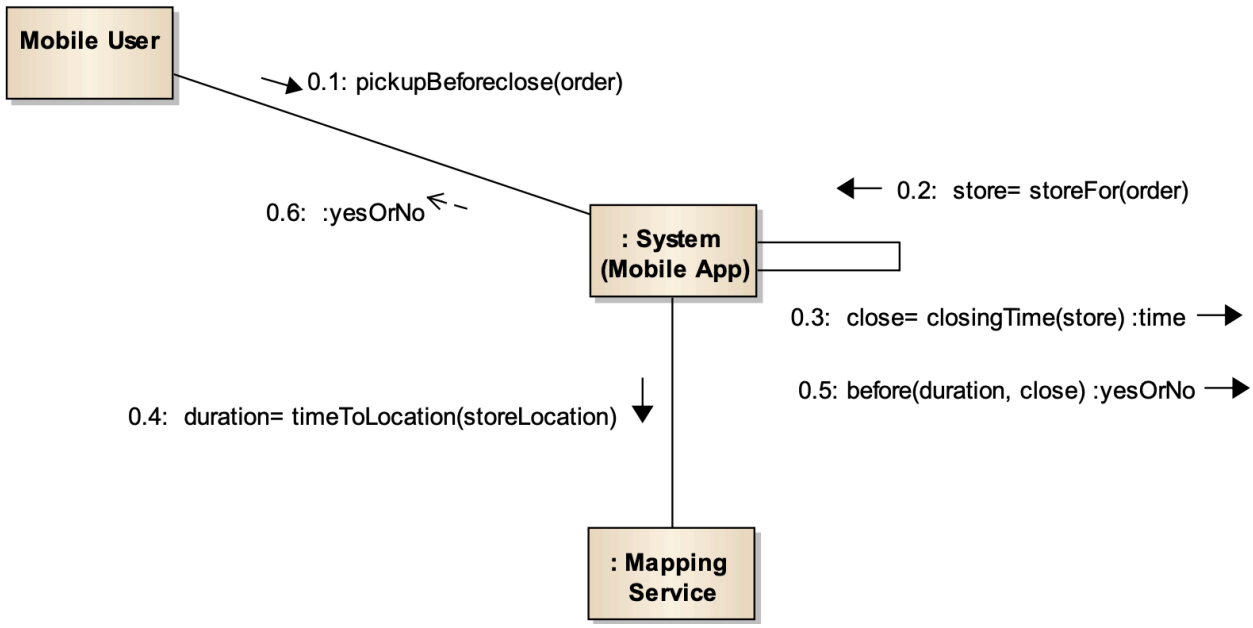
## Mobile App: SCD combined



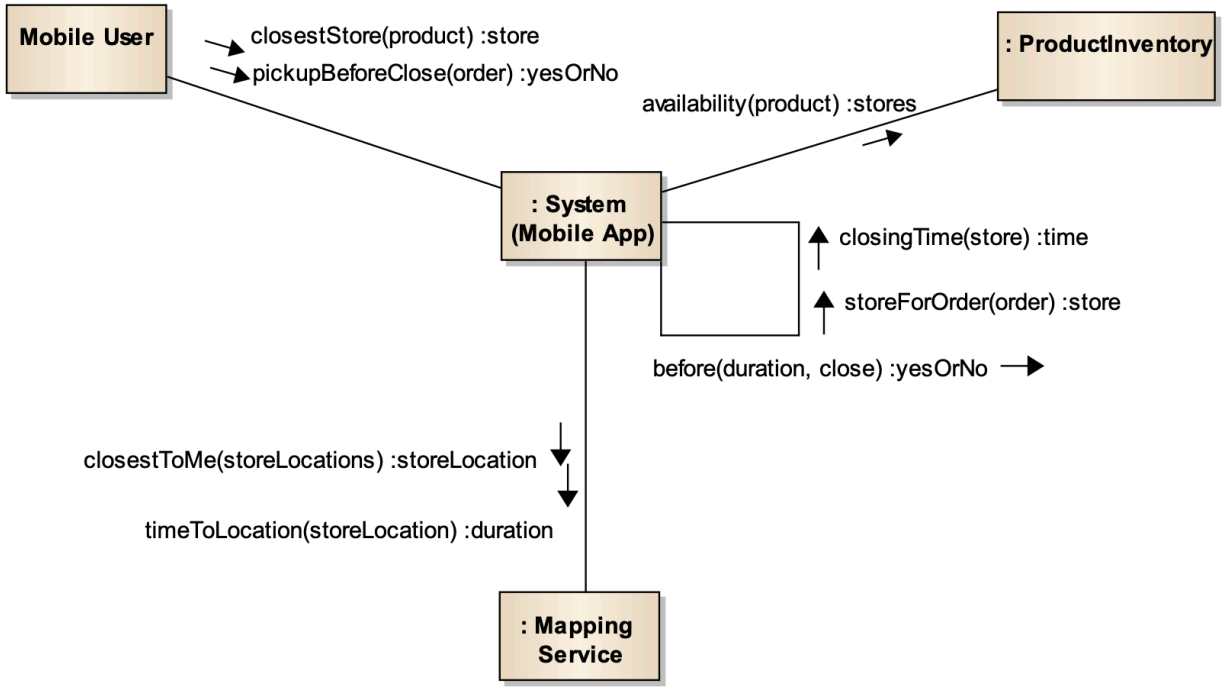

## Mobile App: SSD pickupBeforeclose




## Mobile App: SCD pickupBeforeclose



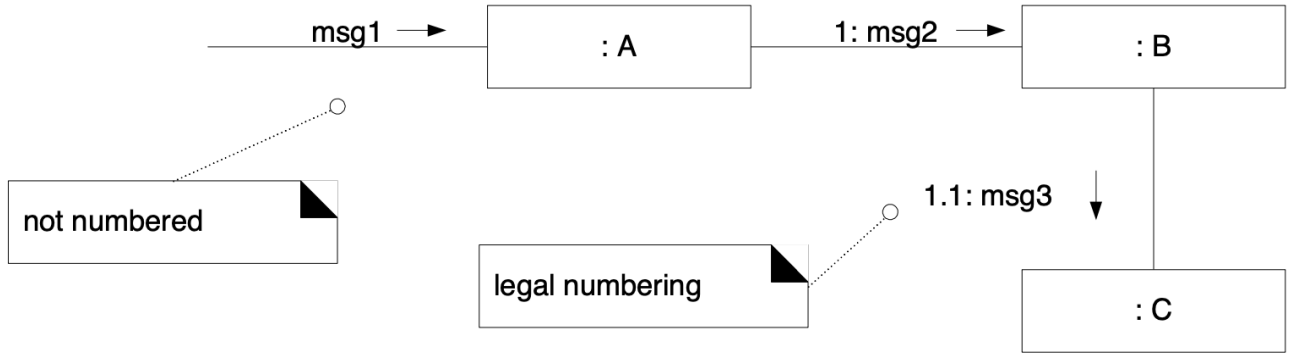
## Mobile App: SCD combined



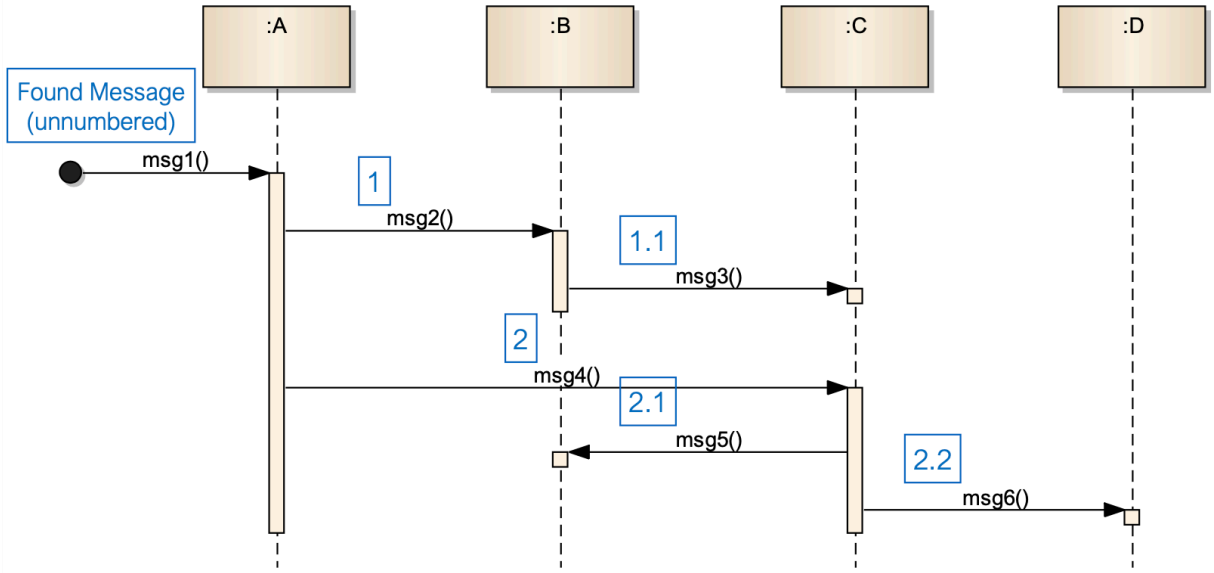

## Sequence vs. communications

| Type | Strengths | Weaknesses |
|---|---|---|
| **sequence** | Clearly shows time ordering of messages<br><br>Can more easily convey the detail of message protocols between objects | Linear layout of instances can obscure relationships<br><br>Linear layout consumes horizontal space |
| **communications** | More layout options<br><br>Clearly shows relationships between object instances<br><br>Can combine scenarios to provide a more complete picture | More difficult to see message sequencing<br><br>Fewer notation options for expressing message patterns |

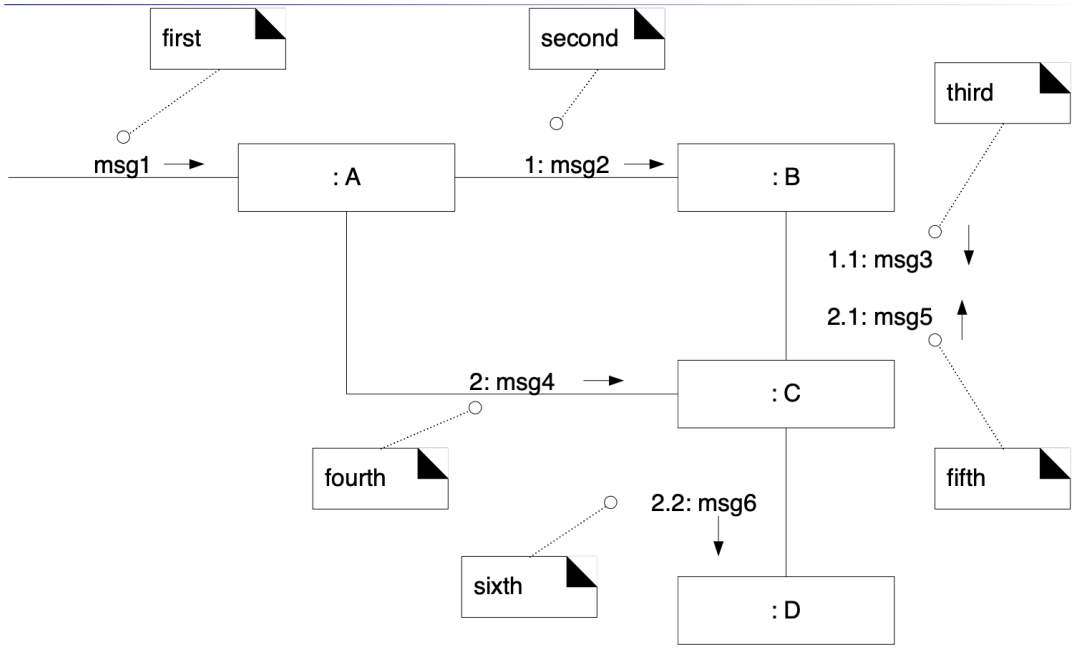
## CD: Message Sequence Numbers




## Complex sequencing




## CD: complex sequencing



# Week05

***Generalization:*** The activity of identifying commonality among concepts
- Define superclass (general concept)
- Define relationships with subclasses (specialized concepts)
- Conceptual subclasses and superclasses are related in terms of set membership

Reasons for using generalization
- Economy of expression
- Reduction in repeated information
- Improved comprehension

**Domain Model: Generalization-Specialization Class Hierarchy (or Class Hierarchy)**
Guideline for creating subclasses:
- Subclass has additional attributes of interest
- Subclass has additional associations of interest
- Subclass is operated on, handled, reacted to, or manipulated differently to the other classes in noteworthy ways

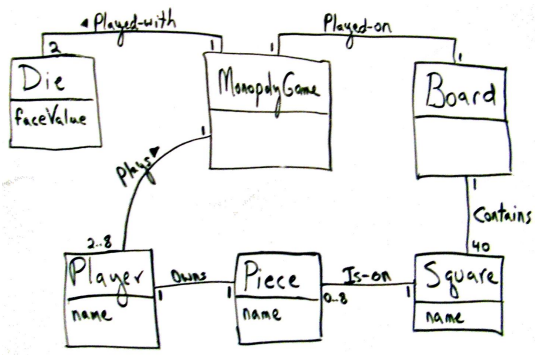
Modeling Guidelines:
a) Declare superclasses abstract
b) Append the superclass name to the subclass

**Polymorphism**
- **Problem (1)**: How handle alternatives based on type?
  - Conditional variation: Adding new alternatives and if-then-else (or case-statement) are required many places
  - E.g., behavior varies based on some conditions
- **Problem (2)**: How to create pluggable software components?
  - – E.g., Client-Server relationship: How to replace Server component without affecting Client?
- **Solution**: When related alternatives or behaviors vary by type (class), assign responsibility for the behavior—using polymorphic operations—to the types for which the behavior varies
  - Polymorphic operations: Operations with the same interface
  - **Corollary**: Do not test for the type of an object and use conditional logic to perform varying alternatives based on type.

Polymorphism is a *fundamental* pattern in designing how a system is organized to *handle similar variations*

**Contraindications**:
- Designing systems with interfaces and polymorphism for speculative “future-proofing” against an unknown variation could be useful.
- However, the unnecessary effort might occur if polymorphism is applied at the points where variations, in fact, are improbable and will never actually arise.

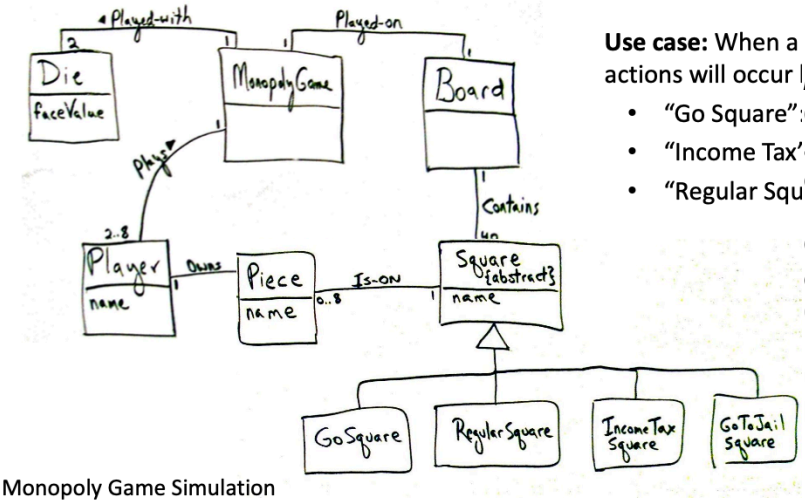
**Example: Monopoly**



Monopoly Game Simulation Domain Model

**Use case**: When a player lands on a square, different actions will occur based on the types of square, e.g.,
•“Go Square”: A player receives $200
•“Income Tax”: A player pays a tax of the income
•“Regular Square”: A player does nothing
**Realization**: The different types of Square can be seen as concepts in the domain as well. It should be captured in the domain model
→ Since the concepts are similar, use **a generalization specialization class hierarchy (class hierarchy)** to organize these concepts

## Example: Monopoly – Update *Domain* Model




Monopoly Game Simulation Domain Model

**Use case:** When a actions will occur

- "Go Square"
- "Income Tax"
- "Regular Squ

## Example: Monopoly – Create *Design* Model (1)

**Observation:** Based on Use case and domain model, a different type of square has a different rule (i.e., behaviour)

**Corollary:** We should not design of our software objects with case logic (e.g., a switch statement)

→ If a switch case (or if-else) is used in Square class, the class can have *low cohesion* and can be bloated!

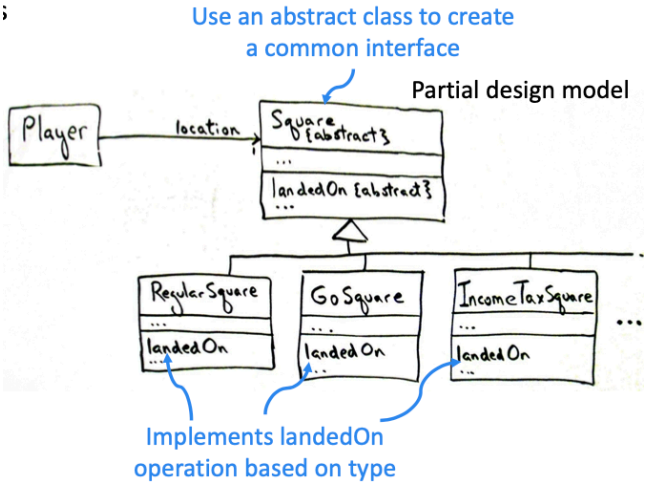
## Example: Monopoly – Create *Design* Model (2)

**Analysis:** There are alternatives based on type of Squares

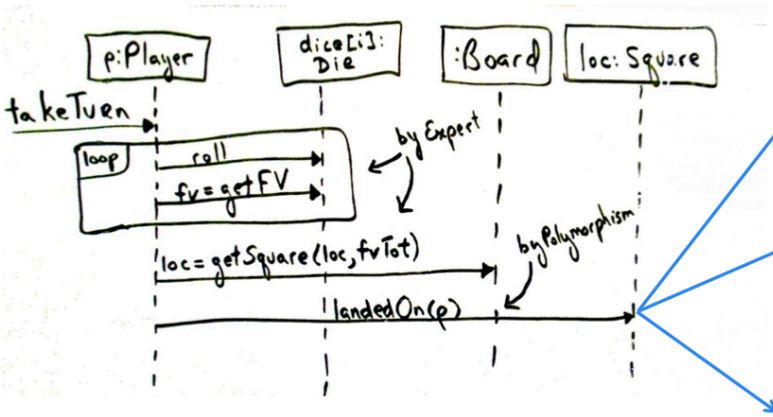
→ Polymorphism should be used.

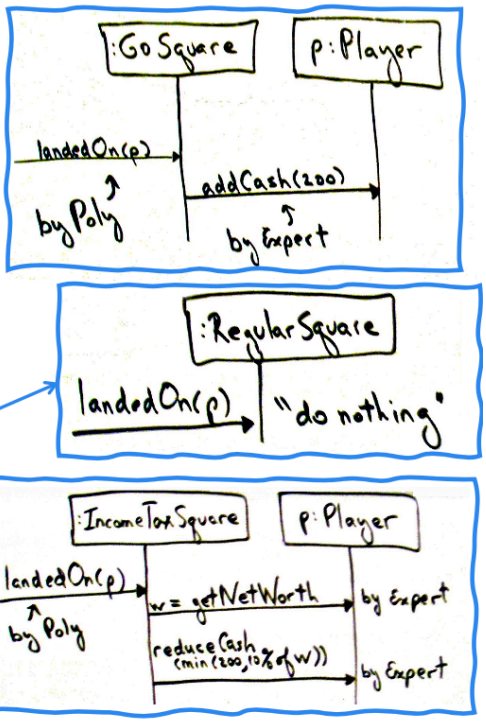
**Approach**: Based on Polymorphism, we should create a polymorphic operation for each type for which the behavior varies.

- What are types that Square varies?
  → "Go Square", "RegularSquare", "IncomeTaxSquare", so on
- What is the operation that varies?
  → The behaviour of Square will vary when a player *lands on* a square
  → Then, the responsibility of "landedOn" should be a *polymorphic operation*




## Example: Monopoly




The behaviour of square is based on the type of Square that Board returns to Player

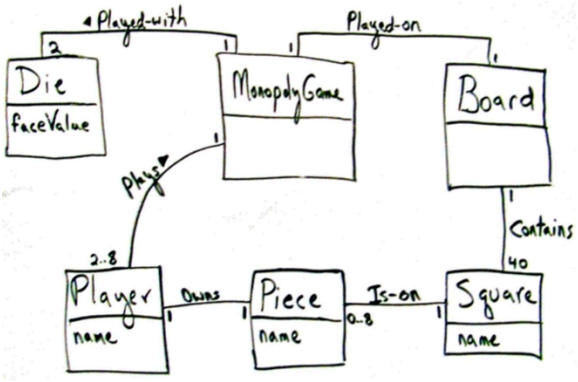
**Pure fabrication**

- **Problem:** Which object should have responsibility when you do not want to violate *High Cohesion* and *Low Coupling*, but solutions offered by other patterns (e.g. Expert) are not appropriate?
  - Domain model often inspires the design of software objects (achieving low representational gap)
  - In many situations, assigning responsibilities only based on the domain model often leads to the problem of poor cohesion or coupling.
- **Solution:** Assign a highly cohesive set of responsibilities to an *artificial* or convenience class that does not represent a problem domain concept
- Made up a class to support high cohesion, low coupling, and reuse

**Pure Fabrication: Contraindications**

- Pure Fabrication should not be used as excuse to add new objects
- If overused Pure Fabrication, the design can end up with one class has one responsibility
  - Too many behavior objects that have responsibilities not co-located with the information required for their fulfillment
  - can adversely affect coupling (i.e., high dependencies)

# Example: Monopoly

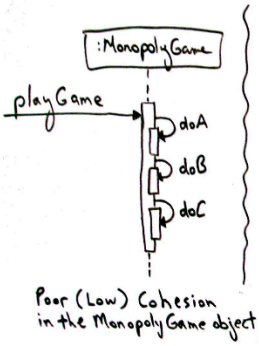



Monopoly Game Simulation Domain Model

Who should be responsible for rolling the dice?

- Based on the domain model & Expert, MonopolyGame class should be assigned for this responsibility

Discussion: It is acceptable, but leads to low cohesion



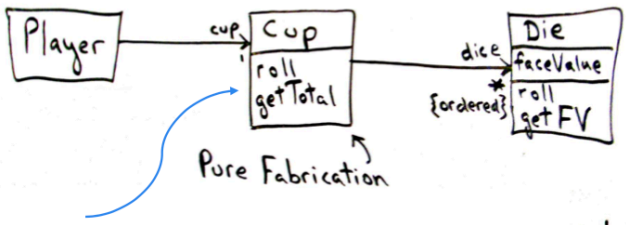

Based on the domain model, the design can violate high cohesion

Let's use **Pure Fabrication**!

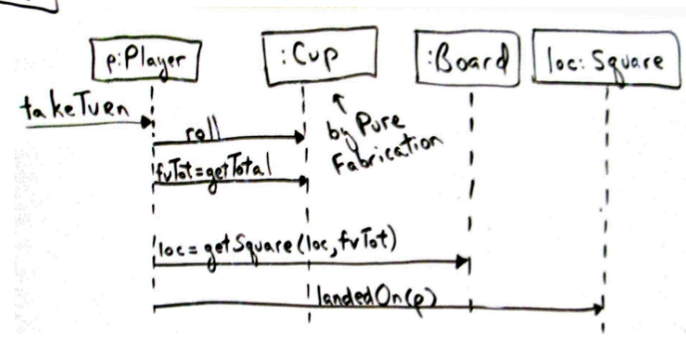
2

Who should be responsible for rolling the dice?

Solution: By Pure Fabrication, let's create an artificial class which is responsible for rolling dice




In the domain, Cup was not used in Monopoly Game

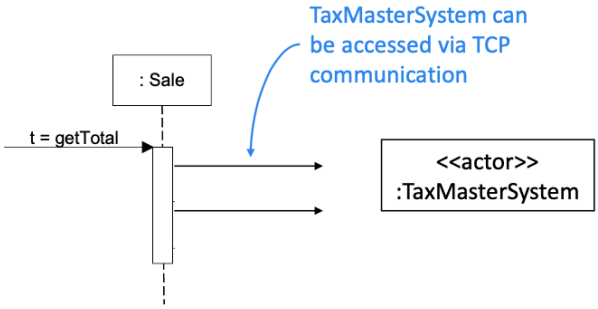


**Indirection**

- **Problem**: Where to assign a responsibility, to avoid direct coupling between two (or more) things?
  – How to de-couple objects so that low coupling is supported and reuse potential remains higher?
- **Solution**: Assign the responsibility to an intermediate object to mediate between other components or services so that they are not directly coupled.
  – The intermediary creates an indirection between the other components.

- Many Pure Fabrications are generated because of Indirection.
  - Indirection: Assigning any class as an intermediary
  - Pure Fabrications: Assign an artificial class (that does not represent the domain) as an intermediary
- The motivation for Indirection is usually Low Coupling (i.e., decouple objects for future reuse)
- Old adage:
  "Most problems in computer science can be solved by another level of indirection"
- Counter adage:
  "Most problems in performance can be solved by removing another layer of indirection!"

## Example: POS




POS will use Tax Master System (an external service) to calculate tax. Who should be responsible for calculating tax?

- By Expert, Sale is responsible for calculating total. So, Sale should also calculate tax
- Should Sale be directly use TaxMasterSystem via TCP communication?

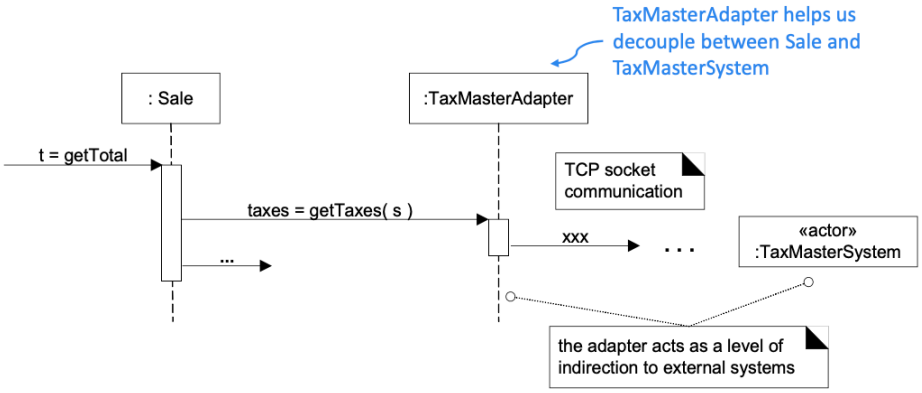
**Discussion:** If this responsibility is assigned to Sale,

→ a lot of TCP communication to TaxMasterSystem will be implemented in Sale

→ Sale class is highly coupled with TaxMasterSystem

## Example: POS

**Solution:** By Indirection, the responsibility of communicating with TaxMasterSystem should be assigned to an intermediate class, i.e., TaxMasterAdapter



**Protected Variation**

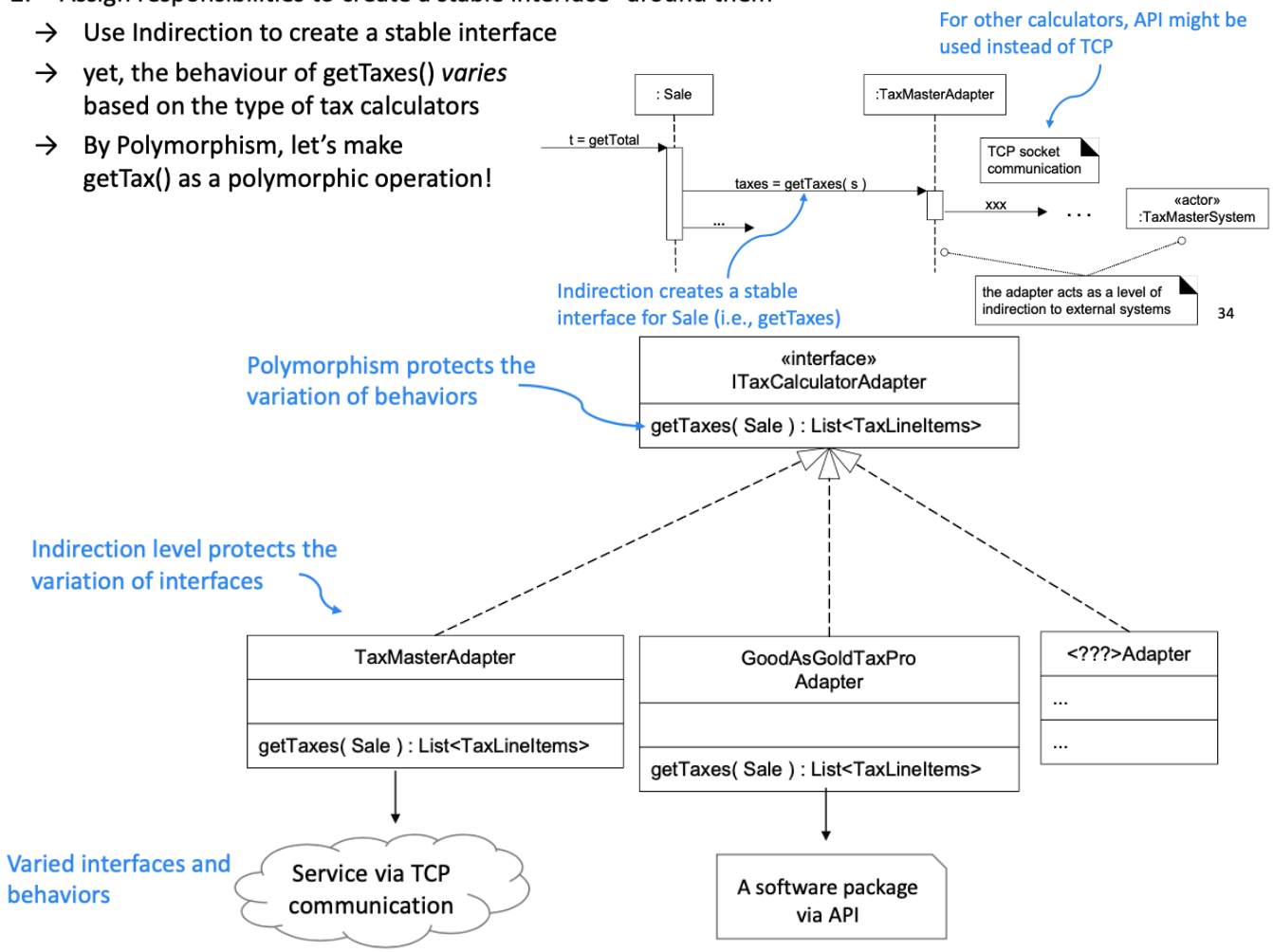
- **Problem**: How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?
  – Points of change include:
    Variation point: variations in existing system or requirements
    Evolution point: speculative variations that may arise in the future
- **Solution**:
  - Identify points of known or predicted variation or instability;
  - Assign responsibilities to create a stable interface* around them
  - *Interface: a means of access (not only a programming language interface or Java interface)
- Protected Variation (PV) is very important, fundamental principle of software design!
- PV is a root principle motivating most of the mechanisms and patterns in programming and design.
- For example: Core Protected Variations Mechanisms, Data-Driven Design, Service Lookup, Interpreter-Driven Design, (See more in textbook)
- PV is equivalent to **Open-Closed Principle** (OCP), described by Bertrand Meyer
  - OCP: Modules should be both open (for extension; adaptable) and closed (to modification that affects clients).
  - A class can be closed w.r.t. attribute access (access methods only – no changes), but open to modification of underlying attributes (+ addition of new methods).
- **Contraindications**: Similar to Polymorphism
  - Cost of speculative "future-proofing" at evolution points can outweigh the benefits
  - It may be cheaper/easier to rework a simple "brittle" design

# Example: POS

POS currently uses 'TaxMasterSystem' to calculate tax. In the future, other tax calculators might be used.

**Analysis:** To protected variation:

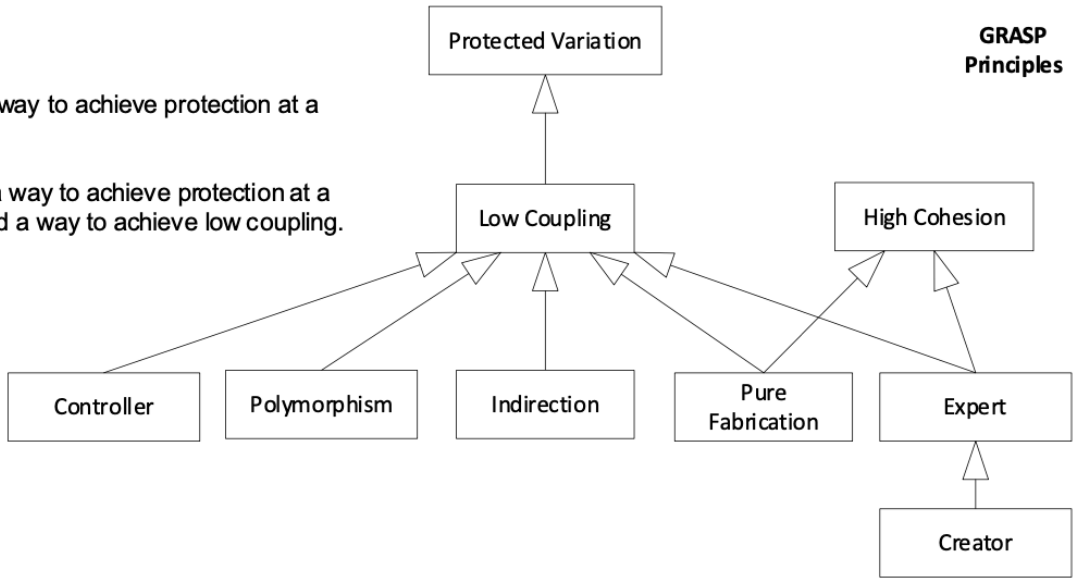
1. What is the point that will be known or predicted variation or instability?
   → The point of instability or variation is the different interfaces or APIs of external tax calculators
2. Assign responsibilities to create a stable interface* around them
   → Use Indirection to create a stable interface
   → yet, the behaviour of getTaxes() *varies* based on the type of tax calculators
   → By Polymorphism, let's make getTax() as a polymorphic operation!



*For example:*

Low coupling is a way to achieve protection at a variation point.

Polymorphism is a way to achieve protection at a variation point, and a way to achieve low coupling.




Achieve Low coupling: Controller, polymorphism, indirection, pure fabrication, expert
Achieve high cohesion: pure fabrication, expert
Achieve expert: creator
Achieve protection at a variant point: low coupling

**Test-Driven Development**
- Development practise in which test code is written before the code that it will test, e.g.
  - acceptance tests at start of iteration
  - unit tests before the corresponding class
- General approach:
  - alternate between test code & production code
  - ensure production code passes tests before proceeding
- Promoted in iterative and agile practice (esp. XP)

**Advantages of Writing Tests First**
- Tests actually get written
- Programmer satisfaction leading to more consistent test writing
- Clarification of detailed interface and behaviour
- Proven, repeatable, automated verification
  - Build up a suite of tests, easy to re-run
- The confidence to make changes
  - Tests can check for unwanted change, and can be changed to check for wanted change

**Refactoring**
- Structured, disciplined method for rewriting or restricting existing code without changing its external behaviour.
- Small behaviour preserving transformations (refactors) can be applied, one at a time.
- Unit tests can be re-executed to show that the refactoring id not cause a regression (failure).
- A series of small test transformations can result in a major restructuring of the code and design (for the better) with no behaviour change.

**Bad Smelling Code**
- duplicated code
- big method
- class with many instance variables
- class with lots of code
- strikingly similar subclasses
- little or no use of interfaces in the design
- high coupling between many objects

| Refactoring | Description |
|---|---|
| Extract Method | Transform a long method into a shorter one by factoring out a portion into a private helper method. |
| Extract Constant | Replace a literal constant with a constant variable. |
| Introduce Explaining Variable (specialization of extract local variable) | Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose. |
| Replace Constructor Call with Factory Method | In Java, for example, replace using the new operator and constructor call with invoking a helper method that creates the object (hiding the details). |

```
        // good method name, but the logic of the body is not clear
boolean isLeapYear( int year )
{
        return  ( ( ( year % 400 ) == 0 ) ||
                ( ( ( year % 4 ) == 0 ) && ( ( year % 100 ) != 0 ) ) );
}
```

Explaining Variables

```
           // that's better!
   boolean isLeapYear( int year )
   {
           boolean isFourthYear = ( ( year % 4 ) == 0 );
           boolean isHundrethYear = ( ( year % 100 ) == 0);
           boolean is4HundrethYear = ( ( year % 400 ) == 0);
           return (
                   is4HundrethYear
                   || ( isFourthYear && ! isHundrethYear ) );
   }
```

```
public void takeTurn()
{
                // roll dice
        int rollTotal = 0;
        for (int i = 0; i < dice.length; i++)
        {
                dice[i].roll();
                rollTotal += dice[i].getFaceValue();
        }

        Square newLoc = board.getSquare(piece.getLocation(), rollTotal);
        piece.setLocation(newLoc);
}
```

```
public void takeTurn()
{
                // the refactored helper method
        int rollTotal = rollDice();

        Square newLoc = board.getSquare(piece.getLocation(), rollTotal);
        piece.setLocation(newLoc);
}

private int rollDice()
```

Extracted Method

```
{
        int rollTotal = 0;
        for (int i = 0; i < dice.length; i++)
        {
                dice[i].roll();
                rollTotal += dice[i].getFaceValue();
        }
        return rollTotal;
}
```

## Conclusion

- Test driven development can play a key role in an Agile process
- A set of successfully passed tests represents a behaviour base-line for the system and its components
- Making design changes can be essential to keeping a maintainable, modifiable and understandable design
- Refactoring, supported by regression testing, is a disciplined approach to achieving design changes

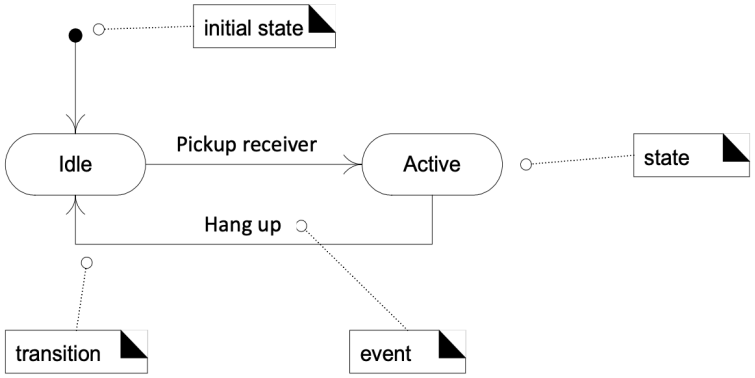
# Week06

**State Machines**

**Definition**: A state machine is a behavior model that captures the dynamic behavior of an object in terms of states, events, and state transitions.

- A ***state*** is the condition of an object at a moment in time
- An ***event*** is a significant or noteworthy occurrence that affects the object to change a state
- A **transition** is a directed relationship between two states such that an event can cause the object to change from the prior state to the subsequent state

**A visual model**: UML State Machine Diagram



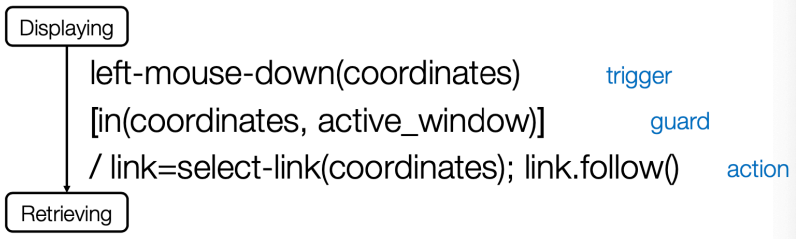

**How to Apply State Machine Diagrams?**

Determines the behavior of an object

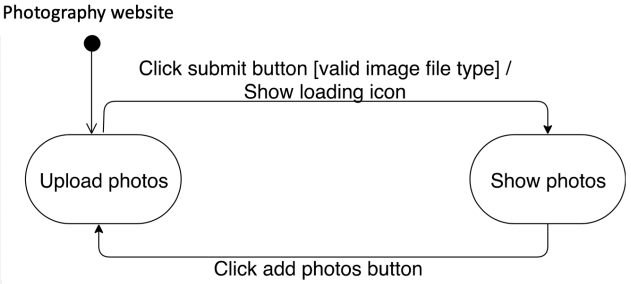
- **State-dependent object:**
  – Reacts differently to events depending on the object's state
- **State-independent object:**
  – For all events of interest, an object always reacts to the event the same way
- **State-independent w.r.t. an event:**
  – Always responds to event the same way
- **Guideline 1**: Consider state machines for state-dependent objects with complex behavior.
  – Model behavior of complex reactive objects
- **Guideline 2**: Complex state-dependent classes are less common for business information systems, and more common in communications and control applications.
- Use Enum to implement

**Transition Actions and Guards**

- A transition action is an action (an object does something) when a transition happens
  – In a software implementation, this may represent the invocation of a method of the class of the state machine diagram.
- A guard is a pre-condition to a transition, i.e., a transition won't happen if the guard condition is false.



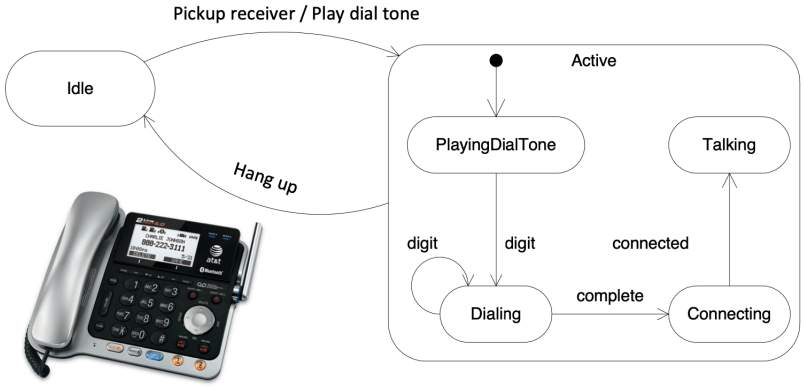

When Browser is *Displaying*:
  if *left-mouse-down(coordinates)* occurs and
    in(*coordinates, active_window)*
  then
    *link=select-link(coordinates); link.follow();*
    transition Browser to *Retrieving;*



Transitions represent events that occur within a system, and are generally one of three types:

1. *External Event*: An event that is caused by something outside our system. For example, receiving touch inputs from a user.
2. *Internal Event*: An event caused by something inside our system. For example, the notification component of the observer pattern.
3. *Temporal Event*: An event caused by a specific date or time, generally controlled by checks against a system clock.

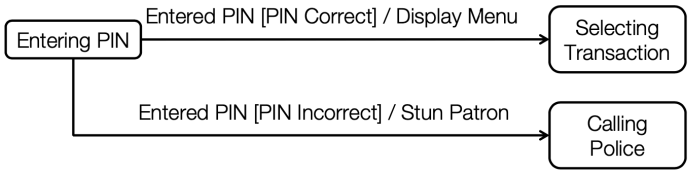
**Nested States**

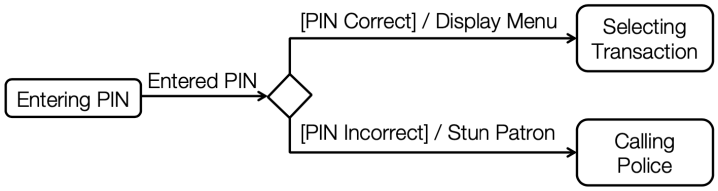



A state allows nesting to contain substates; a substate inherits the transitions of its superstate (the enclosing state)

- Transition into *Active* (via *pickup receiver*) transitions into substate *PlayingDialTone*
- All substates of *Active* inherit the *Hang up* transition.

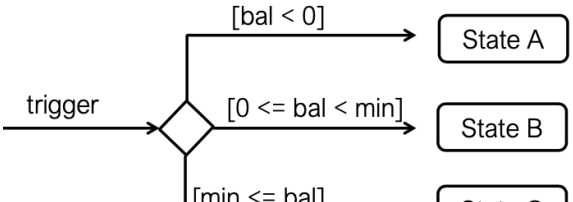
**Choice Pseudostate** realizes a dynamic conditional branch. It evaluates the guards of the triggers of its outgoing transitions to select only one outgoing transition.



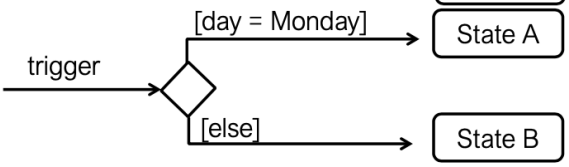




Choice Pseudostate can

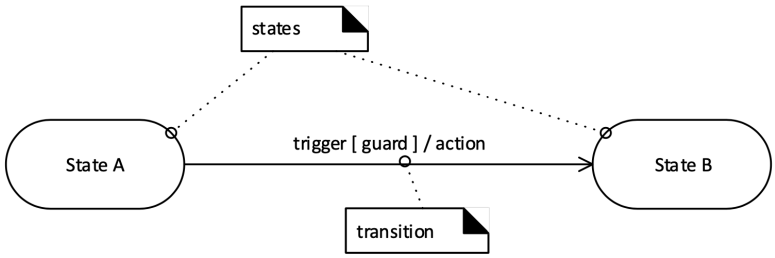
- Have two or more outgoing transitions




- Use predefined [else] guard
  – [else] outgoing transition chosen if no other guards are true



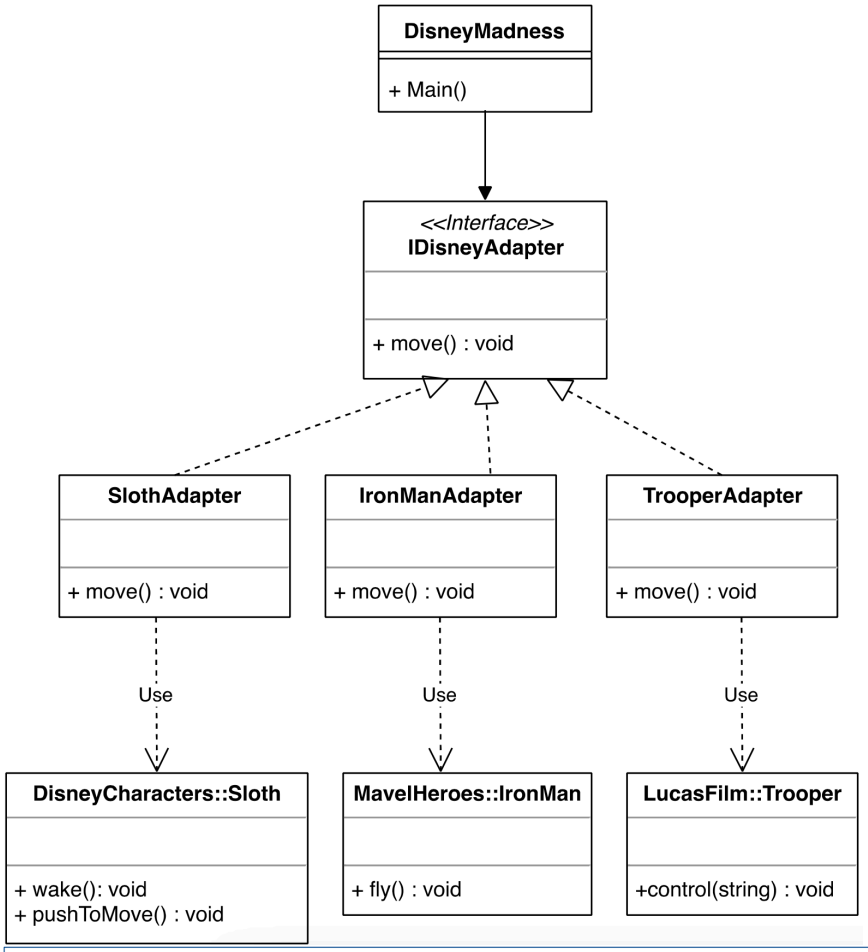

**Notation**




When object is in *State A*:
if *trigger* event occurs and *guard* is true then perform the behaviour *action* and transition object to *State B*.

# Week07

**Adapter (GoF)**
**Problem**: How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces?
**Solution (advice)**: Convert the original interface of a component into another interface, through an intermediate adapter object.
Separation of Concerns/High Cohesion
Creating adapters is not pure application logic
Adapter object is not a domain object
Use GRASP Pure Fabrication pattern



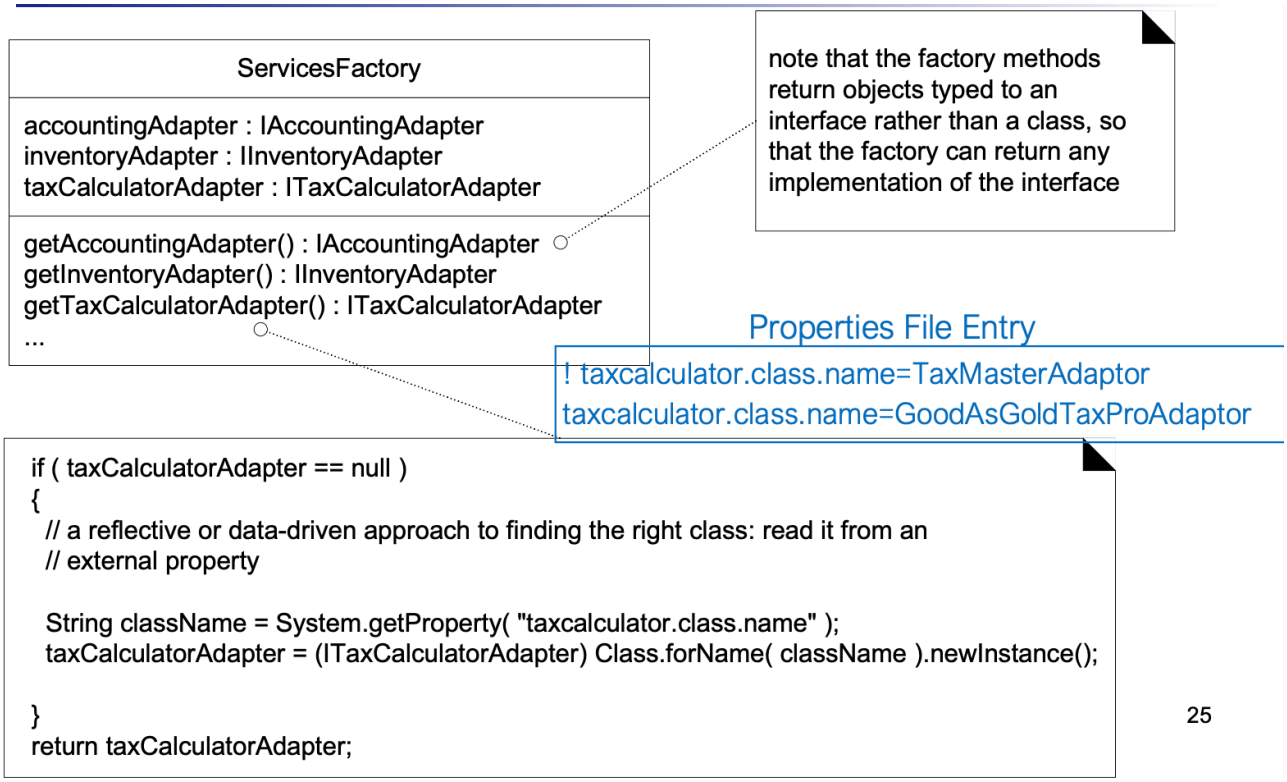

```
public interface IDisneyAdapter {
  public void move();
}
```

```
public class SlothAdapter implements IDisneyAdapter {
  private Sloth theSloth;
  public SlothAdapter(String name) {
    theSloth = new Sloth(name);
  }
  public void move() {
    theSloth.wake();
    theSloth.pushToClimb();
  }
}
```

```
public class DisneyMadess {
  public static void main(String[] args)  {
   //With adapter
   IDisneyAdapter slothAdt = new SlothAdapter("Flash");
   slothAdt.move();

   IDisneyAdapter ironManAdt = new  IronManAdapter("Tony");
   ironManAdt.move();

   IDisneyAdapter trooperAdt = new TrooperAdapter(”Storm");
   trooperAdt.move();
  }
}
```

**Factory**
**Problem**: Who should be responsible for creating objects when there are special considerations, such as complex creation logic, a desire to separate the creation responsibilities for better cohesion, and so forth? E.g. create adapters
**Solution (advice)**: Create a Pure Fabrication object called a Factory that handles the creation.
(Concrete) Factory aka (also known as) **Simple Factory**
Simplified GoF Abstract Factory pattern
**Advantages:**
- Separate responsibility of complex creation into cohesive helper objects.
- Hide potentially complex creation logic.
- Allow introduction of performance-enhancing memory management strategies, such as object caching or recycling.

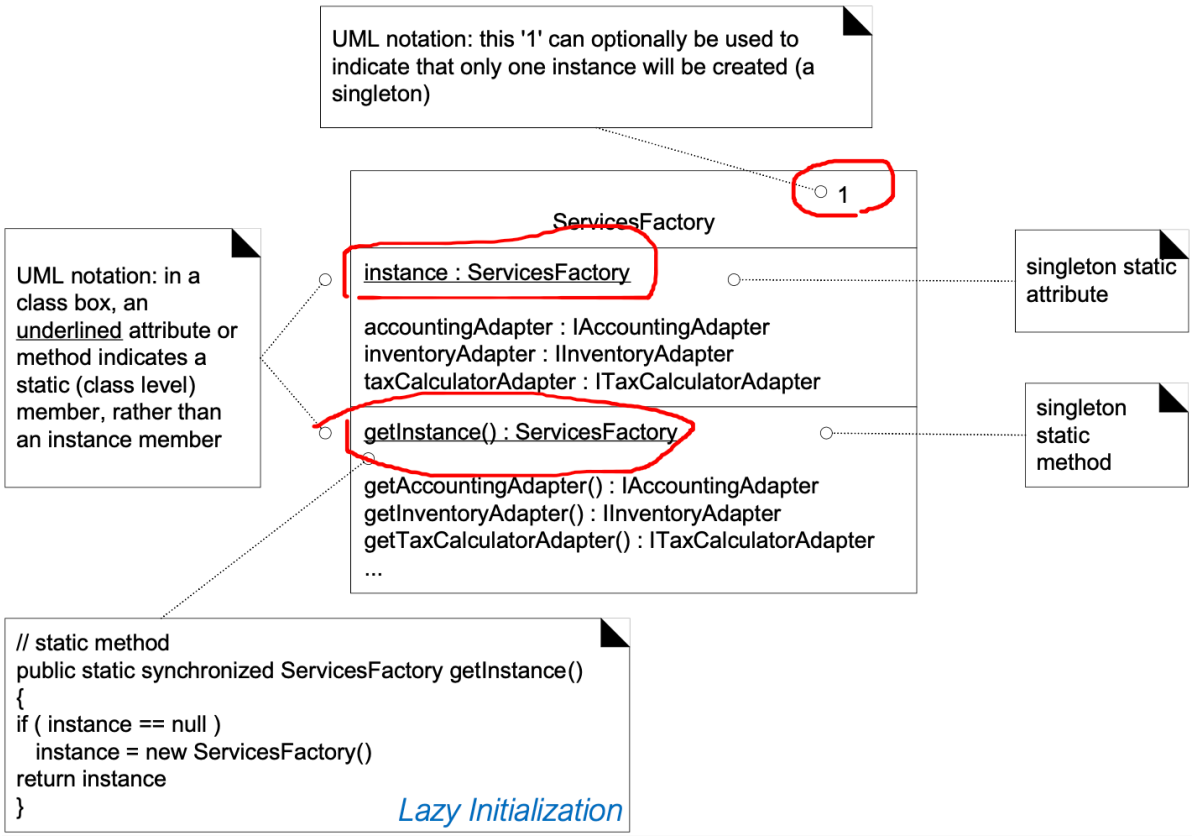
Only one instance of the factory is needed
Where required, pass through to methods or
initialise objects with a ref? No.
Use Singleton pattern to provide a single access point through global visibility



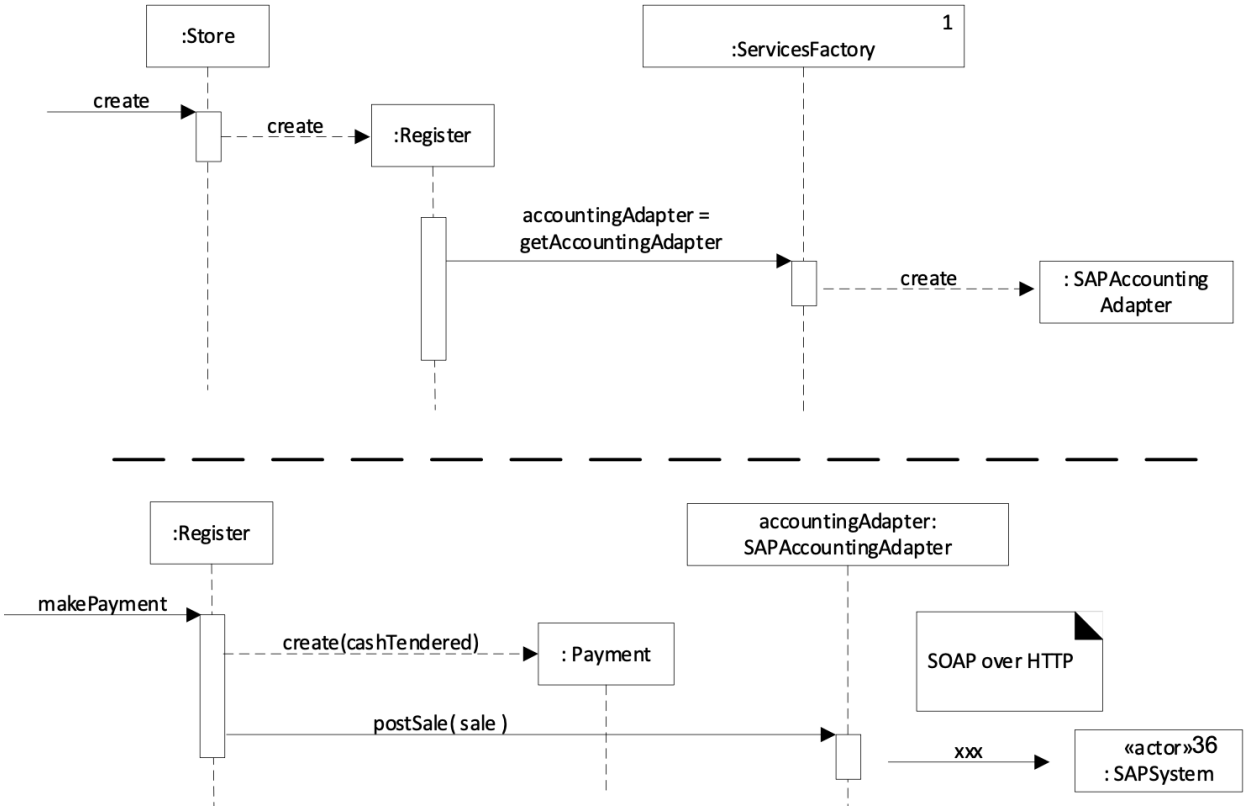

```
public class DisneyFactory {
  private IDisneyAdapter characterAdapter = null;

  public IDisneyAdapter getDisneyAdapter(String  character, String name){
    if(character.equals("ironman")) {
      characterAdapter = (IDisneyAdapter) new IronManAdapter(name);
    }else if(character.equals("sloth")) {
      characterAdapter = (IDisneyAdapter) new SlothAdapter(name);
    }else if(character.equals("trooper")) {
      characterAdapter = (IDisneyAdapter) new TrooperAdapter(name);
    }
    return characterAdapter;
  }
}
```

**Singleton (GoF)**
**Problem**: Exactly one instance of a class is allowed—it is a "singleton." Objects need a global and single point of access.
**Solution (advice)**: Define a static method of the class that returns the singleton.

- May want subclasses: static methods are not polymorphic (virtual); no overriding in most languages
- Most remote communication mechanisms (e.g. Java's RMI) don't support remote-enabling of static methods.
- A class is not always a singleton in all application contexts.




SingletonClass.getInstance() is globally visible
External Services with Varying Interfaces
Possible solution: "To handle this problem, let's use Adapters generated from a Singleton Factory."

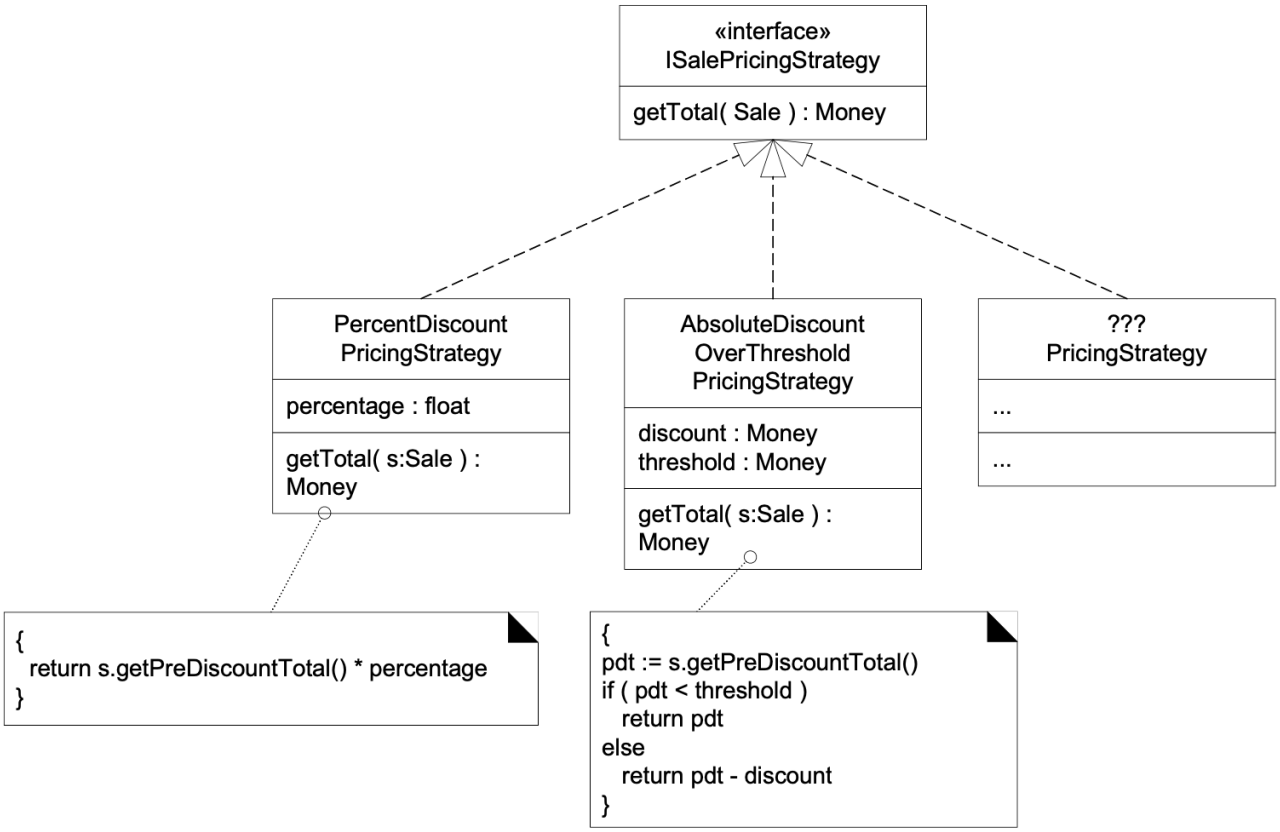


# Week08

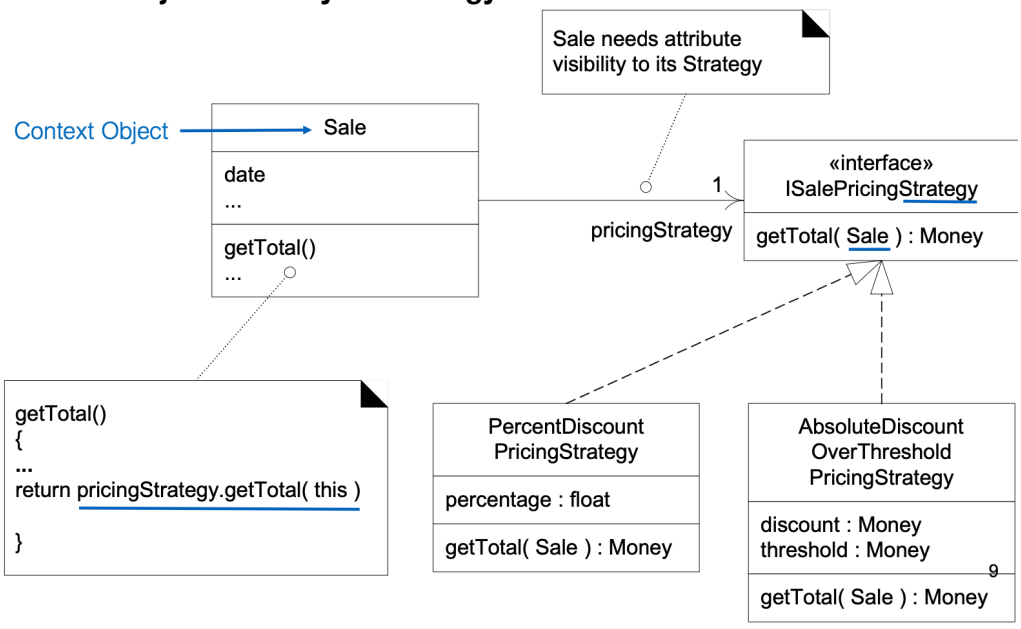
**Strategy (GoF)**
**Problem**:
- How to design for varying, but related, algorithms or policies?
- How to design for the ability to change these algorithms or policies?

**Solution (advice):**
- Define each algorithm/policy/strategy in a separate class, with a common interface.
- Difference with adapter: adapter is structural design pattern and strategy is a behavior pattern

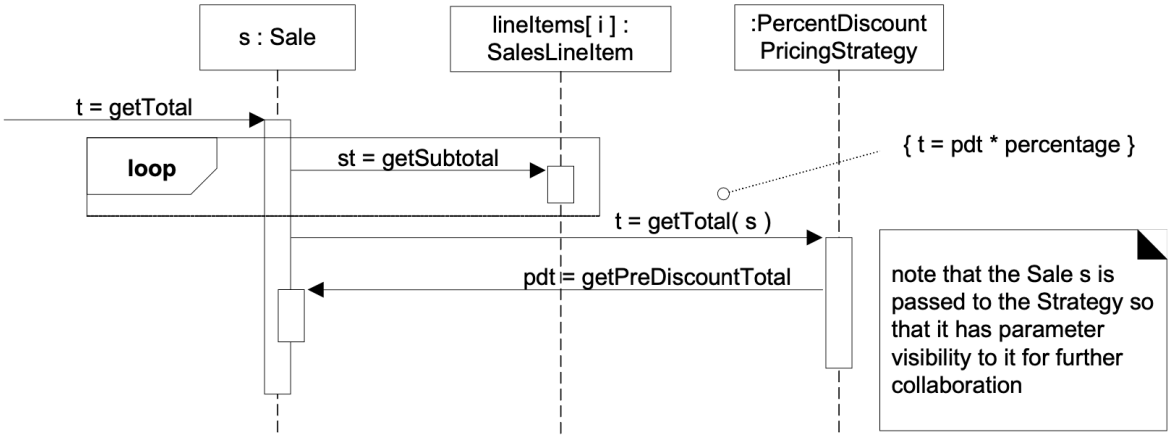
### Pricing Strategy Classes




### Context Object Visibility to Strategy



## Strategy in Collaboration



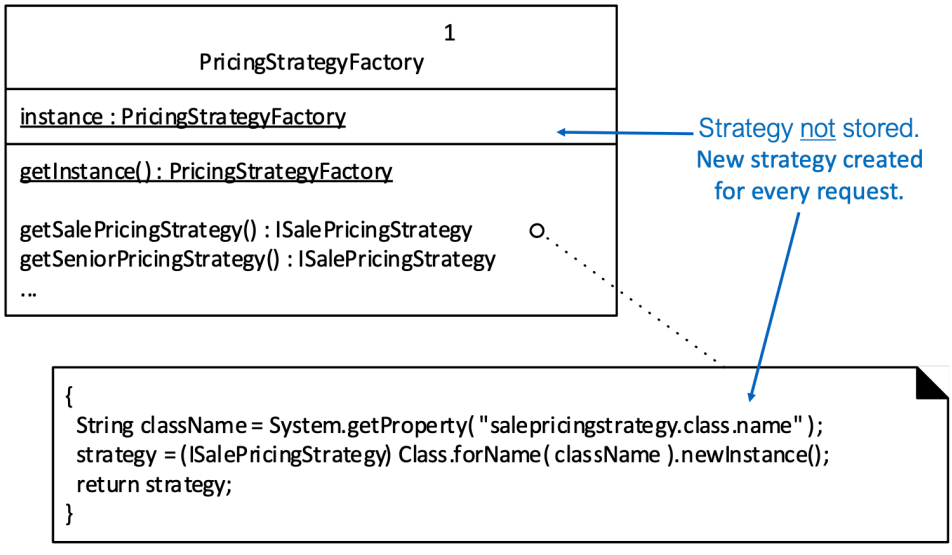

## Code Example

```
public interface ISalePricingStrategy {
        public double getTotal(Sale s);
}
```
**ISalePricingStrategy.java**

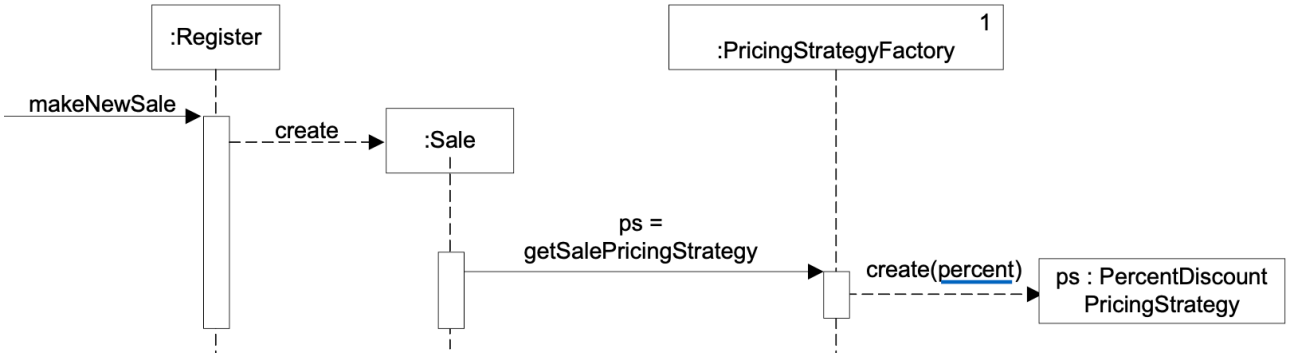
```
public class PercentDiscountPricingStrategy implements ISalePricingStrategy{
        private double percentage = 0.05;
        public double getTotal(Sale s) {
                return s.getPreDiscountTotal() -
        (s.getPreDiscountTotal() * percentage);
}}
```
**PercentDiscountPricingStrategy.java**

```
public class AbsoluteDiscountOverThresholdPricingStrategy implements ISalePricingStrategy {
        private double threshold = 50;
        private double discount = 5;
        public double getTotal(Sale s) {
                double pdt = s.getPreDiscountTotal();
                if(pdt >= threshold) {
                        return pdt - discount;
                }else {
                        return pdt;
        }
}}
```
**AbsoluteDiscountOverThethresholdPricingStrategy.java**

## Creation: Factory for Strategies



## Creating a Strategy




## Combining Policies

- conflict resolution strategy:
- when multiple policies are applicable, how are these policies resolved?
  - Some discounts cannot be combined with others
  - Possible policies: Best for customer or Best for store
- Composite pricing strategy:
  - Determine which pricing strategies are applicable
  - Apply the relevant conflict resolution strategy

## Composite (GoF)

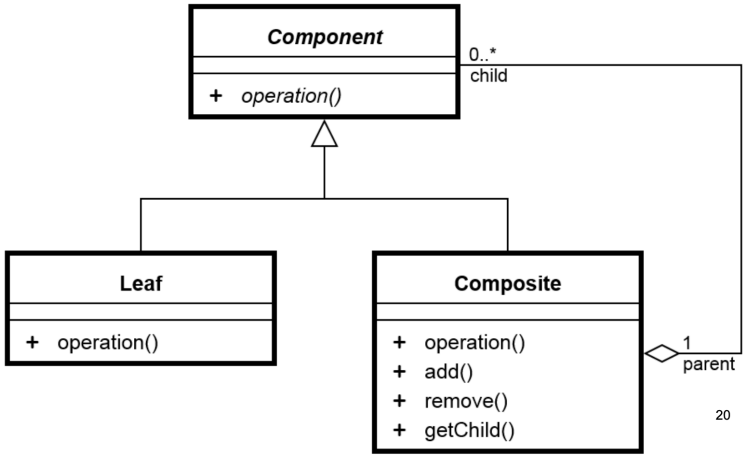
Problem:

- How to treat a group or composition structure of objects the same way (polymorphically) as a non-composite (atomic) object?

Solution (advice):

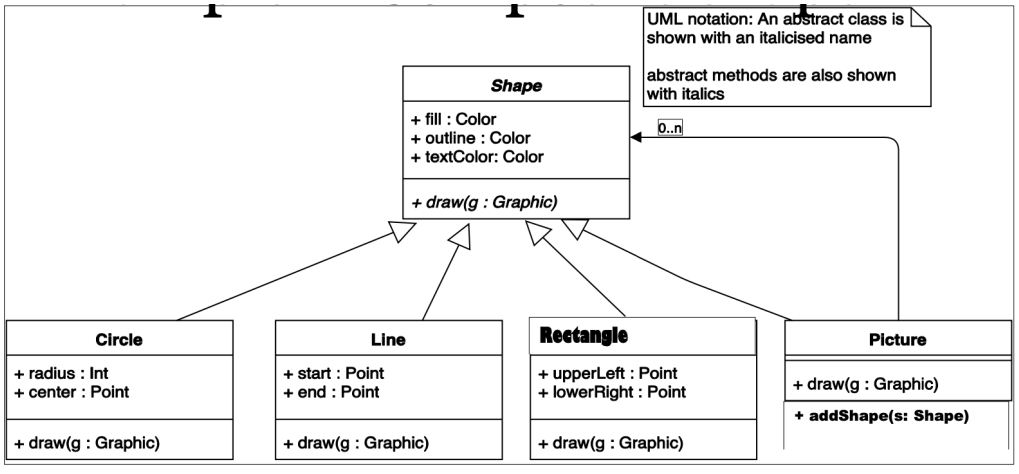
- Define classes for composite and atomic objects so that they implement the same interface.

## Composite: Generalised Structure




20

## Example




```
Circle.draw(g)   Line.draw(g)   Rectangle.draw(g)   Picture.draw(g)
```

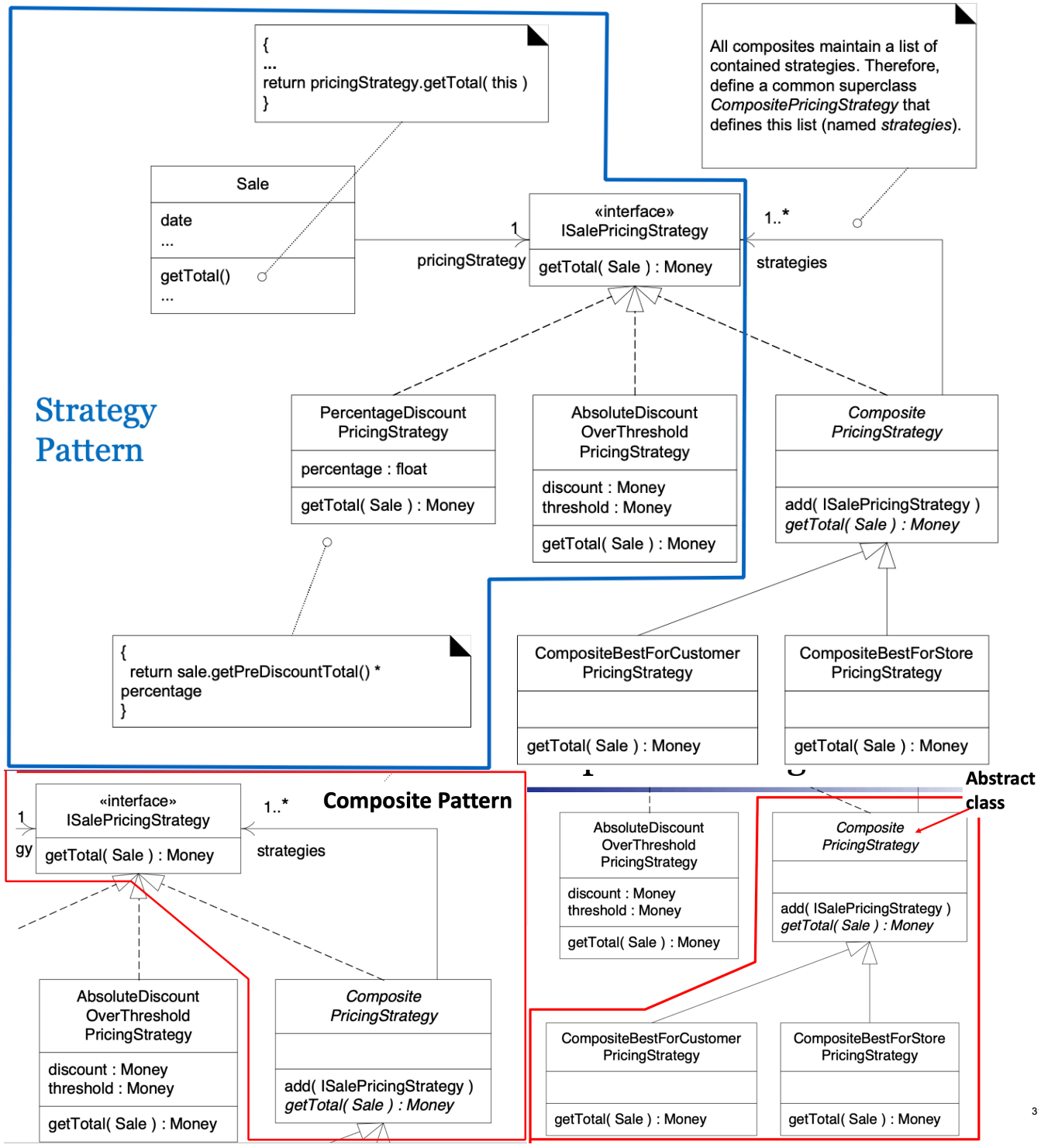
21

## Example code

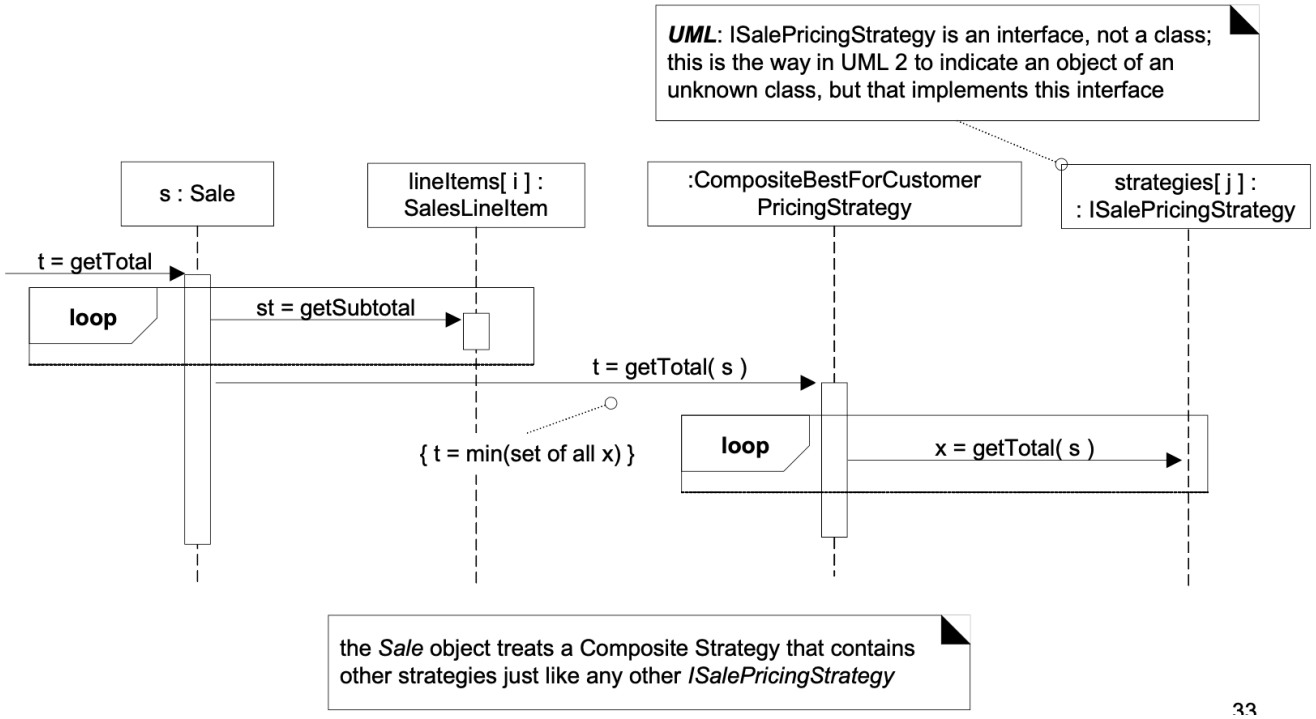
```
public class CompositeDrawing {
  public static void main(String[] args) {
  Circle circle1 = new Circle(5,new Point(0,0));
  Rectangle rectangle1 = new Rectangle(new Point(0,10),new Point(10,20));
  Line line1 = new Line(new Point(5,10),new Point(0,10));
  Picture myPicture = new Picture();

  myPicture.addShape(circle1);
  myPicture.addShape(rectangle1);
  myPicture.addShape(line1);
  myPicture.draw();
 }
}
```

```
public class Picture extends Shape{
  ArrayList<Shape> shapes;
  public Picture() {
    shapes = new ArrayList<Shape>();
  }

  public void addShape(Shape shape) {
   shapes.add(shape);
  }

  public void draw() {
   for(Shape s: shapes) {
     s.draw();
   }
  }
}
```

## Composite Strategies

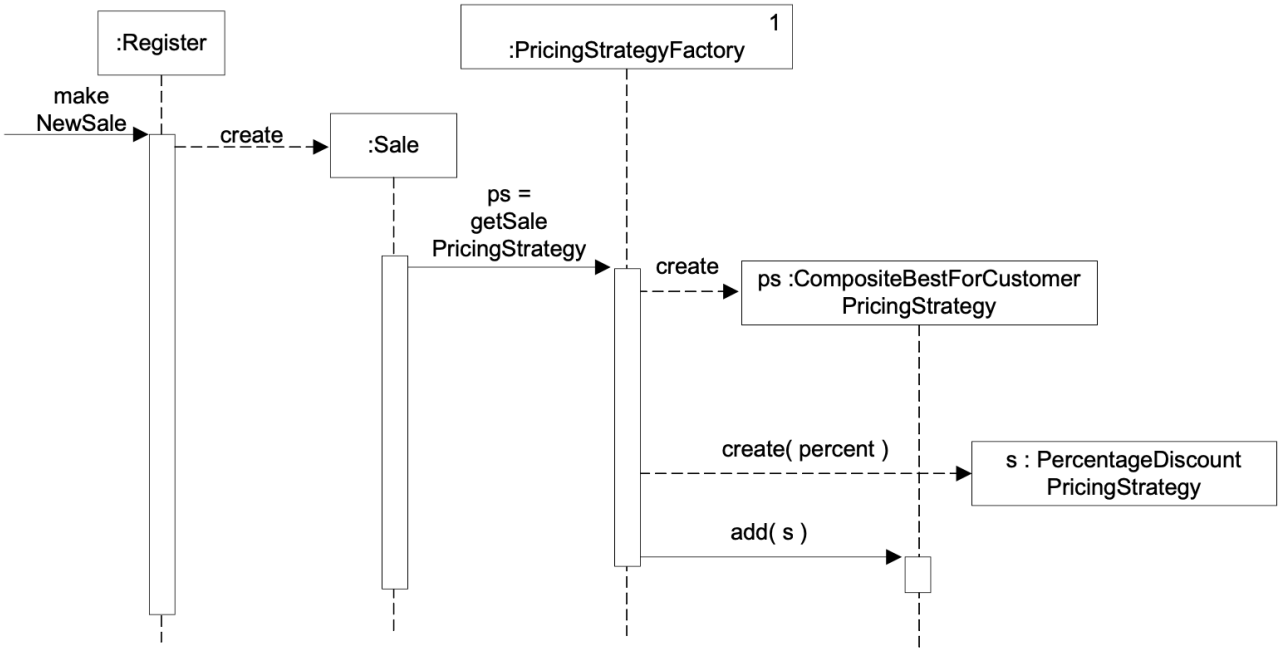


## Collaboration with a Composite






## Create a Composite Strategy



## Code example

```
public interface ISalePricingStrategy {
        public double getTotal(Sale s);
        public String getStrategyName();
}
```
ISalePricingStrategy.java

```
abstract class CompositePricingStrategy implements ISalePricingStrategy {

        protected ArrayList<ISalePricingStrategy> pricingStrategies = new
ArrayList<ISalePricingStrategy>();

        public void add(ISalePricingStrategy strategy) {
        pricingStrategies.add(strategy);
        }

        public abstract double getTotal(Sale s);
}
```
CompositePricingStrategy.java

35

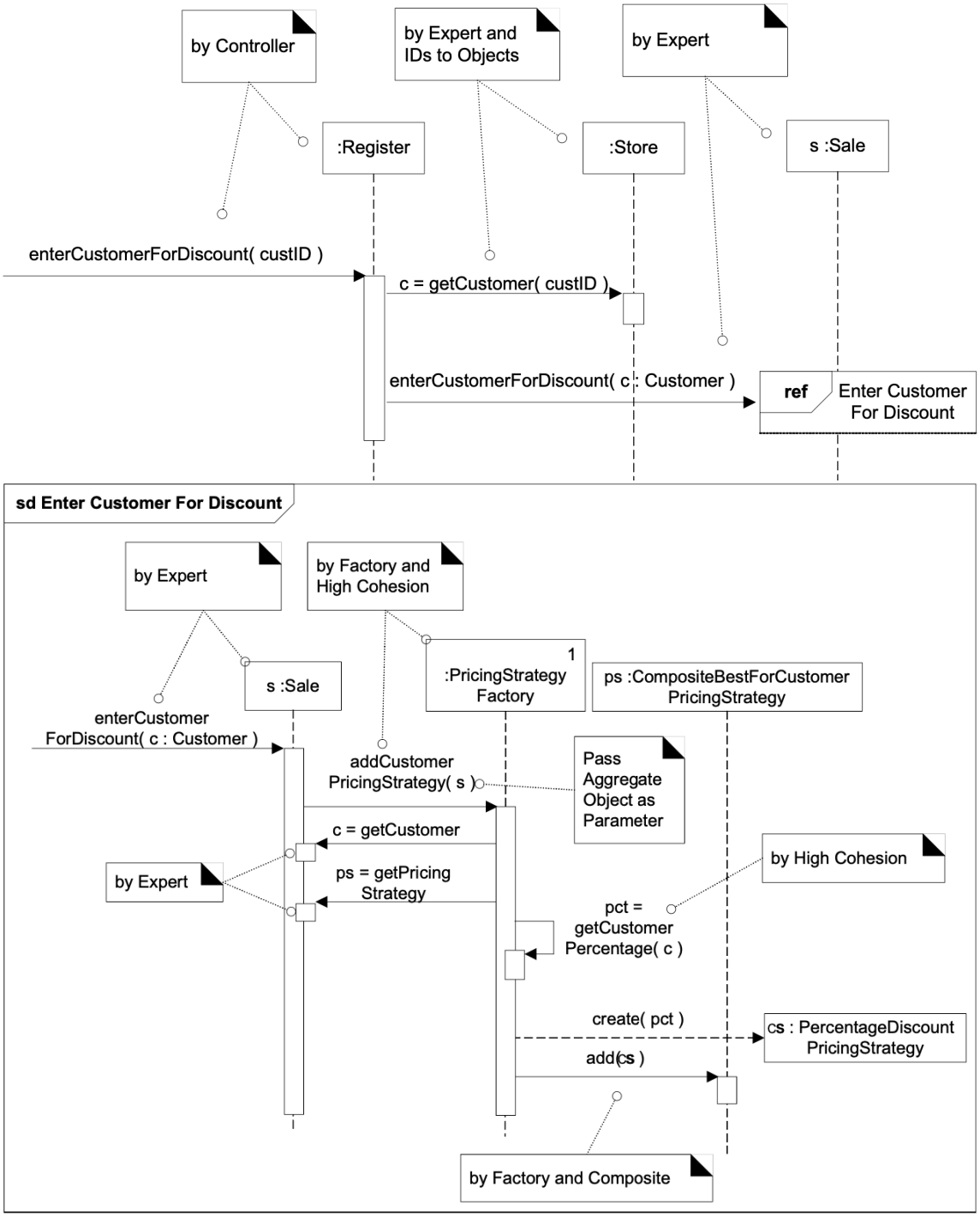
```
public class CompositeBestForCustomerPricingStrategy extends CompositePricingStrategy {
 private String selectedStrategy = null;
 //Get minimum total
 public double getTotal(Sale s) {
   double lowestTotal = s.getPreDiscountTotal();

   for(ISalePricingStrategy strat: this.pricingStrategies){
            double total = strat.getTotal(s);
            if(lowestTotal > total) {
                    lowestTotal = total;
            }
           }
           return lowestTotal;
 }
}
```

36

```
public class PricingStrategyFactory {
 public ISalePricingStrategy getCompositeSalePricingStrategy() {
  ISalePricingStrategy applicableStrategies = null;
  LocalDateTime today = LocalDateTime.now();
  switch(today.getDayOfWeek()) {
   case MONDAY:
     CompositePricingStrategy composStrat = new
             CompositeBestForCustomerPricingStrategy();
     composStrat.add(new PercentDiscountPricingStrategy());
     composStrat.add(new TeaDiscountPricingStrategy());
     applicableStrategies = composStrat;
     break;
    case FRIDAY:
      applicableStrategies = new
        AbsoluteDiscountOverThresholdPricingStrategy();
      break;
}
return applicableStrategies;
 }
}
```

37

**Extension (or Alternative Flows)**
Customer says they are eligible for a discount (e.g., employee, preferred customer)
- Cashier signals discount request.
- Cashier enters Customer identification.
- System presents discount total, based on discount rules.




**Summary: Complex Pricing Logic**
POS has various pricing strategies/discounts and some of them cannot be combined
Solution:
“To handle this problem, let’s use Composite Strategy”
Design reasoning based on: Protected Variation, Polymorphism, High Cohesion, Low Coupling
Strategy, Composite, Factory, Singleton

# Week09

Problem 3: Pluggable Business Rules

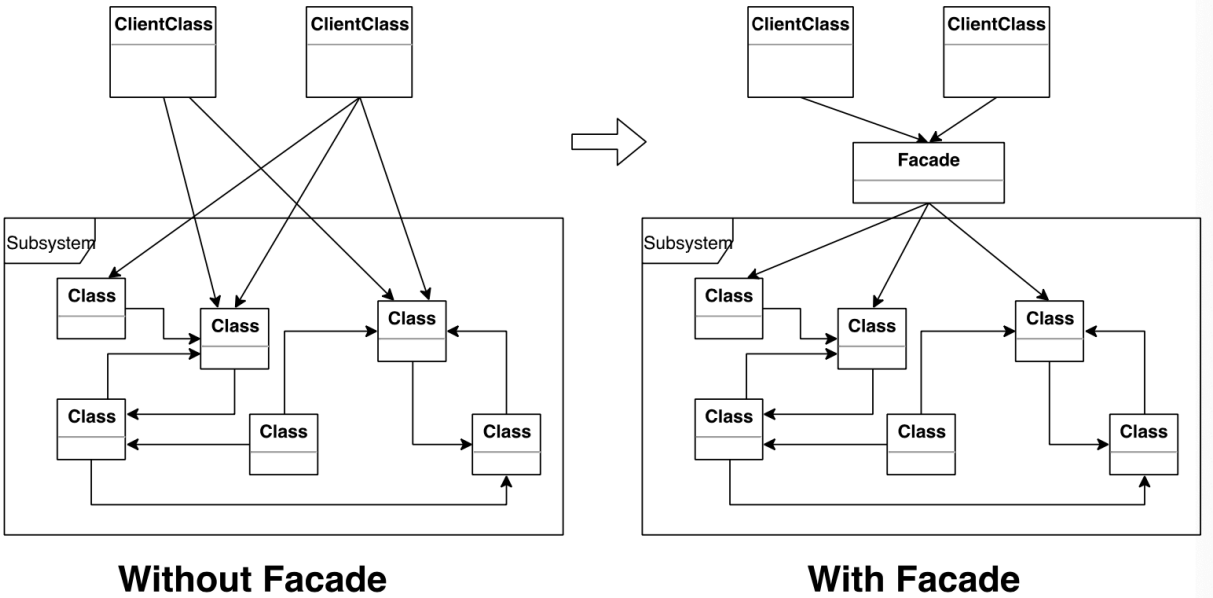
- Different companies who wish to purchase the NextGen POS would like to customise its behaviour slightly.
- E.g. to invalidate an action:
  - Paid by a Gift card
    - Allow one item to be purchased -> Invalidate subsequent EnterItem operations
    - Invalidate request for change as cash or store a/c credit.
  - Charitable donation (by store) sale
    - If cashier is not manager, CreateCharitableSale invalidated
    - EnterItem operation for items over $250 invalidated.

**Analysis**

- This customization should have **low impact** on the existing software components
- A “rule engine” subsystem, whose specific implementation is not yet known. It may be implemented with
- the Strategy pattern; or
- Free open-source rule interpreters; or
- Commercial rule interpreters; or
- Other solutions.

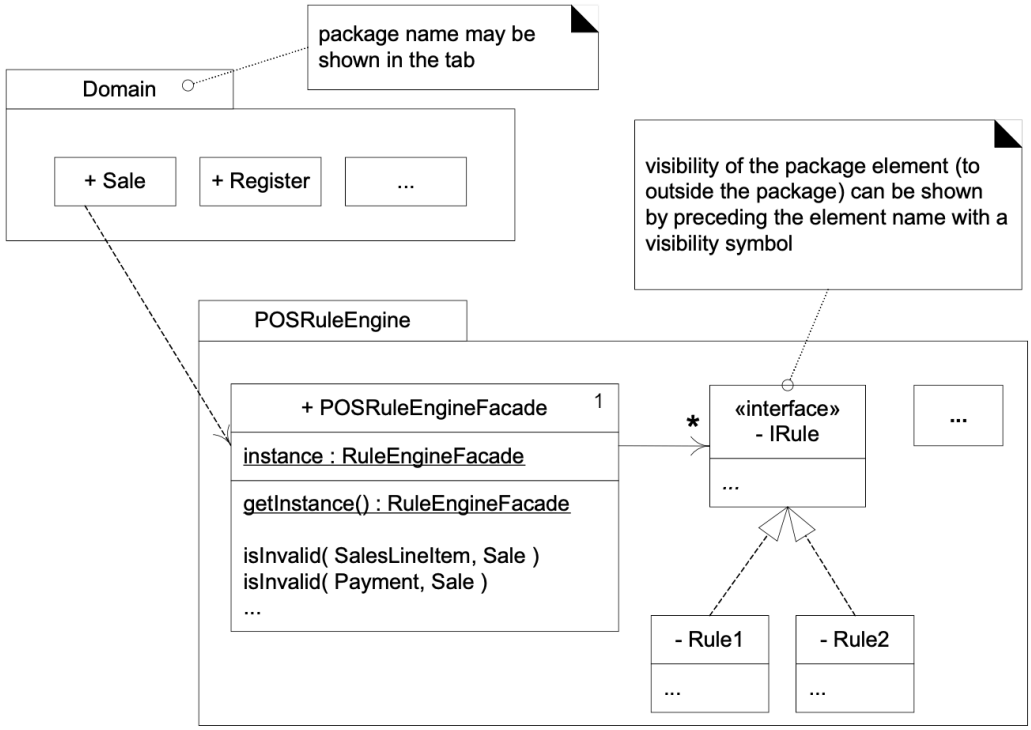
**Façade (GoF)**

- **Problem**:
- Require a common, unified interface to a disparate set of implementations or interfaces—such as within a subsystem—is required. There may be undesirable coupling to many objects in the subsystem, or the implementation of the subsystem may change. What to do?
- **Solution (advice)**:
- Define a single point of contact to the subsystem—a facade object that wraps the subsystem. This facade object presents a single unified interface and is responsible for collaborating with the subsystem components.




**Without Facade** **With Facade**

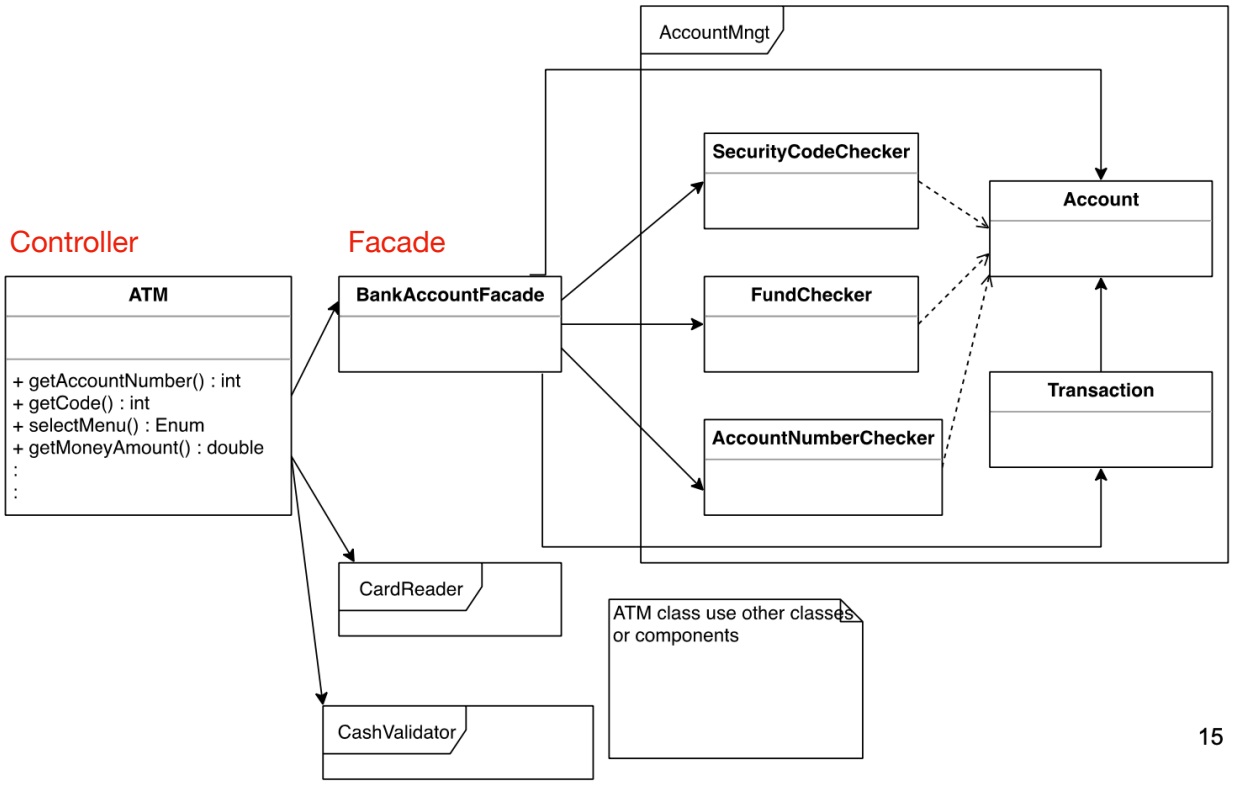
## UML Package Diagram: Facade




## Code example

```
public class Sale {
   public void makeLineItem( ProductDescription desc, int quantity )
   {
      SalesLineItem sli = new SalesLineItem( desc, quantity );
      // call to the Facade
      if ( POSRuleEngineFacade.getInstance().isInvalid( sli, this ) )
         return;
      lineItems.add( sli );
   }
// ...
} // end of class
```

## ATM with Facade






## Example code

```
BankAccountFacade accessingAccount = new
                BankAccountFacade(accountNumber, code);
boolean end = false;
while(!end) {
    switch(selectMenu()) {
    case Deposit:
       double cashToDeposit = getMoneyAmount();
       accessingAccount.depositCash(cashToDeposit);
       break;
    case Withdraw:
       double cashToGet = getMoneyAmount();
       accessingAccount.withdrawCash(cashToGet);
       break;
    case Exit:
       end = true;
       break;
     }
}
```

```
public class BankAccountFacade {
  private int accountNumber;
  private int securityCode;
  private AccountNumberCheck acctChecker;
  private SecurityCodeCheck codeChecker;
  private FundsCheck fundChecker;
  private Account account;
  private Transaction transaction;

  public BankAccountFacade(int newAcctNum, int newSecCode){
     //Instantiate related objects
  }
  public void withdrawCash(double cashToGet){
   //Check account number, security code, available fund
   //If all is valid, decrease cash in account
  }
  public void depositCash(double cashToDeposit){
    //Check account number, security code
    //If all is valid, increase cash in account
  }
}
```

**Façade VS GRASP Controller**
Façade provides a simpler interface of a complex subsystem for a client class
Controller handles user inputs based on business logics and workflow

**Façade VS Adapter**
Façade wraps access to a subsystem or system with a single object
Adaptor wraps each API with the required mapping to provide a single interface

Problem 4: Dynamic Behavior at Run-time

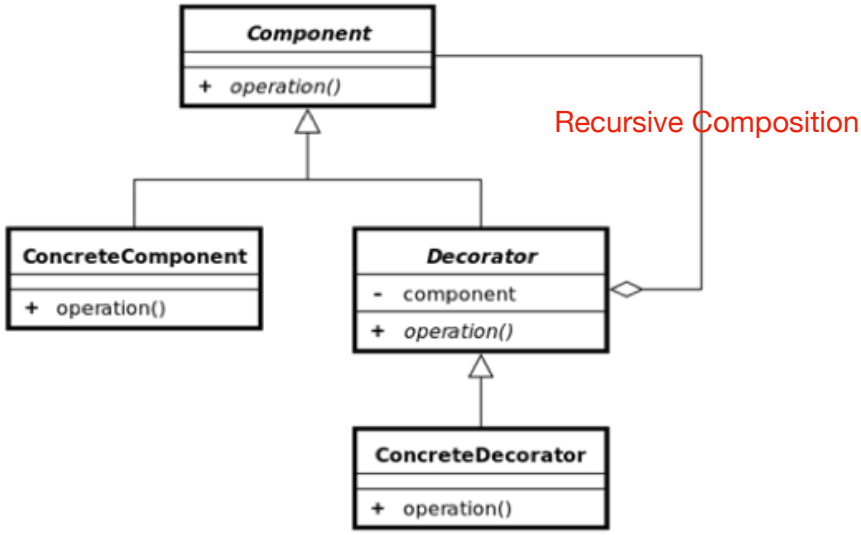
- You want to add behavior or state to specific individual objects at run-time.
  - Irrespective of combination of number of features or behaviours added, you don't want to change the interface presented to the client.
  - Inheritance is not feasible because it is static and applies to an entire class.

**Decorator (GoF)**

- **Problem**:
  - How to dynamically add behaviour or state to individual objects at run-time without changing the interface presented to the client. **What to do?**
- **Solution (advice)**:
  - Encapsulate the original concrete object inside an abstract wrapper interface. Then, let the **decorators** that contain the dynamic behaviours also inherit from this abstract interface. The interface will then use recursive composition to allow an unlimited number of decorator "layers" to be added to each core object.

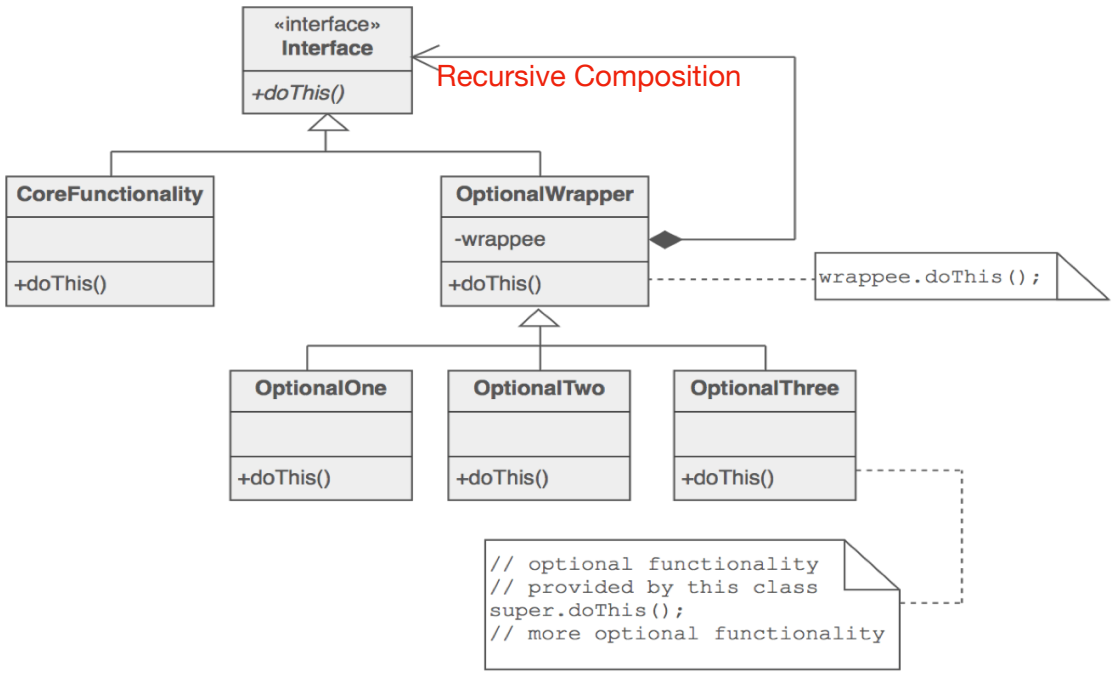
**Decorator: General Form**

A client is always interested in ConcreteComponent.operation() but may or may not be interested in ConcreteDecorator.operation(). More optional ConcreteDecotrators can be added to provide more behaviours.

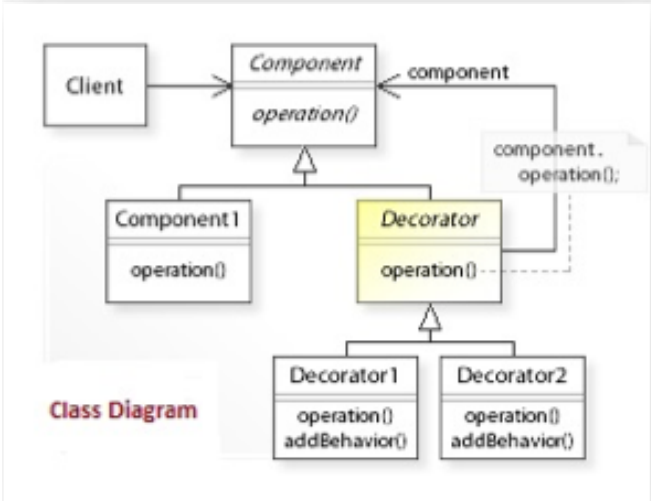


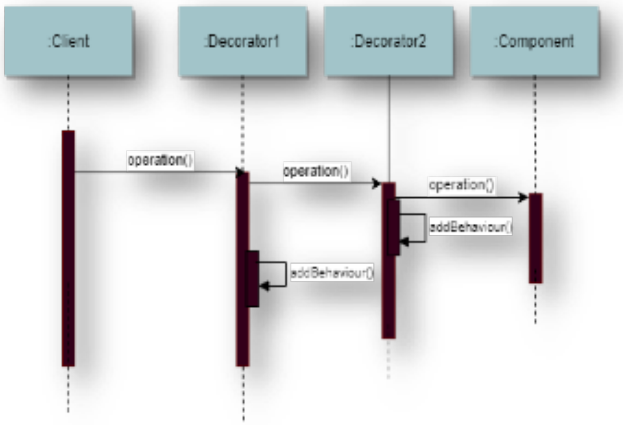

A client is always interested in ConcreteComponent.operation() but may or may not be interested in ConcreteDecorator.operation(). More optional ConcreteDecotrators can be added to provide more behaviours.

**Decorator: Multiple "options"**

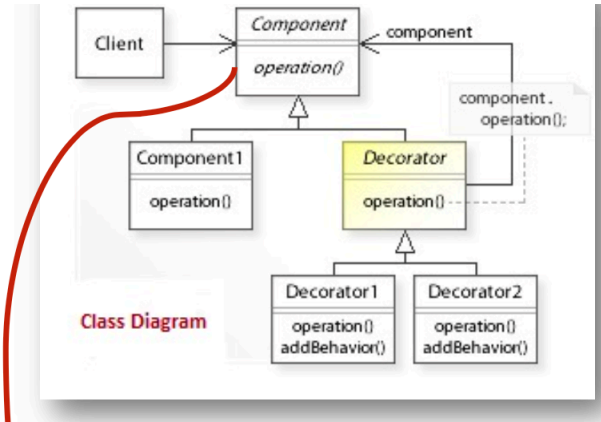


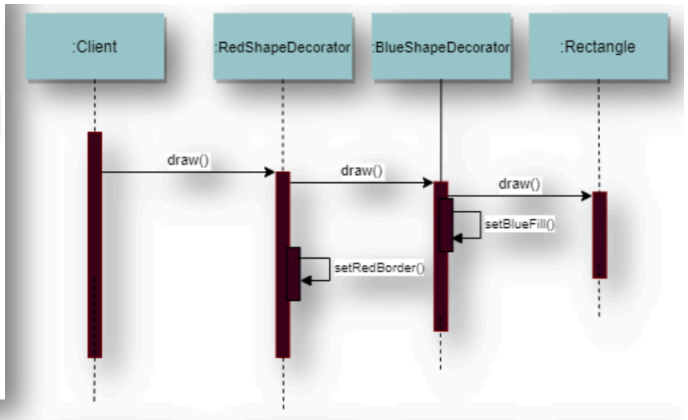
## Collaboration and Sequencing

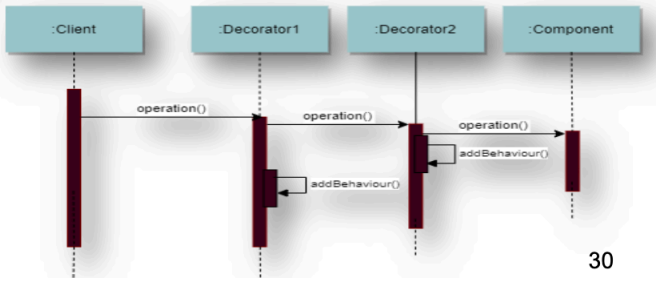



## Shape example



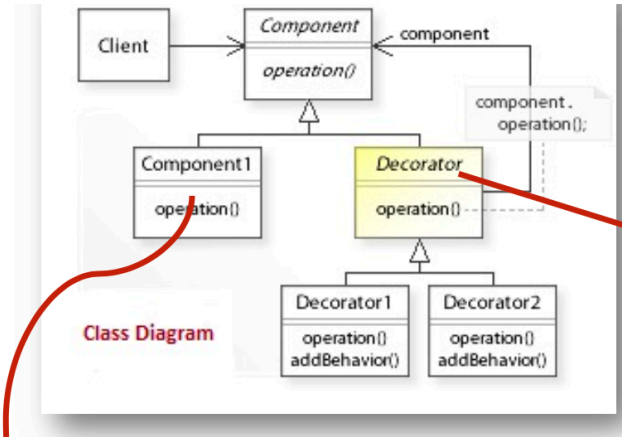



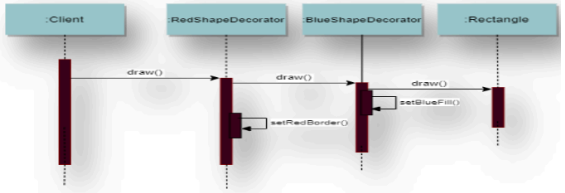




*Shape.java*

```
public interface Shape {
   void draw();
}
```





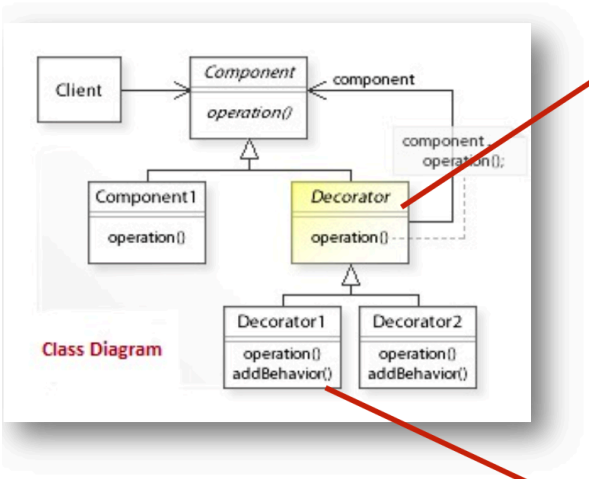




*Rectangle.java*

```
public class Rectangle implements Shape {

   @Override
   public void draw() {
      System.out.println("Shape: Rectangle");
   }
}
```

*ShapeDecorator.java*

```
public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }

   public void draw(){
      decoratedShape.draw();
   }
}
```

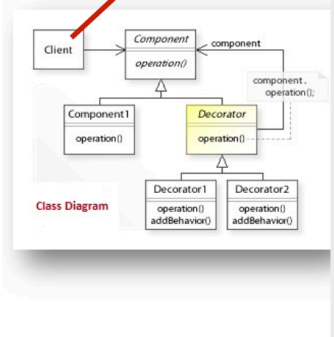



*ShapeDecorator.java*

```java
public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }

   public void draw(){
      decoratedShape.draw();
   }
}
```

*RedShapeDecorator.java*

```java
public class RedShapeDecorator extends ShapeDecorator {

   public RedShapeDecorator(Shape decoratedShape) {
      super(decoratedShape);
   }

   @Override
   public void draw() {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }

   private void setRedBorder(Shape decoratedShape){
      System.out.println("Border Color: Red");
   }
}
```




```java
public class DecoratorPatternDemo {
   public static void main(String[] args) {

      Shape circle = new Circle();

      Shape redCircle = new RedShapeDecorator(new Circle());

      Shape redRectangle = new RedShapeDecorator(new Rectangle());
      System.out.println("Circle with normal border");
      circle.draw();

      System.out.println("\nCircle of red border");
      redCircle.draw();

      System.out.println("\nRectangle of red border");
      redRectangle.draw();
   }
}
```

**Composite Vs Decorator**

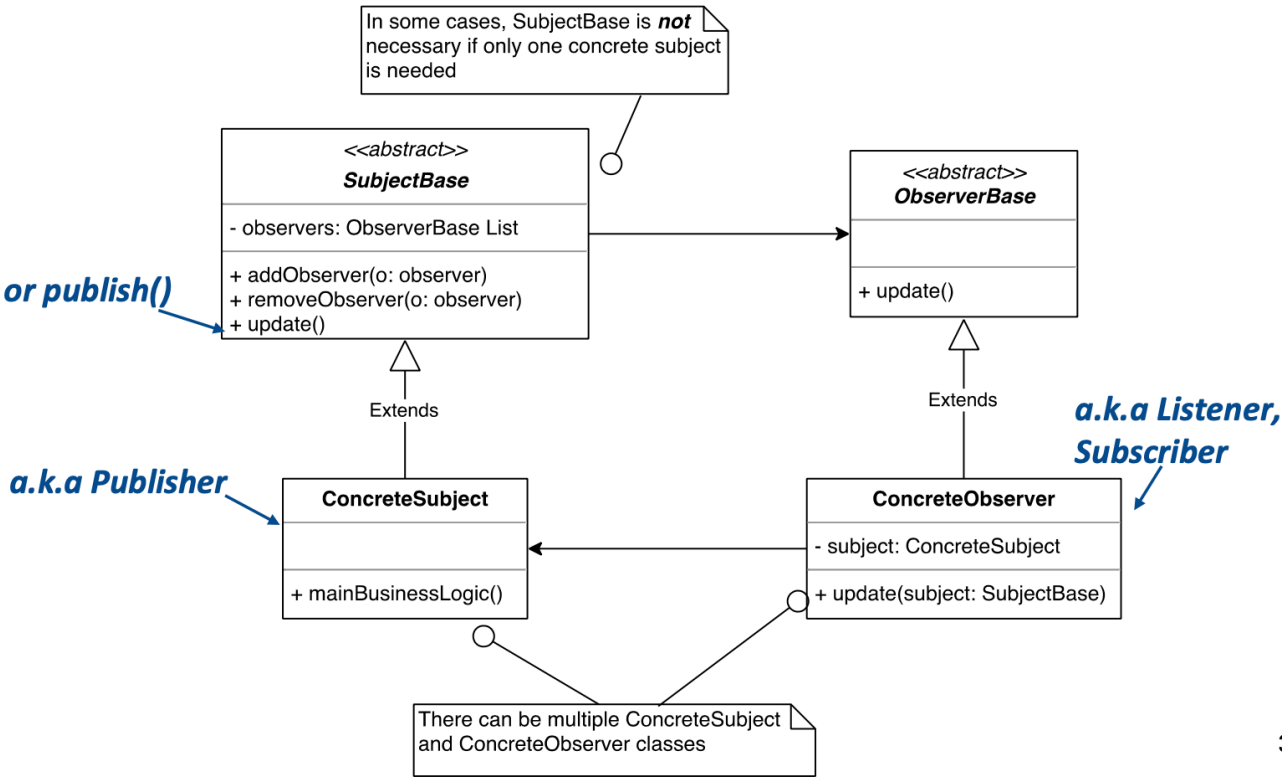
Irrespective of the close similarity in the UML models, Composites and Decorators solve different kinds of problem.

- Decorators decorate one object, which is the primary focus. Zero or more decorators are used as needed to embellish that object.
- Composites exist out of necessity; while a composite needs at least one leaf object, the composite itself is the primary focus.

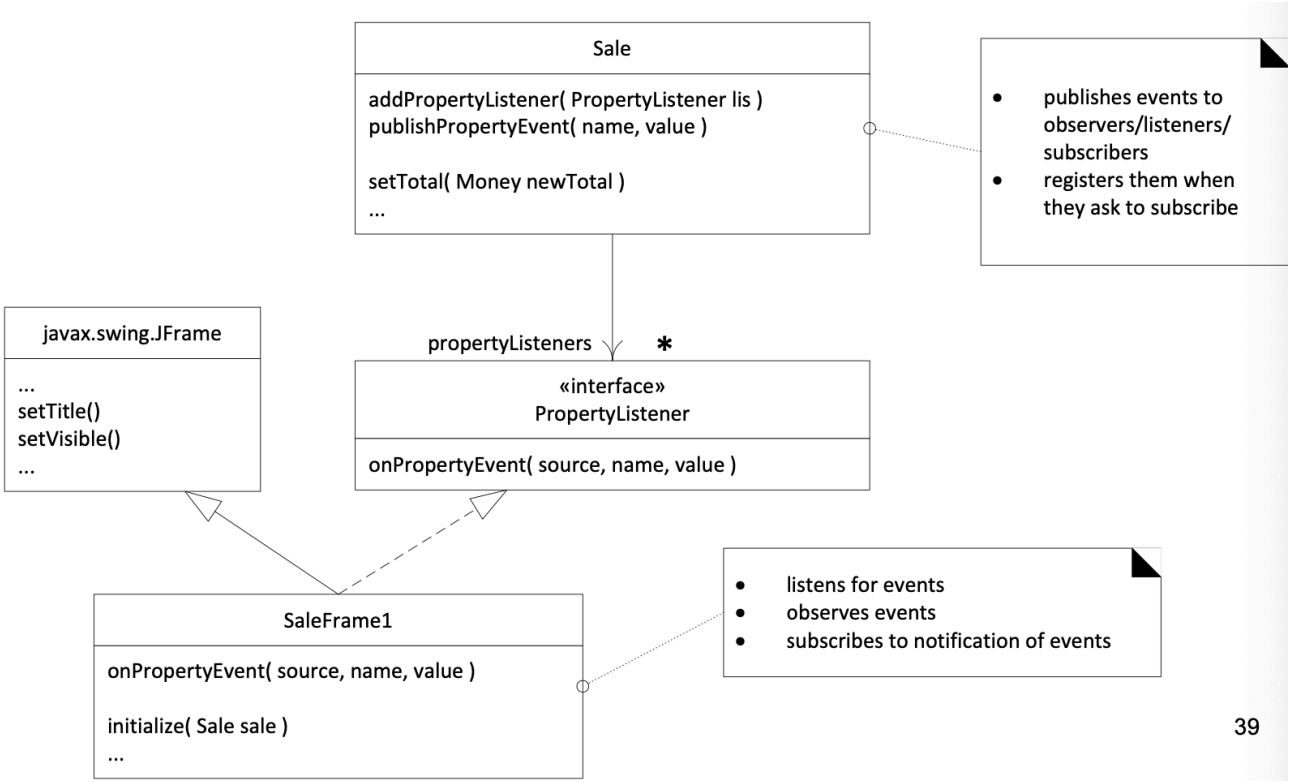
Problem 5: Refreshing Sales Total UI
- a GUI window refreshes its display of the sale total when the total changes
- Sale object should not send a message to a window, asking it to refresh its display
  - Low coupling from other layers to the UI layer

**Observer (aka Publish-Subscribe) (GoF)**
- Problem:
  - Different kinds of subscriber objects are interested in the state changes or events of a publisher object, and want to react in their own unique way when the publisher generates an event. Moreover, the publisher wants to maintain low coupling to the subscribers. What to do?
- Solution (advice):
  - Define a “subscriber” or “listener” interface. Subscribers implement this interface. The publisher can dynamically register subscribers who are interested in an event and notify them when an event occurs.

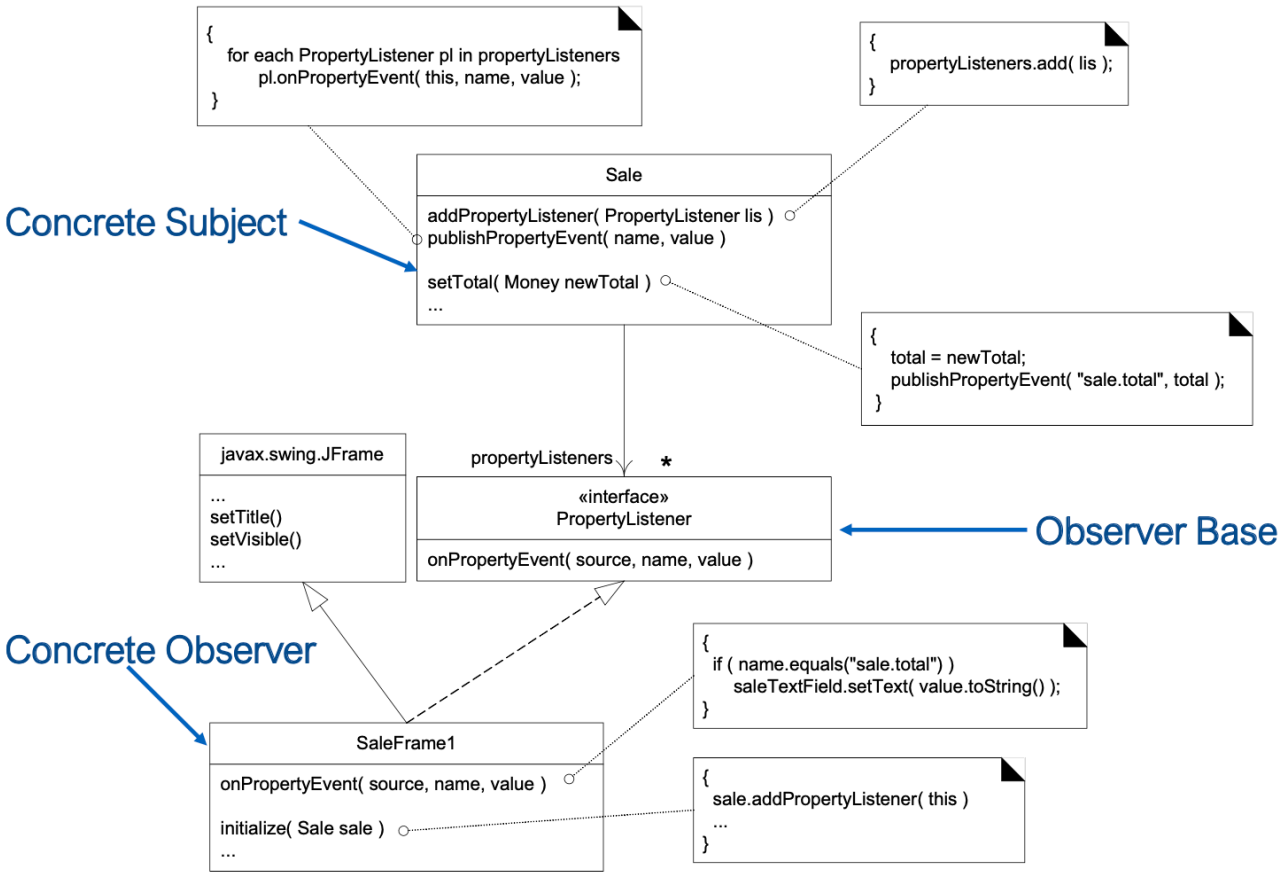
**Observer: Generalised Structure**



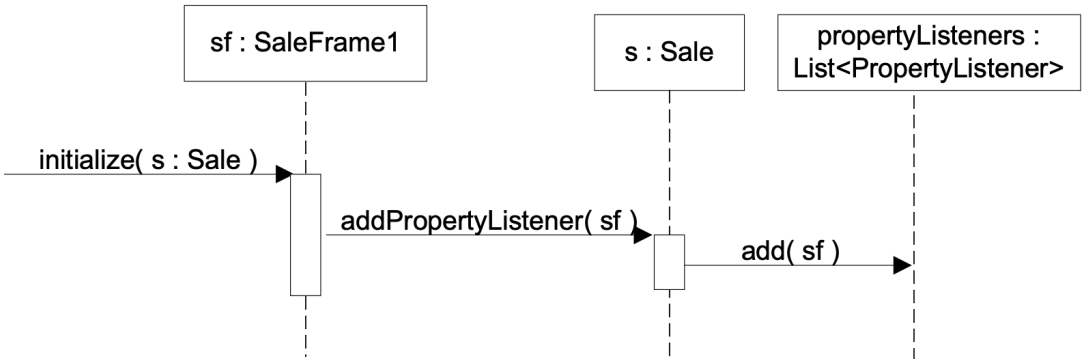
## Sale example





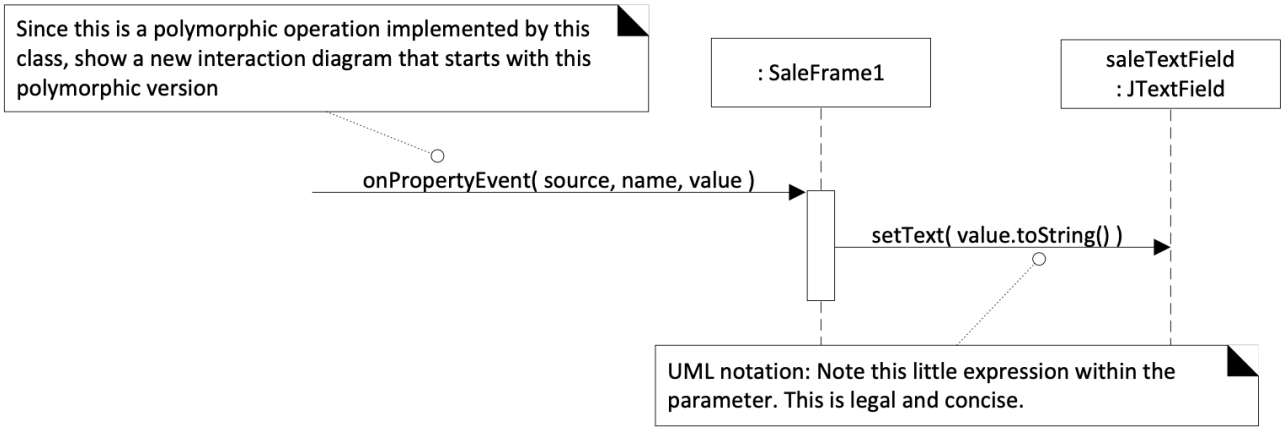



**Ob. *SalesFrame1* Subscribes to Pub. *Sale***




**Sale Publishes Property Event to Subscribers**

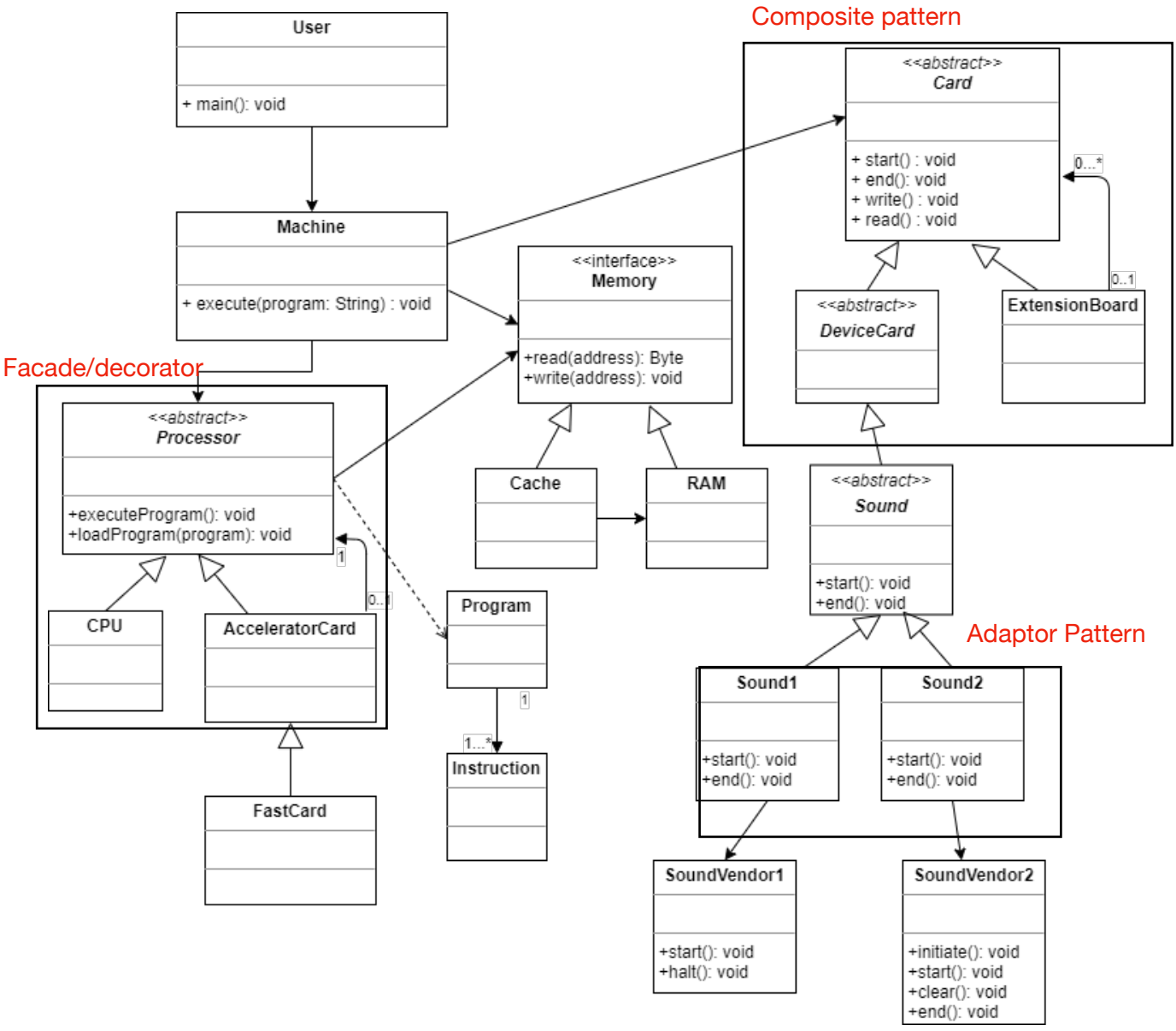

**Subscriber *SaleFrame1* Receives Notification**




**Summary of related GoF Patterns**

- Adapter provides a different interface to its subject. Decorator provides an enhanced interface.
- Composite and Decorator have similar structure, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects.
- Composite, unlike Decorator, is not focused on embellishment but on representation. These intents are distinct but complementary. Consequently, Composite and Decorator are often used in concert.
- Decorator lets you change the skin of an object. Strategy lets you change the guts.

**Exercise**


Composite pattern
User
+ main(): void
Machine
+ execute(program: String) : void
<<interface>>
Memory
+read(address): Byte
+write(address): void
<<abstract>>
Card
+ start() : void
+ end(): void
+ write() : void
+ read() : void
0...*
<<abstract>>
DeviceCard
ExtensionBoard
0..1
Facade/decorator
<<abstract>>
Processor
+executeProgram(): void
+loadProgram(program): void
1
0..1
Cache
RAM
<<abstract>>
Sound
+start(): void
+end(): void
CPU
AcceleratorCard
Program
1
1...*
Instruction
FastCard
Adaptor Pattern
Sound1
+start(): void
+end(): void
Sound2
+start(): void
+end(): void
SoundVendor1
+start(): void
+halt(): void
SoundVendor2
+initiate(): void
+start(): void
+clear(): void
+end(): void

# Week10

**Software Architecture**

**Definition:** Software Architecture is the large-scale organization of elements in a software system.

- A design that concerns whether the system will align with business logic and be easy to maintain and evolve

Software Architecture involves a set of significant ***decisions*** which concern:

- *Structural elements*: How should the software classes/components be in the system?
- *Interfaces*: How should each element is composed altogether?
- *Collaboration*: How should these elements work together according to the business logic?
- *Composition*: How should these elements be grouped progressively into larger subsystems?

**Architectural Analysis**

**Definition:** An activity to identify *factors* that will influence the architecture, understand their variability and priority, and resolve them

• To identify and resolve system's non-functional requirements in the context of its functional requirements

**Goal:** To reduce risk of missing a critical factor in the *design* of a system, focuses effort on high priority requirements, and aligns the product with business goals

Architectural Analysis includes identifying and analysing:

- *Architecturally significant requirement*: The requirement that can have a large impact on the design (especially when it was not considered at the beginning)
- Variation points
- Potential evolution points

**Examples: Architecturally Significant Functional Requirements**

| Function | Description |
|---|---|
| Auditing | Provide audit trails of system execution. |
| Licensing | Provide services for tracking, acquiring, installing, and monitoring license usage. |
| Localization | Provide facilities for supporting multiple human languages. |
| Mail | Provide services that allow applications to send and receive mail. |
| Online help | Provide online help capability. |
| Printing | Provide facilities for printing. |
| Reporting | Provide reporting facilities. |
| Security | Provide services to protect access to certain resources or information. |
| System management | Provide services that facilitate management of applications in a distributed environment. |
| Workflow | Provide support for moving documents and other work items, including review and approval cycles. |

**Examples: Architecturally Significant Non-Functional Requirements**

- Usability
  – e.g. aesthetics and consistency in the UI.
- Reliability
  – e.g. availability (the amount of system "up time"), accuracy of system calculations, and the system's ability to recover from failure.
- Performance
  – e.g. throughput, response time, recovery time, start-up time, and shutdown time.
  Supportability
- – e.g. testability, adaptability, maintainability, compatibility, configurability, installability, scalability, and localizability.

**Effects of requirements on design**

Examples of how the requirements can affect the design decision.

- Reliability and fault-tolerance requirements:
  – Ex: POS uses remote services (e.g., tax calculator). Will the remote services failover to local services be allowed?
- Adaptability and configurability requirements:
  – Ex: What if we want to sell POS to many retailers but they have variations in business rules. Can the retailers change the rules? Should POS be configurable?
- Brand name and branding requirements:
  – Ex: Retailers want to put their own brand in the POS user interfaces. Can the branding be changed in POS?

The design can be significantly changed based on the answer of this question.

**Common Steps in Architectural Analysis**

*Steps* (occurs in early elaboration):

1. Identify/analyse *architectural factors:* requirements with impact on the architecture (especially non-functional requirements)
   - overlaps with requirements analysis
   - some identified/recorded during inception, now investigated in more detail
2. For the architectural factors, analyse alternatives and create solutions: *architectural decision*
   - e.g. remove requirement; custom solution; stop project; hire expert

**Priorities**

- Inflexible constraints
  – e.g. safety; security, legal compliance
- Business goals
  – e.g. demo for clients: tradeshow in 18 months
  – e.g. competitor-driven window-of-opportunity
- Other goals
  – requirements all relate back to higher-level goals, and eventually to business goals
  – e.g. extendible → new release every 6 months

**Architectural Factor Table**

A documentation that records the influence of the factors, their priorities, and their variability (immediate need for flexibility and future evolution)

1. Factor
2. Measures and quality scenarios
3. Variability (current flexibility and future evolution)
4. Impact of factor (and its variability) on stakeholders, architecture and other factors
5. Priority for success
6. Difficulty or risk

**Example: Architecture Factor Table**
Reliability—Recoverability of POS

| Factor | Recovery from remote service (e.g., Tax Calculator) failure |
|---|---|
| Measures and quality scenarios | When remote service fails, re-establish connectivity with it within 1 min. of its detected re-availability, under normal store load in a production environment. |
| Variability (current flexibility and future evolution) | current flexibility - our SME says local client-side simplified services are acceptable (and desirable) until reconnection is possible.<br><br>evolution - within 2 years, some retailers may be willing to pay for full local replication of remote services (such as the tax calculator). Probability? High. |
| Impact of factor (and its variability) on stakeholders, architecture and other factors | High impact on large-scale design. Retailers really dislike it when remote services fail, as it prevents them from using a POS to make sales. |
| Priority for Success | High |
| Difficulty or Risk | Medium |

**Technical Memo**
A documentation that records alternative solutions, decisions, influential factors, and motivations for the noteworthy issues and decisions
1. Issue
2. Solution Summary
3. Factors
4. Solution
5. Motivation
6. Unresolved Issues
7. Alternatives Considered

**Example: Technical memo**
**Technical Memo: Issue: Reliability—Recovery from Remote Service Failure**
**Factors:** Robust recovery from remote service failure, e.g., tax calculator, inventory
**Solution:** To satisfy the quality scenarios of reconnection with the remote services ASAP, use smart Proxy objects for the services, that on each service call test for remote service reactivation, and redirect to them when possible.
Where possible, offer local implementations of remote services. For example, implementing a small cache to store data (e.g., tax rates)
Achieve protected variation with respect to location of services using an Adapter created in a ServicesFactory.
**Motivation:** Retailers really don't want to stop making sales! Therefore, if the NextGen POS offers this level of reliability and recovery, it will be a very attractive product, as none of our competitors provide this capability.
The small product cache is motivated by very limited client-side resources. The real third-party tax calculator is not replicated on the client primarily because of the higher licensing costs, and configuration efforts (as each calculator installation requires almost weekly adjustments).
This design (Adapter and ServiceFactory) also supports the evolution point of future customers willing and able to permanently replicate services such as the tax calculator to each client terminal.
**Unresolved Issues:** none
**Alternatives Considered:** A “gold level” quality of service agreement with remote credit authorization services to improve reliability. It was available, but much too expensive.

**Summary of Architectural Analysis**

1. Concerns are especially related to non-functional requirements with awareness of business context, but addressing functional requirements and their variability.
2. Concerns involve system-level, large-scale, broad problems, with resolution involving large-scale or fundamental design decisions.
3. Must address interdependencies and trade-offs (e.g. security/performance, or anything/cost)
4. Involves the generation/evaluation of alternatives.

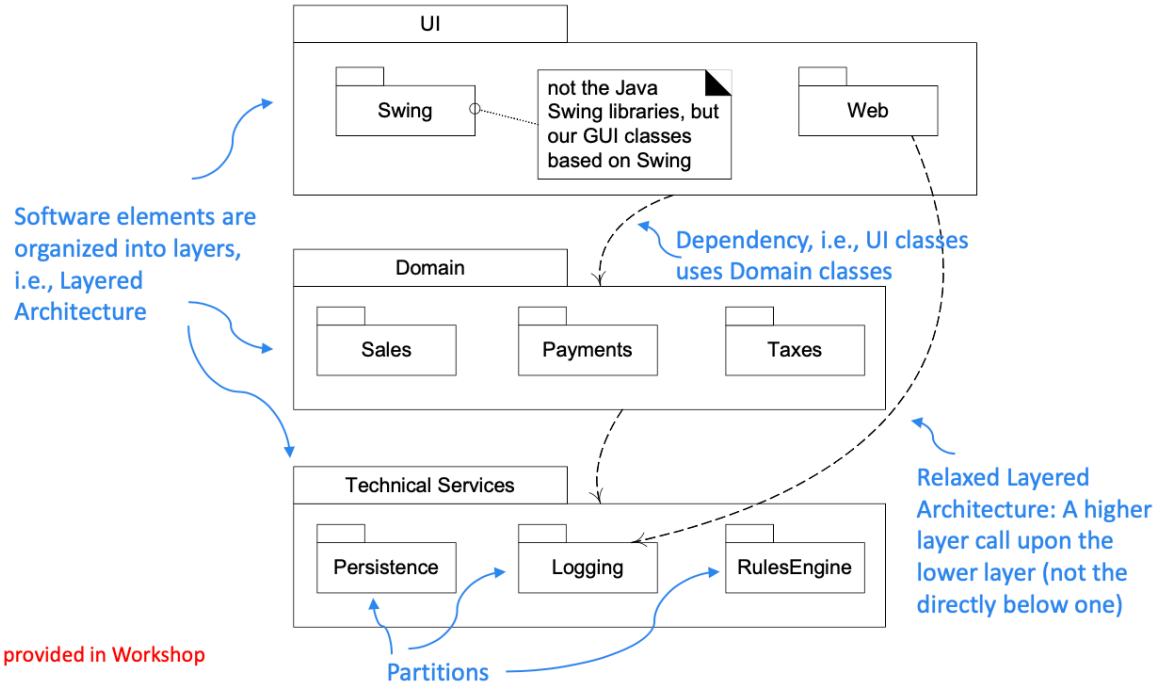
**Logical Architecture (LA)** - Definition:
The large-scale organisation of the software classes into *packages*, *subsystems* and *layers*
– Logical: Not concerned with networking, physical computers, or operating system processes (such concerns are for the *deployment architecture*)
LA defines the packages in which software classes are defined.

**Layered Architecture**
**Layers - Definition**: Coarse-grained grouping of classes, packages, or subsystems that has cohesive responsibility for a major aspect of the system.

- Example layers:
  - **User Interface**.
  - **Application Logic and Domain Objects**—software objects representing domain concepts (Sale class) that fulfill application requirements.
  - Technical Services—general purpose objects and subsystems that provide supporting technical services (e.g., interfaces with DB)

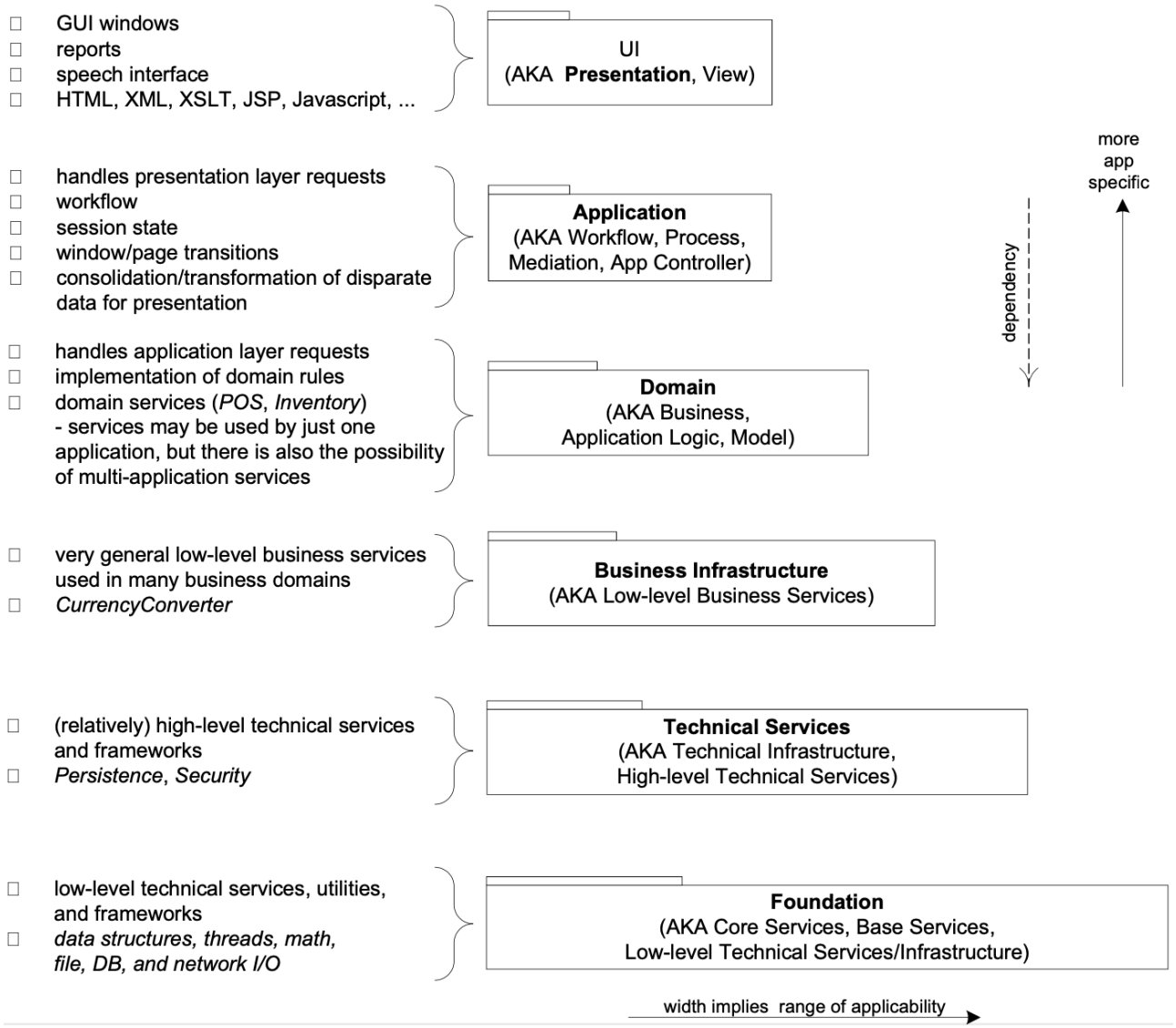
**Strict layered architecture**: A layer only calls upon the services of the layer directly below it (e.g., a network protocol stack), not common in information systems
**Relaxed layered architecture**: A higher layer calls upon several lower layers. Common in IS.

**Layers and Partitions**
UML Package diagram is used to illustrate the Logical Architecture




provided in Workshop

## Common Layers: IS Logical Architecture



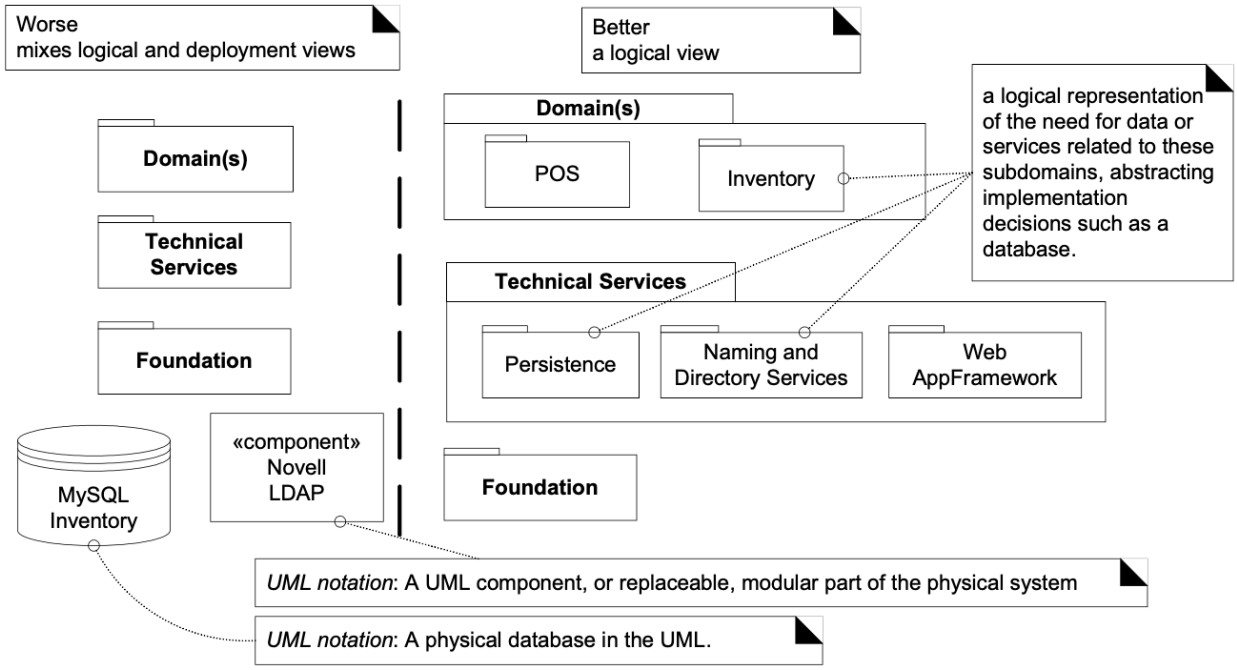

## Mapping UML Packages to Code

**// --- UI Layer**
com.mycompany.nextgen.ui.swing
com.mycompany.nextgen.ui.web
**// --- DOMAIN Layer**
// packages specific to the NextGen project
com.mycompany.nextgen.domain.sales
com.mycompany.nextgen.domain.payments
**// --- TECHNICAL SERVICES Layer**
// our home-grown persistence (database) access layer
com.mycompany.service.persistence
**// third party**
org.apache.log4j
org.apache.soap.rpc
**// --- FOUNDATION Layer**
// foundation packages that our team creates
com.mycompany.util

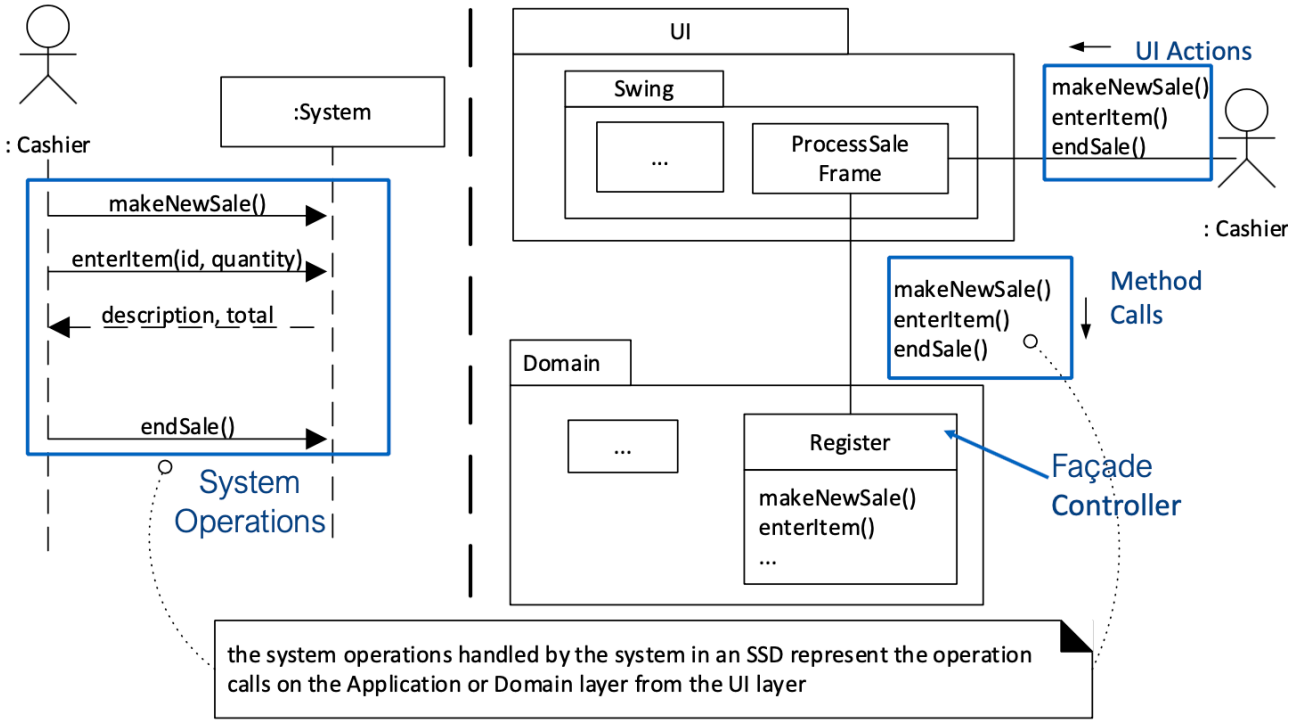
**Guidelines**:

- Organise large-scale logical structure of system into distinct cohesive layers from high application specific to low general services
  – Maintain separation of concerns, e.g. No application logic in UI objects
- Collaboration and coupling from higher layers to lower layers. Lower to higher layer coupling is avoided.
- Don't Show External Resources as the Bottom Layer

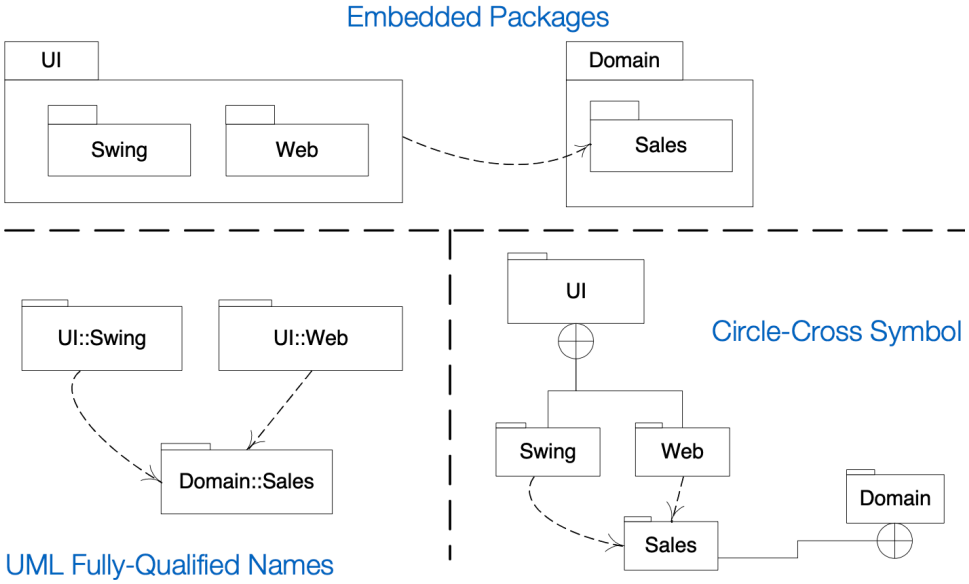
## Examples: Logical Architecture




## System Operations: SSDs and Layers



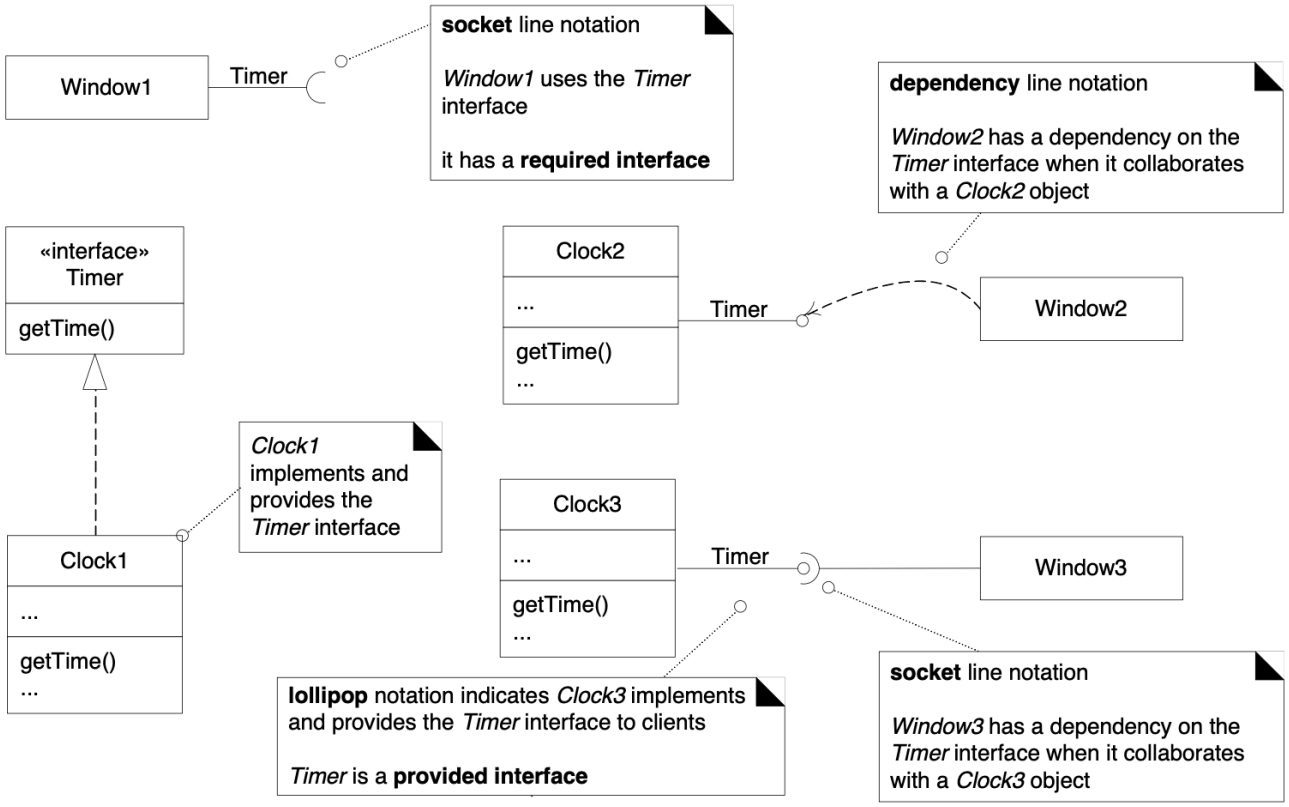
**Package Nesting: Alternatives**
The three pics are same, just different notations



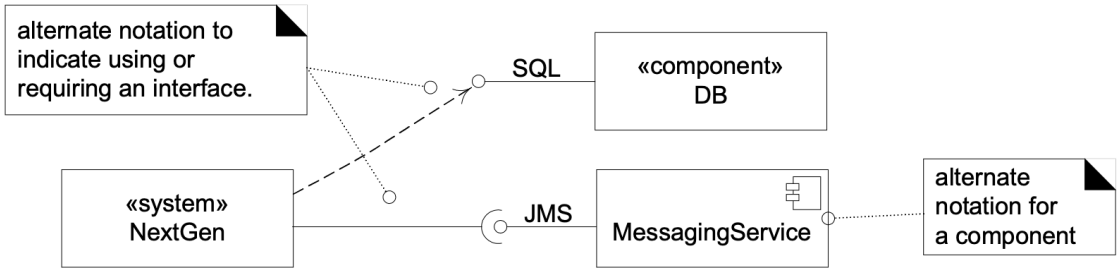

**Benefits of Layered Architecture**
Using layers helps address the following problems:
- Changes rippling through system due to coupling
- Intertwining of application logic and UI, reducing reuse and restricting distribution options
- Intertwining of general technical services or business logic with application specific logic, reducing reuse, restricting distribution, and complicating replacement
- High coupling across areas of concern, impacting division of development work

**Implementation View Architecture**
- ***Component*** *is* a modular part of a system that encapsulates its contents and replaceable within its environment
- **Component Diagram**
  - Concerns on how to implement the software system in high level
  - Replaceable within its environment
  - Provides the initial architecture landscape for the system
  - Defines its behaviour in terms of provided and required interfaces
- How is this different from a class?
  – It can be a class! External resources (e.g., DB) and services are also considered as a component

## UML component diagram notations



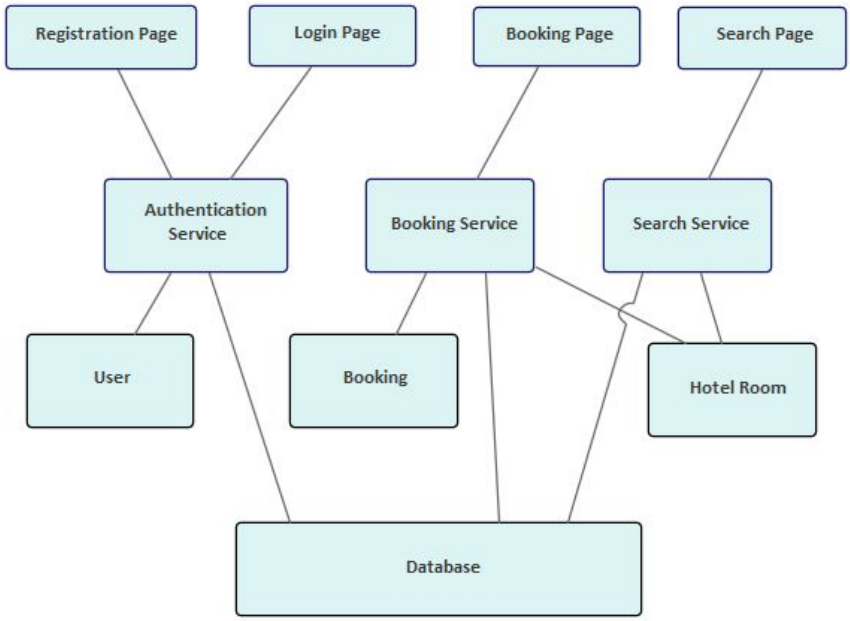

## UML Components



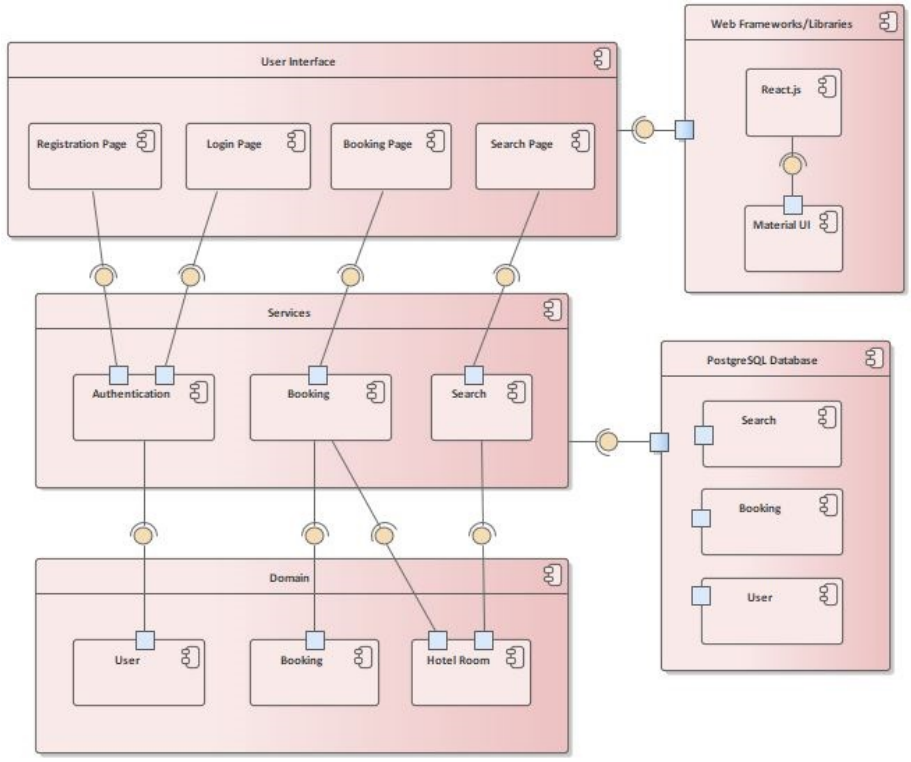

*Interface Standards:*

- SQL (Structured Query Language)
- JMS (Java Message Service)

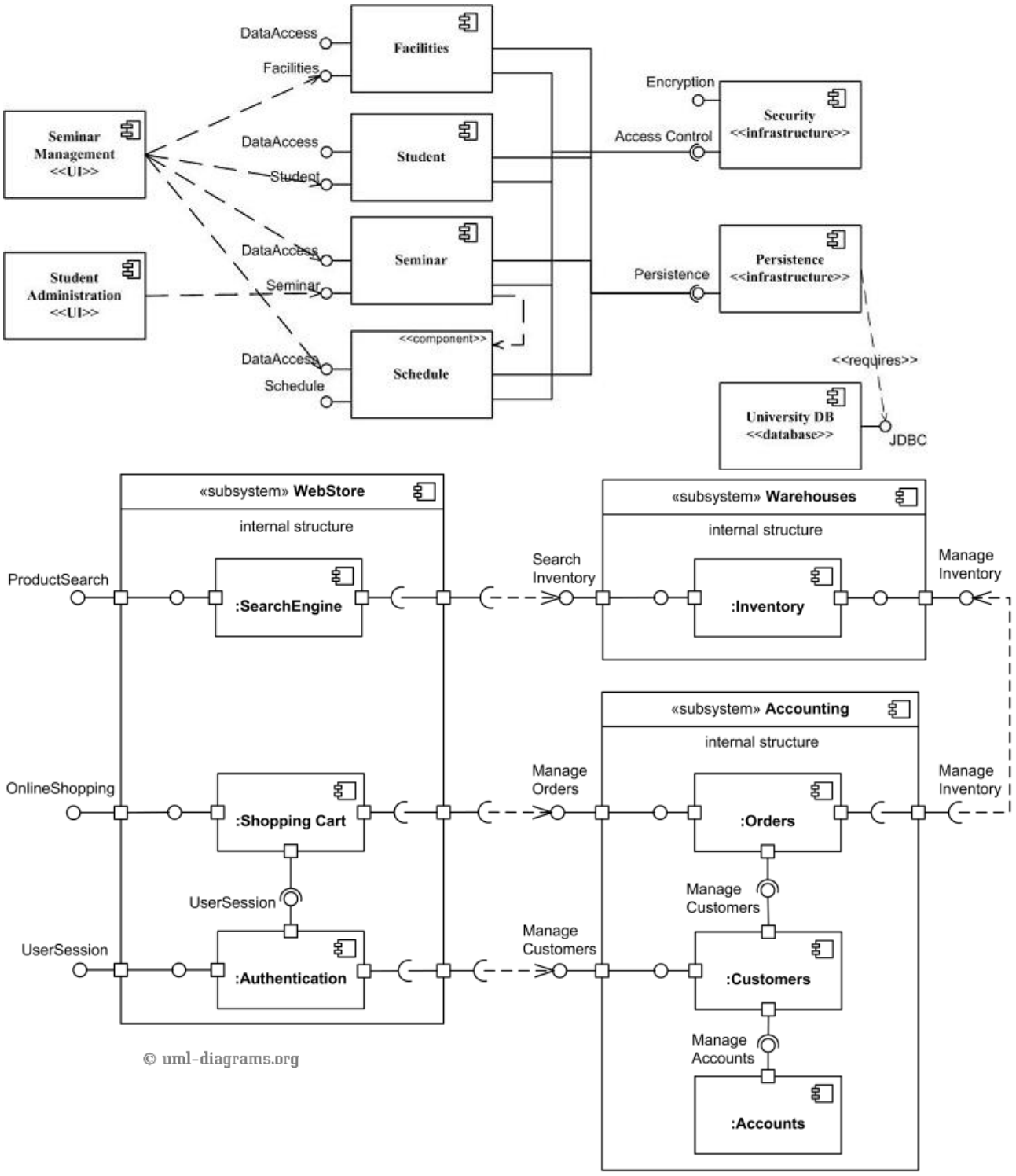
**Block Diagram**
Before we draw the required component diagram, we will start with a block diagram. A block diagram is used to provide a high-level visual overview of a system. Such a diagram does not have a formal notation and can simply include named blocks connected by lines. Each block should define a feature or component of a system and each line should connect blocks that are associated in some way. Below is an example of a block diagram for the online hotel booking application.




**Component Diagram**
Just like the block diagram, a component diagram is also used to depict a high-level visual overview of a system. However, this diagram is slightly more detailed and formal with its notation. Each component is labelled with the component symbol in the top right corner and associated components are connected by a ball and socket. Consider the relationship between the Authentication Service and the Registration Page in which the Authentication Service is directly providing an interface to the Registration Page. The ball indicates that a component is providing a service or interface and the socket indicates that a component is using that service/interface. It is also possible for components to be embedded within other components. Below is a component diagram for the online hotel booking application.



**UML Component Example**




**Summary and Remarks**

**Software Architecture** concerns on a high-level organization of software elements in the system and how the element collaborate to address the business logic

**Architectural Analysis** is an activity of analyzing requirements (especially the non-functional requirements) to identify the factors that can have an impact on the design

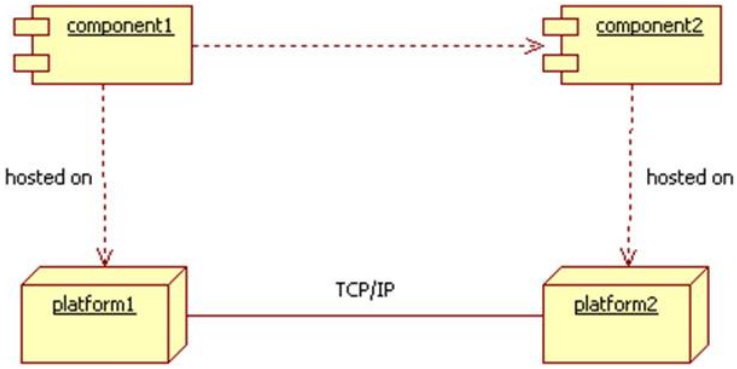
**Logical Architecture (LA)** is the large-scale organisation of the software classes UML Package diagram illustrate the logical architecture

The logical architecture can organize software classes in layers (i.e., ***Layered Architecture***)

UML Component diagram provide a high-level implementation view of the architecture which consider the external services (e.g., databases)

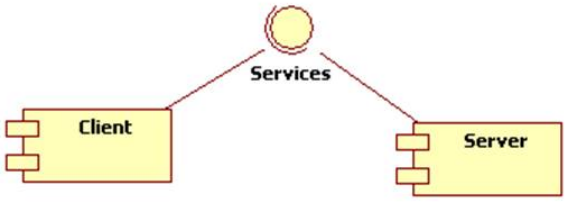
## Distributed Architectures

Components are typically:
- Hosted on different platforms
- Communicate through a network



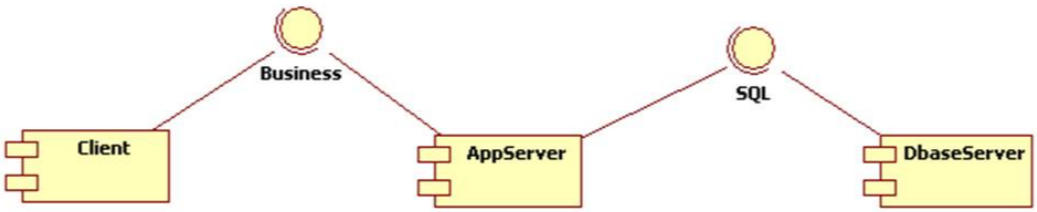

## Client-Server Architecture

Two component types: Clients and Servers
Server: perpetually listens for Client requests
- When request is received, Server processes request, then sends response back to Client
- Servers may be Stateless, or Stateful which allows for transactional interaction (Session)

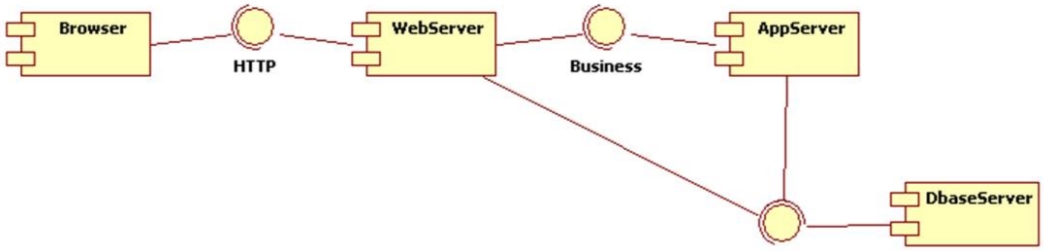



## More Client-Server Architectures

**Three tier: Tier 2 is a tier 1 server and a tier 3 client**

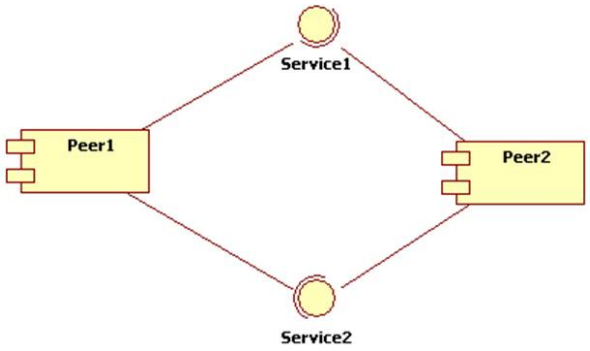



**Four Tier: Tier k is a tier k-1 server and a tier k+1 client**



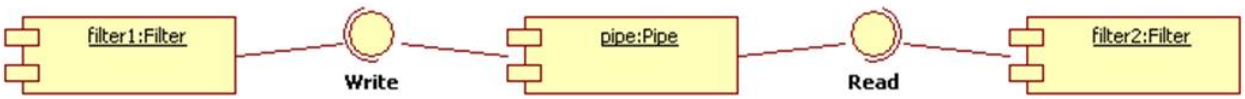

## Peer-to-peer (P2P) Architecture

Roles of client and server switch back and forth between components



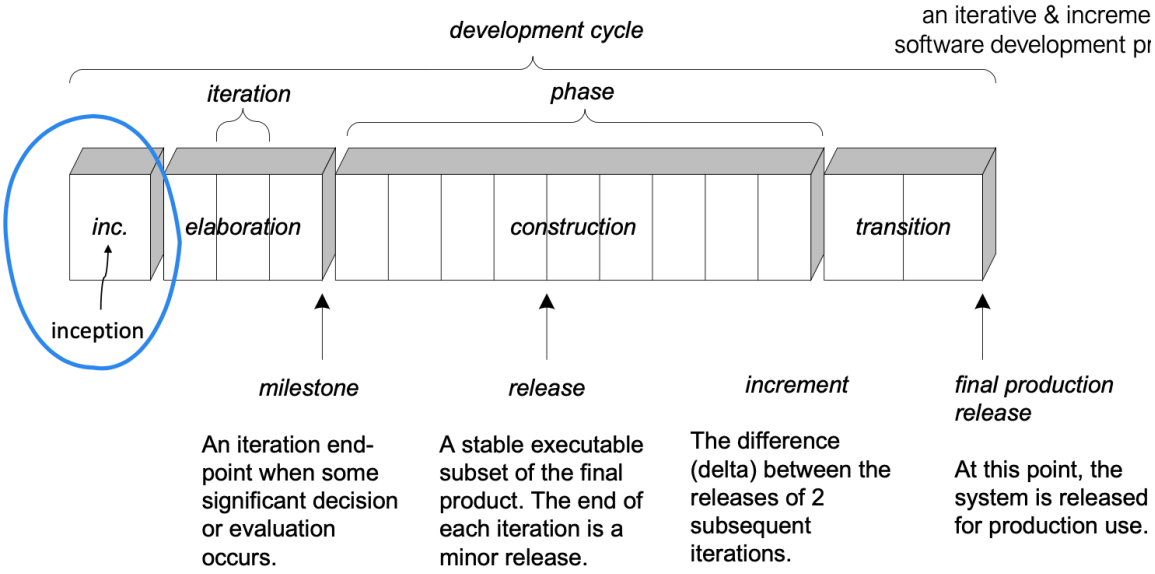

## Pipeline Architecture

One of the oldest distributed architectures
Filter perpetually reads data from an input Pipe, processes it, then writes the result to an output Pipe
Can be static and linear, or can be dynamic and complex



# Week11

Schedule-Oriented Terms in UP
Unified Process:an iterative & incremental software development process




**Inception:** is the initial short step to establish a common vision and basic scope for the project. Focusing on answering questions like:
– What is the vision and business case for this project?
**Vision** is about how will the organization or business operate in the present of the system we are thinking or developing, how will that operate differently.
Business case is how we justify the cost for the project in terms of the benefit we receive.
– Feasible?
– Buy and/or build?
– Rough unreliable range of cost: $10K–100K or $xM?
– Should we proceed or stop?
– Do the stakeholders have basic agreement on the vision of the project, and is it worth investing in serious investigation?
The UP is not the waterfall, and the inception is not the time do all requirements or create believable estimates or plans. That happens during elaboration.

**Outcome of Inception**
• Common vision and basic scope for the project
• Creation of a business case (addressing cost)
• Analysis of ~10% of use cases (difficult ones)
• Analysis of critical non-functional requirements
• Preparation of the development environment
• (Maybe) Prototypes: clarify requirements or technology questions
• Go or no go decision.
“In preparing for battle I have always found that plans are useless, but planning indispensable”- Eisenhower

**Inception Artefacts (Some, Partial)**
*must-haves

| **Artefact** | **Comment** |
|---|---|
| **Vision & Business Case*** | Describes high-level goals and constraints, business case, and provides an executive summary. |
| **Use-Case Model*** | Describes functional requirements. During inception, names of most use cases will be identified; ~10% of use cases analysed in detail. |
| Supplementary Specification | Describes other requirements, mostly non-functional. During inception, useful to have some idea of ***key non-functional requirements with major impact on the architecture*.*** |
| Glossary | Key domain terminology, and data dictionary. |
| Risk List &<br>Risk Management Plan | Describes risks (business, technical, resource, schedule) and ideas for their mitigation or response. |
| Prototypes & Proof-of-concepts | To clarify the vision, and validate technical ideas. |
| **Iteration Plan*** | Describes what to do in the *first elaboration* iteration. |
| Phase Plan &<br>Software Development Plan | Low-precision guess for elaboration phase duration and effort. Tools, people, education, and other resources. |
| Development Case | A description of the customized UP steps and artefacts for this project. In the UP, one always customizes for the project. |

**Not consider as Inception if ...**
1. You usually take more than “a few” weeks for inception.
2. You attempted to define most of the requirements.
3. You expect estimates or plans to be reliable.
4. You defined the architecture.
5. You planned the sequence of work: 1) define the requirements; 2) design the architecture; 3) implement.
6. You don’t produce a Business Case or Vision artefact.
7. You wrote all the use cases in detail.
8. You didn’t write any use cases in detail.

**Elaboration**: is the initial series of iterations for building the core architecture, resolving the high-risk elements, defining most requirements, and estimating the overall schedule and resources.
After the elaboration:
- The core, risky software architecture is programmed and tested
  – Produce an executable architecture
- The majority of requirements are discovered and stabilized
- The major risks are mitigated or retired

Elaboration is not a design phase or a phase when the models are fully developed in preparation for implementation in the construction step.

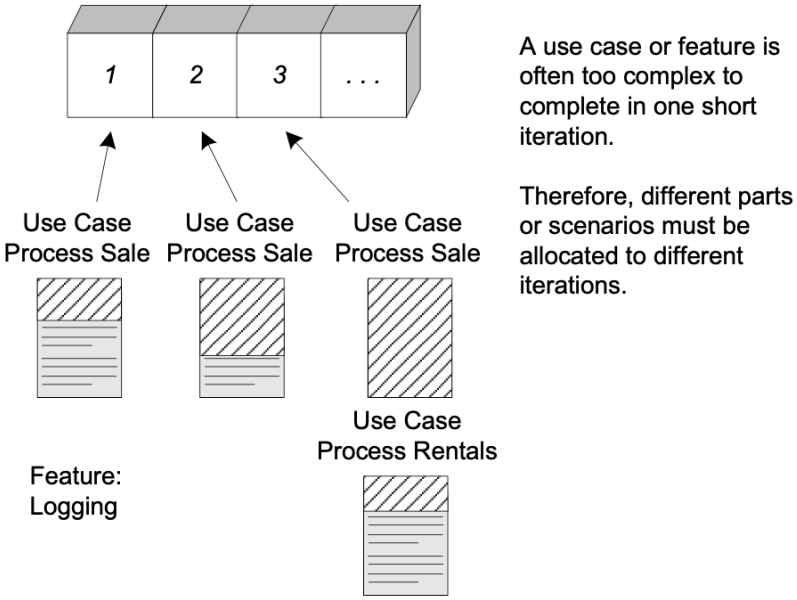
**Elaboration Artifacts**
Sample artifacts that may be started in elaboration
- Domain Model
- Design model
- Software Architecture Document
- Data Model
- Use-Case Story-boards, UI Prototypes

By the end of Elaboration, ~80% of the requirements are reliably defined

**Requirements to Design—Iteratively**

- Iteratively Do the Right Thing, Do the Thing Right
  - Requirements: do the right thing (c.f. validation)
  - Design: do the thing right (c.f. verification)
- Provoking (inevitable) Change Early
  - Although an iterative method embrace change, try to provoke the inevitable change in early iterations
  - Achieving a more stable goal, requirements discovery should stabilized
- Didn't All That Analysis & Modelling Take Weeks?
  - A few hours or days: use case writing, domain modelling
  - Mixed into a few weeks: proof-of-concept dev., finding resources, planning, setting up the environment, ...

**Spreading Use Cases Across Iterations**




A use case or feature is often too complex to complete in one short iteration.

Therefore, different parts or scenarios must be allocated to different iterations.

How do Developers Design Objects?

1. **Code**. From mental model to code. Design-while-coding (Java, C#, ...), ideally with an IDE (e.g., Eclipse or Visual Studio) which supports refactoring and other high-level operations.
2. **Draw, then code**. Drawing UML on a whiteboard or UML CASE tool (e.g. EA), then switching to 1.
3. **Only draw.** The tool generates everything from diagrams! Many a dead tool vendor has washed onto the shores of this steep island. “Only draw” is a misnomer; it still involves a programming language attached to the graphic elements.

If we use Draw, then code (the most popular approach with UML), the **drawing overhead** should be worth the effort

**Agile Modelling & Lightweight UML Drawing**

- Modelling with other developers
- Static and dynamic models
  - Class and interaction diagrams
  - Create several models in parallel
- Hand draw, and/or
  - White boards, large surface area, digital capture
- UML tool
  - IDE integration, reverse or round trip (class and interaction diagrams)
- How long?: few hours to a day near iteration start (3wk iter.)

**Object Design Skill vs UML Notation Skill**

- Drawing UML is a reflection of making decisions about the design
- UML models should use correct notation as a principle of communication
  - However, of **greatest importance** is object design skill, not UML Notation skill
- Object design requires knowledge of
  - principles of responsibility assignment
  - design patterns

**Summary**

In Unified Process (UP), Inception is the initial short step to establish a common vision and basic scope for the project

Elaboration is the phase where software design and modelling is performed The core architecture is programmed and tested (i.e., executable architecture)

Translating requirements to design can be done iteratively