

*Tribhuvan University*  
*Institute of Engineering*  
*Kathmandu Engineering College*  
*Department of Electronics and Computer Engineering*

***Object Oriented Analysis and Design***  
***[Subject Code: CT 651]***

*by*  
***Er. Santosh Giri***  
***Lecturer, IOE, Pulchowk Campus***

# Contents

## 🚫 Object Oriented Fundamentals → 10 hrs, 18 marks

- © *Introduction*
- © *Object Oriented Analysis and Design*
- © *Defining Models*
- © *Case Study (NextGen point-of-sale (POS) system, Monopoly Game)*
- © *Requirement Process*
- © *Use Cases*
- © *Object Oriented Development Cycle*
- © *Overview of the Unified Modeling Language: UML Fundamentals and Notations.*

## 🚫 Object Oriented Analysis → 8 hrs, 14 marks

- © *Building Conceptual Model*
- © *Adding Associations and Attributes*
- © *Representation of System Behavior*

# 🚫 Object Oriented Design

→12 hrs, 21 marks

- © *Analysis to Design*
- © *Describing and Elaborating Use Cases*
- © *Collaboration Diagram*
- © *Objects and Patterns*
- © *Determining Visibility*
- © *Class Diagram*

# 🚫 Implementation

→15 hrs, 27 marks

- © *Programming and Development Process*
- © *Mapping Design to Code*
- © *Creating Class Definitions from Design Class Diagrams*
- © *Creating Methods from Collaboration Diagram*
- © *Updating Class Definitions*
- © *Classes in Code*
- © *Exception and Error Handling.*

**Reference Book: *Applying UML and Patterns***

**→C. Larman**

# Pre-requisites

## **Object Oriented Programming**

- © Objects
- © Attributes
- © Class
- © Methods
- © Message passing & more

## **Software Engineering**

- © SDLC and Process Models.
- © Requirement Engineering Process.
- © Functional and Non-Functional Requirements.
- © Agile Methodology\*
- © RUP-with Reference from Software Engineering\* & more

## **Project Work**

- © UML Diagrams\*
- © Lab work\*

*Tribhuvan University*  
*Institute of Engineering*  
*Kathmandu Engineering College*  
*Department of Electronics and Computer Engineering*

## **Chapter1: Object Oriented Fundamentals**

10 Hrs, 18 Marks

*by*  
***Er. Santosh Giri***  
***Lecturer, IOE, Pulchowk Campus***

# *Object Oriented Fundamentals*

# Chapter One

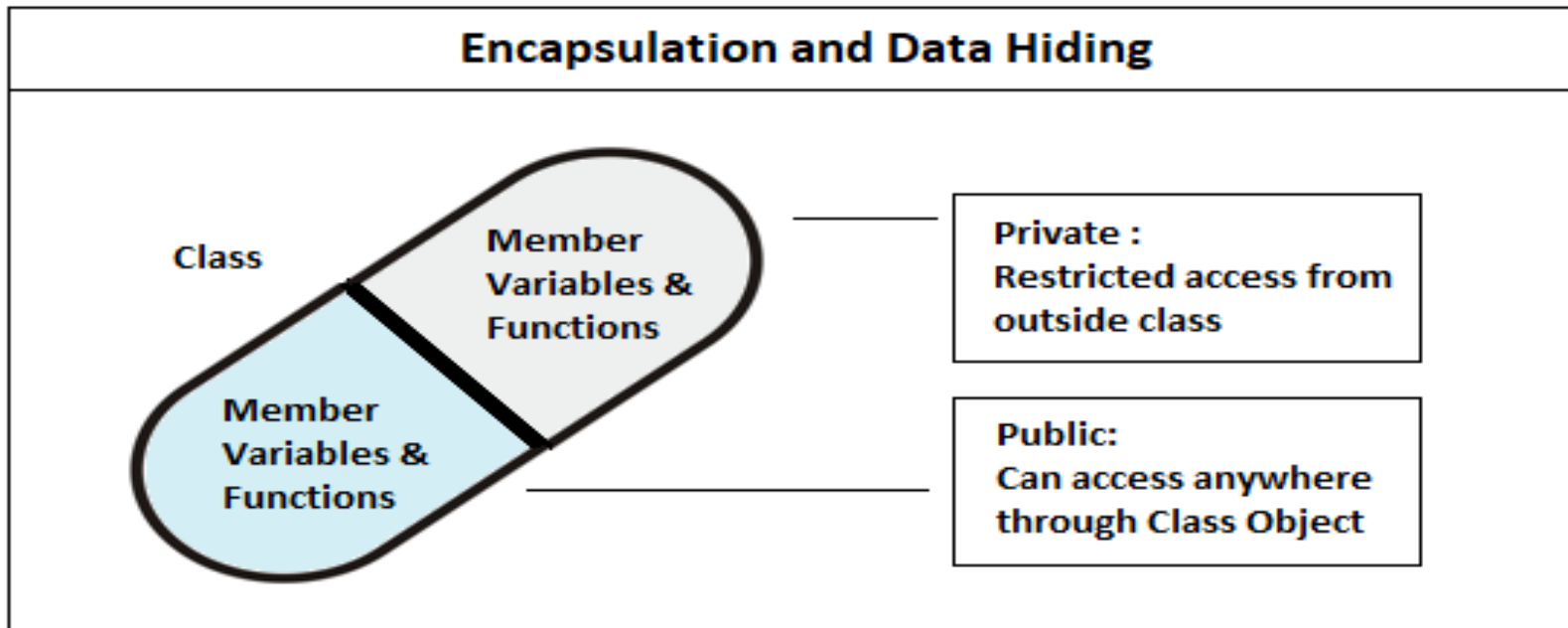
## **Object Oriented Fundamentals** → *10 hours*

- ❌ Introduction
- ❌ Object Oriented Analysis and Design
- ❌ Defining Models
- ❌ Case Study
- ❌ Requirement Process
- ❌ Use Cases
- ❌ Object Oriented Development Cycle
- ❌ Overview of the Unified Modeling Language: UML  
Fundamentals and Notations

# Features of Object Oriented System

## ❌ Encapsulation

- © Encapsulation is the process of combining data and function into a single unit called class.





## ⊘ Abstraction



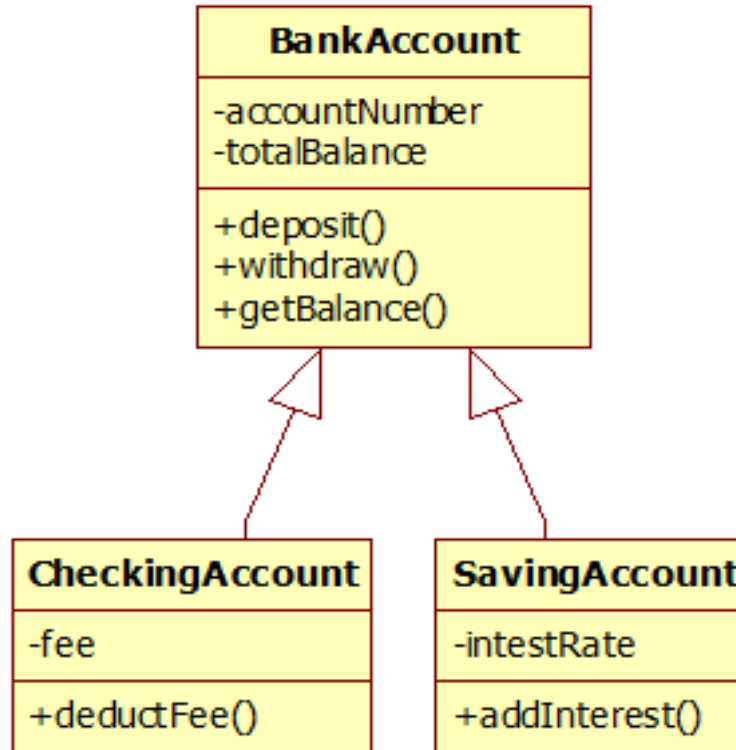
### Example: ATM Machine

© We can perform operations on the ATM machine like cash withdrawal, balance enquiry, retrieve mini-statement...etc.

but what about internal details i.e. How??

© Representing only necessary information, hiding the implementation details is abstraction.

## © Inheritance



© **Advantage:** Reusability

## Polymorphism

- © Polymorphism means having many forms
- © Polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

### © **Example:**

```
sum(a,b);  
sum(a,b,c);
```

# Structured approach vs OO Approach

Structured Approach	Object Oriented Approach
Top-down approach.	Bottom-up approach.
Program is divided into number of sub modules or functions.	Program is organized by having number of classes and objects.
Function call is used.	Message passing is used.
Software reuse is not possible.	Reusability is possible.
Structured design programming usually left until end phases.	Object oriented design programming done concurrently with other phases.
DFD & E-R diagram are used to model the data.	Class diagram, sequence diagram, state chart diagram, and use cases all contribute.

# Object Oriented Analysis and Design

- © It is technical approach for **analyzing** and **designing** an application/system by applying **object-oriented** programming.
- © **During analysis phase**, you try to determine which objects you need and how to organize them.
- © **During design phase**, you take the analyzed objects and add constraints to make them fit within the software and hardware that you are developing for.
- © **But line between them is very thin and many easily tend to cross it.**

# Object Oriented Analysis

- ④ It is an iterative stage of analysis, which takes place during the SDLC, that aims to **model the functional requirements** of the software, while remaining completely independent of any potential **implementation requirements**.
- ④ The source of OOA can be written requirements statements, A formal Vision document, Interviews with stakeholders etc.
- ④ This will be typically presented as a set of use cases, one or more UML class diagrams, interaction diagram etc.
- ④ Use case model:
  - © Use cases
  - © Activity Diagram
  - © Swimlane diagram & more

# Object Oriented Analysis

## **OOA phase consists of five stages:**

- © Find and define the objects.
- © Organize the objects by creating object model diagram.
- © Describe how the objects interact with one another.
- © Define the attributes of the objects.
- © Define the behavior i.e. actions of the objects.

# Object Oriented Design

- ❌ OOD is the Implementation of the conceptual model produced during OOA.
- ❌ It is really just an extension of the OOA process except with the consideration and implementation technological or environmental of constraints (i.e. How?).
- ❌ The constraints are:
  - © response time
  - © run-time platform
  - © development environment or programming language
  - © Time and budgetary limitations
  - © developer aptitude (ability), and so forth



# Object Oriented Design

## **OOD Tasks:**

- © Physical DB design
- © Design system architecture
- © Design classes
  - ✓ Combining classes
  - ✓ Splitting classes
  - ✓ Eliminate classes
- © Design components
- © GUI design

## **OOD phases:**

- © Restructuring the class data (if necessary),
- © Implementation of methods, i.e., internal data structures and algorithms,
- © Implementation of control, and
- © Implementation of associations and so on

→ If **OOA** is the **what**, then **OOD** is the **how**.

# OOAD

## OOAD Basic Terminologies

- © Object, Attributes, Class, Methods, Message Passing, Encapsulation, Abstraction, Inheritance.
- © **Already Discussed in previous slides**

## OOAD Modeling

- © Models helps to visualize, specify, construct and document the artifacts of software intensive system.
- © OOAD modeling involves development of various diagrams that describes the system under consideration.
- © It helps to manage complexity, understand requirements and properly derive the implementation.

# OOAD

**OOAD Modeling types** → Details on later slides

- © Conceptual model
- © Structural model
- © Behavioral model
- © Specification Model
- © Implementation model

# OOAD

## 🚫 Requirement Process:

- © Requirement process is a systematic approach to find, document, organize and track the needs of the users and response on changing requirements of a system.
- © Requirements are the aspects that the system must conform.

## 🚫 Requirement Types → Assignment

- © Functional Requirement- What?
- © Non-Functional Requirement- How?
- © See S/W slides

## 🚫 Requirement Elicitation Methods → Assignment

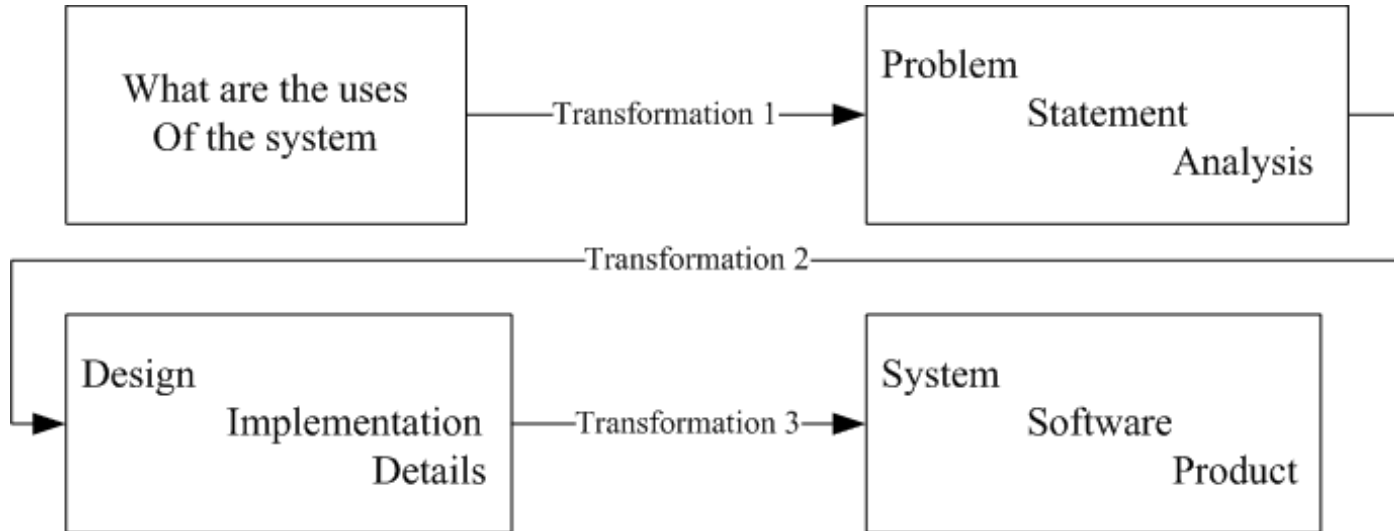
- © Questionnaire
- © Task Analysis
- © Scenario
- © Case study

# *Object Oriented Development Cycle*

# Object Oriented Development Cycle

- ❌ Object oriented systems development is a way to develop software by building self – contained (independent) modules or objects that can be easily replaced, modified and reused.
- ❌ In an object–oriented environment, software or application is a collection of discrete/separate objects that encapsulate their data as well as the functionality.
- ❌ It works by allocating tasks among the objects of the application.
- ❌ The software development approach for object oriented modeling goes through following stages:
  - © Analysis
  - © Design
  - © Implementation and Testing

# Object Oriented Development Cycle



## ❌ Transformation 1 (analysis):

© Translates the users' needs into system requirements & responsibilities.

## ❌ Transformation 2 (design):

© Begins with a problem statement and ends with a detailed design that can be transformed into an operational system.

## ❌ Transformation 3 (implementation)

© Refines the detailed design into the system deployment that will satisfy the users' needs.

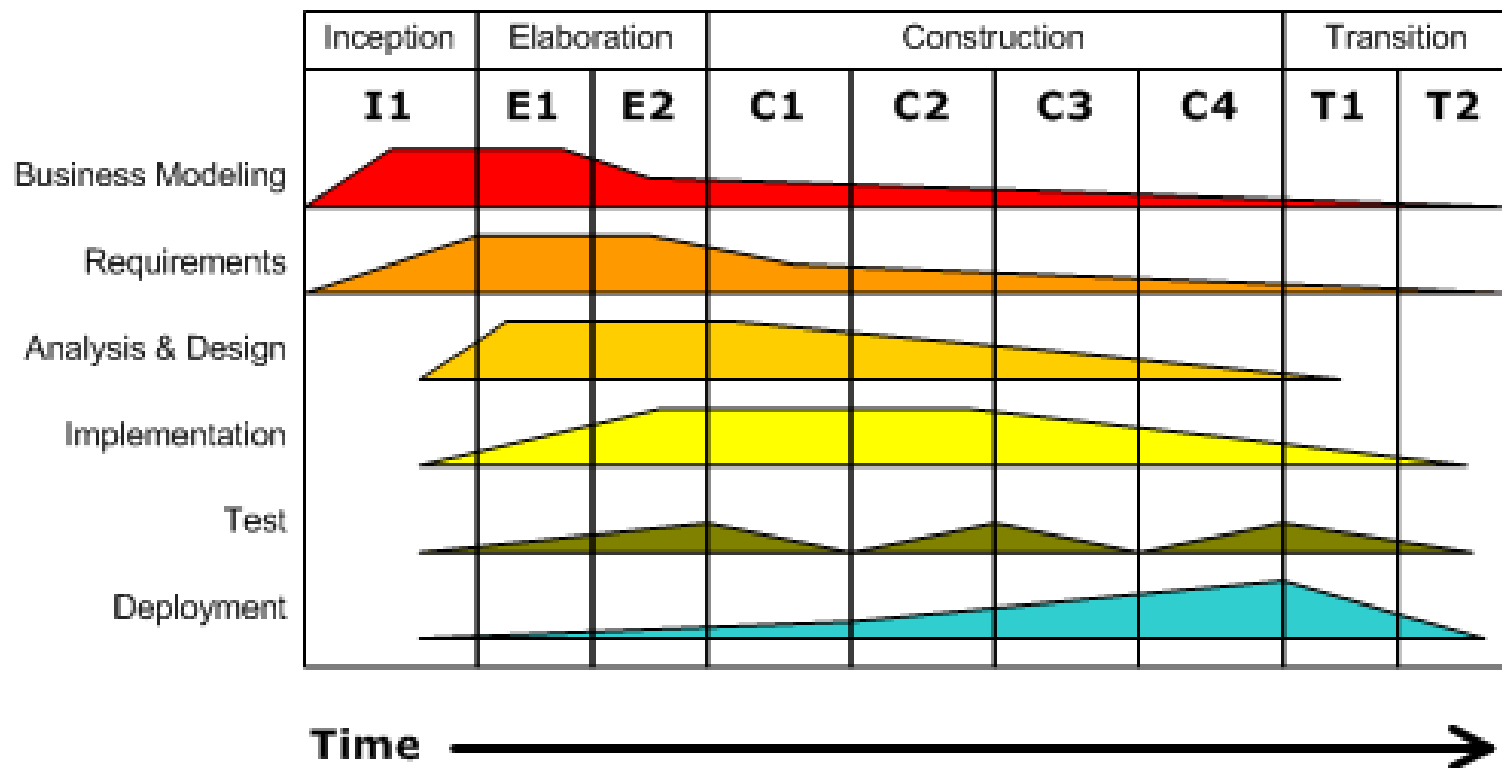
© It represents embedding s/w product within its operational environment.

# *Rational Unified Process*



# Rational Unified Process

- ❌ The most successful approach for object oriented software development is Rational Unified Process (RUP).
- ❌ It is an approach that combines iterative and risk driven development into a well documented process description.



# RUP

<b>Discipline</b>	<b>Purpose</b>
<b>Business Modeling</b>	©A business context (scope) of the project /product /software showing how your system fits into its overall environment. © <i>outlining the scope /feasibility of the product</i>
<b>Requirements Phase</b>	© Gathering requirements
<b>Design Phase</b>	©System design and planning
<b>Development Phase</b>	©Coding and unit testing;
<b>System Testing</b>	©Whole system testing and integration;
<b>Deployment</b>	©Deployment and release to production.

# RUP Phases (*Abstract*)

## **Inception:**

- © The idea for the project is stated. The development team determines if the project is worth pursuing and what resources will be needed.

## **Elaboration**

- © The project's architecture and required resources are further evaluated. Developers consider possible applications of the software and costs associated with the development.

## **Construction**

- © The project is developed and completed. The software is designed, written, and tested.

## **Transition**

- © The software is released to the public. Final adjustments or updates are made based on feedback from end users.

# RUP Phases (*description*)

## Inception

- © The requirements are gathered.
- © Feasibility study and scope of the project are determined.
- © Actors and their interactions are analyzed.

## Elaboration

- © Project plan is developed.
- © Risk assessment is performed.
- © Non-functional requirements are elaborated.
- © Software architecture is described.
- © Use case model is completed.

# RUP Phases

## **Construction**

- © All the components are developed and integrated.
- © All features are tested.
- © In each iteration, refactoring (clarifying and simplifying the design of existing code, without changing its behavior) is done.
- © Stable product should be released.

## **Transition**

- © Software product is launched to user.
- © Deployment baseline should be complete.
- © Final product should be released.

## **RUP Conclusion:**

- © Input to a process is the needs or requirements of the system.
- © Process is the set of activities to reach goal.
- © Output is the software product.

**Agile Unified Process → Assignment**

# *UML fundamental and Notations*

## 🚫 UML fundamental and Notations

- © UML (Unified Modeling Language) is a standardized modeling language consisting of an integrated set of diagrams, developed to help system and software developers for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems.
- © The UML represents a collection of best engineering practices that have proven successful in the modeling of large and complex systems.
- © The UML uses mostly graphical notations to express the design of software projects.
- © Using the UML helps project teams communicate, explore potential designs, and validate the architectural design of the software.

## 🚫 *Model Diagrams in UML*

- © The diagrams that are used to visualize the models of the system are called as model diagrams.
- © Constraint specifications help to explain the model diagrams and also indicates its boundary.



# *Five Views of UML*

# UML Views

## **User's view**

- © This contains the diagrams in which the user's part of interaction with the software is defined.
- © No internal working of the software is defined in this model.
- © The diagrams contained in this view are:

*Use case Diagram*

## **Structural view**

- © In the structural view, only the structure of the model is explained.
- © This gives an estimate of what the software consists of. However, internal working is still not defined in this model.
- © The diagram that this view includes are:

*Class Diagrams*

*Object Diagrams*

# UML Views

## Behavioral view

- © The behavioral view contains the diagrams which explain the behavior of the software.
- © These diagrams are:

*Sequence Diagram*

*Collaboration Diagram*

*State chart Diagram*

*Activity Diagram*

## Environmental view

- © The environmental view contains the diagram which explains the after deployment behavior of the software model.
- © This diagram usually explains the user interactions and software effects on the system.
- © The diagrams that the environmental model contain are:

**Deployment diagram**

# UML Views

## **Implementation view**

- © The implementation view consists of the diagrams which represent the implementation part of the software.
- © This view is related to the developer's views.
- © This is the only view in which the internal workflow of the software is defined.
- © The diagram that this view contains is as follows:

***Component Diagram***

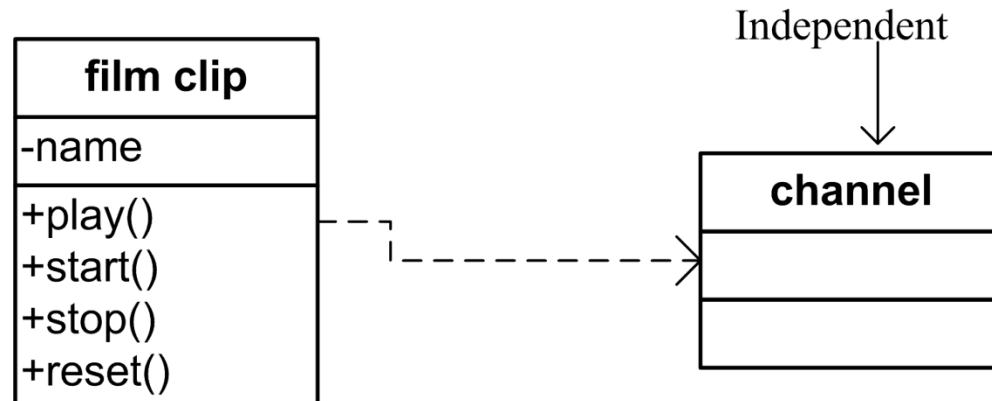
# *UML*

## *Notations*

# UML Notations

## Dependency

- © A dependency relationship indicates that changes to one model element (independent model) can cause changes in another model.
- © The change in independent thing will affect the dependent thing.
- © Dependency is graphically represented as a dashed directed line.
- © The arrow head points towards the independent thing.



# UML Notations

## Association:

- © describe links between objects.
- © It is presented as solid line connecting the participating classes, labeled and with adornments (multiplicity and role name).

*Various degree of association:*

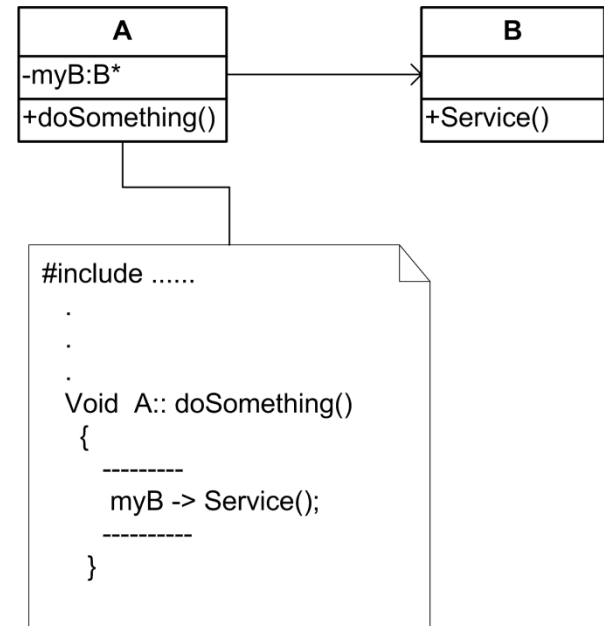
### 1. Unary

- © Class A knows about class B.
- © Class B knows nothing about Class A.
  - \*Usually 'knows about' means a pointer or reference*

### Example:

A person knows its address

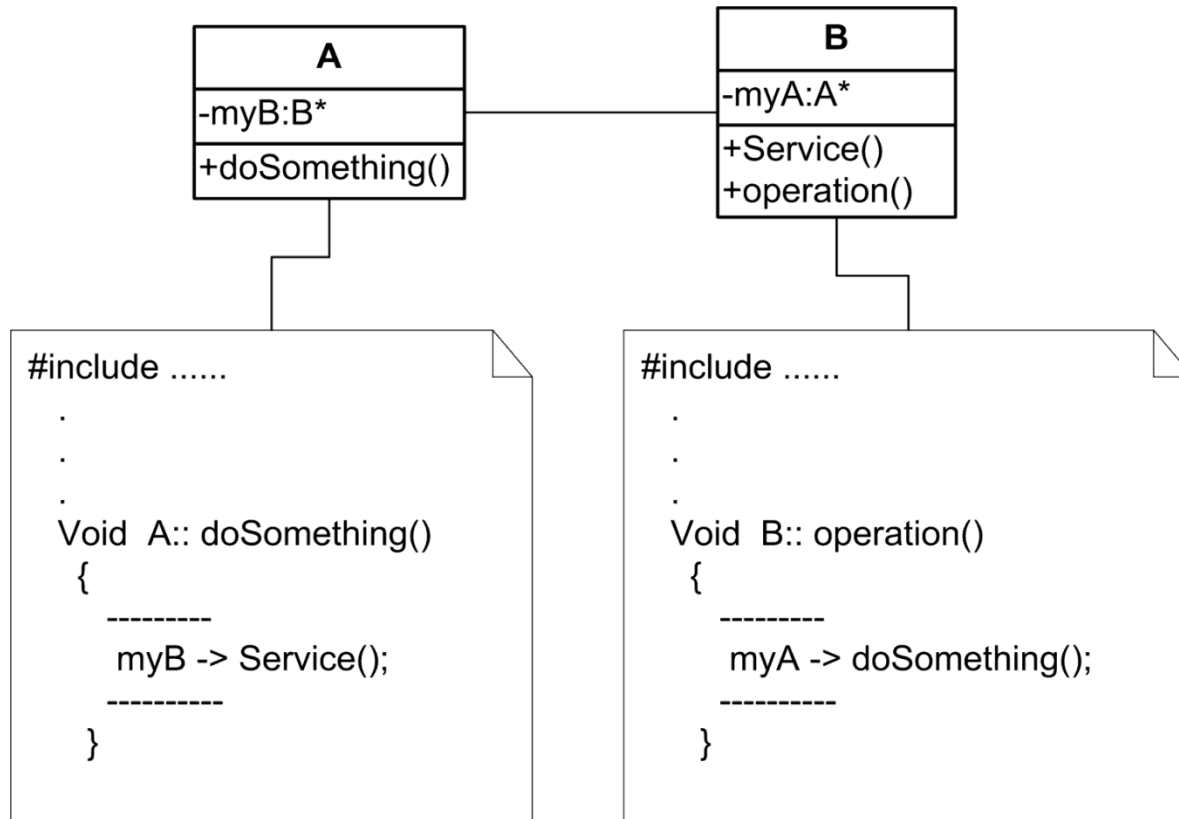
Address doesn't know who is living there



# UML Notations

## 2. Binary

© Class A knows about class B and vice-versa.



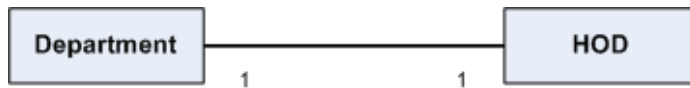


# UML Notations

## 2. Binary

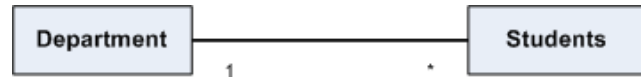
### i. One to one

Example: **Department** and **HOD** have one to one associations.



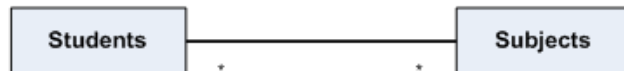
### ii. One to many

Example: **department** and **students** have one to many associations.



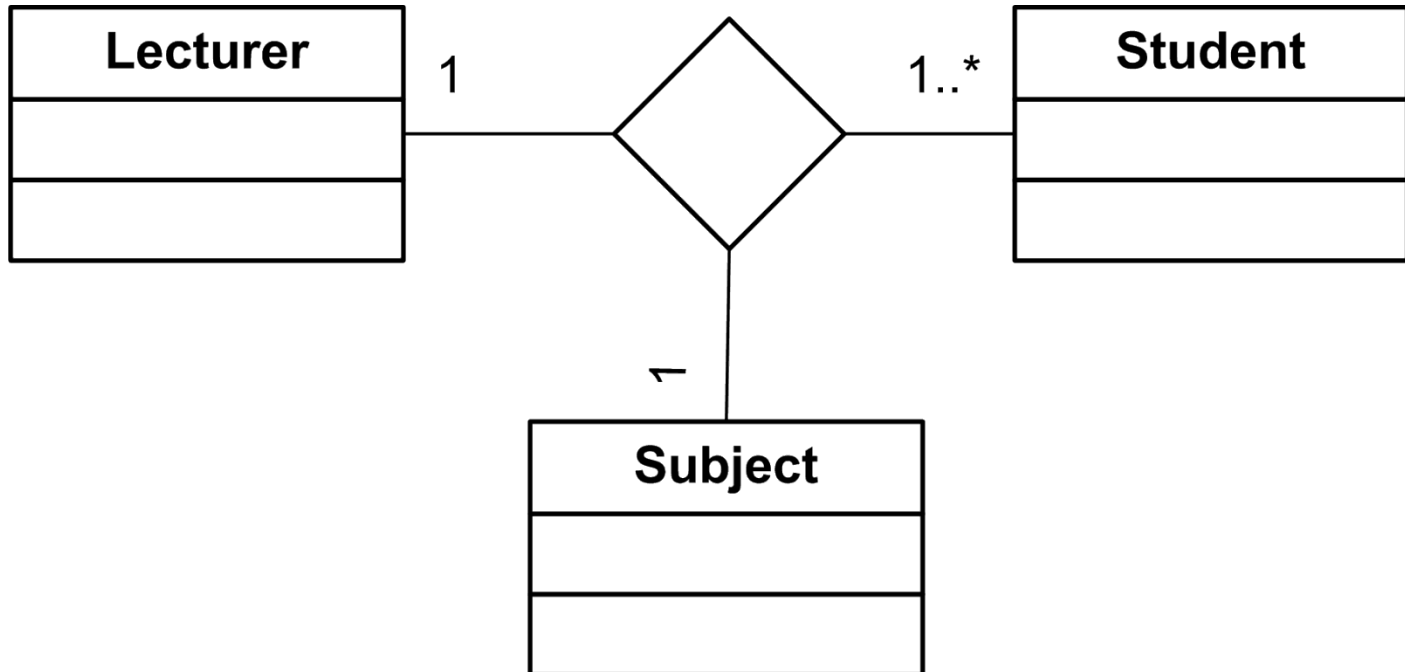
### iii. Many to Many

Example: **students** and **subjects** have many to many relationship.



# UML Notations

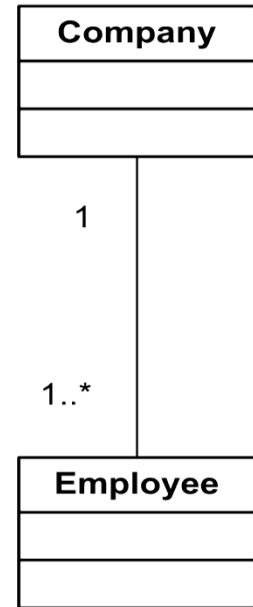
## 3. Ternary



# Multiplicities Example

**Format of multiplicity:**  $\langle \text{Lower\_bound} \rangle .. \langle \text{Upper\_bound} \rangle$

1	Exactly one, no more/ no less.
0..1	Zero or one
*	many
0..*	Zero or many
1..*	one or many

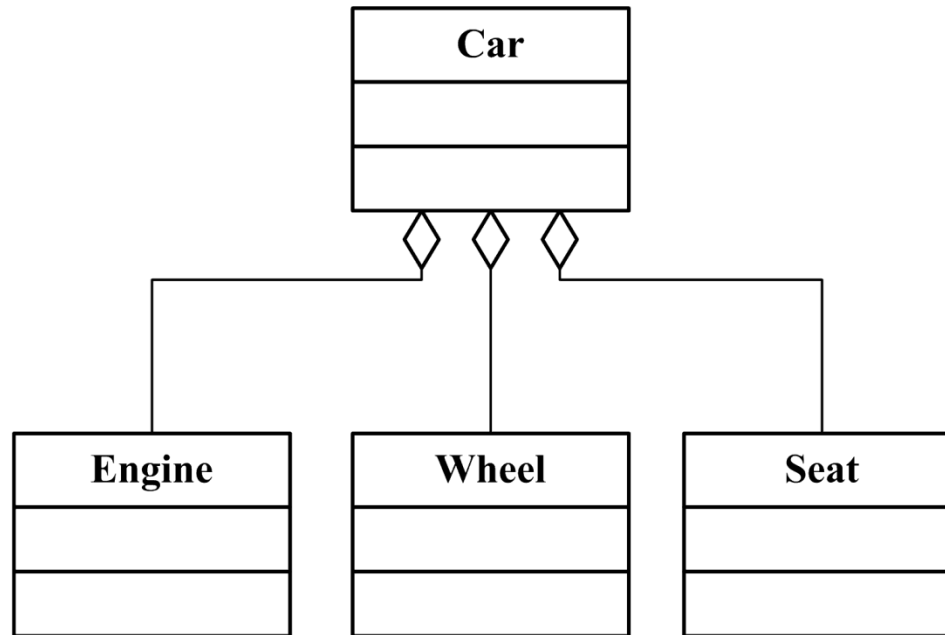


- © In association, each role has multiplicity, which shows the number of objects that participate in an association.
- © E.g.: 4 .. 8 means minimum of 4 and maximum of 8.
- © The upper bound can be \*, which means infinite upper bound. If only single integer is shown then the range includes only that value

# UML Notations

## Aggregation

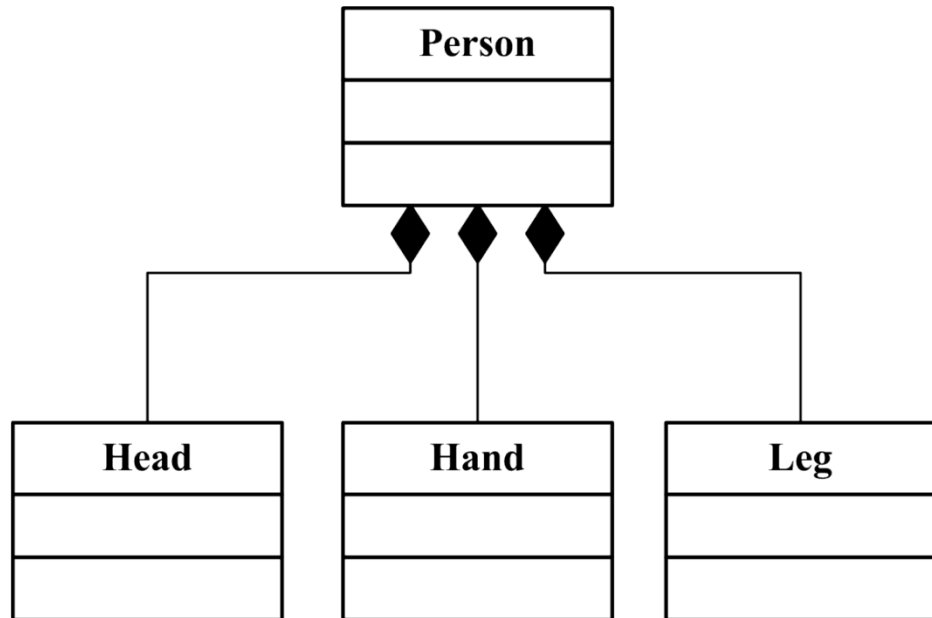
- © Aggregation represents part-of relationship between a component object and an aggregate object. Objects are assembled to create a new, more complex object.
- © Aggregation is represented with a hollow diamond at the aggregate end.



# UML Notations

## Compositions:

- © **Aggregation** implies a relationship where the child can exist independently of the parent.
- © E.g.: Class (parent) and Student (child). Delete the Class and the Students still exist.
- © **Composition** implies a relationship where the child cannot exist independent of the parent. Example: House (parent) and Room (child). Rooms don't exist separate to a House.



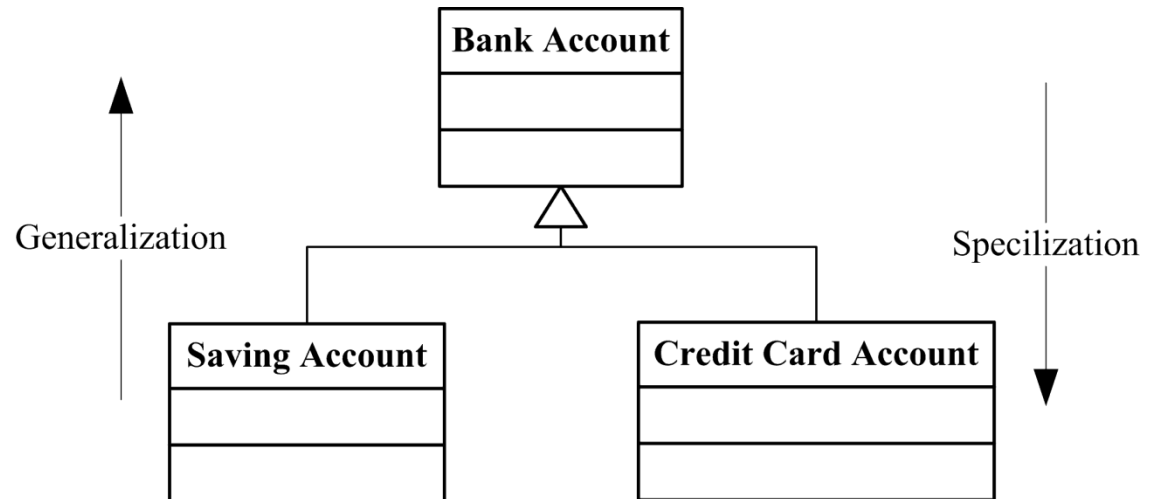
# UML Notations

## Generalization

- © Generalization is a mechanism for combining similar classes of objects into a single, more general class.

## Specialization

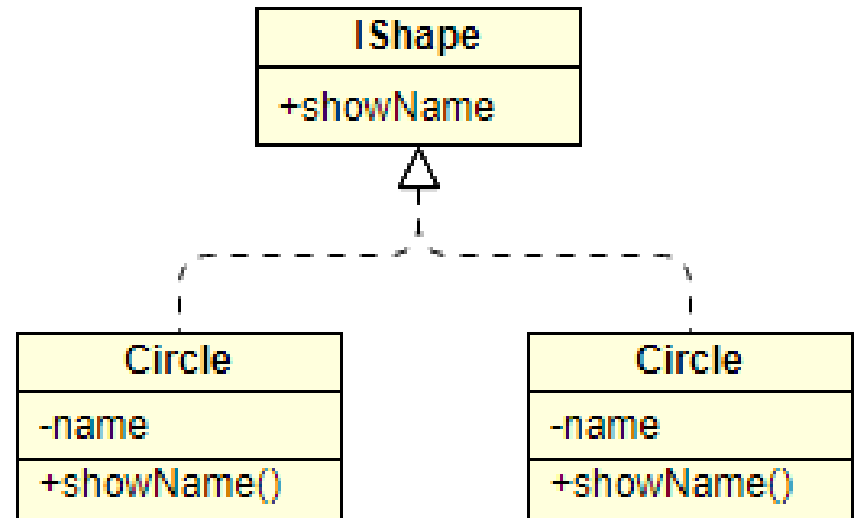
- © It is reverse process of Generalization means creating new sub classes from an existing class.



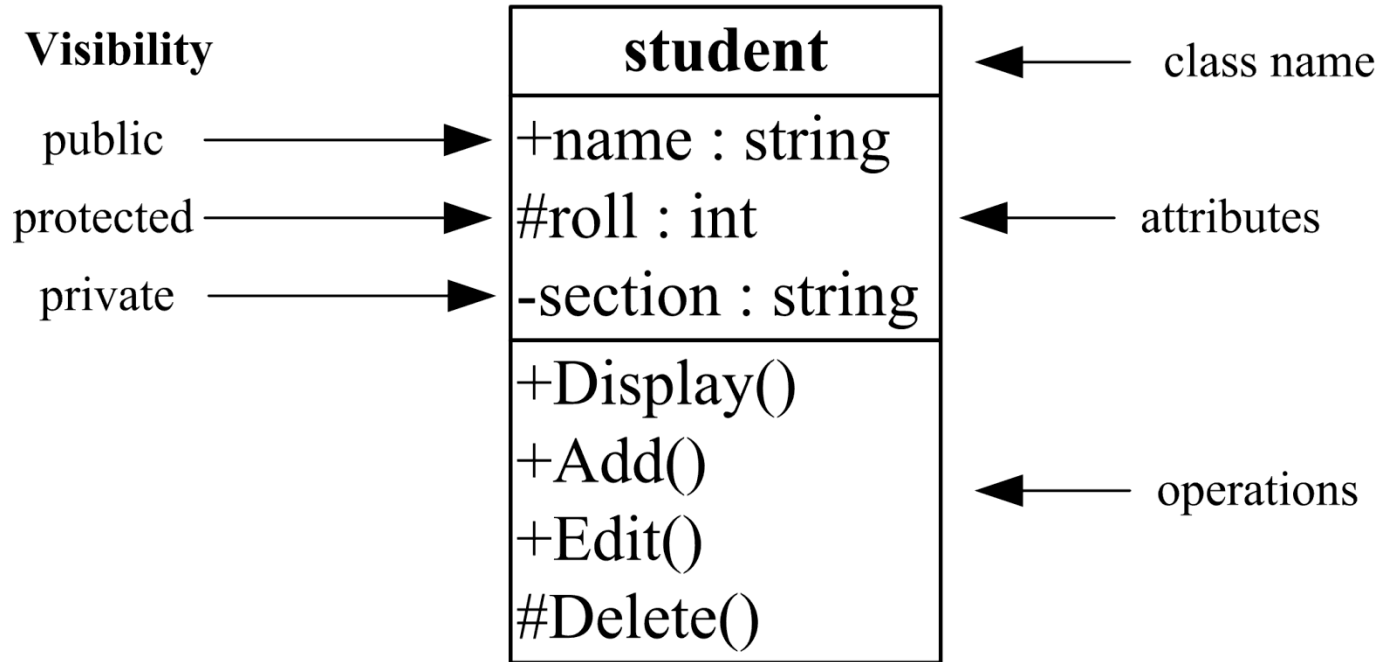
# UML Notations

## Realization:

- © It only inherits operations and cannot inherit attributes and association.
- © It is represented by a hollow triangle shape on the interface end of the *dashed* line. End corresponds to the realized element (the **supplier**).



# Class Notations



- © UML *class* is represented by the diagram shown above
- © The top section is used to name the class.
- © The second one is used to show the attributes of the class
- © The third section is used to describe the operations performed by the class.



# Symbols

©**Plus**:"+", indicates **public** visibility for an operation or attribute.

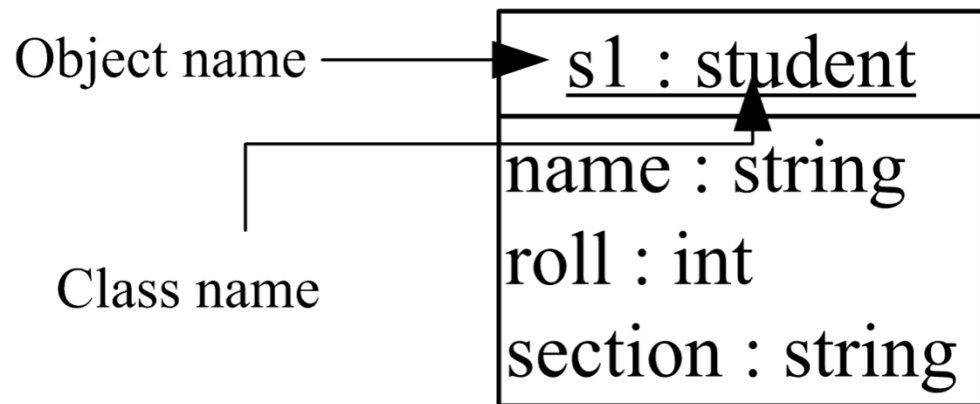
©**Minus**:"- ", indicates **private** visibility for an operation or attribute.

©**Hash**:"#", indicates **protected** visibility for an operation or attribute.

©**Colon**:":", used to separate a name of a parameter or attribute or operation from a class.

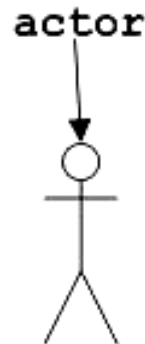
# Object Notation:

- © The *object* is represented in the same way as the class.
- © The only difference is the *name* which is underlined as shown.



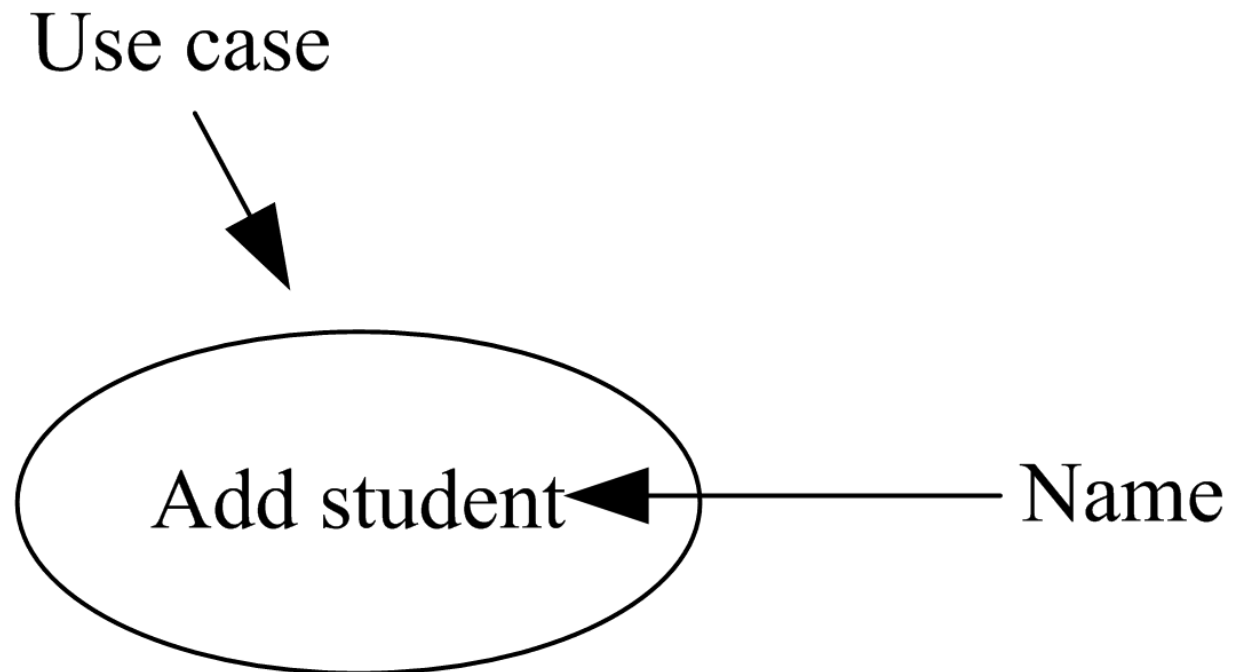
# Actor Notation

- © An **actor** can be defined as some internal or external entity that interacts with the system.
- © **Actor** is used in a use-case diagram to describe the internal or external entities.



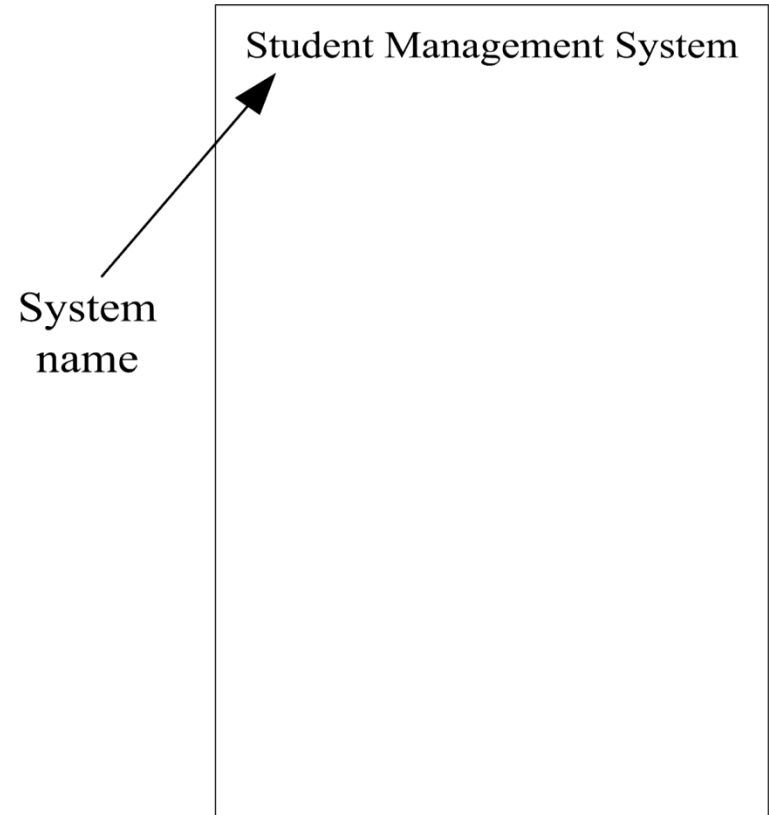
## Use-Case Notation:

© **Use-Case** is represented as an eclipse with a name inside it. It may contain additional responsibilities.



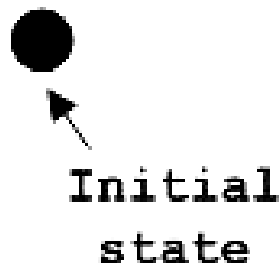
# System Boundary

- © It helps to identify what is an external vs. internal in use case.
- © Represented as a rectangle.



# Initial State Notation:

- © **Initial state** is defined to show the start of a process. This notation mainly used in Activity diagram and State machine diagram.
- © The usage of Initial State Notation is to show the starting point of a process



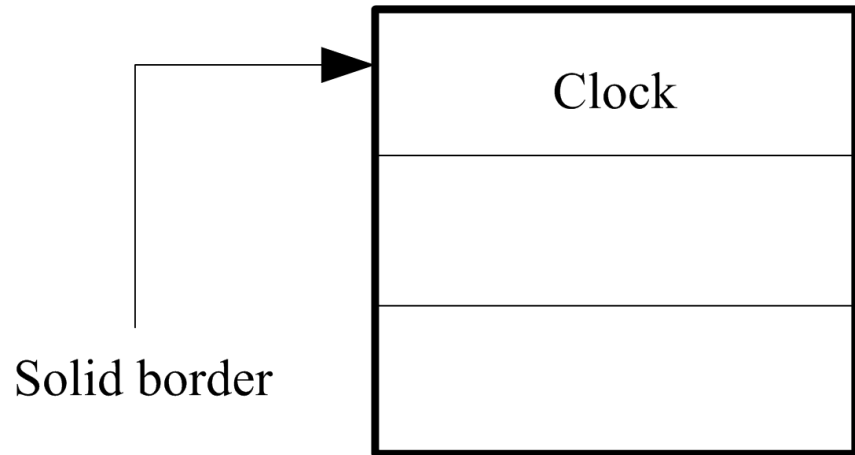
# Final State Notation:

- © **Final state** is used to show the end of a process. This notation is mainly used in activity and State machine diagram to describe the end.
- © The usage of Final State Notation is to show the termination point of a process.



# Active class Notation:

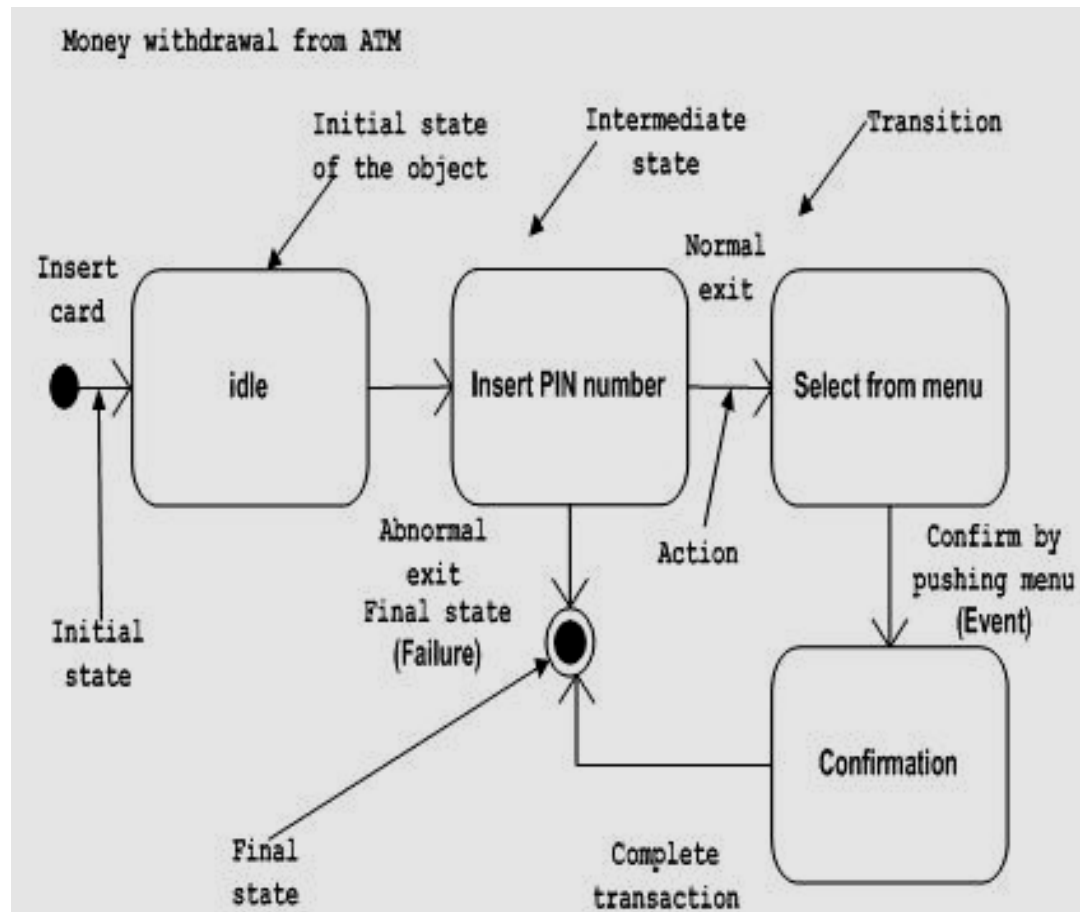
- © Active class looks similar to a class with a solid border.
- © An Active Class indicates that, when instantiated, the Class controls its own execution., it can operate standalone and define its own thread of behavior.
- © An **active object** runs on and controls its own thread of execution rather than being invoked or activated by other objects
- © **Example:** Clock





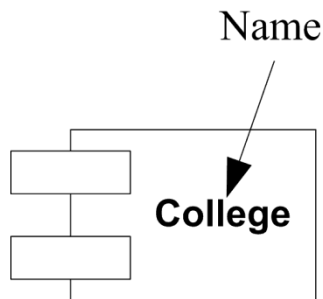
# State machine Notation:

- © State machine describes the different states of a component in its life cycle.



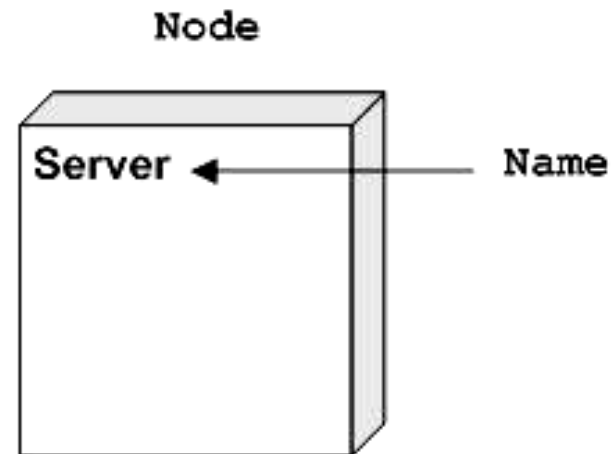
# Component Notation:

- © A component is a logical unit block of the system, a slightly higher abstraction than classes i.e. *shows "white-box" view*.
- © A **component** in UML is shown as below with a name inside. Additional elements can be added wherever required.



# Node Notation:

- © A node in UML is represented by a square box as shown below with a name.
- © It is a computational resource upon which system may be deployed for execution. It may be
  - A hardware device
  - An execution environment represents software containers (such as OS, JVM, servlet/EJB containers, application servers, portal servers etc.)

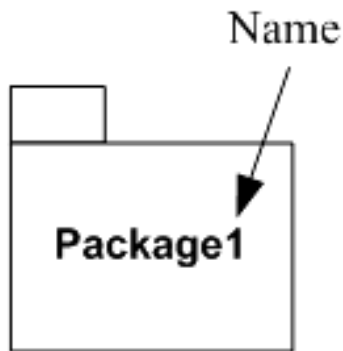


# Behavioral Things

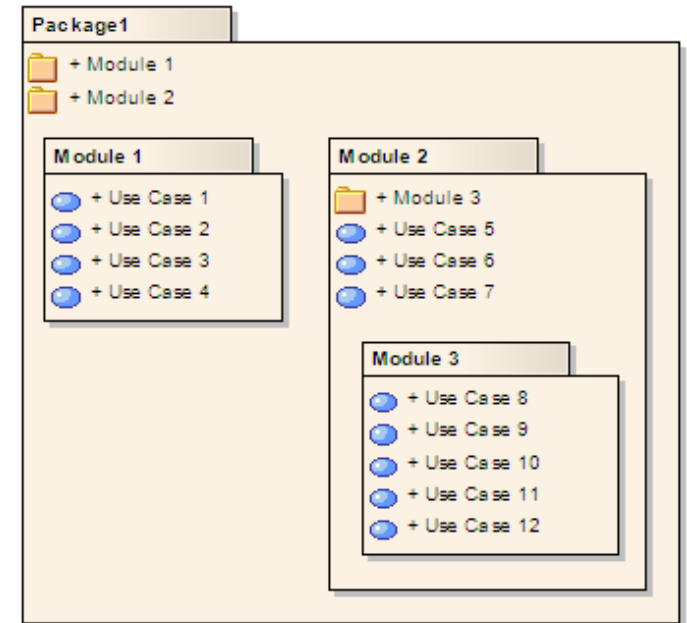
- © **Dynamic** parts are one of the most important elements in UML
- © UML has a set of powerful features to represent the dynamic part of software and non software systems
  - Sequential** (Represented by sequence diagram)
  - Collaborative** (Represented by collaboration diagram)

# Grouping Things: Package Notation

- © **Package** notation is shown below and this is used to group elements or wrap the components of a system.
- © Pretty much all UML elements can be grouped into packages. Thus, classes, objects, use cases, components, nodes, node instances etc. can all be organized as packages



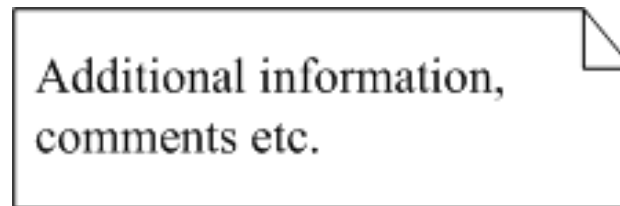
Symbol



Example

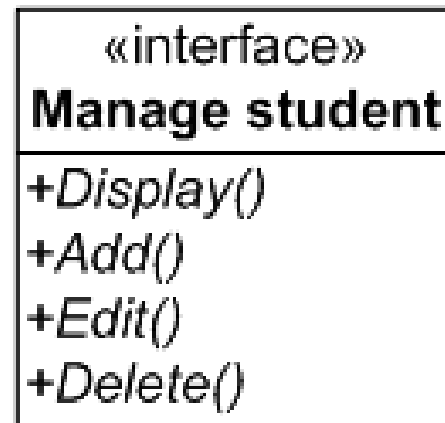
# Note Notation

© This notation is used to provide Additional information of a system.

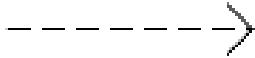







## Interface Notation:

- © An interface is a collection of operations i.e. Each interface specifies a well-defined set of operations that have public visibility.
- © Interface is used to describe functionality without implementation. Interface is just like a template where you define different functions not the implementation.

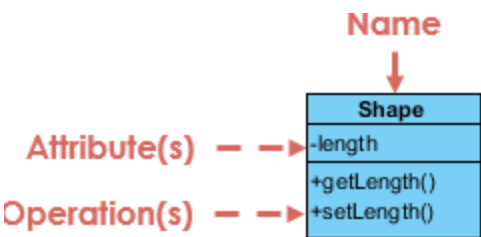




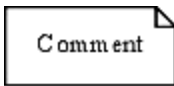


# Types of Relations in UML

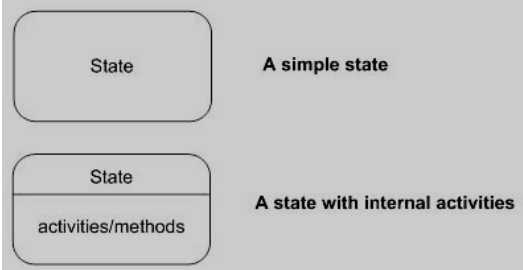
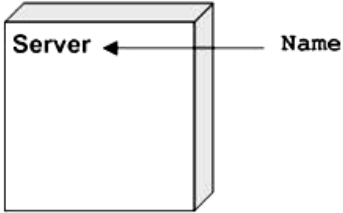

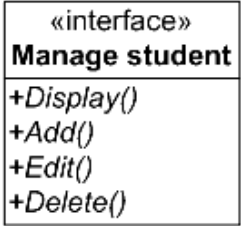
Name	Symbol	Description
<b>Dependency</b>		Changes in one model( independent) can cause changes in another model (dependent)
<b>Association</b>		-describes link between objects.
<b>-Aggregation</b>		-Part of relationship -objects are assembled to create new and more complex object.
<b>-Composition</b>		-Same as aggregation only difference is that child cannot exist independent of the parent.
<b>Generalization</b>		-combining similar classes of objects into a more general single class.
<b>Specialization</b>		-creating new sub classes from an existing class.
<b>Realization</b>		-It only inherits operations and cannot inherit attributes and association.



# Types of Things in UML

Name	Symbol	Description
<b>Class</b>	 <p>The diagram shows a class box for 'Shape'. The name 'Shape' is at the top. Below it, the attribute '-length' is listed. At the bottom, two operations are listed: '+getLength()' and '+setLength()'. Red arrows point from the labels 'Name', 'Attribute(s)', and 'Operation(s)' to their respective parts in the class box.</p>	-An object with defined attributes and operations.
<b>Use case</b>	 <p>A simple oval shape representing a use case.</p>	-A sequence of actions that a system performs that yields an observable result
<b>Actor</b>	 <p>A stick figure representing an actor.</p>	-some internal or external entity that interacts with the system
<b>Initial state</b>	 <p>A solid black circle representing the initial state.</p>	-show the starting point of a process
<b>Final state</b>	 <p>A bullseye symbol (a solid black circle inside a larger circle) representing the final state.</p>	-show the Ending point of a process
<b>Note</b>	 <p>A rectangular box with a folded corner, containing the word 'Comment'.</p>	-A symbol to display comments.

# Types of Things in UML

Name	Symbol	Description
<b>State Machine</b>		<p>- A behavior that specifies the sequences of states an object or interaction goes through during its' lifetime in response to events.</p>
<b>Node</b>		<p>-A physical element existing at run time and represents a resource. -e.g.: A server</p>
<b>Components</b>		<p>-A physical and replaceable part of a system that implements a number of interfaces. Example: a set of classes, interfaces, and collaborations.</p>
<b>Interface</b>		<p>-A collection of functions that specify a service of a class or component, i.e. externally visible behavior of that class.</p>

*Tribhuvan University*  
*Institute of Engineering*  
*Kathmandu Engineering College*  
*Department of Electronics and Computer Engineering*

## **Chapter 2 & 3: Object Oriented Analysis & Design**

20 Hrs, 35 Marks

*by*  
***Er. Santosh Giri***  
***Lecturer, IOE, Pulchowk Campus***

# *Case Studies from chapter 1*

# Case One: **The NextGen POS System**

- © A point-of-sale POS system is a computerized application used (in part) to record sales and handle payments; it is typically used in a retail store. It includes hardware components such as a computer and bar code scanner, and software to run the system. It interfaces to various service applications, such as a third-party tax calculator and inventory control. These systems must be relatively fault-tolerant; that is, even if remote services are temporarily unavailable (such as the inventory system), they must still be capable of capturing sales and handling at least cash payments (so that the business is not crippled).
- © A POS system increasingly must support multiple and varied client-side terminals and interfaces. These include a thin-client Web browser terminal, a regular personal computer with something like a Java Swing graphical user interface, touch screen input, wireless PDA(Personal Digital Assistant) , and so forth.

**More Cases:**

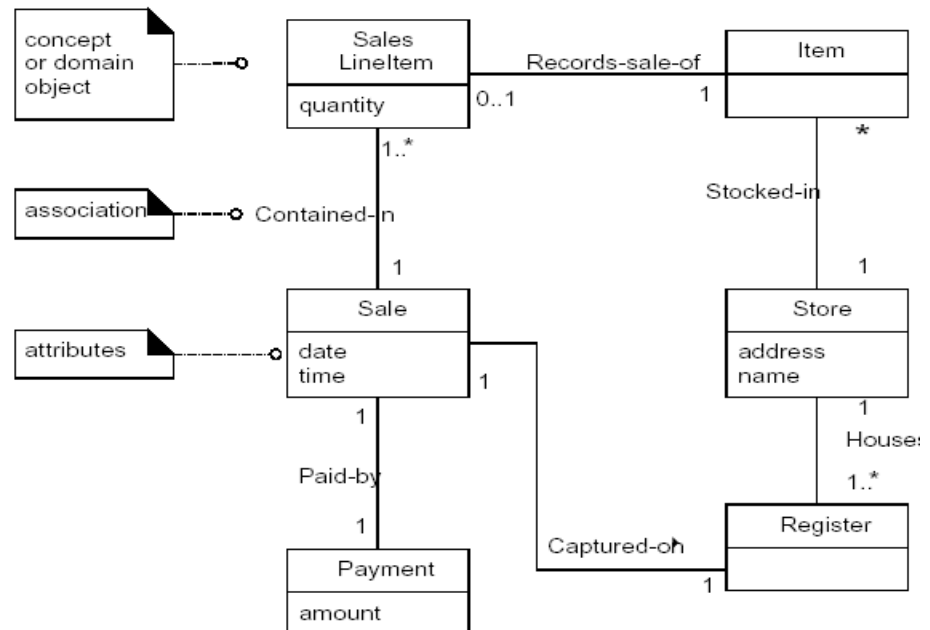
**The Monopoly Game System**  
**Airlines Reservation system**

*→ See by yourself*

# *Domain Modeling*

# Domain Modeling

- © A domain model contains conceptual classes, associations between conceptual classes, and attributes of a conceptual class.
- © A domain model in UML is illustrated with a set of class diagrams omitting the operations.
- © The domain model also identifies the relationships among all the entities within the scope of the problem domain, and commonly identifies their attributes.
- © The model is shown as a class diagram.





# Domain Modeling

## Why?

- © The domain model is created in order to represent key concepts of the problem domain.
- © The domain model can be effectively used to verify and validate the understanding of the problem domain among various stakeholders.
- © It is especially helpful as a communication tool and a focusing point both amongst the different members of the business team as well as between the technical and business teams.

## How?

Using UML notation, a domain model is illustrated with a set of class diagrams in which no operations are defined. It may show:

- ✓ domain objects or conceptual classes
- ✓ associations between conceptual classes
- ✓ Attributes.

# Domain Modeling Guidelines

→ List the candidate conceptual classes

*How?*

© *Reuse or modify the existing model if one exists*

© *Use a category list*

© *Identify noun phrases in your use-cases*

→ Draw them in a domain model using UML

→ Add the associations necessary to record relationships.

→ Add the attributes necessary to fulfill the information requirements.

# Domain Modeling Guidelines

## → *On Naming and Modeling Things*

- © Use the vocabulary of the domain **i.e. domain specific naming**, when naming conceptual classes and attributes.
  - ✓ For example, if developing a model for a library, name the customer a "Borrower" or "Patron"—the terms used by the library staff.
- © A domain model may exclude conceptual classes in the problem domain that are not relevant to the requirements.
  - ✓ For example, we may exclude Pen and PaperBag from our domain model (for the current set of requirements) since they do not have any obvious noteworthy role.

# Domain Modeling Guidelines

## → Common Mistake in Identifying Conceptual Classes

Perhaps the most common mistake when creating a domain model is to *represent something as an attribute when it should have been a concept*.

### Example:

© *should store be an attribute of Sale, or a separate conceptual class Store?*

The term suggests a legal entity, an organization, and something that occupies space.

*Therefore, Store should be a concept.*

© *As another example, consider the domain of airline reservations. Should destination be an attribute of Flight, or a separate conceptual class Airport?*

In the real world, a destination airport is not considered attributes as it is a massive thing that occupies space.

*Therefore, Airport should be a concept.*

# Finding Conceptual Classes

## Three Strategies to Find Conceptual Classes

### 1. Reuse or modify existing models.

- © This is the first, best, and usually easiest approach.
- © here we used published, well-crafted domain models and data models (which can be modified into domain models) for many common domains, such as inventory, finance, health, and so forth..

### 2. Use a category list.

### 3. Identify noun phrases.

*Reusing existing models is excellent, but outside our scope.*

# Finding Conceptual Classes

## Use a category list.

- © We can kick-start the creation of a domain model by making a list of candidate conceptual classes.
- © Figure (in next slide represented as table) contains many common categories that are usually worth considering, with an emphasis on business information system needs.
- © The guidelines also suggest some priorities in the analysis.

**Examples are drawn from:** → see C. Larman's Book for details

- © Next-Gen-POS (Point-of-Sale)
- © Monopoly
- © Airline Reservation domains.

# Finding Conceptual Classes

Use a category list (*Airlines RS*)

Conceptual Class Category	Examples
<b>business transactions</b> Guideline: These are critical (they involve money), so start with transactions.	Sale, Payment <u>Reservation</u>
<b>transaction line items</b> Guideline: Transactions often come with related line items, so consider these next.	SalesLineItem
<b>product or service related to a transaction or transaction line item</b> Guideline: Transactions are for something (a product or service).	Item <u>Flight</u> , <u>Seat</u> , Meal
<b>where is the transaction recorded?</b> Guideline: Important.	<u>Register</u> , <u>Ledger</u>
<b>roles of people or organizations related to the transaction; actors in the use case</b> Guideline: We usually need to know about the parties involved in a transaction.	Cashier, Customer, Store MonopolyPlayer <u>Passenger</u> , <u>Airline</u>

# Finding Conceptual Classes

Continue...

Conceptual Class Category	Examples
place of transaction; place of service	Store <u>Airport</u> , <u>Plane</u> , <u>Seat</u>
noteworthy events, often with a time or place we need to remember	Sale, Payment MonopolyGame <u>Flight</u>
<b>physical objects</b> Guideline: This is especially relevant when creating device-control software, or simulations.	Item, Register Board, Piece, Die <u>Airplane</u>
<b>descriptions of things</b>	ProductDescription <u>FlightDescription</u>
<b>catalogs</b> Guideline: Descriptions are often in a catalog.	ProductCatalog <u>FlightCatalog</u>
<b>containers of things (physical or information)</b>	Store, <u>Bin Board Airplane</u>
<b>things in a container</b>	Item Square (in a Board) <u>Passenger</u>
<b>other collaborating systems</b>	CreditAuthorizationSystem <u>AirTrafficControl</u>



# Finding Conceptual Classes

Continue...

Conceptual Class Category	Examples
records of finance, work, contracts, legal matters	Receipt, Ledger <u>MaintenanceLog</u>
financial instruments	Cash, <u>Check</u> , LineOfCredit <u>TicketCredit</u>
schedules, manuals, documents that are regularly referred to in order to perform work	DailyPriceChangeList <u>RepairSchedule</u>

# Finding Conceptual Classes

## Noun Phrase Identification(Method 3)

- © Identify the nouns and noun phrases in textual descriptions of a domain, and consider them as candidate conceptual classes or attributes.
- © For example, the current scenario of the *Process Sale* use case can be used.

## Main Success Scenario (or Basic Flow) for POS system:

- © **Customer** arrives at a **POS checkout** with **goods** and/or **services** to purchase.
- © **Cashier** starts a new **sale**.
- © **Cashier** enters **product identifier**.
- © **Customer** select **Item**.
- © System records **sale line item** and presents **product description, price,** and running **total**. Price calculated from a set of price rules. **Cashier** repeats steps 2-3 until it indicates done.
- © System presents total with **taxes** calculated.

# Finding Conceptual Classes

## Continue

- © Cashier tells Customer the total, and asks for **payment**.
- © Customer pays and System handles payment.
- © System logs the completed **sale** and sends sale and payment information to the external **Accounting** “*Ledger*” (for accounting and **commissions**) and **Inventory** “*Register*” systems (to update inventory).
- © System presents **receipt**.
- © Customer leaves with receipt and goods

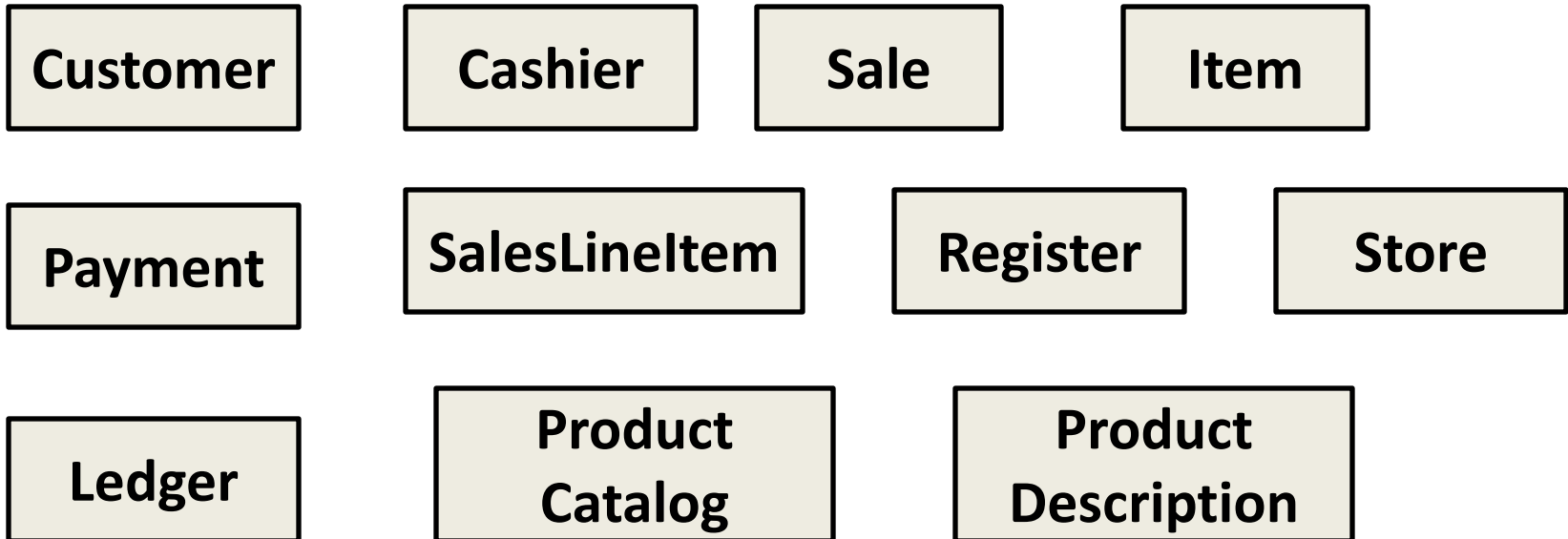
# Find and draw Conceptual Classes

## Case Study: POS Domain

- © From the category list and noun phrase analysis, a list is generated of candidate conceptual classes for the domain.

**Note:** There is no correct list of conceptual classes

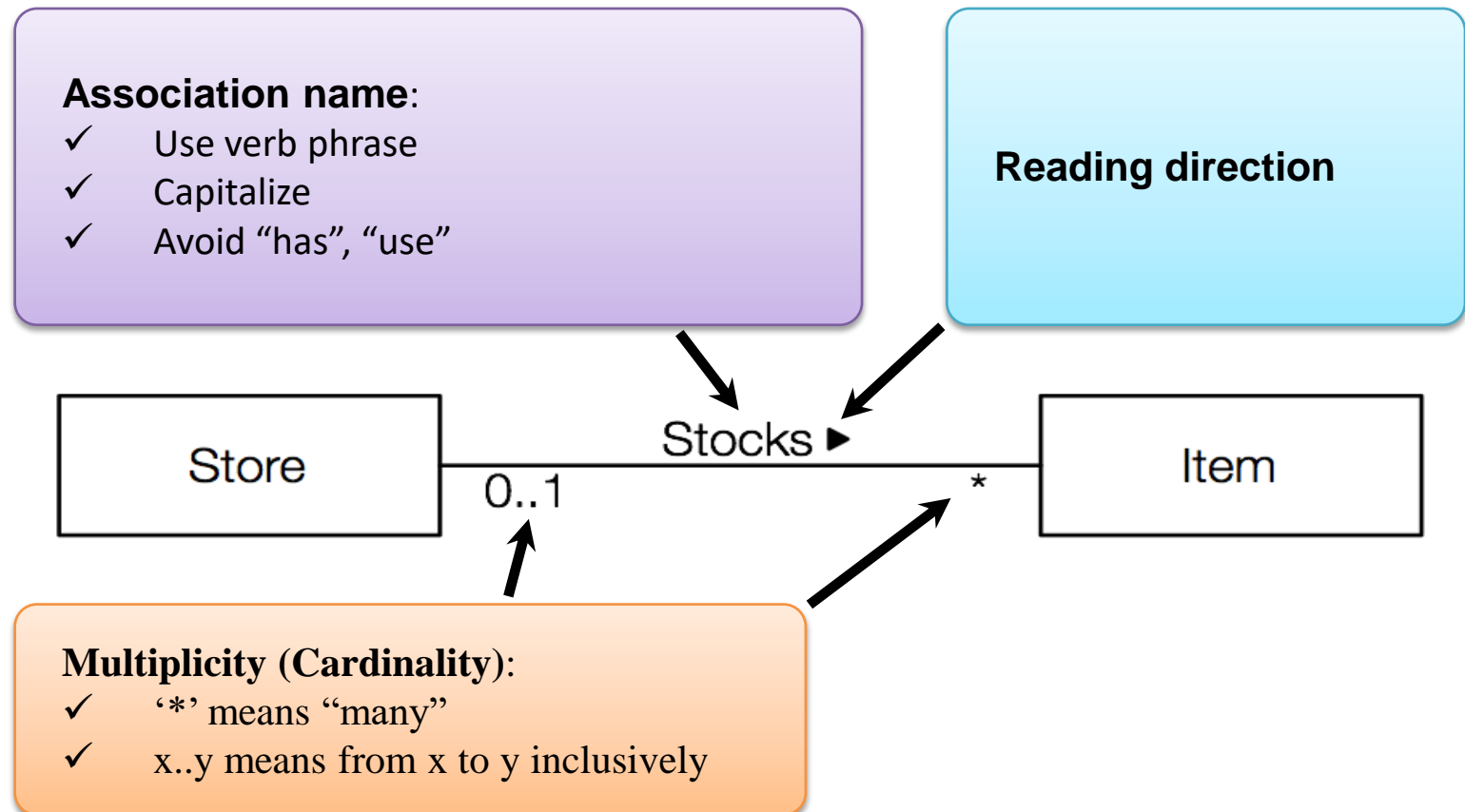
### List of conceptual classes of initial POS domain model



# Adding Associations and Attributes

## Adding Associations

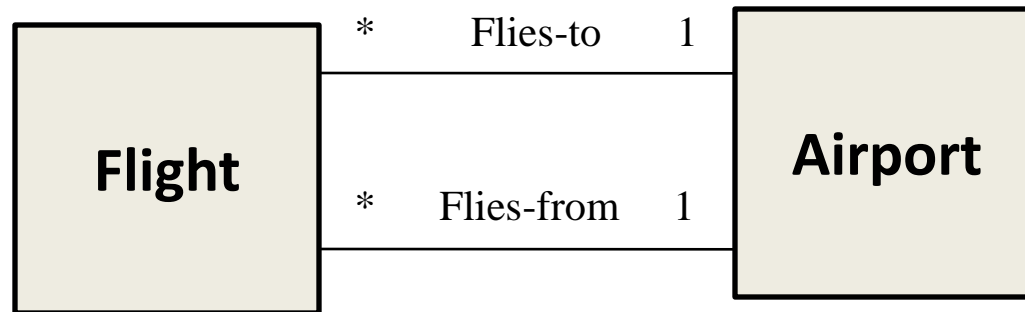
- © An association is a relationship between classes that indicates some meaningful and interesting relationship.



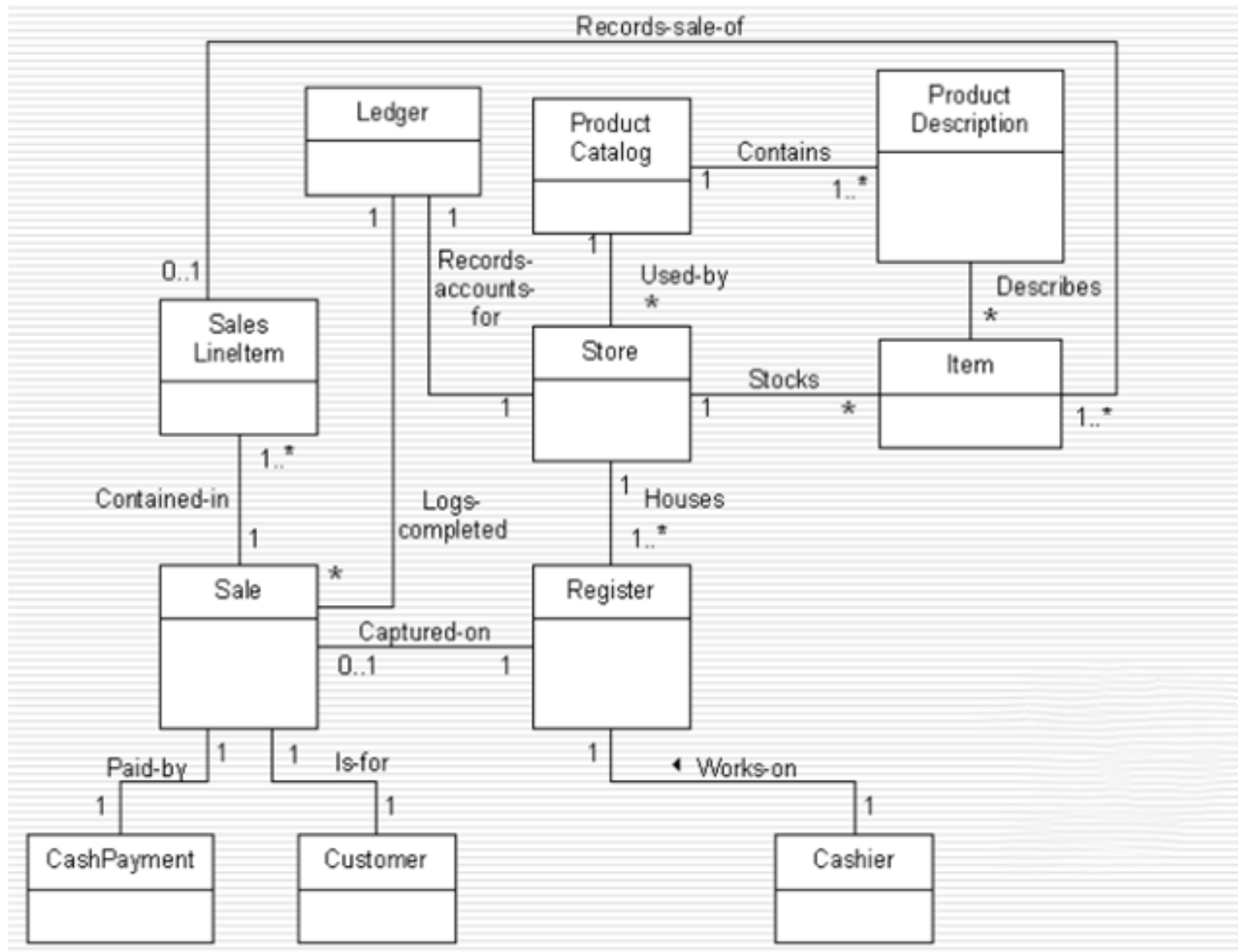
# Adding Associations and Attributes

## Multiple Associations between two classes:

- © Two classes may have multiple associations between them in a UML class diagram; this is not uncommon.
- © There is no outstanding example in the POS or Monopoly case study.
- © but an example from the domain of the airline is the relationships between a *Flight* and an *Airport* (see Figure); the flying-to and flying-from associations are distinctly different relationships, which should be shown separately.



# POS with associations



# Adding Associations and Attributes

## Adding attributes

It is useful to identify those attributes of conceptual classes that are needed to satisfy the information

## When to Show Attributes?

Include attributes that the requirements (for example, use cases) suggest or imply a need to remember information.

### For example, in POS:

- © A receipt (which reports the information of a sale) in the *POS* normally includes a date and time
- © The store includes name and address and
- © The cashier includes ID, among many other things.

## Therefore,

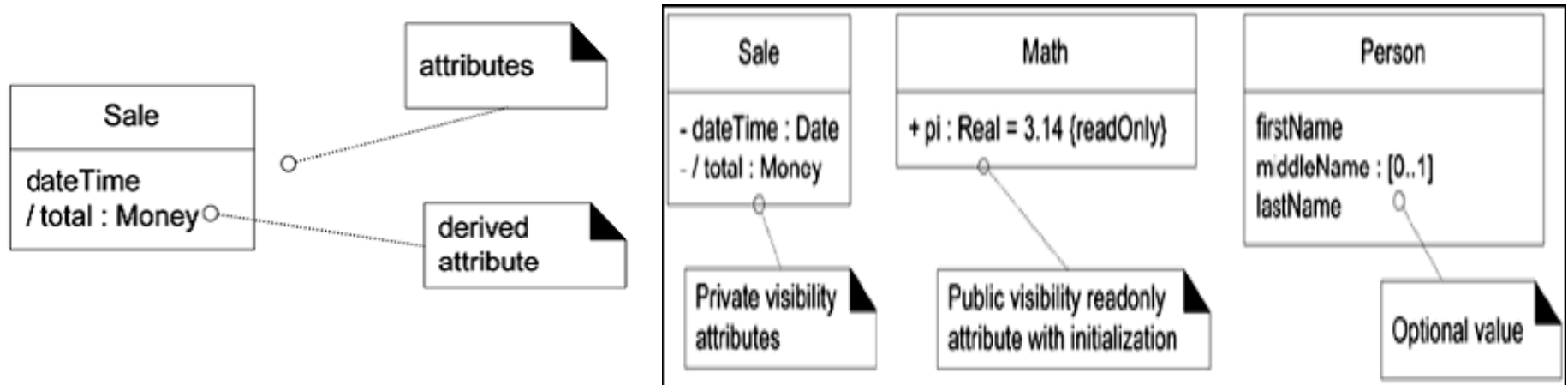
- © *Sale* needs a *dateTime* attribute.
- © *Store* needs a *name* and *address*.
- © *Cashier* needs an *ID*.



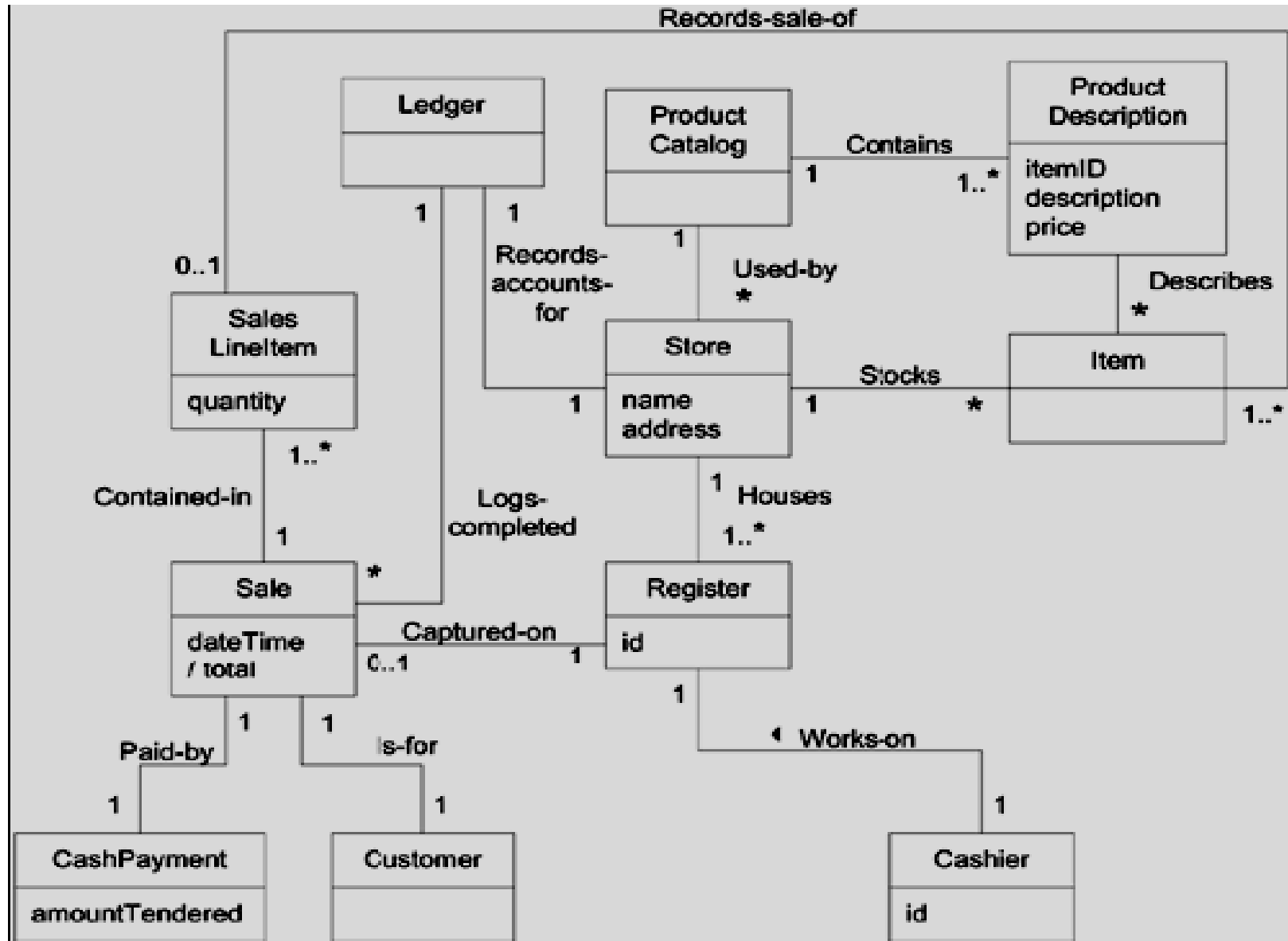
# Adding Associations and Attributes

## Attributes Notations in UML

- © The expression *middleName* : [0..1] indicates an optional value 0 or 1 are present.
- © For derived attributes we use the UML convention (a forward slash '/' symbol before the attribute name)



# Domain model for POS after adding attributes



*UML*

*Use Case Diagram*

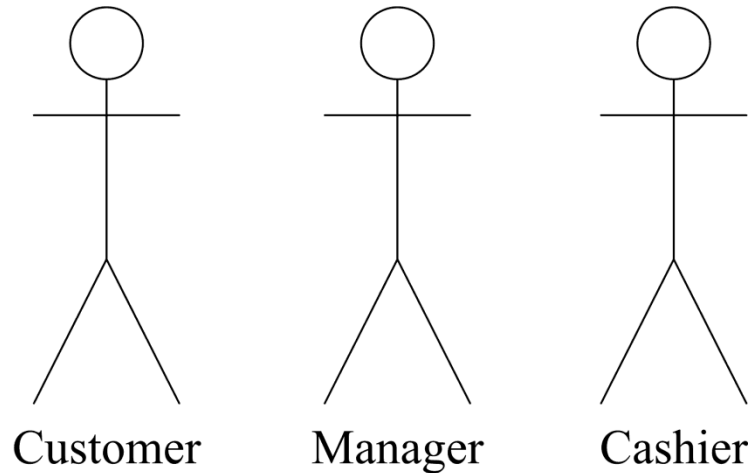
# UML Use Case Diagram

- ❌ Use case is used to model the system or subsystem of an application.
- ❌ A single use case diagram captures a particular functionality of a system.
- ❌ The purpose of use case diagrams can be as follows:
  - © Used to gather **requirements of a system**
  - © Used to get an **outside view** of a system
  - © Identify **external factors** influencing the system
  - © Show the **interaction among the requirements** also called use cases.
- ❌ To draw a use case diagram we should have following items identified or components of use case diagrams are:
  - © Actor
  - © Use case
  - © System boundary
  - © Relationship

# Use Case Diagram

## Actor

- © An actor is someone or something that must interact with the system under development.
- © An UML notation of Actor is represented as:



- © Actors are not part of the system they represents anyone or anything that must interact with the system.
- © An single actor may perform more than one use cases (functionality)

# Use Case Diagram

- ❌ An actor may be:
  - © Input information to the system
  - © Receive information from the system
  - © Input to and out from the system.

- ❌ How do we find the actors?

**Ask the following questions:**

- © Who uses the system?
- © Who install the system?
- © Who starts up the system?
- © What other system use this system?
- © Who gets the information to the system?

**Note:** *An actor is always external to the system.*

# Use Case Diagram

Categories of Actor:

- © **Principle**

who uses the main system functions

- © **Secondary**

who takes care of administration and maintenance

- © **External h/w**

The h/w devices which are part of application domain and must be used

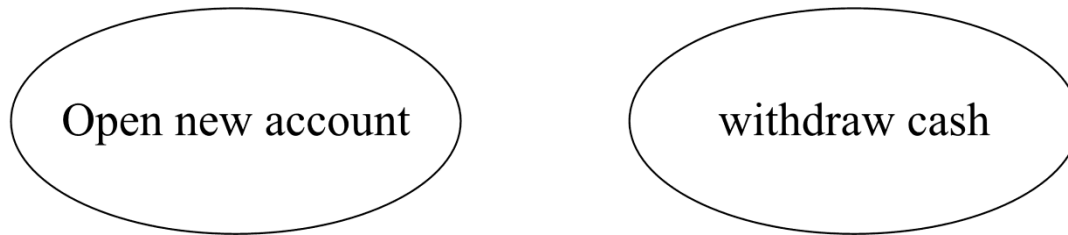
- © **Other system**

The other system with which the system must interact.

# Use Case Diagram

## Use cases:

- © Use cases represents functionality of a system
- © which are the specific roles played by the actors within and around the system



## How do we find the use cases?

- © What functions will the actor want from the system?
- © Does the system store information? If yes then which actors will create, read, update or delete that information?
- © Does the system need to notify an actor about changes in its internal state?



# Use Case Diagram

## Generic format for documenting an use case

Pre condition:	if any
Use case:	name of the use case
Actors:	list of actors, indicating who initiates the use case
Purpose:	intention of the use case.
Overview:	Description.
Type:	primary/secondary.
Post condition:	if any

# Use Case Diagram

## Example:

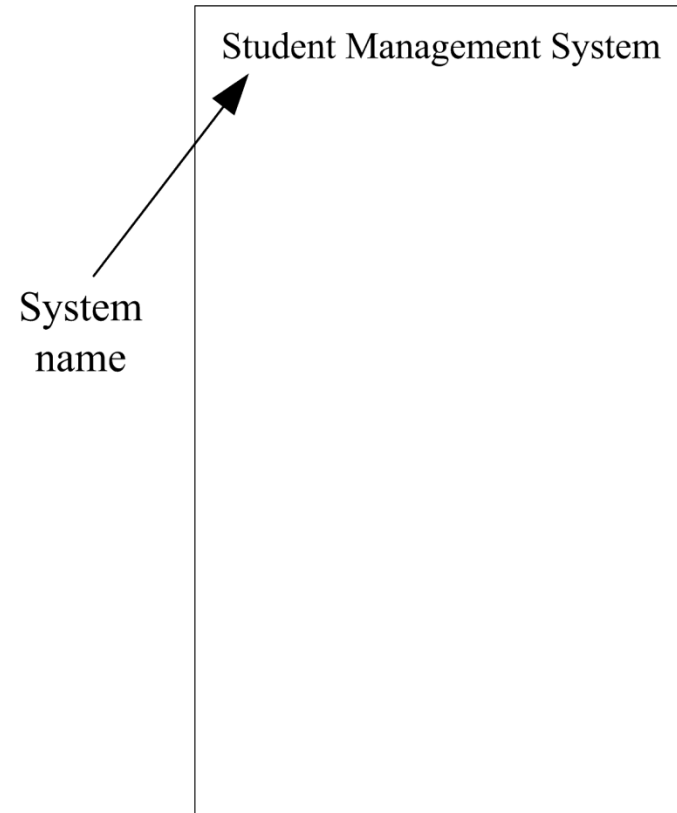
### Description of opening a new account in the bank

Use case	Open new account
Actors	Customer, Cashier, Manager
Purpose	Like to have new saving account.
Description	<p>A customer arrives in the bank to open the new account.</p> <p>Customer requests for the new account form, fill the form and submits, along with the minimal deposit.</p> <p>At the end of complete successful process customer receives the passbook.</p>

# Use Case Diagram

## System boundary:

- © It helps to identify what is an external verse internal.
- © External environment is represented only by actors.
- © Represented as a rectangle.



# UML Use Case Diagram

## Relationship:

Relationship between use case and actor

© communicates

Relationship between two use cases

© Include

© extend

Notation used to show the include and extend relationship

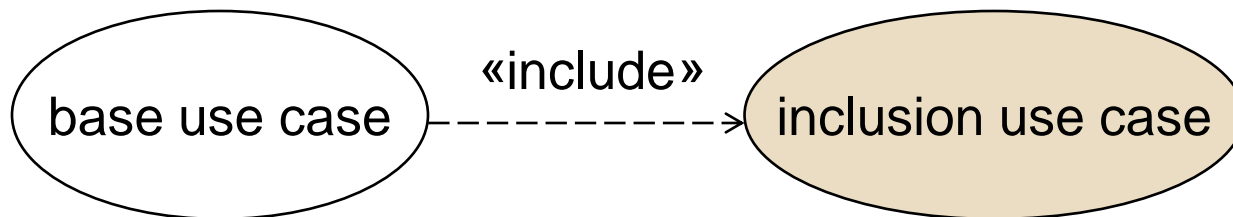
«include»  
----->

«extend»  
----->

# Use Case Diagram

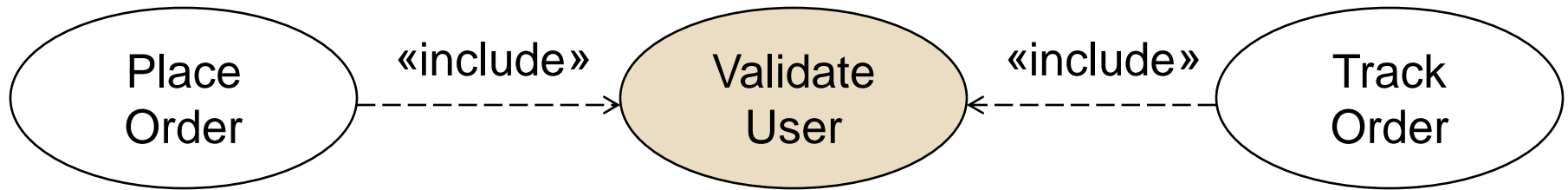
## «include» Relationship:

- © A use cases may contain the functionality of another use case.
- © It is used to show how the system can use a pre existing components
- © An include relationship is a relationship in which one use case (the base use case) includes or uses the functionality of another use case (the inclusion use case).
- © Represented as a dashed line with an open arrow pointing from the base use case to the inclusion use case. The keyword «include» is attached to the connector.



# Use Case Diagram

## «include» relationship example:

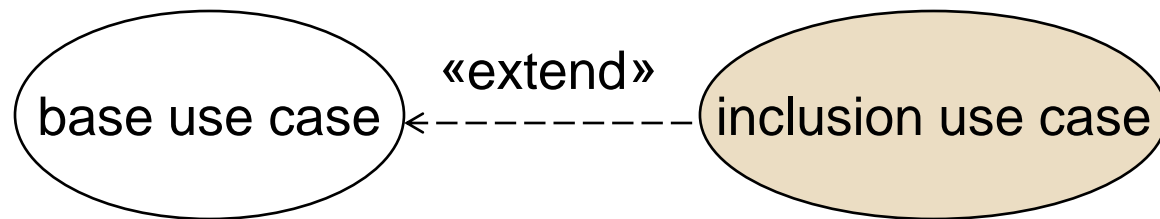


- © The following figure illustrates an restaurant order management system that provides customers with the option of placing orders as well as tracking orders. This behavior is modeled with two base use cases called *PlaceOrder* & *TrackOrder* that has an inclusion use case called *ValidateUser*. The *ValidateUser* use case is a separate inclusion use case because it contains behaviors that several other use cases in the system use. That include relationship points from the *PlaceOrder* & *TrackOrder* use cases to the *ValidateUser* use case indicate that, the *PlaceOrder* & *TrackOrder* use cases always includes the behaviors of the *ValidateUser* use case.

# UML Use Case Diagram

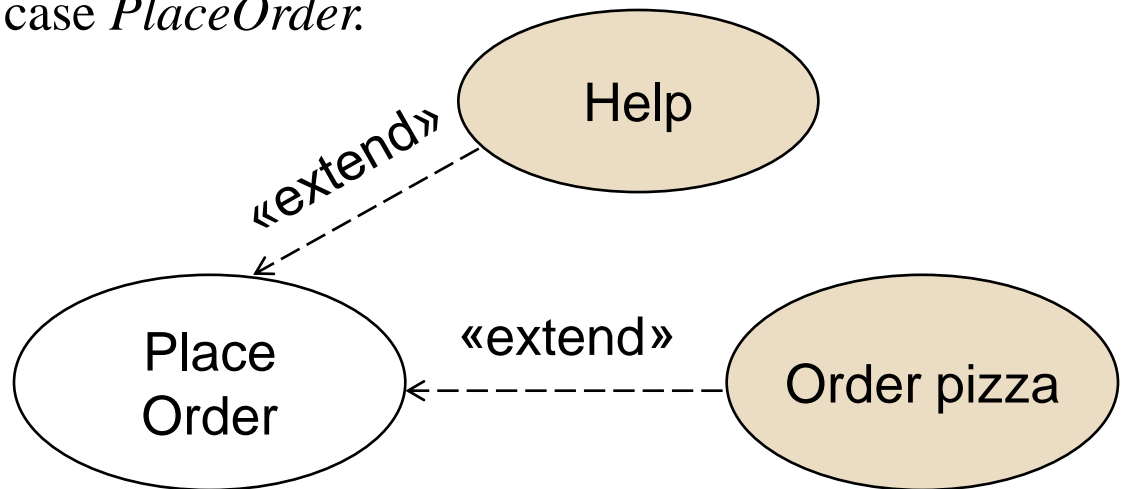
## «extend» relationship

- © Used to show optional behavior, which is required only under certain condition. This is typically used in exceptional circumstances.
- © Represented as dotted line labeled «extend» with an arrow toward the base case.



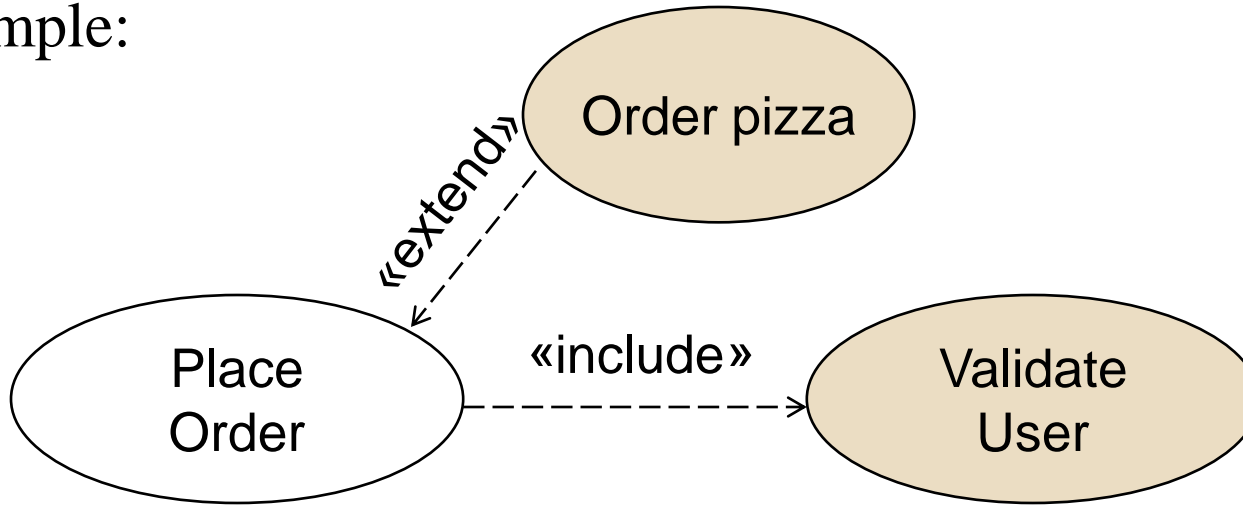
## Example:

Here in following example the *OrderPizza & Help* use cases are optional to base use case *PlaceOrder*.



# Includes vs. Extend

Example:



Key Points:

	<b>include</b>	<b>extend</b>
Is this use case optional?	<i>No</i>	<i>Yes</i>
Is the base use case complete without this use case?	<i>No</i>	<i>Yes</i>
Is the execution of this use case conditional?	<i>No</i>	<i>Yes</i>
Does this use case change the behavior of the base use case?	<i>No</i>	<i>Yes</i>



# Description: Withdraw money from ATM.

**Use case Scenario name:** Withdraw money from ATM.

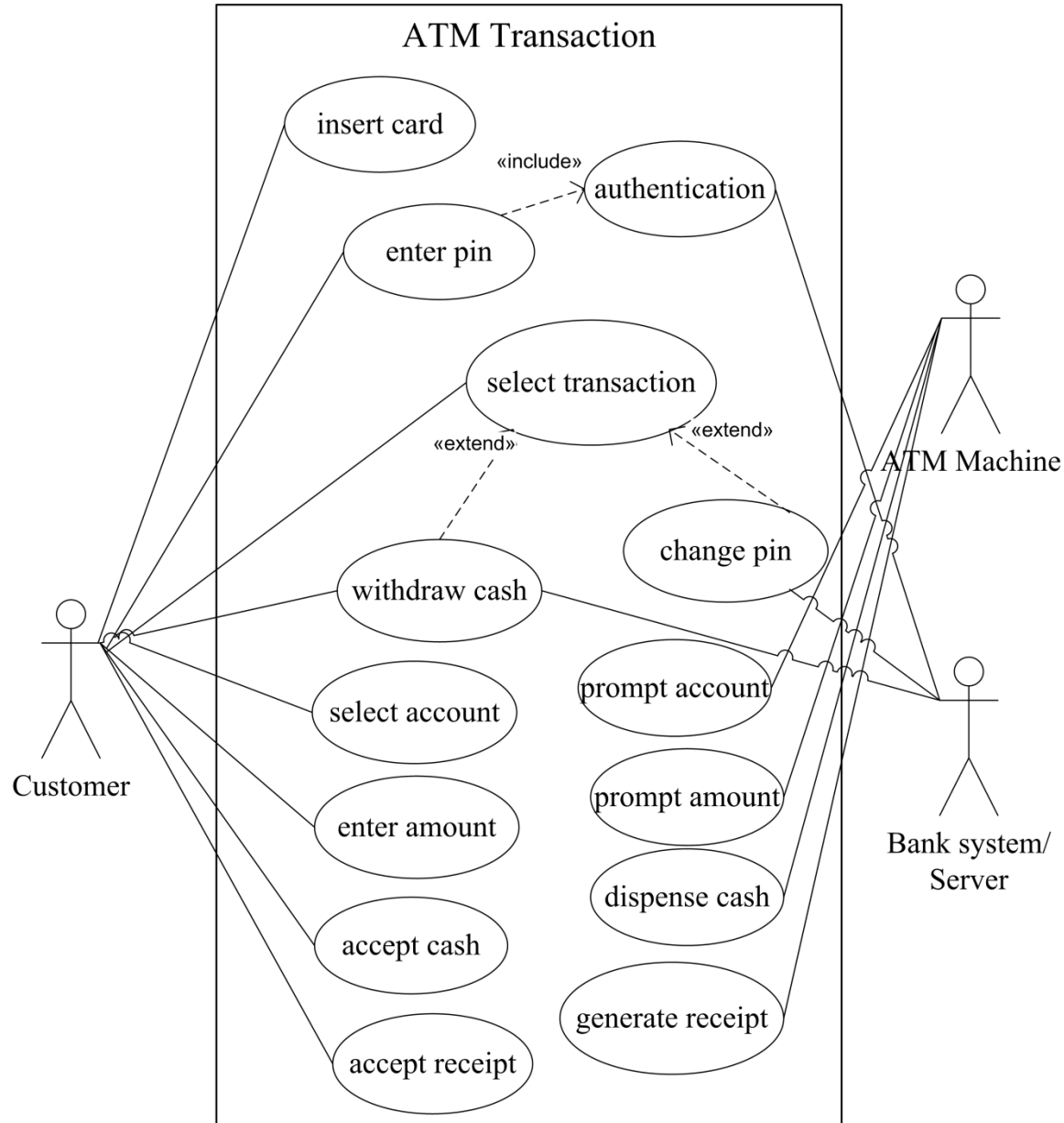
**Participating actors:** Customer  
ATM Machine  
Bank

**Preconditions:** Network connection is active  
ATM has available cash

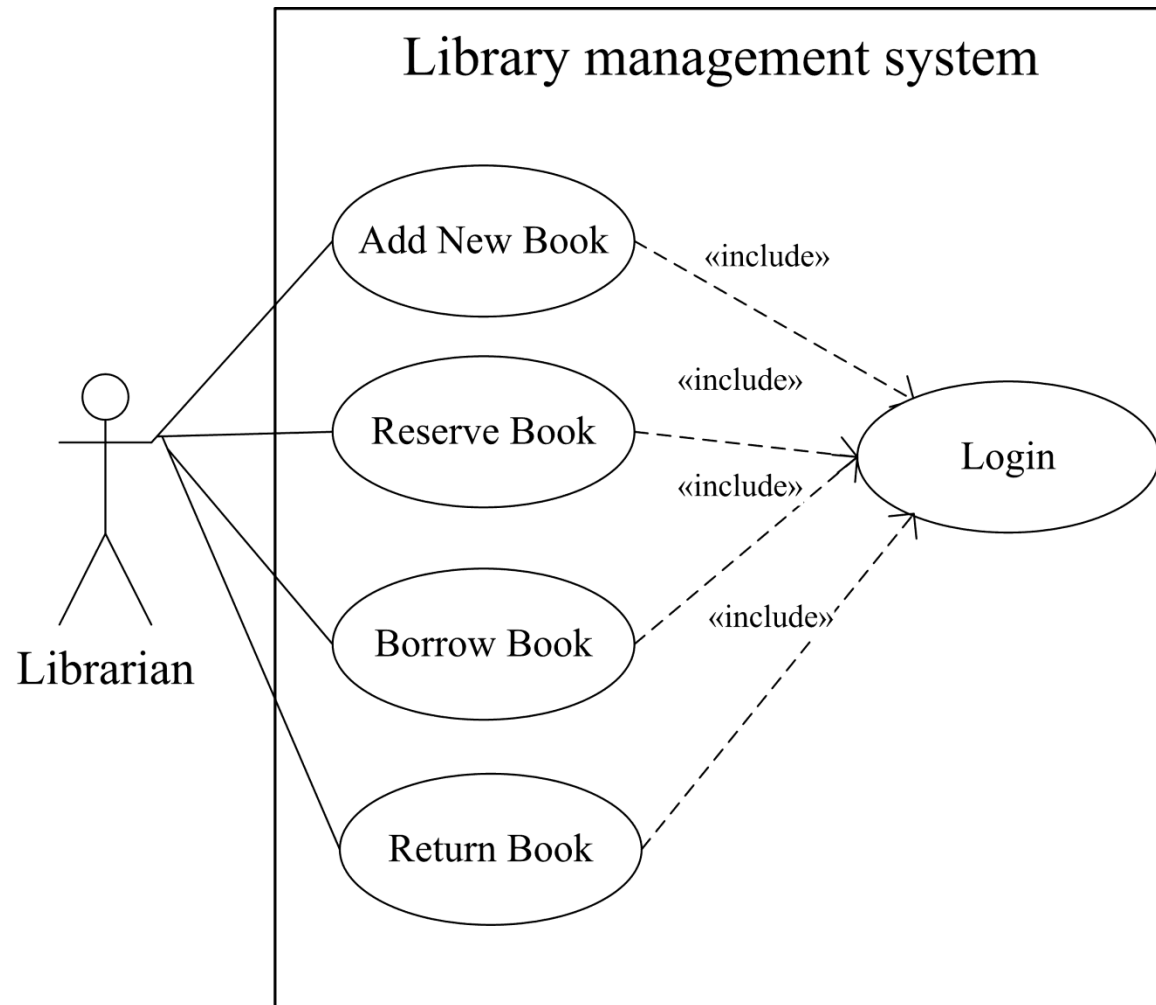
## Flow of events:

Bank customer inserts ATM card and enters PIN.  
Customer is validated.  
ATM displays actions available on ATM unit. Customer selects Withdraw Cash.  
ATM prompts account.  
Customer selects account.  
ATM prompts amount.  
Customer enters desired amount.  
Information sent to Bank, inquiring if sufficient funds/allowable withdrawal limit.  
Money is dispensed and receipt prints.

# ATM Transaction Example



# Library Management System



# Example

A coffee vending machine dispenses coffee to customers. Customers order coffee by selecting a recipe from a set of recipes. Customers pay using coins. Change is given back if any to the customer. The service staff loads ingredients into machine. The service staff can also add a recipe by indicating name of coffee, units of coffee powder, milk, sugar, water and chocolate to be added as well as the cost of coffee.

## Actors:

Customer, Service staff

## Use Cases:

Dispense coffee

Order

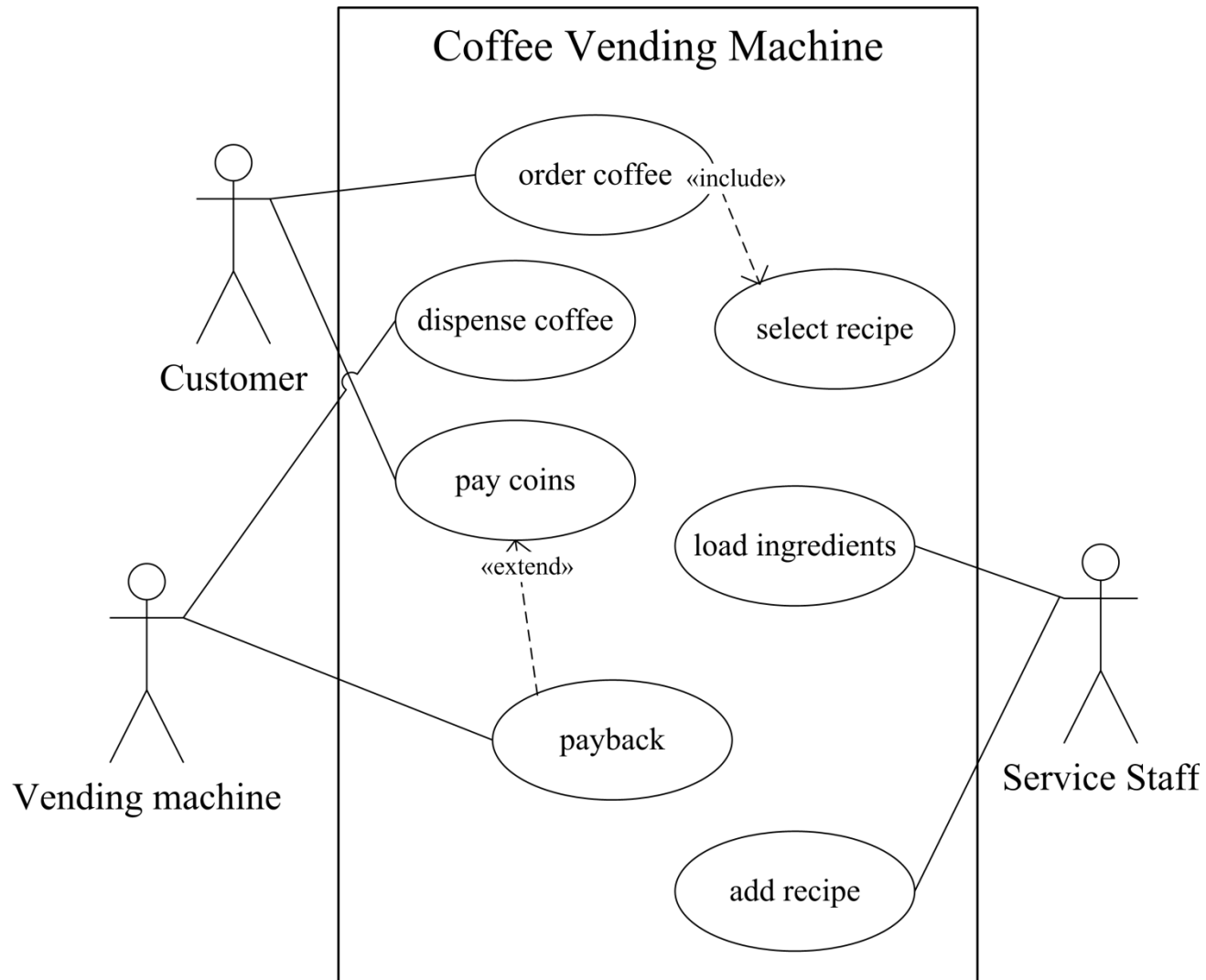
Pay coins

Payback

Load ingredients

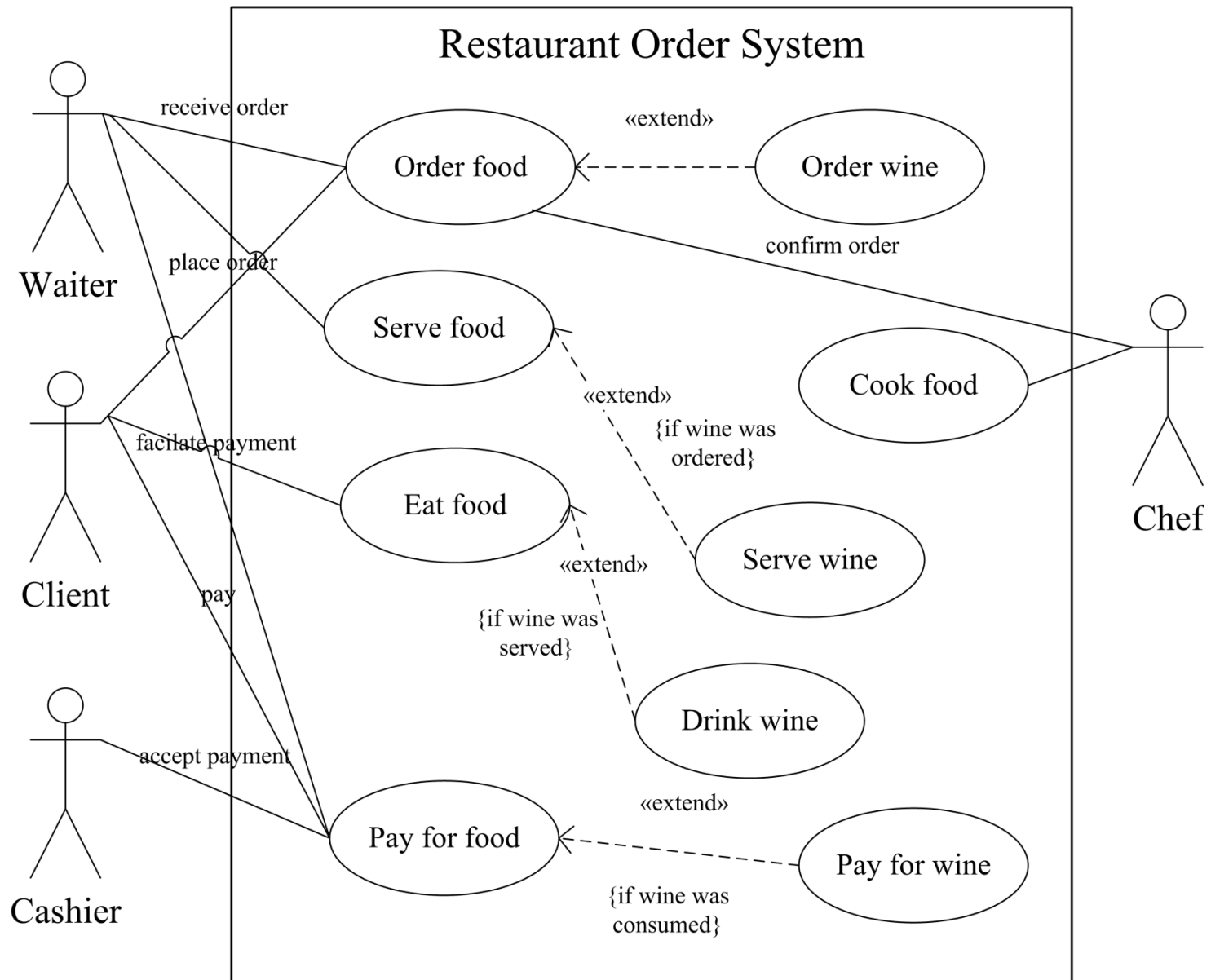
Add recipe

# Use Case Diagram Example



# Restaurant order System (Class Work)

# Restaurant order System (Class Work)



# *UML*

## *System Sequence*

### *Diagram*

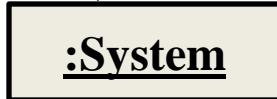


# System Sequence Diagram

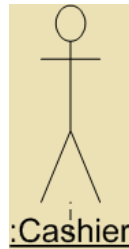
- ❌ SSD is used to visualize a use case i.e. it shows interaction between system and actors.

**For a use case scenario, an SSD shows:**

- © The System (as a black box):



- © The external actors that interact with System



- © The System events that the actors generate.

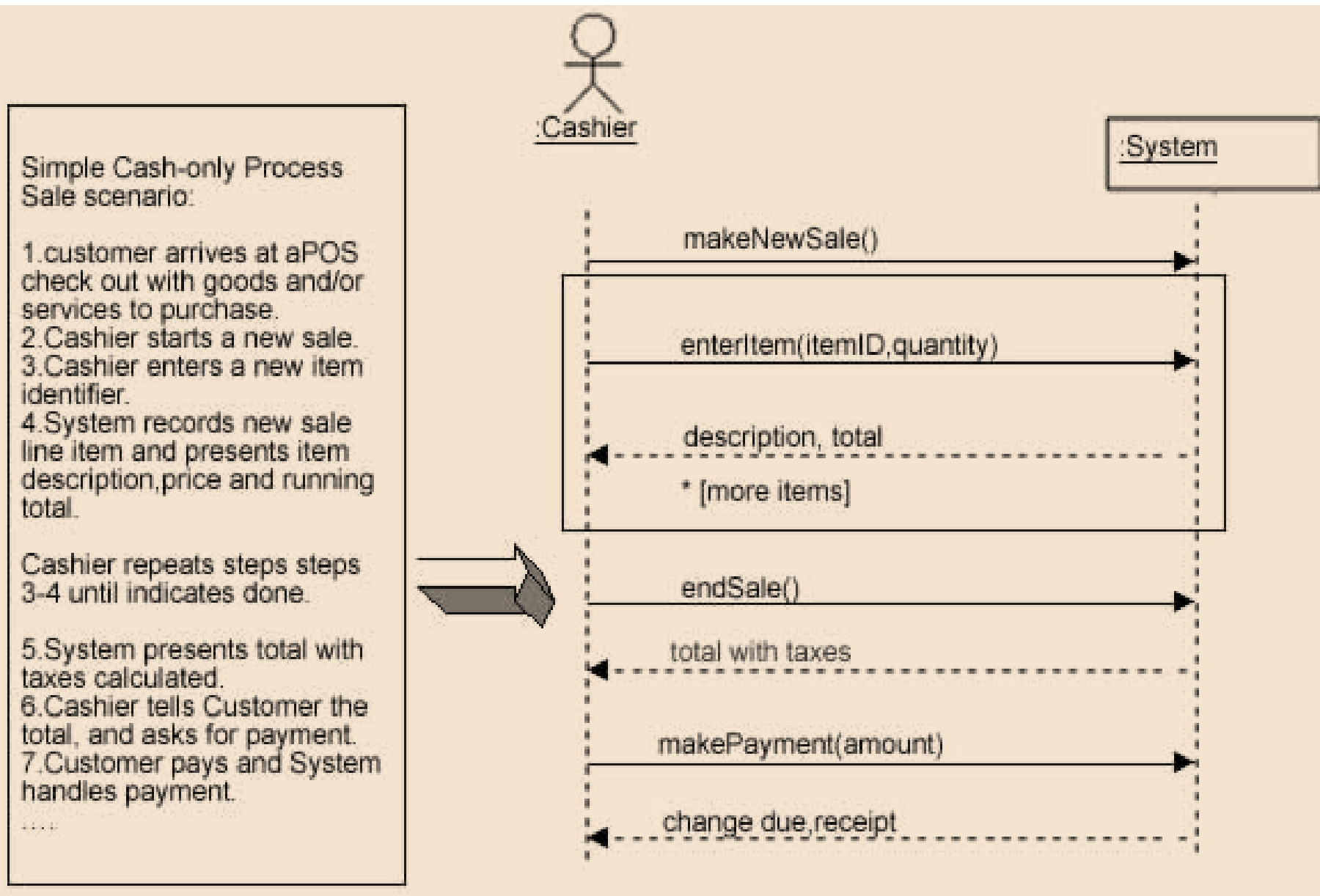
- ❌ SSD shows operations of the System in response to events, in temporal order
- ❌ Develop SSDs for the main success scenario of a selected use case, then frequent and salient alternative scenarios

# From Use case to SSD

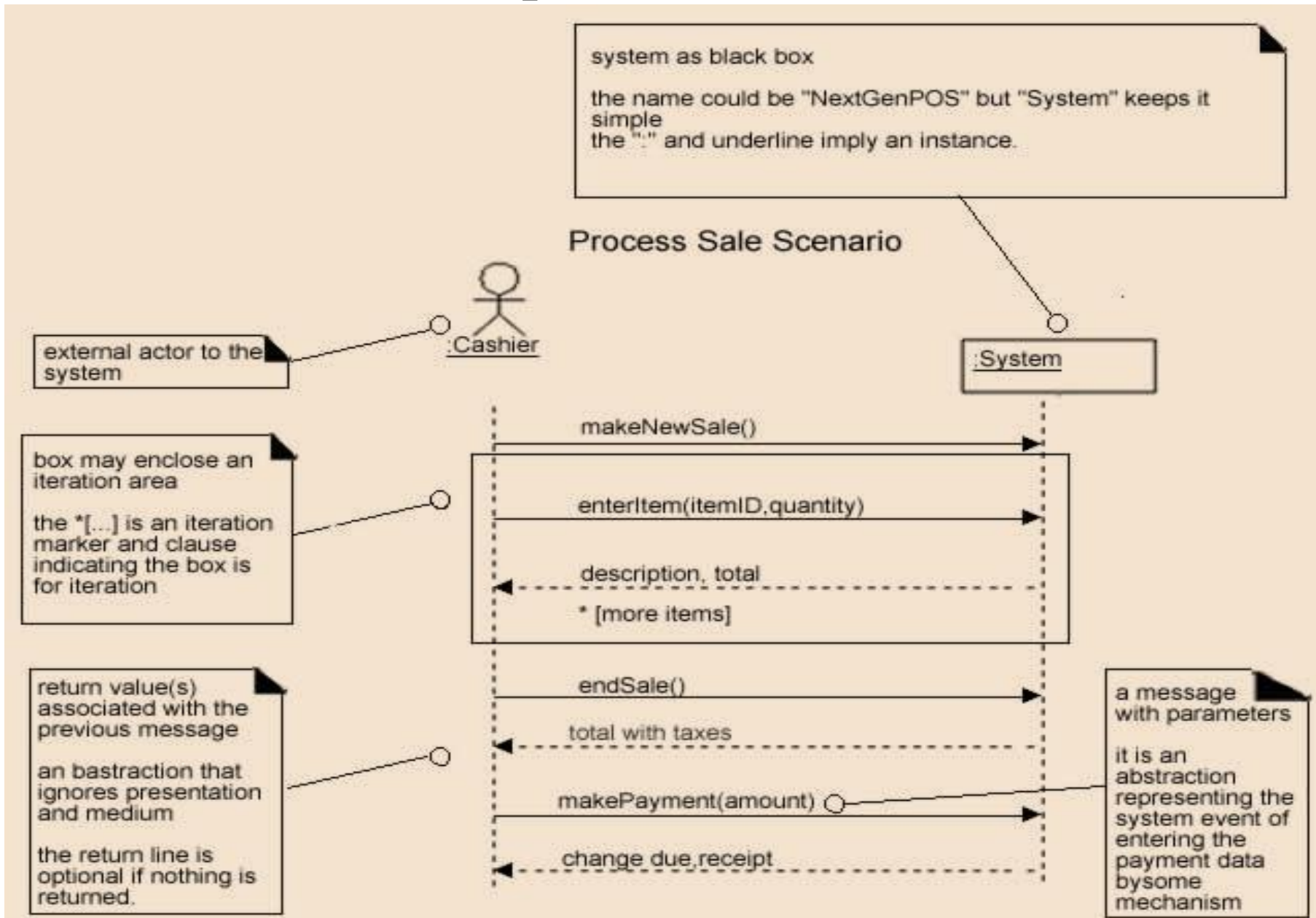
## How to construct an SSD from a use case:

- ❌ Draw System as black box on right side
- ❌ For each actor that directly operates on the System, draw a stick figure and a lifeline.
- ❌ For each System events that each actor generates in use case, draw a message.
- ❌ Optionally, include use case text to left of diagram.

# Example: use cases to SSD (Process Sale scenario )→ Larman, page 175

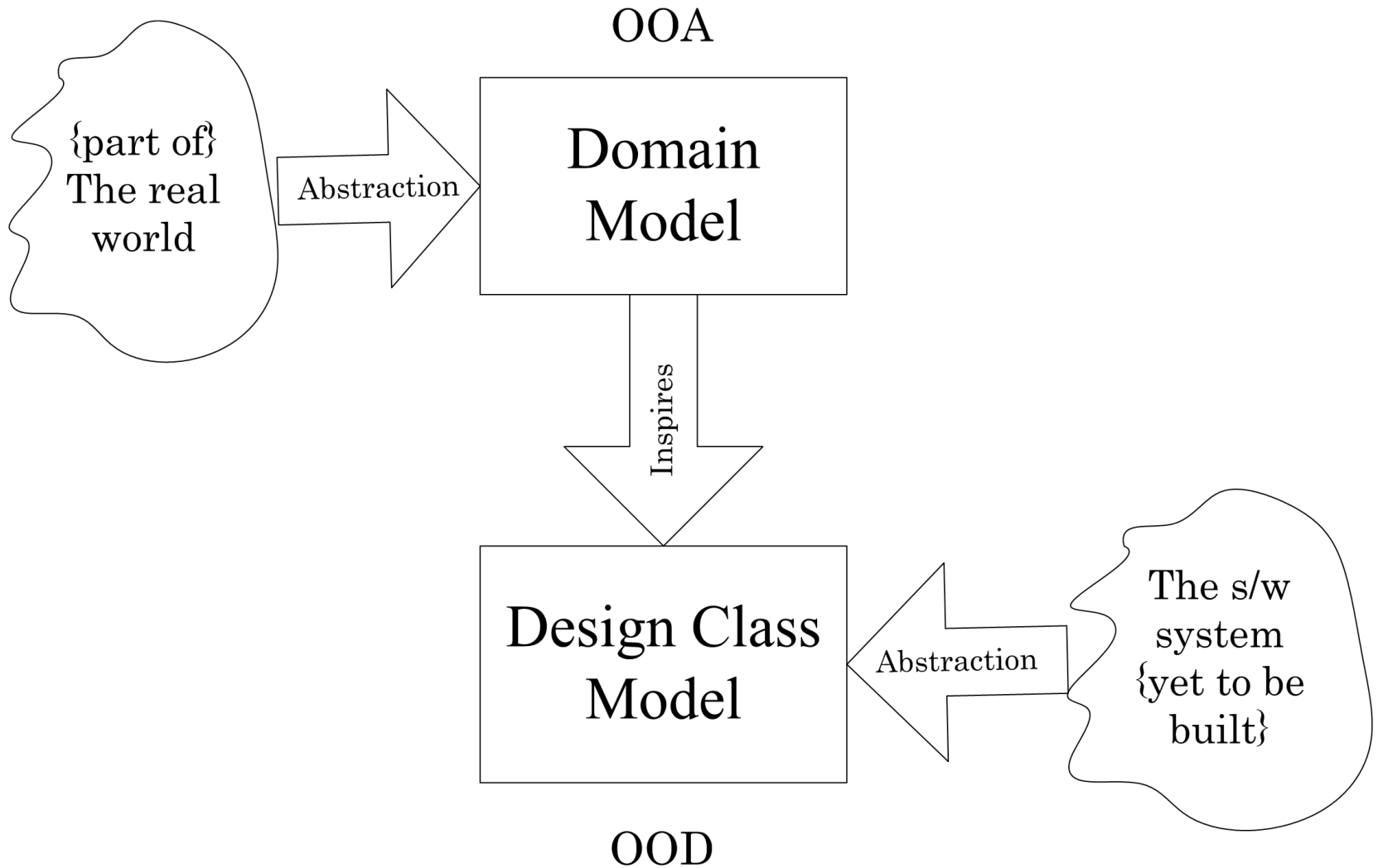


# Description Process Sale



*Analysis*  
*To*  
*Design*

# OOA to OOD



*Fig: Transition from OOA to OOD*

# OOA to OOD

## **Input (sources) for object-oriented design**

The input for object-oriented design is provided by the output of object-oriented analysis. Realize that an output artifact does not need to be completely developed to serve as input of object-oriented design. Analysis and design may occur in parallel, and in practice the results of one activity can feed the other in a short feedback cycle through an iterative process. Both analysis and design can be performed incrementally, and the artifacts can be continuously grown instead of completely developed in one shot.

Some typical input artifacts for object-oriented design are:

### **❖ Conceptual model**

Conceptual model is the result of object-oriented analysis, it captures concepts in the problem domain. The conceptual model is explicitly chosen to be independent of implementation details, such as concurrency or data storage.

# OOA to OOD

## ❖ Use case

- © Use case is a description of sequences of events that, taken together, lead to a system doing something useful. Each use case provides one or more scenarios that convey how the system should interact with the users called actors to achieve a specific business goal or function.
- © Use case actors may be end users or other systems. In many circumstances use cases are further elaborated into use case diagrams. Use case diagrams are used to identify the actor (users or other systems) and the processes they perform.

## ❖ System Sequence Diagram

- © System Sequence diagram (SSD) is a picture that shows interaction between actors and system (as a abstract), for a particular scenario of a use case, the events that external actors generate, their order, and possible inter-system events.



# OOA to OOD

## ❖ **User interface documentations (if applicable)**

© Document that shows and describes the look and feel of the end product's user interface. It is not mandatory to have this, but it helps to visualize the end-product and therefore helps the designer.

## ❖ **Relational data model (if applicable)**

© A data model is an abstract model that describes how data is represented and used. If an object database is not used, the relational data model should usually be created before the design, since the strategy chosen for object-relational mapping is an output of the OO design process. However, it is possible to develop the relational data model and the object-oriented design artifacts in parallel, and the growth of an artifact can stimulate the refinement of other artifacts.

# OOA to OOD

## Output (deliverables) of object-oriented design

Some typical deliverables artifacts for object-oriented design are:

### ❖ Interaction diagram (Sequence Diagrams/Collaboration)

- © Extend the System Sequence Diagram to add specific objects that handle the system events. A sequence diagram shows, as parallel vertical lines, different processes or objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur.

### ❖ Design Class diagram

- © A class diagram is a type of static structure UML diagram that describes the structure of a system by showing the system's classes, their attributes, and the relationships between the classes. The messages and classes identified through the development of the sequence diagrams can serve as input to the automatic generation of the global class diagram of the system.

*UML*

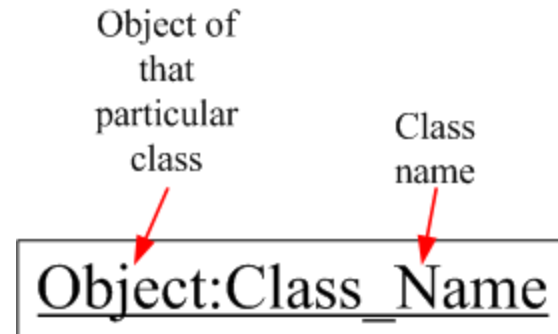
*Sequence Diagram*

# Sequence Diagram

- ❖ Sequence diagram is interaction diagram that shows message exchanged or interaction between objects in the system.
- ❖ It mainly emphasizes on time ordering of messages between objects.
- ❖ It is used to illustrate the dynamic view of the system

## Object or participants:

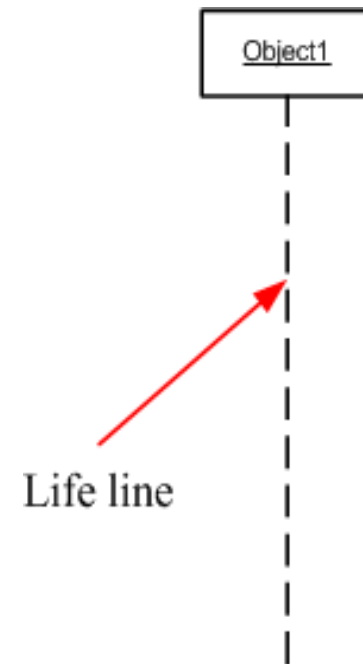
- ✓ The sequence diagram is made up of collection of **participants or objects**. Participants are system parts that interact each other during sequence diagram.
- ✓ The participants interact with each other by sending and receiving message.
- ✓ The object is represented by as:



# Sequence Diagram

## Lifeline:

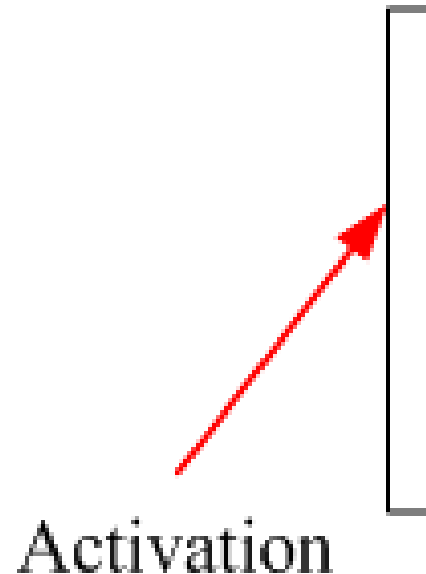
- ❖ Lifeline represents the existence of an object over a period of time.
- ❖ It is represented by vertical dashed line
- ❖ Most objects that appeared in 'Interaction diagram' will be in existence for the duration of an interaction. So, these objects are aligned at top of diagram with their lifeline from top to bottom of diagram.



# Sequence Diagram

## Activation bar:

- ❖ It is represented by tall thin rectangle.
- ❖ The top of rectangle is aligned with start of the action.
- ❖ The bottom is aligned with its completion and can be marked by a written message
- ❖ It is also called as focus of control. It shows the period of time during which an object is performing an action or operation.



# Sequence Diagram

## Messages:

- ❖ Messages can be flow in whatever direction required for interaction from left to right and right to left.
- ❖ The messages on sequence diagram are specifies using an arrow from participant that wants to pass the messages to the participant that receive the messages.
- ❖ The interaction in a sequence diagram between the objects can be shown by using messages.

## Message types:

### 1) Asynchronous messages:-

- ✓ It is a message where the sender is not blocked and can continue executing.
- ✓ Representing by solid line with open arrowhead.

Asynchronous message



# Sequence Diagram

## 2) Synchronous messages

- ❖ It is a message where the sender is blocked and waits until the receiver has finished processing of message.
- ❖ It is invoked the caller waits for the receiver to return from the message invocation.
- ❖ It is represented by solid line with filled arrowhead.

Synchronous message





# Sequence Diagram

## 3) Reflexive messages/self message:-

- ❖ If the object sends the message to itself then it is called as 'Reflexive message'.
- ❖ It is represented by solid line with loops the lifeline of object.

Self call



## 4) Return messages:-

- ❖ It can be used at the end of activation bar to show that control flow of activation returns to the participant that pass the original message.
- ❖ It is represented by dashed line from sender to receiver.

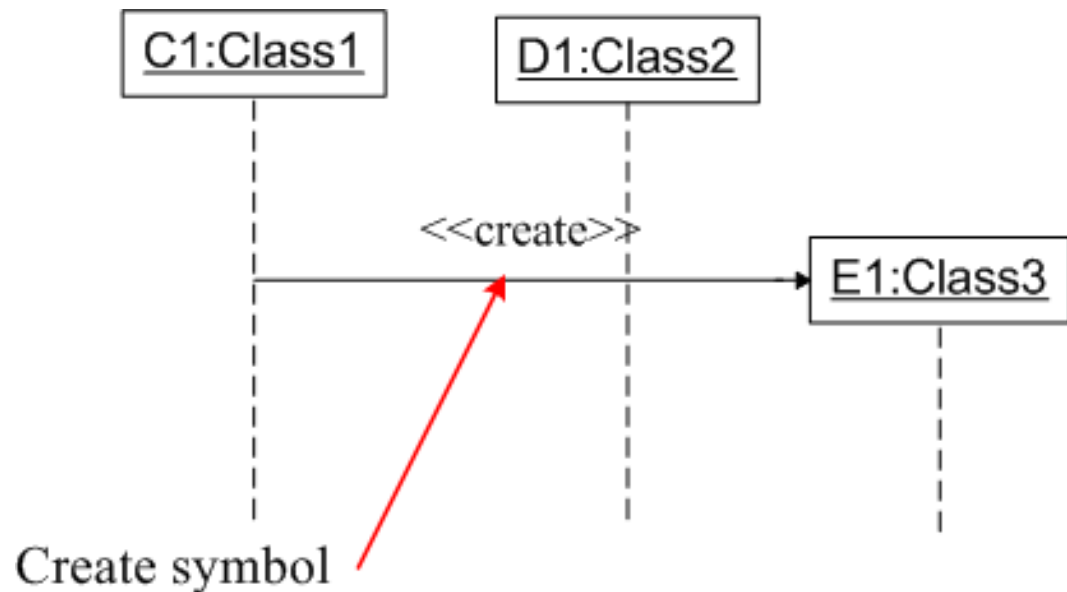
Return message



# Sequence Diagram

## 5) Create messages:

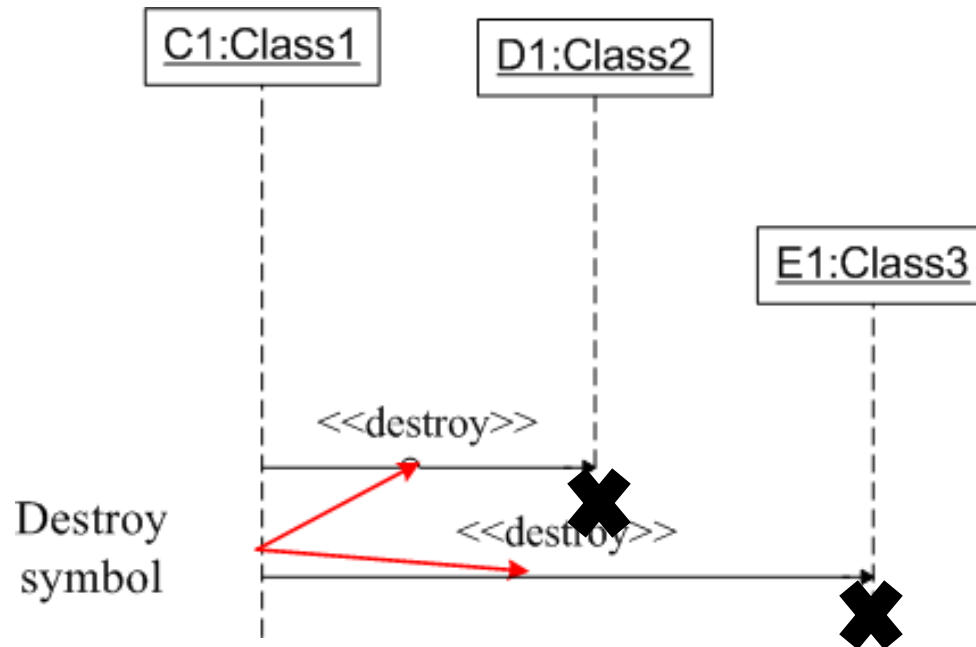
- ❖ It is used to create object during interaction.
- ❖ The object can be created by using <<create>> to indicate the timing of creation.
- ❖ Creating message can be shown as below:



# Sequence Diagram

## 6) Destroy messages:

- ❖ It is used to destroy the objects during interaction.
- ❖ The objects can be terminated using <<destroy>> which points to an “x”.
- ❖ It indicates that object named message is terminated.

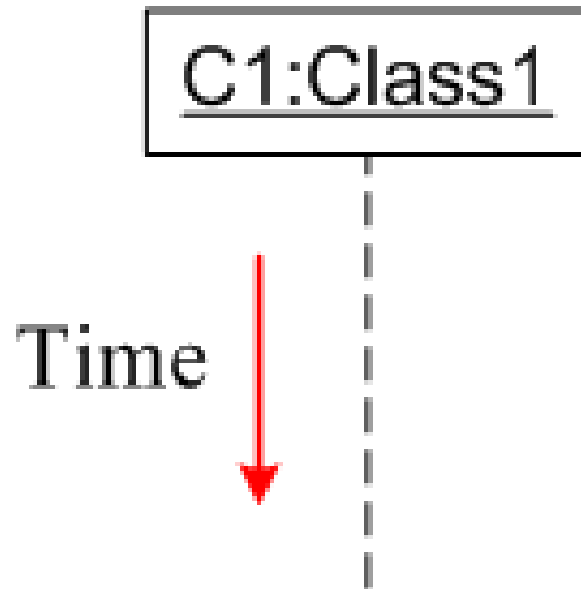


*Note: Avoid modeling object destruction unless memory management is critical.*

# Sequence Diagram

## Time:

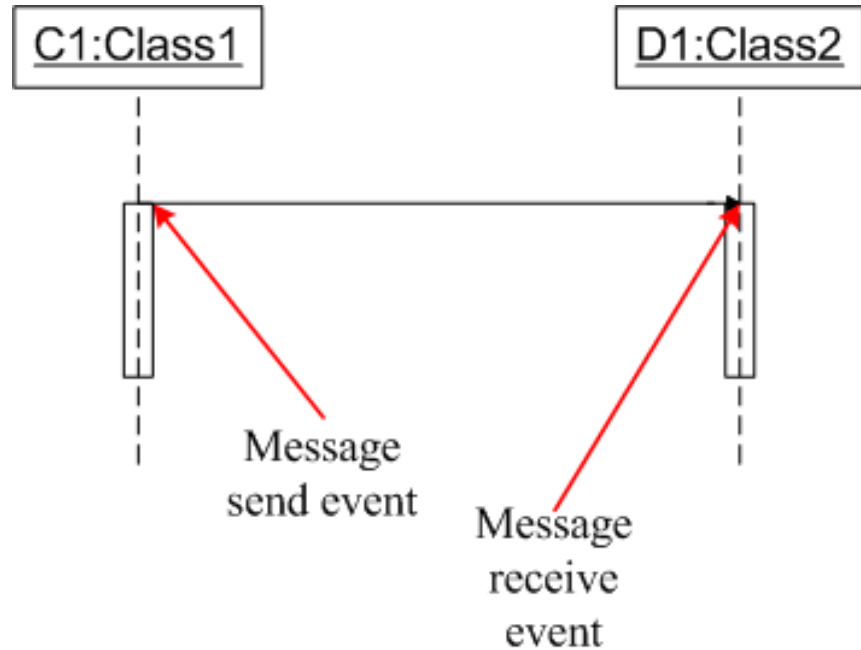
- ❖ Time is all about ordering but not duration.
- ❖ So time is an important factor.
- ❖ The time on sequence diagram starts at top of the page just below the object and then progresses down the page.
- ❖ The sequence diagram describes the order in which interaction takes place.



# Sequence Diagram

## Event:

- ❖ Event is created while sending and receiving message.
- ❖ When interaction takes place, Events are called as build in blocks for messages and signals.
- ❖ It can be referred as smallest part of an interaction and event can occur of at any given point in a Time.



# Sequence Diagram: Control Information

## Condition

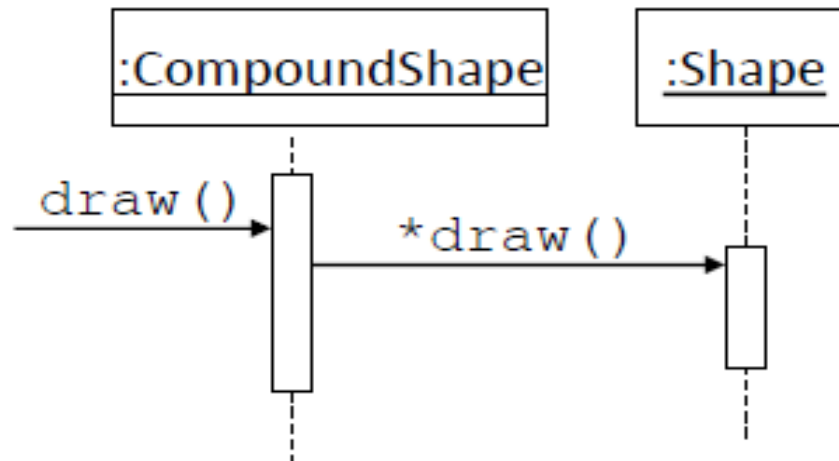
- ❖ **syntax:** [expression or condition] message-label
- ❖ The message is sent only if the condition is true
- ❖ example: [ user valid= “true”] give access

## Iteration

- ❖ **syntax:** \* [expression] message-label or \*message-label

**Note:** \* [expression] message-label is not standard syntax

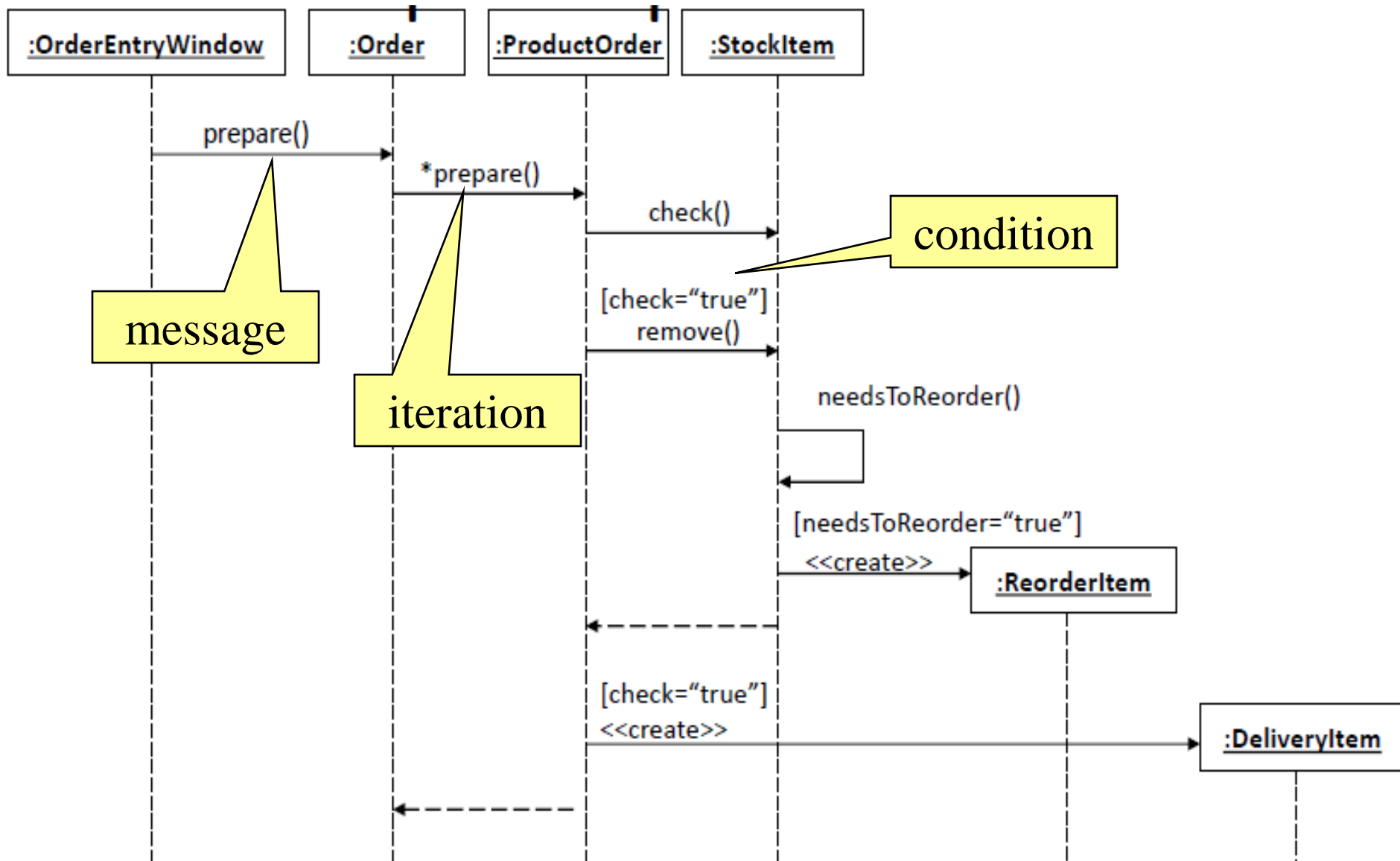
- ❖ The message is sent many times to possibly multiple receiver objects.



# Sequence Diagram

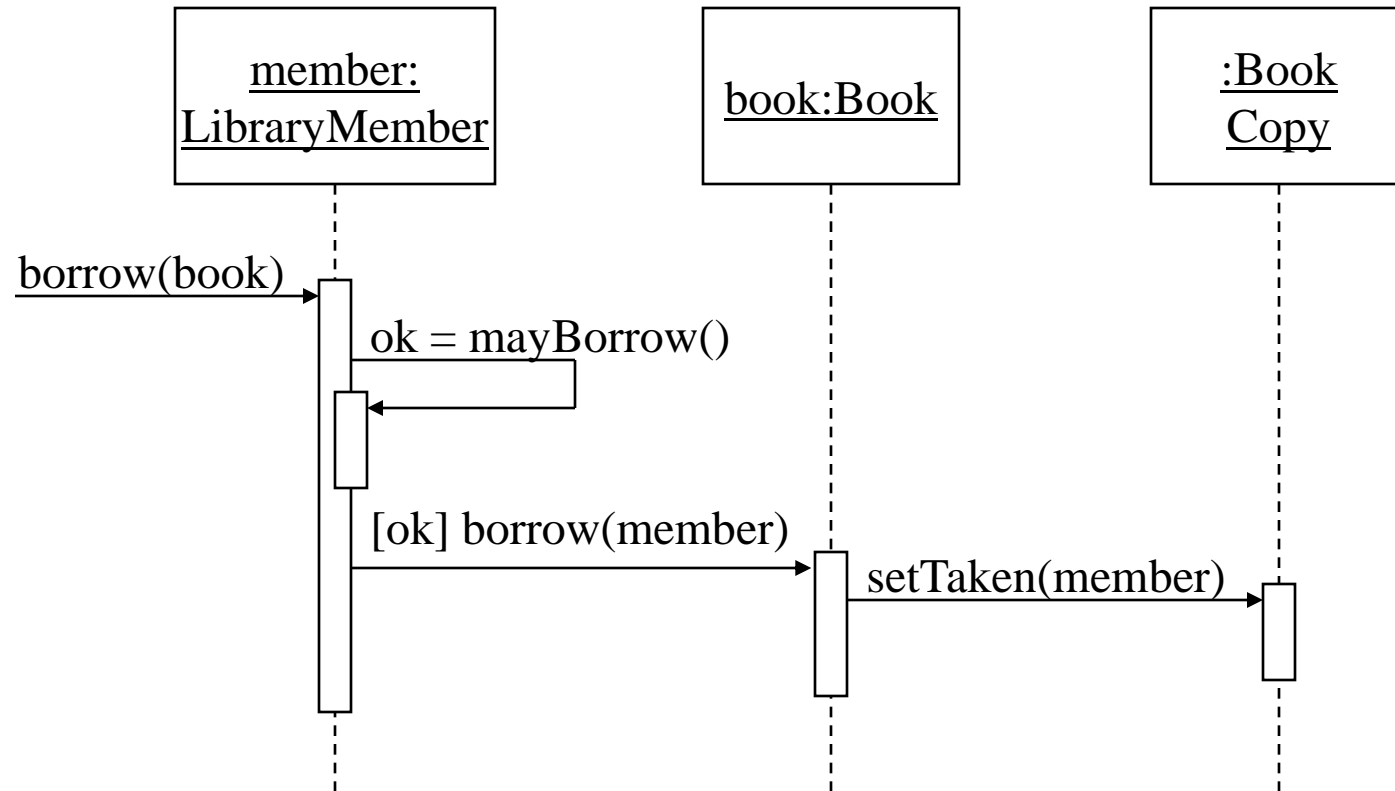
- ❖ The control mechanisms of sequence diagrams suffice only for modeling simple alternatives.
- ❖ Consider drawing several diagrams for modeling complex scenarios.
- ❖ Don't use sequence diagrams for detailed modeling of algorithms (this is better done using *activity diagrams*, *pseudo-code* or *state-charts*).

# Order processing → POS





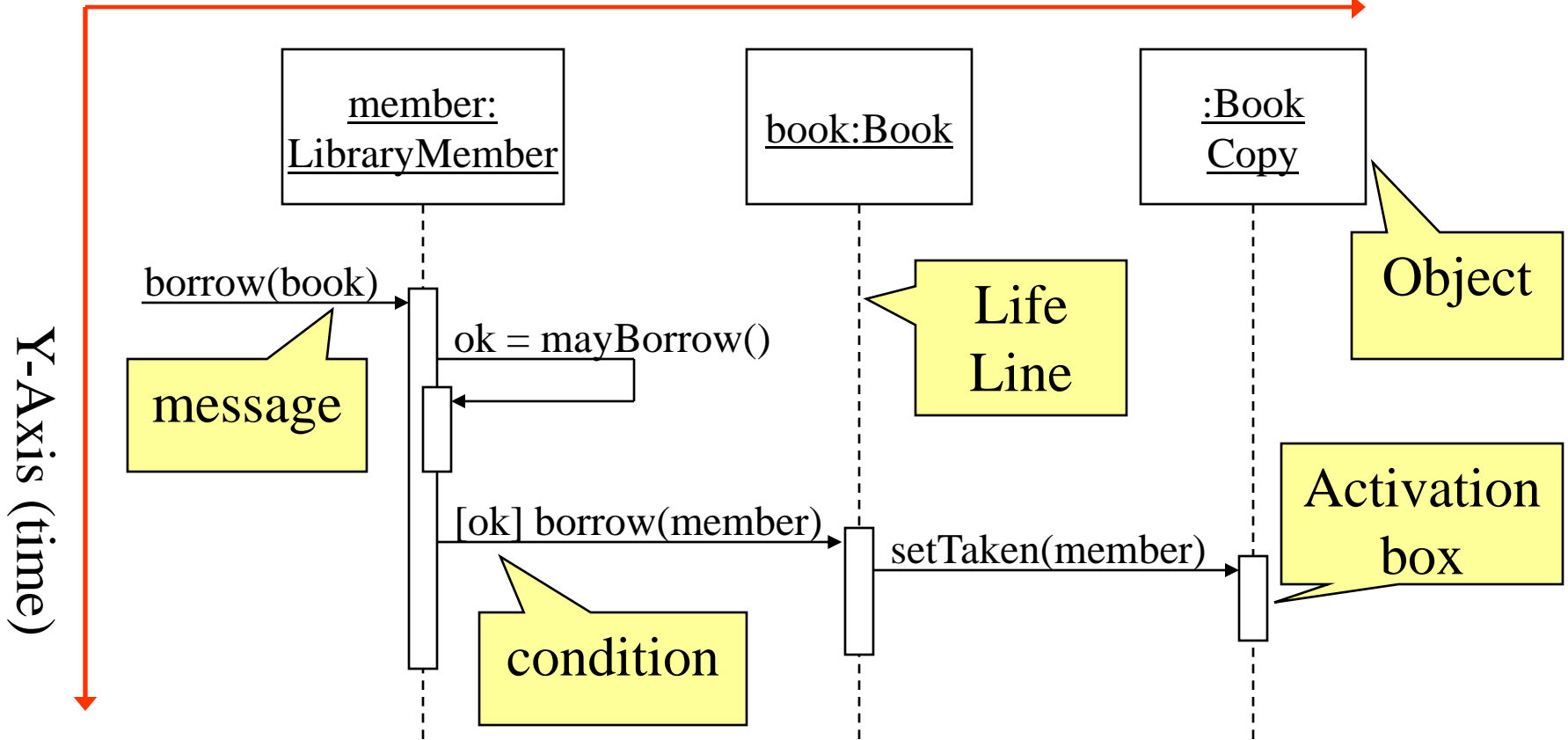
# Borrowing book from library



Sequence Diagrams

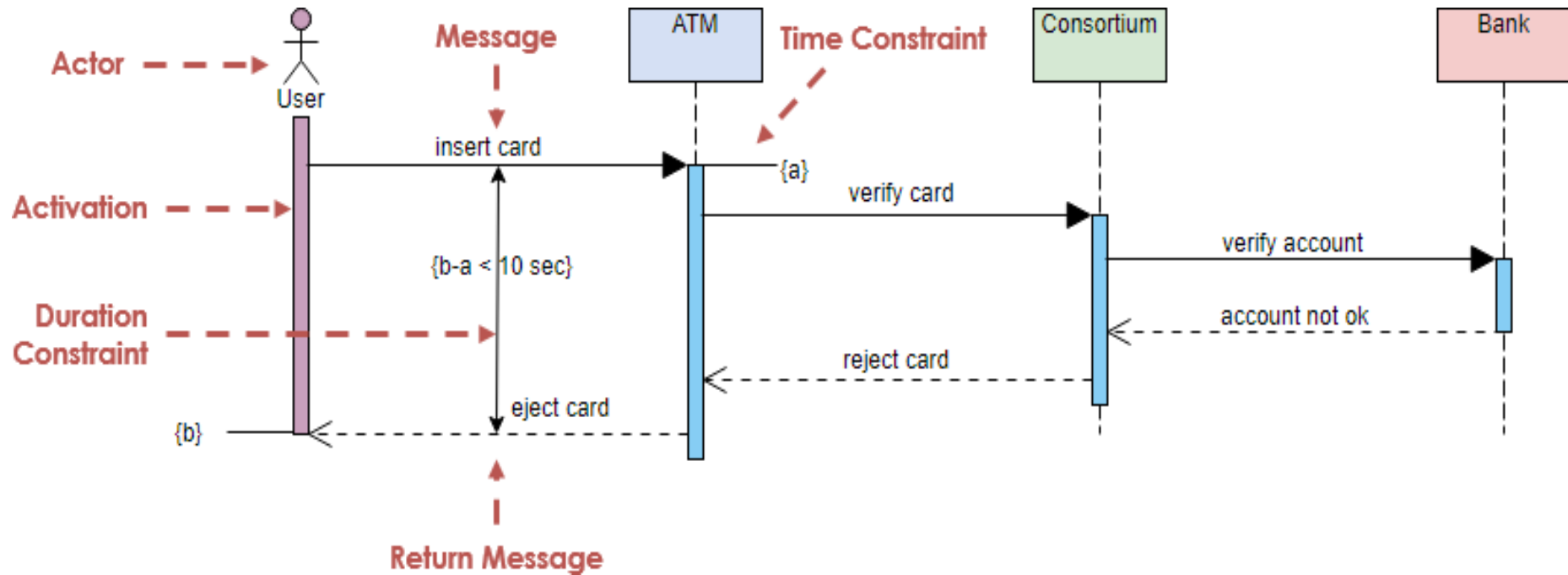
# Borrowing book from library

X-Axis (objects)

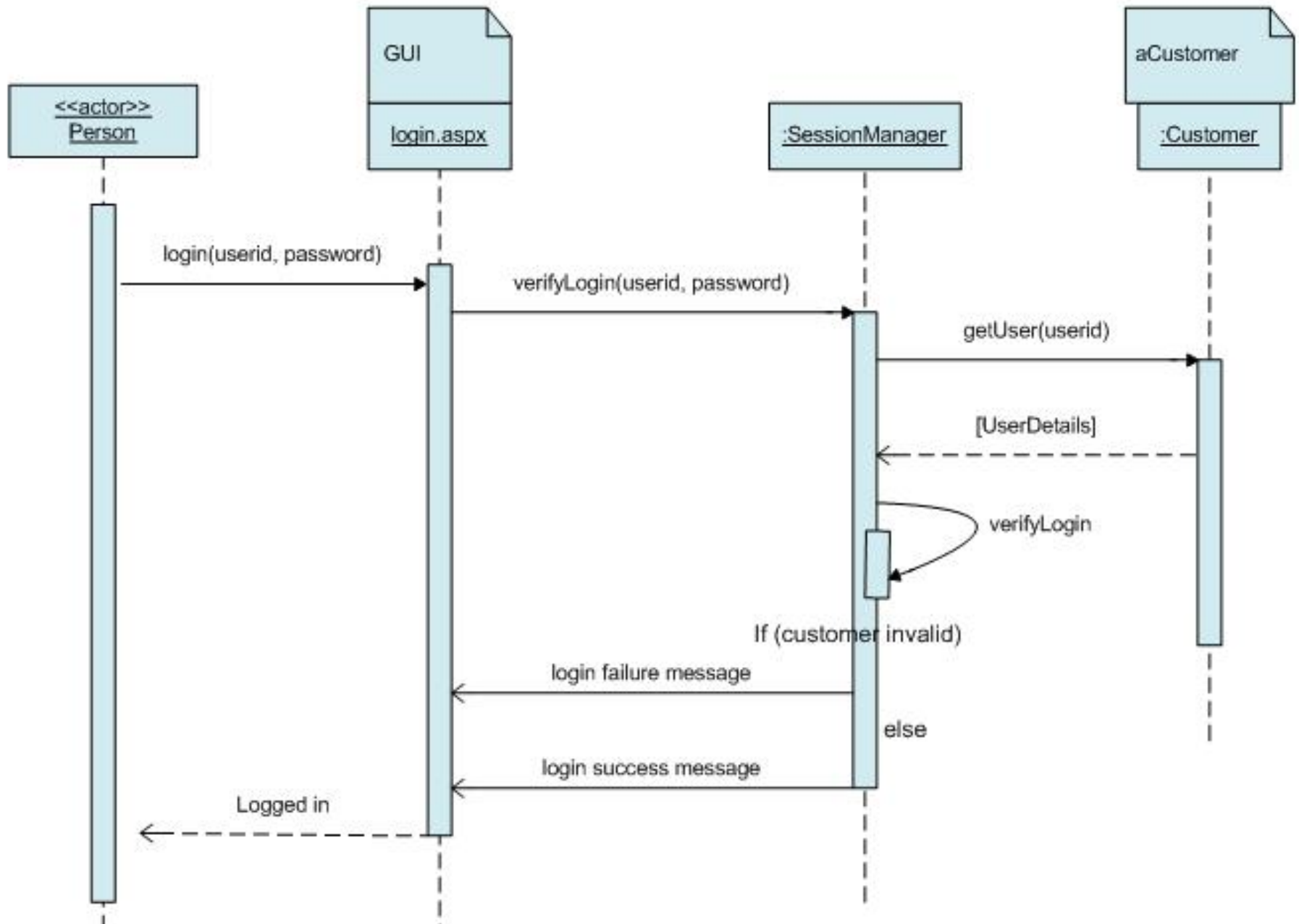


Sequence Diagrams

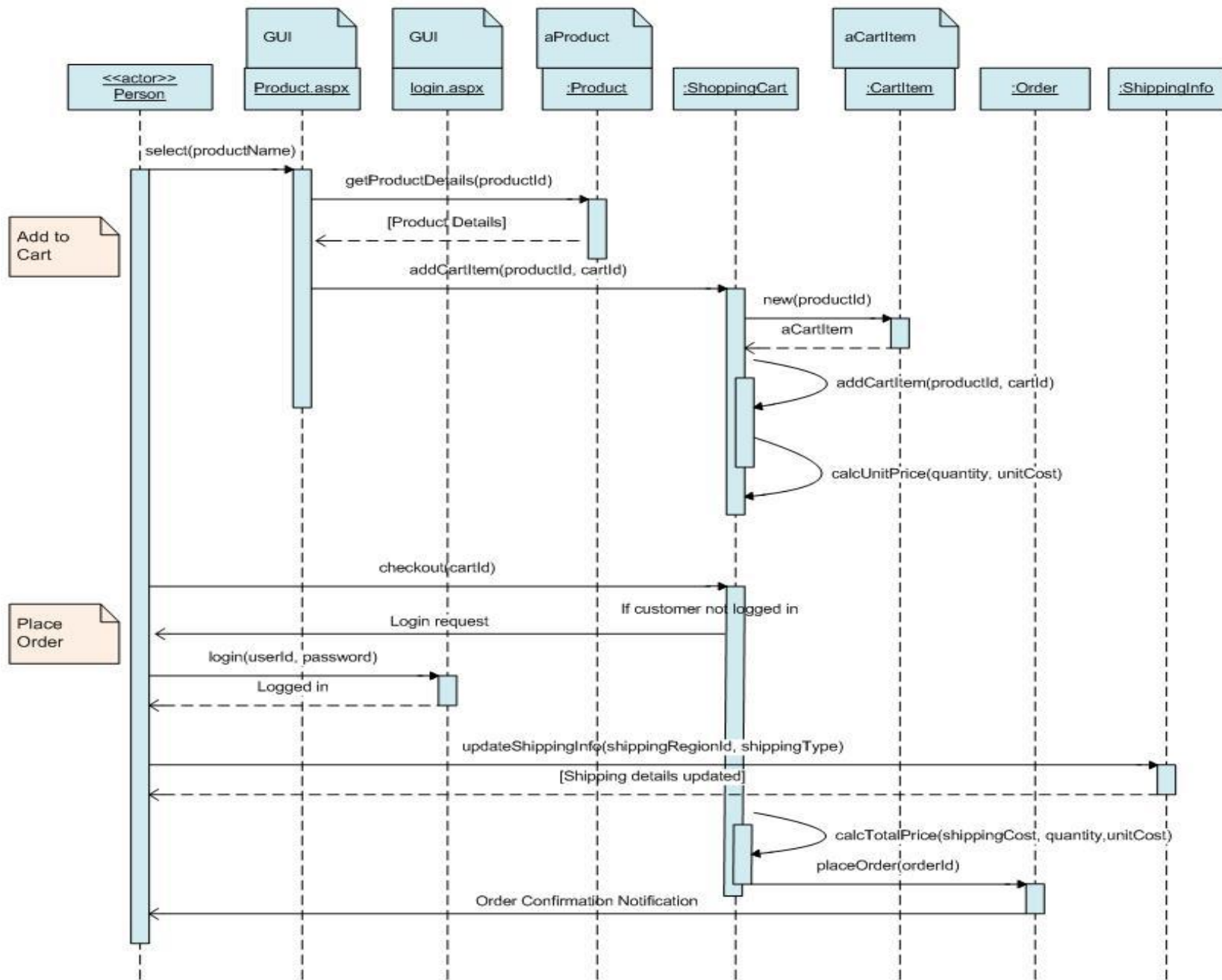
# ATM (Invalid Pin)



# Login



# Online shopping: Explanation [Assignment]



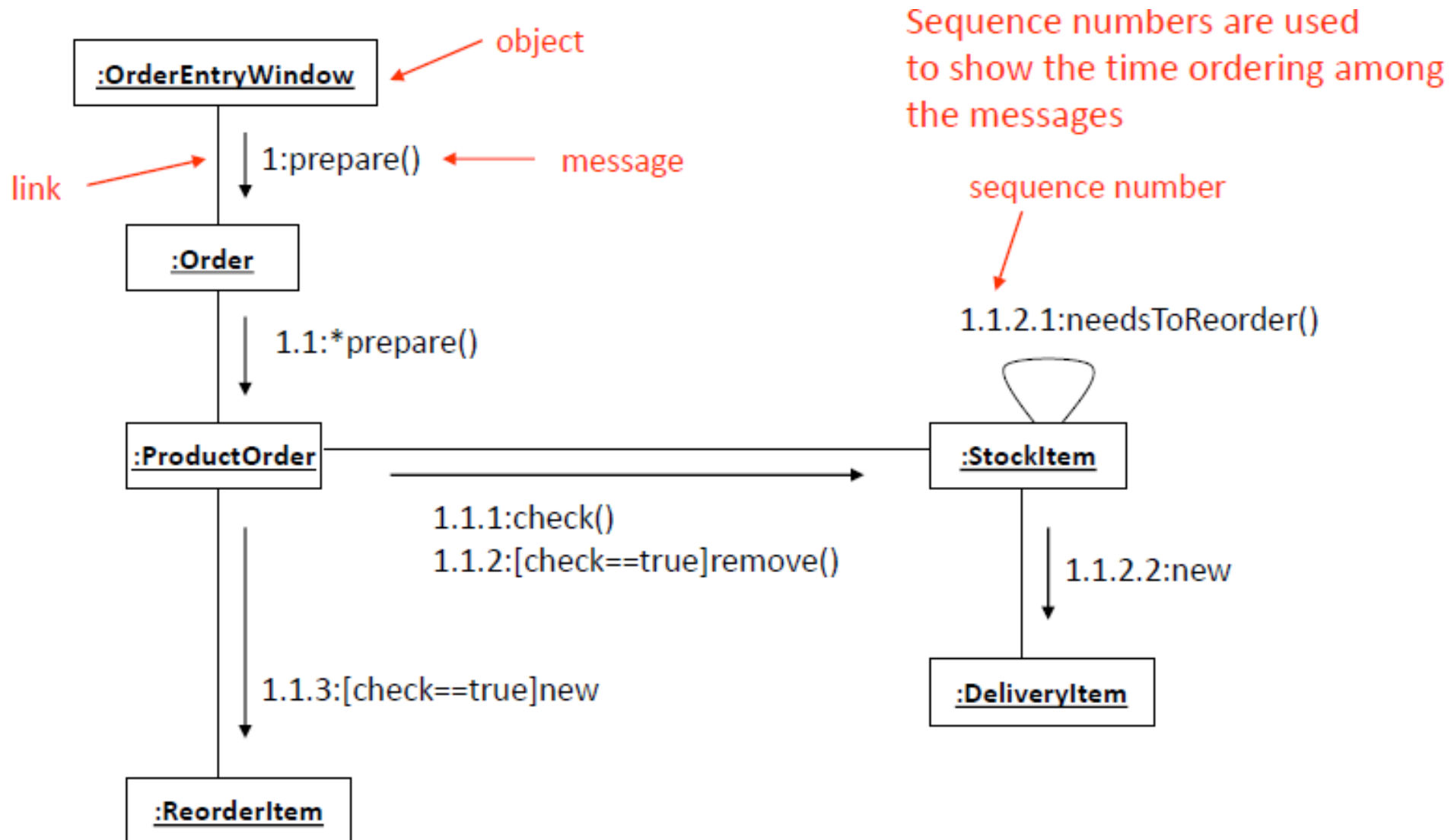
# *UML*

## *Collaboration/ Comm<sup>n</sup> Diagram*

# Collaboration (Communication) Diagrams

- ❖ Collaboration diagrams (also called Communication diagrams) show a particular sequence of messages exchanged between a number of objects
  - ✓ This is what sequence diagrams do too!
- ❖ Sequence diagram highlight more the temporal aspect of the system i.e. it shows object interaction in timely manner(so no need of numbering the messages).
- ❖ In Collaboration diagram, the temporal aspect can be shown here too, by numbering the interactions with sequential labels. (so need to numbering the messages).
- ❖ So Sequence numbers are used to show the time ordering among the messages.

# Collaboration (Communication) Diagrams





*UML*

*Activity Diagram*

# Activity Diagram

- ❖ An activity diagram visually represents a series of actions or flow of control in a system similar to a flowchart.
- ❖ Activities modeled can be sequential and concurrent.
- ❖ In both cases an activity diagram will have a beginning (an initial state) and an end (a final state) and in between them series of actions to be performed by the system.

## Symbols in Activity Diagram

### Initial State or Start Point

- ❖ A small filled circle followed by an arrow represents the initial action state or the start point for any activity diagram.



# Symbols in Activity Diagram

## Activity or Action State

- ❖ An activity represents execution of an action or performing some operation.
- ❖ It is represented using a rectangle with rounded corners.



## Action Flow

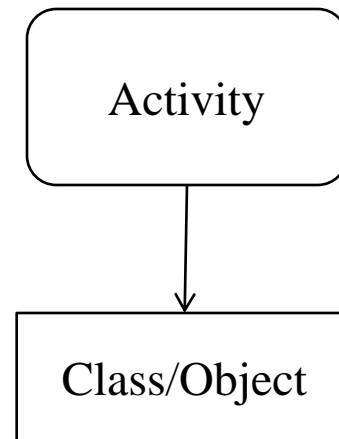
- ❖ Action flows, also called edges and paths, illustrate the transitions from one action state to another.
- ❖ They are usually drawn with an arrowed line.



# Symbols in Activity Diagram

## Object Flow

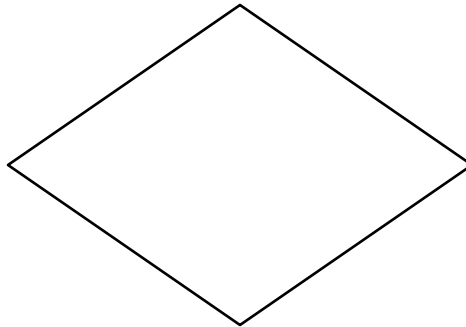
- ❖ Object flow refers to the creation and modification of objects by activities.
- ❖ An object flow arrow from an action to an object means that the action creates or influences the object.
- ❖ An object flow arrow from an object to an action indicates that the action state uses the object.



# Symbols in Activity Diagram

## Decisions and Branching

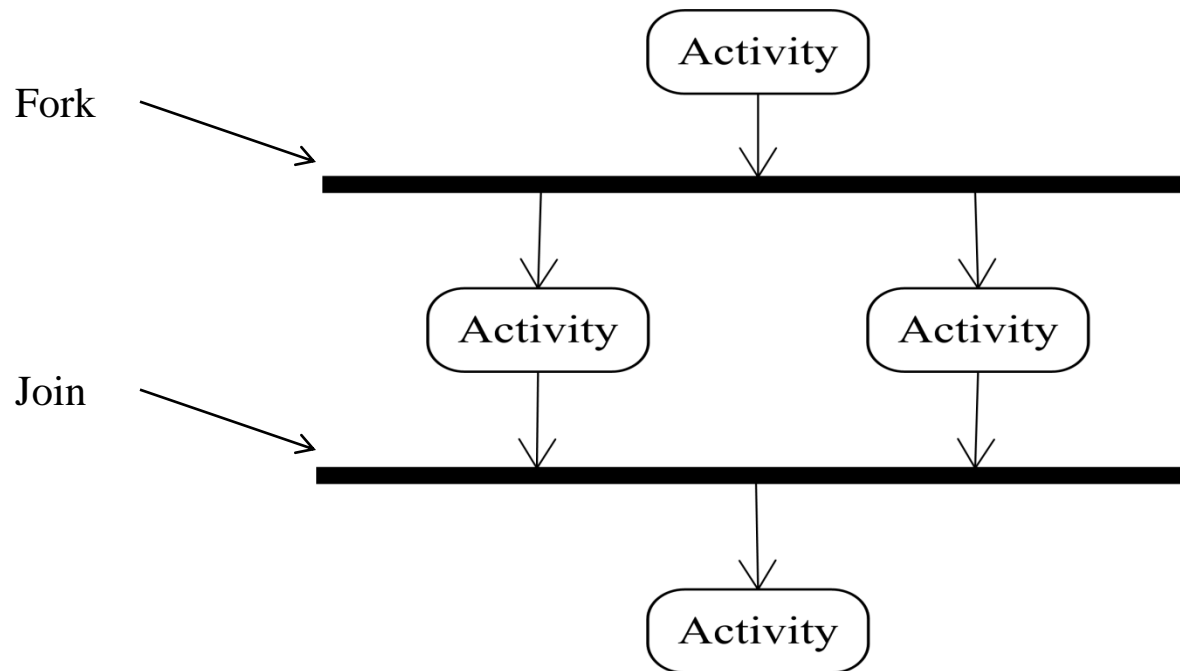
- ❖ A diamond represents a decision with alternate paths.
- ❖ When an activity requires a decision prior to moving on to the next activity, add a diamond between the two activities.
- ❖ The outgoing alternates should be labeled with a condition or guard expression. You can also label one of the paths "else."



# Symbols in Activity Diagram

## Synchronization

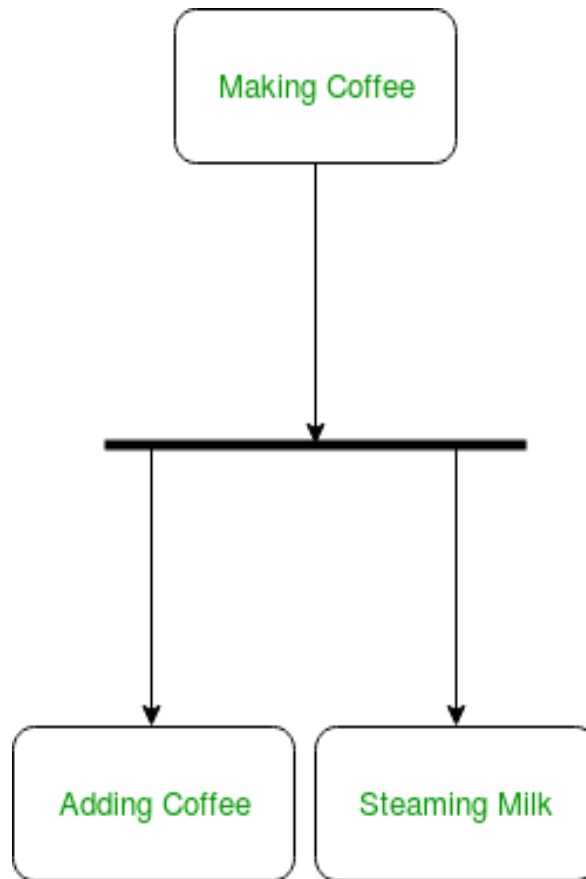
- ❖ A fork node is used to split a single incoming flow into multiple concurrent flows. It is represented as a straight, slightly thicker line in an activity diagram.
- ❖ A join node joins multiple concurrent flows back into a single outgoing flow.
- ❖ A fork and join node used together are often referred to as synchronization.



# Symbols in Activity Diagram

## Fork

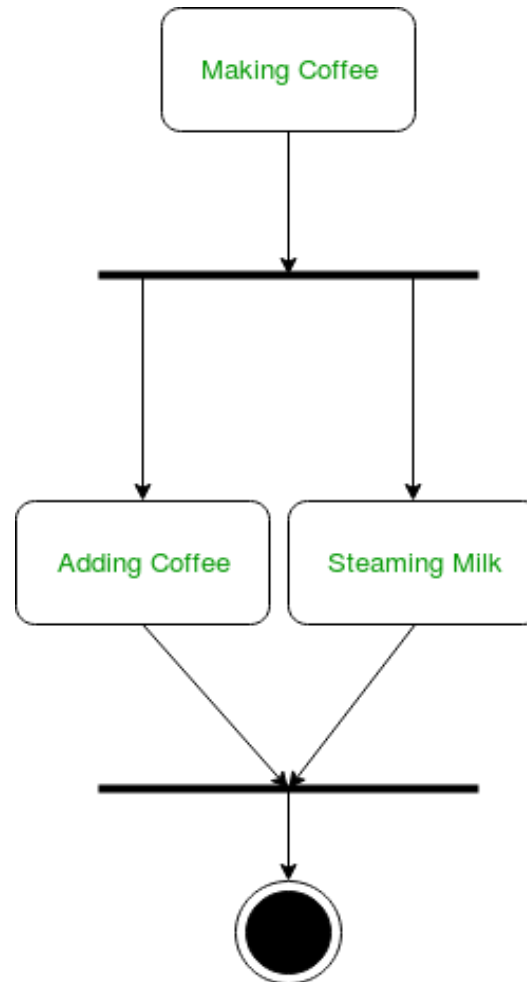
- ❖ A fork node is used to split a single incoming flow into multiple concurrent flows.
- ❖ It is represented as a straight, slightly thicker line in an activity diagram.



# Symbols in Activity Diagram

## Join

- ❖ A join node joins multiple concurrent flows back into a single outgoing flow.

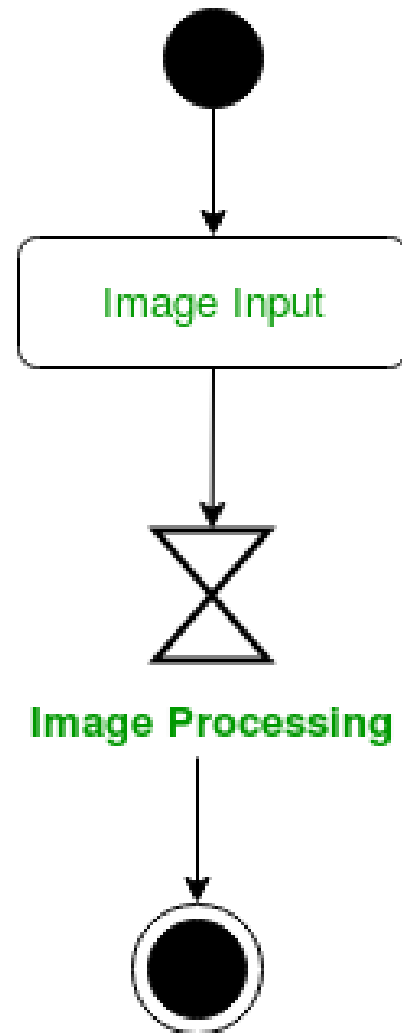




# Symbols in Activity Diagram

## Time Event

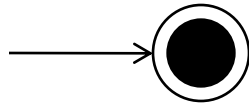
- ❖ This refers to an event that stops the flow for some amount of time.
- ❖ It is represented by a hourglass.

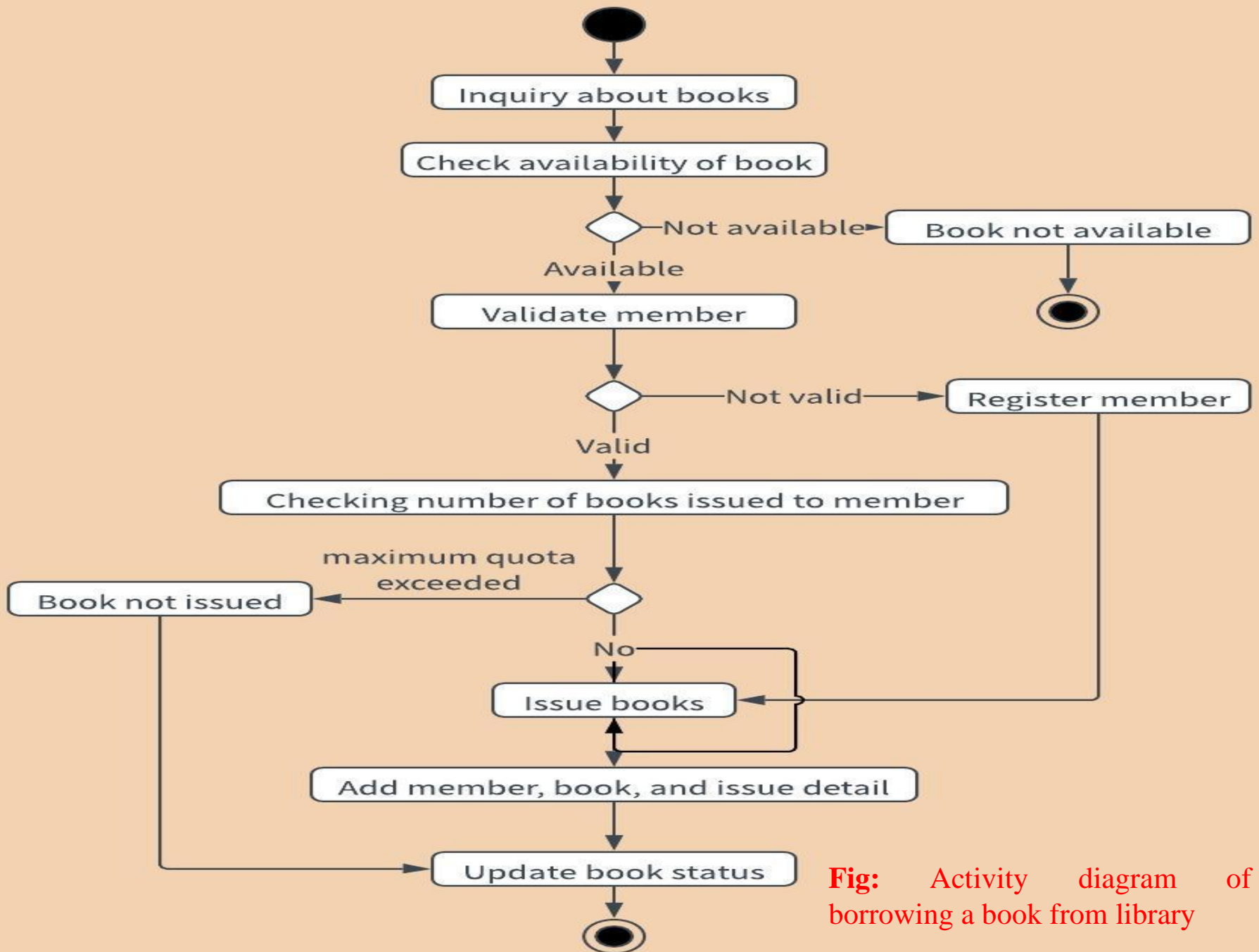


# Symbols in Activity Diagram

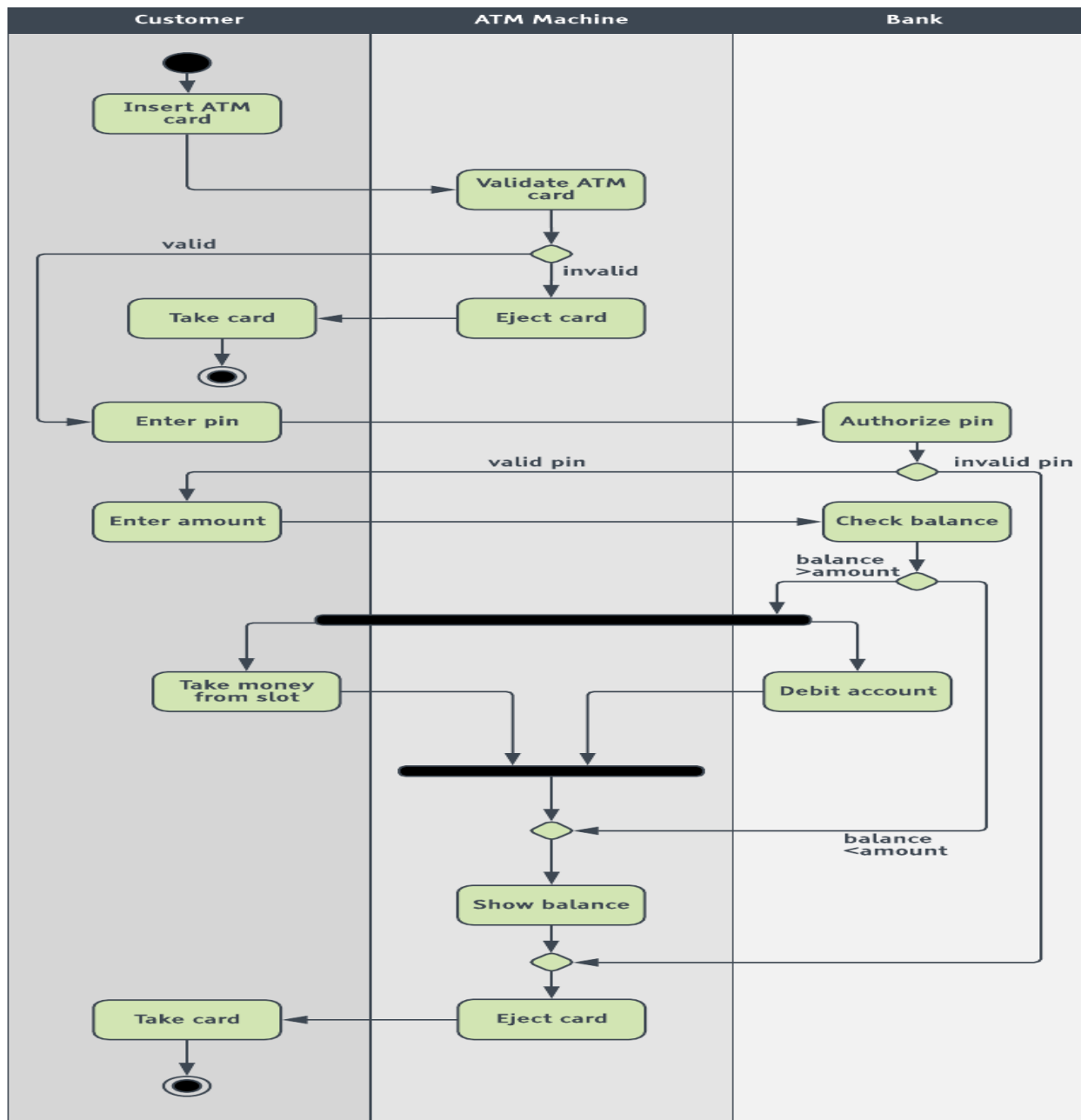
## Final State or End Point

- ❖ An arrow pointing to a filled circle nested inside another circle represents the final action state.





**Fig:** Activity diagram of borrowing a book from library

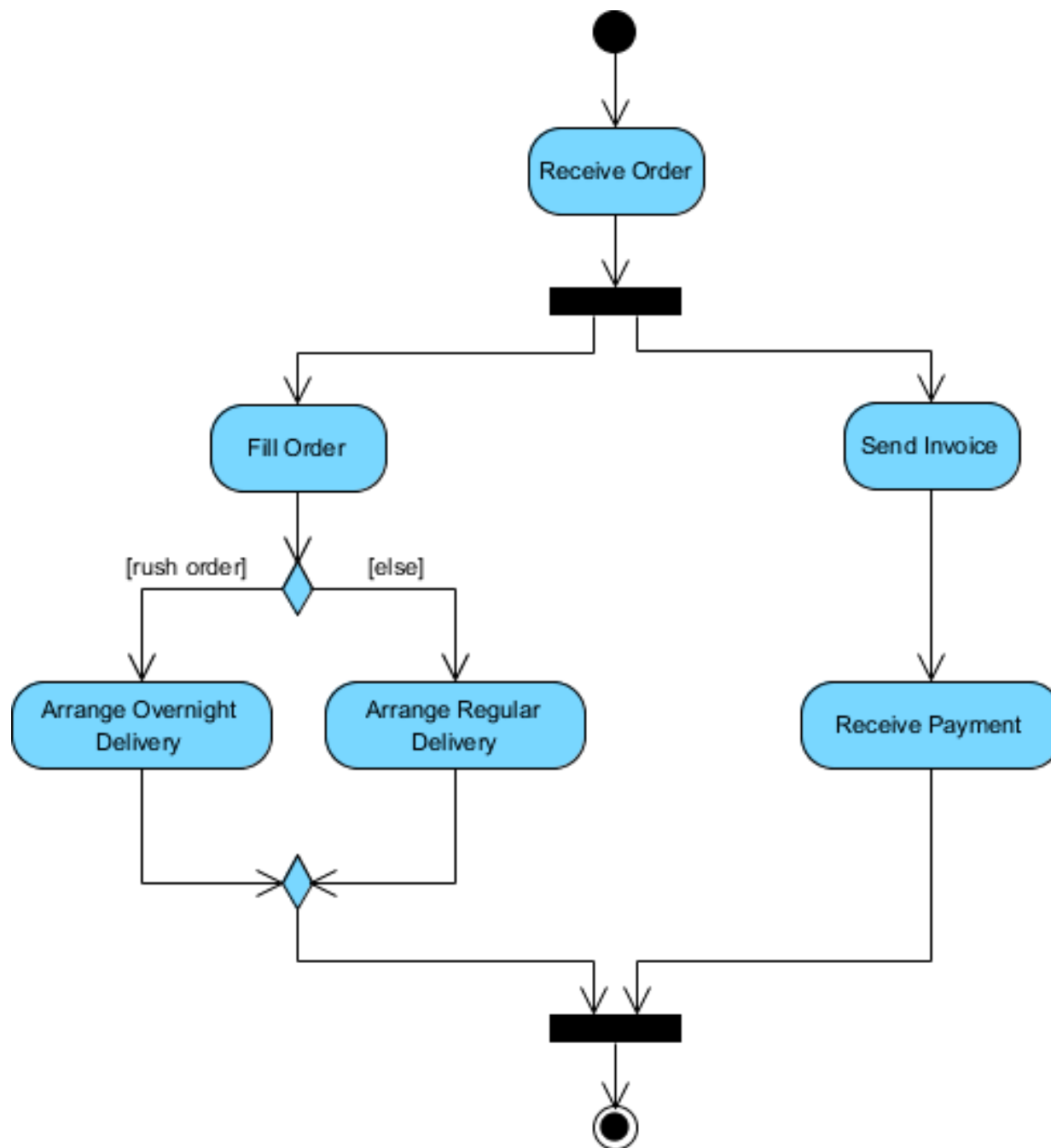


**Fig:** Swimlane diagram of ATM transaction

# Activity Diagram

## Process Order - Problem Description (Class Work)

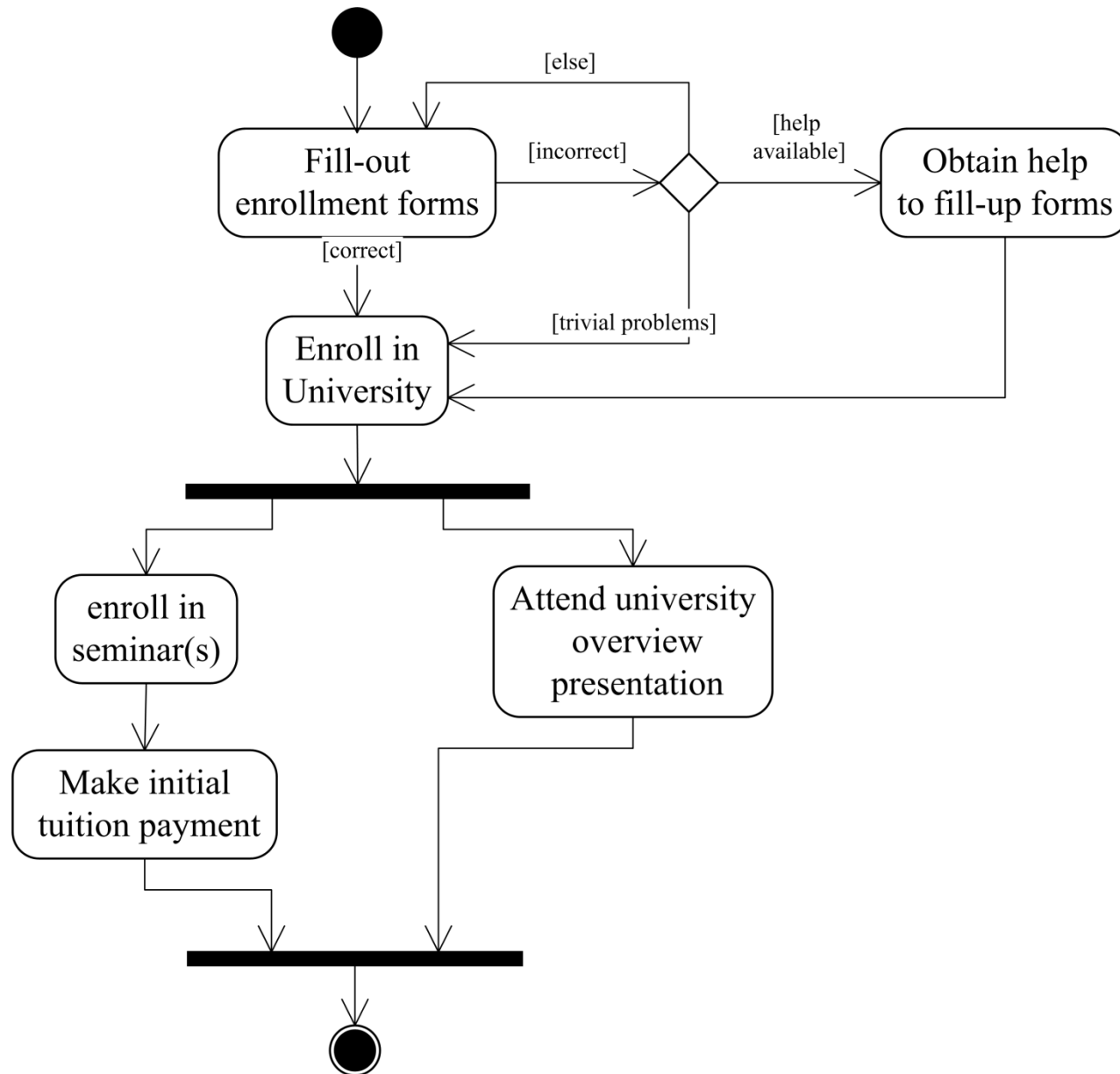
- ❖ Once the order is received, the activities split into two parallel sets of activities. One side fills and sends the order while the other handles the billing.
- ❖ On the Fill Order side, the method of delivery is decided conditionally. Depending on the condition either the Overnight Delivery activity or the Regular Delivery activity is performed.
- ❖ Finally the parallel activities combine to close the order.



# Activity Diagram

## Activity Diagram Example - Student Enrollment (CW)

- ❖ An applicant wants to enroll in the university.
- ❖ The applicant hands a filled out copy of Enrollment Form.
- ❖ The registrar inspects the forms.
- ❖ The registrar determines that the forms have been filled out properly.
- ❖ The registrar informs student to attend in university overview presentation.
- ❖ The registrar helps the student to enroll in seminars
- ❖ The registrar asks the student to pay for the initial tuition.





# *UML*

## *State Machine*

### *Diagram*

# State Machine Diagram

- ❖ An object responds differently to the same event depending on what state it is in.
- ❖ A state machine diagram models the behavior of a single object, specifying the sequence of events that an object goes through during its lifetime in response to events.
- ❖ State diagrams are used to show possible states a single object can get into
  - ✓ i.e. shows states of an object .
- ❖ How object changes state in response to events
  - ✓ shows transitions between states

# State Machine Diagram

## States

- ❖ A state is denoted by a **round-cornered rectangle** with the name of the state written inside it.



## Initial and Final States

- ❖ The **initial state** is denoted by a **filled black circle** and may be labeled with a name. The **final state** is denoted by a **circle with a dot inside** and may also be labeled with a name.



Initial state

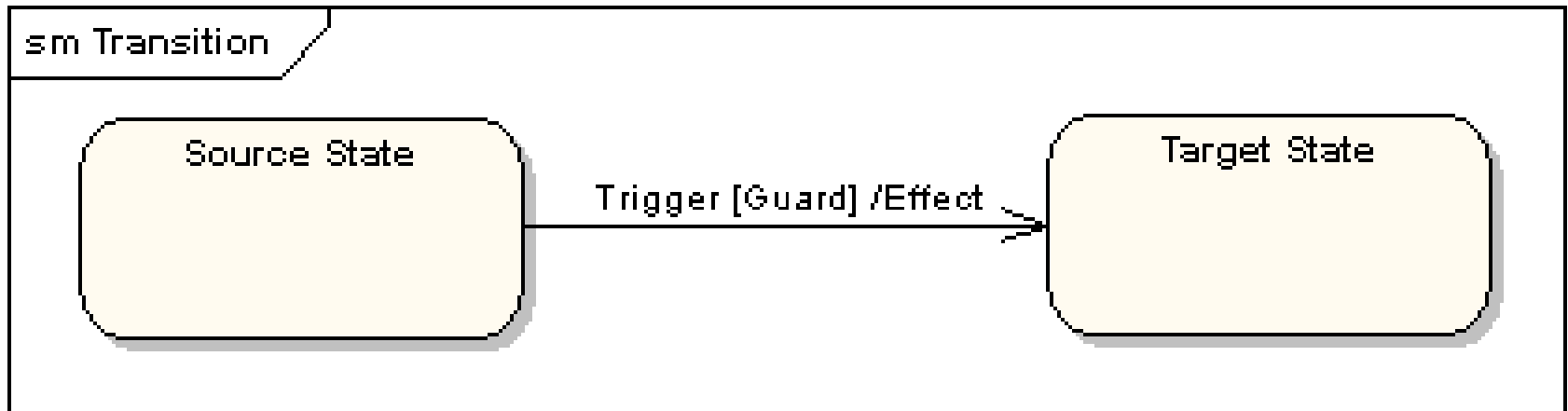


Final state

# State Machine Diagram

## Transitions

- ❖ Transitions from one state to the next are denoted by lines with arrowheads. A transition may have a trigger, a guard and an effect, as below.

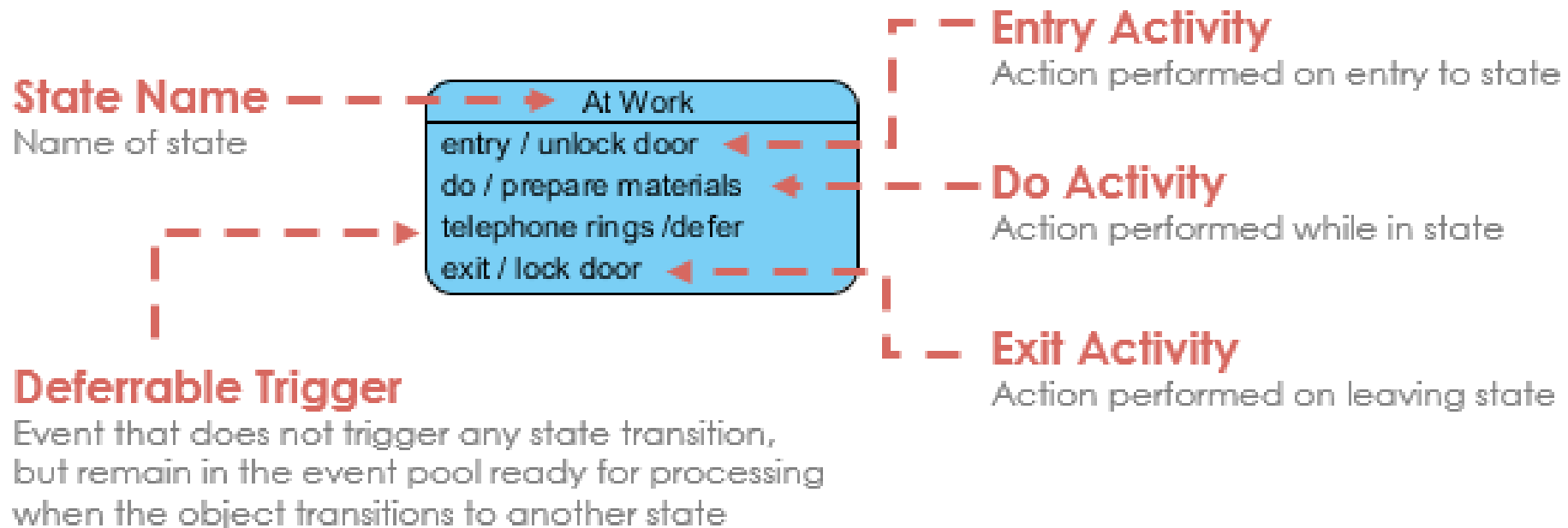


- ❖ "Trigger" is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time. "Guard" is a condition which must be true in order for the trigger to cause the transition. "Effect" is an action which will be invoked directly on the object that owns the state machine as a result of the transition.

# State Machine Diagram

## State Actions

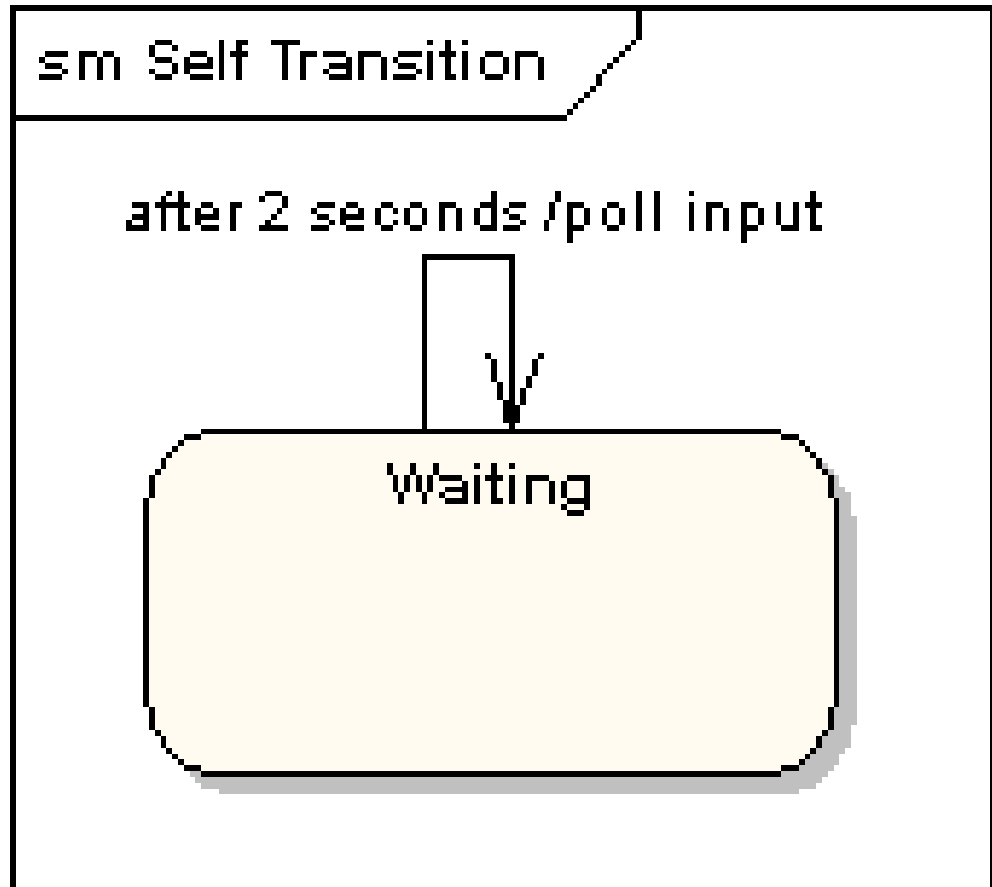
- ❖ For each transition, an effect was associated with the transition.
  - ✓ Entry
  - ✓ Exit
  - ✓ Do
  - ✓ Defer



# State Machine Diagram

## Self-Transitions

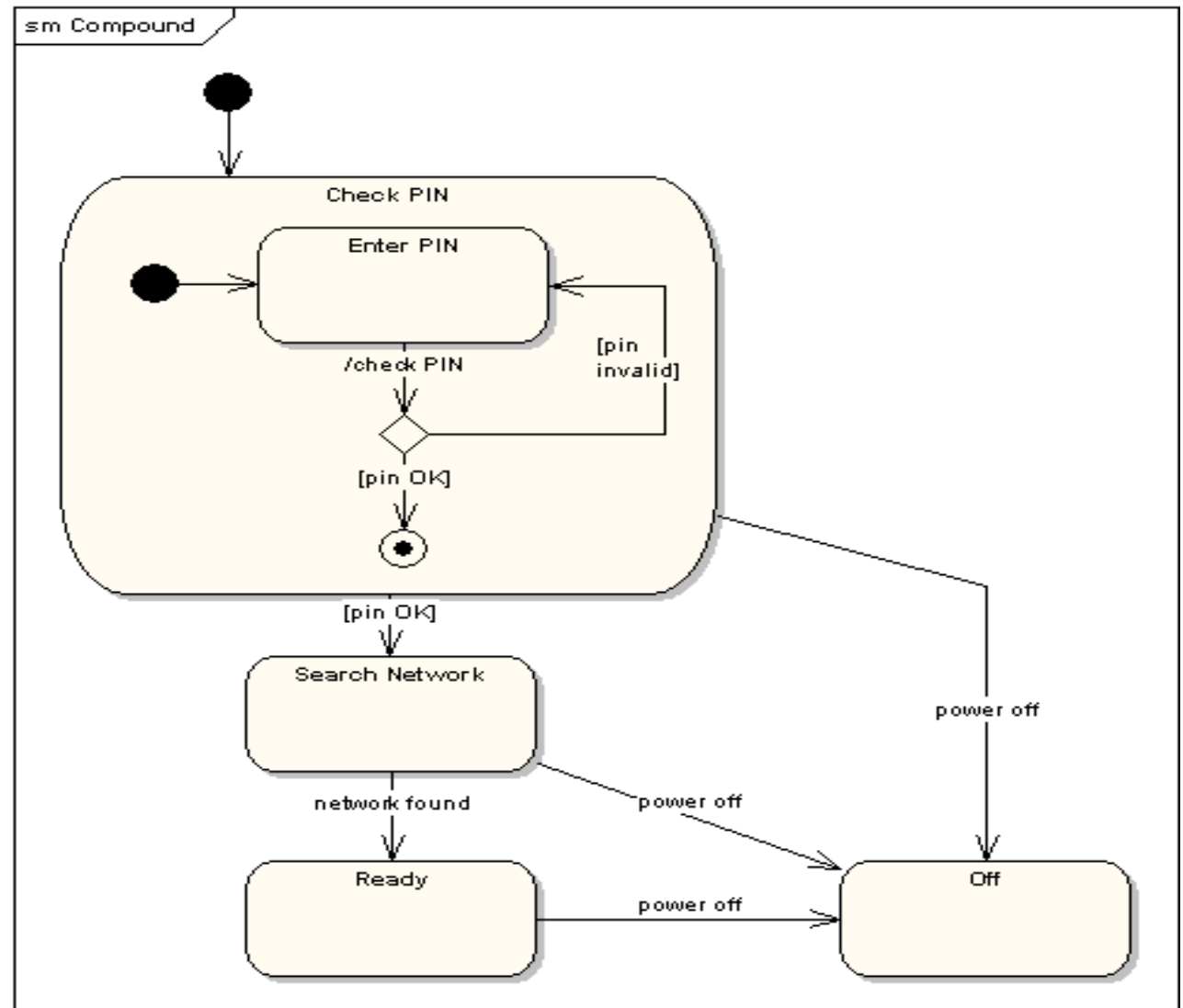
- ❖ A state can have a transition that returns to itself, as in the following diagram



# State Machine Diagram

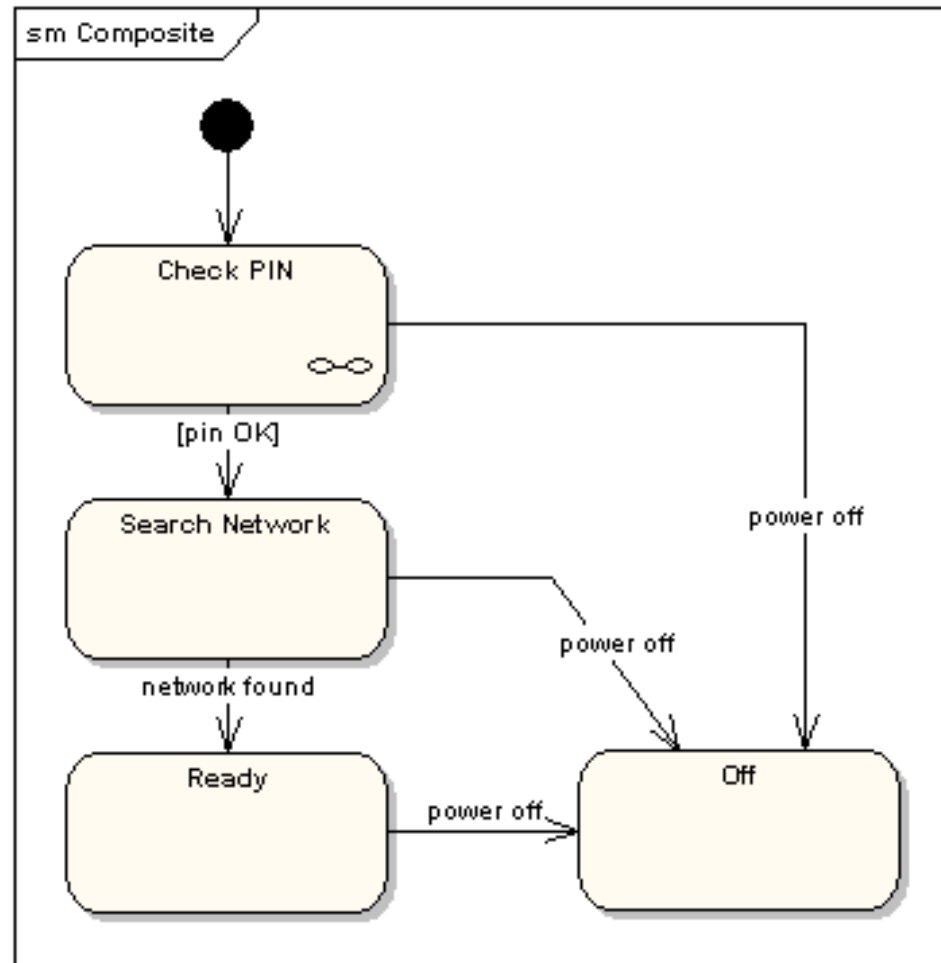
## Compound States or Sub States

- A state machine diagram may include sub-machine diagrams, as in the example:



# State Machine Diagram

- ❖ The alternative way to show the same information is as follows.



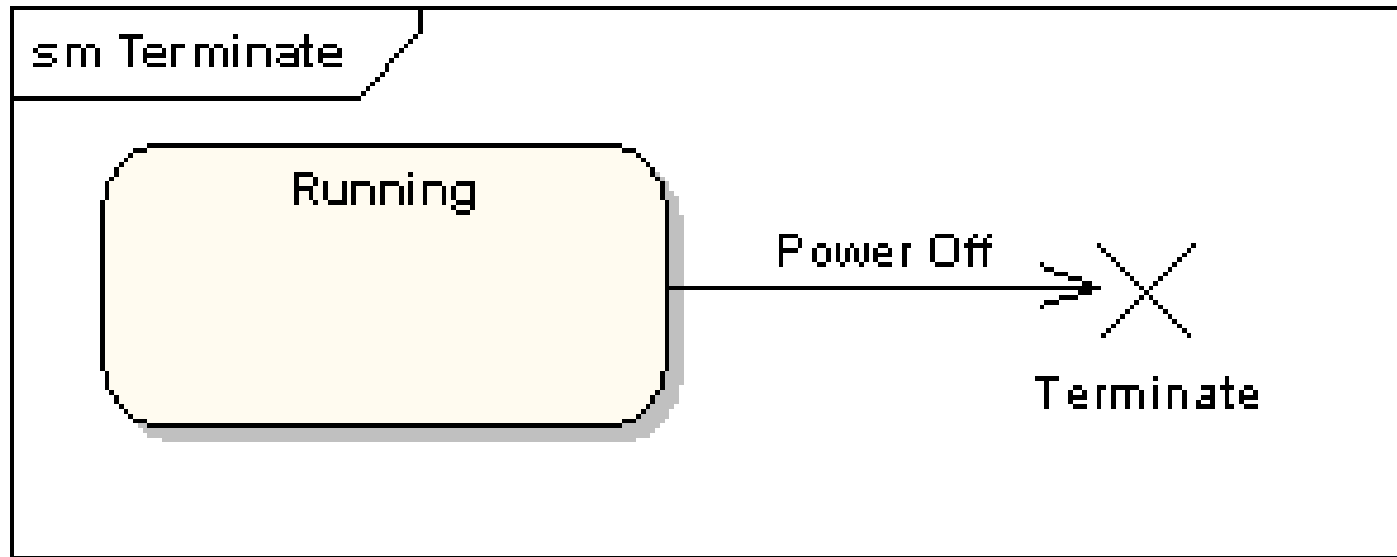
- ❖ The notation in above fig. indicates that the details of the **Check PIN** sub-machine are shown in a separate diagram.



# State Machine Diagram

## Terminate Pseudo-State

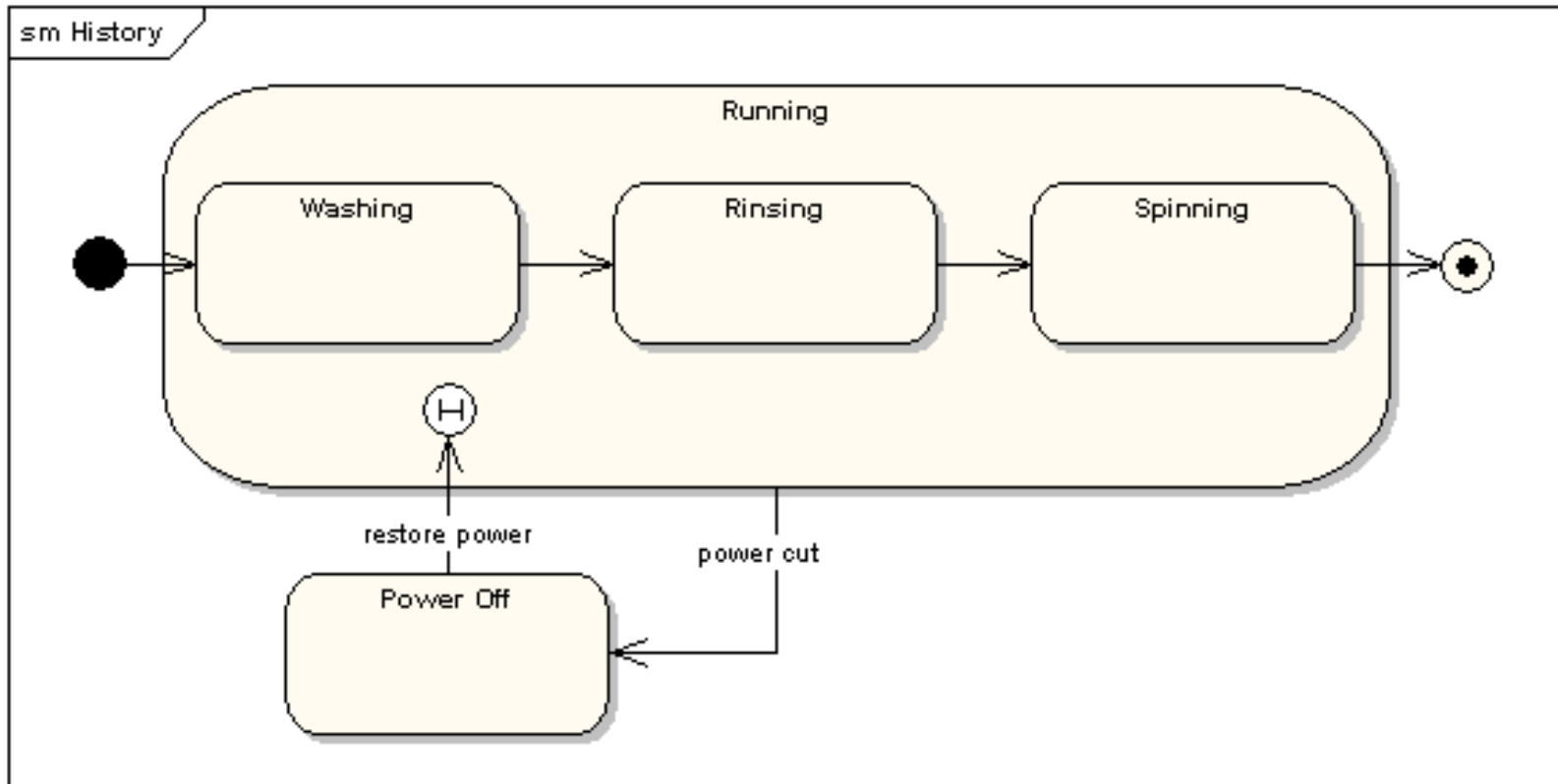
- ❖ Entering a terminate pseudo-state indicates that the lifeline of the state machine has ended.
- ❖ A terminate pseudo-state is notated as a cross.



# State Machine Diagram

## History States

- ❖ A history state is used to remember the previous state of a state machine when it was interrupted.



- ❖ The above diagram illustrates the use of history states. The example is a state machine belonging to a washing machine.

# State Machine Diagram

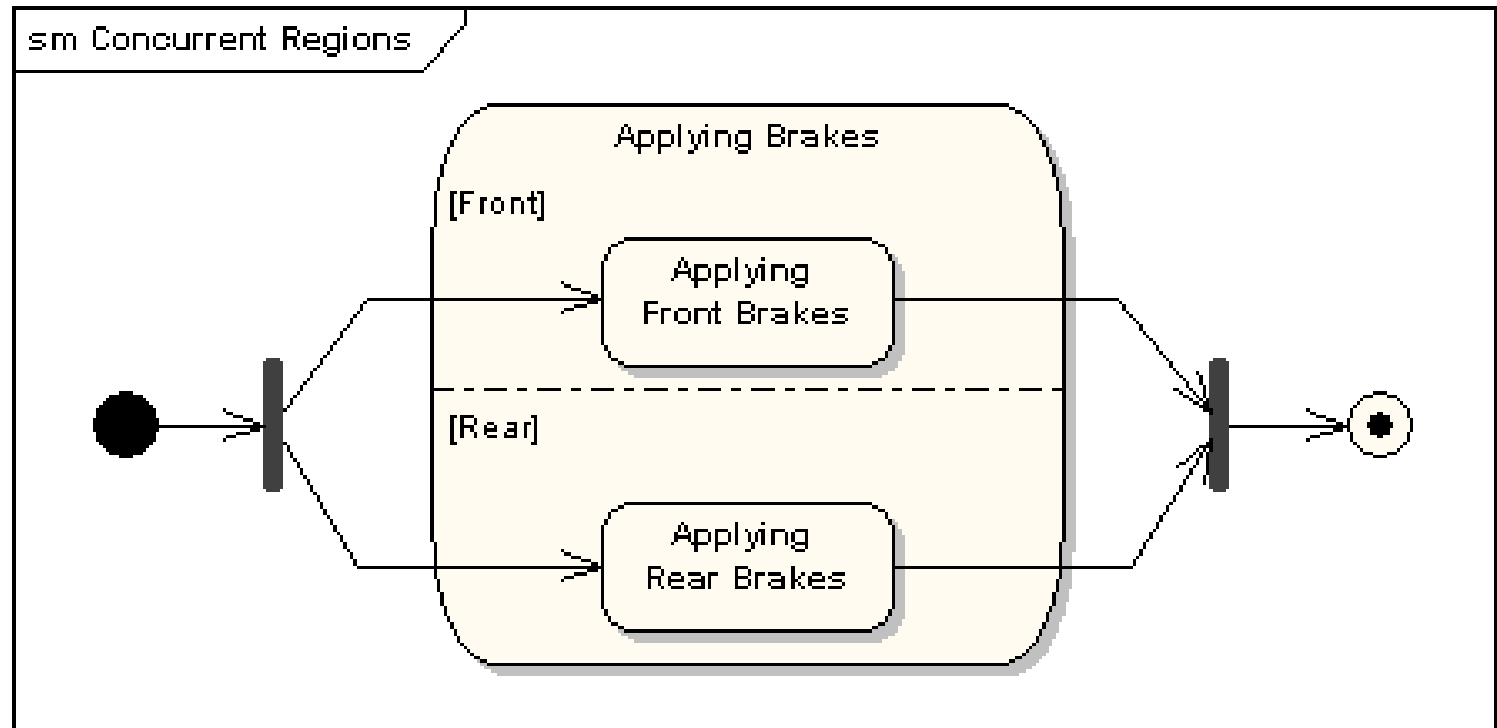
## History States

- ❖ In this state machine shown in previous slide, when a washing machine is running, it will progress from "Washing" through "Rinsing" to "Spinning".
- ❖ If there is a power cut, the washing machine will stop running and will go to the "Power Off" state. Then when the power is restored, the Running state is entered at the "History State" symbol meaning that it should resume where it last left-off.

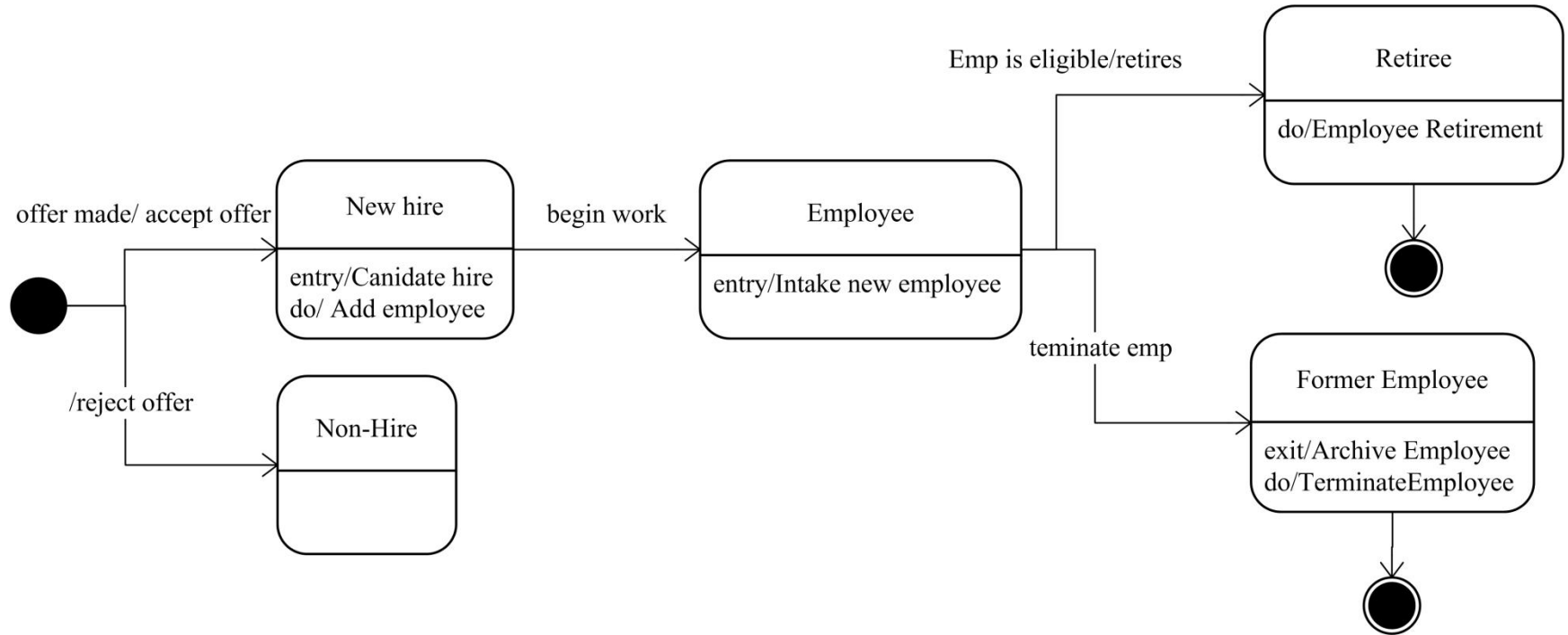
# State Machine Diagram

## Concurrent state machine

- ❖ A state may be divided into regions containing sub-states that exist and execute concurrently. The example below shows that within the state "Applying Brakes", the front and rear brakes will be operating simultaneously and independently. Notice the use of fork and join pseudo-states, rather than choice and merge pseudo-states. These symbols are used to synchronize the concurrent threads.

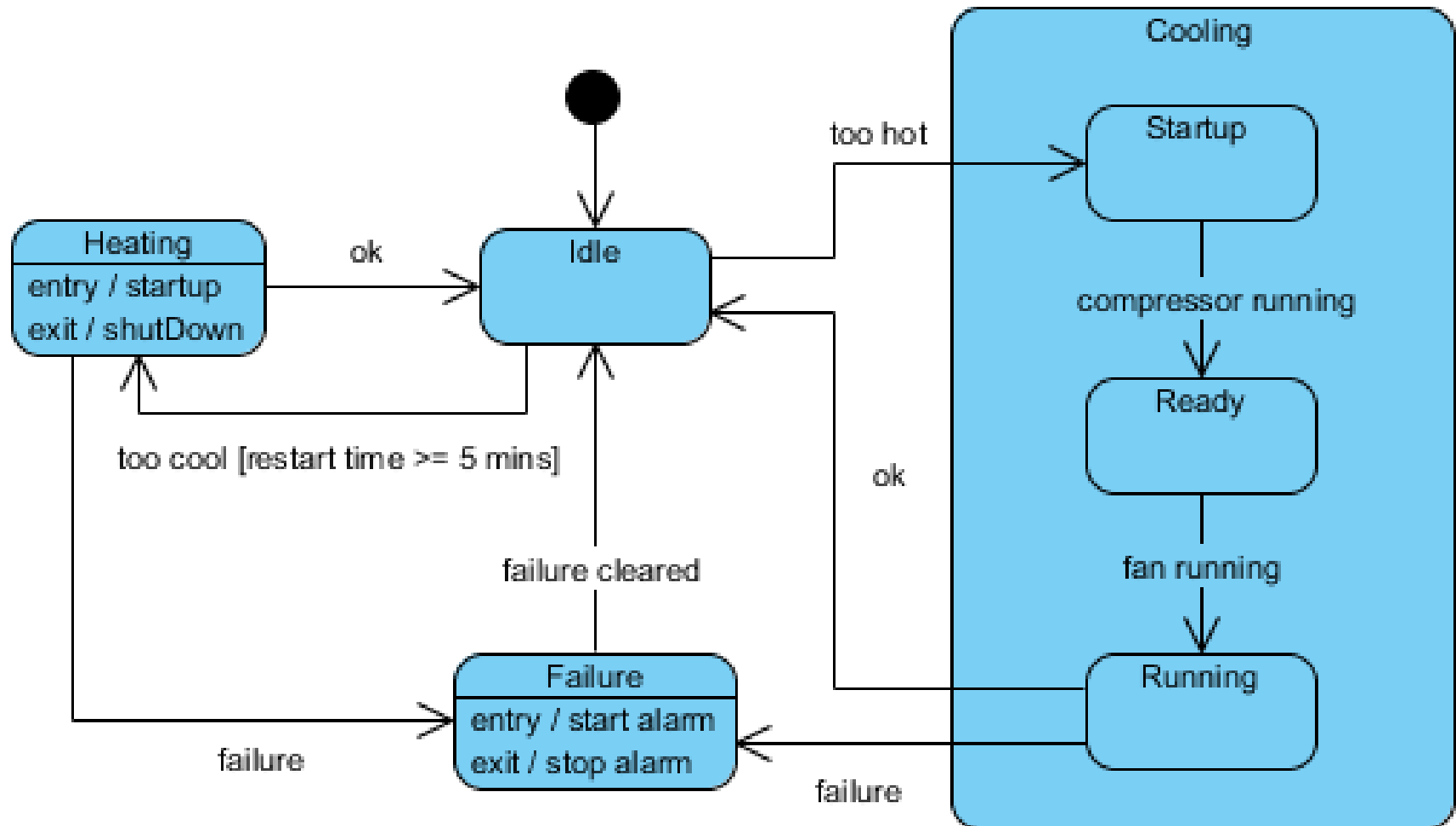


# SMD of Recruitment process



# State Machine Diagram

## ❖ Sub states or compounded states example



## ❖ Class work (description)

->Even roll -> **Too Hot**

->Odd roll -> **Too Cool**

# *Designing objects/ Object models*

# Designing Objects

## Object Models:

There are two kinds of object models:

- ❖ **Dynamic models**, such as UML interaction diagrams sequence diagrams or communication diagrams), help design the logic, the behavior of the code or the method bodies. They tend to be the more interesting, difficult, important diagrams to create.
- ❖ **Static models**, such as UML class diagrams, help design the definition of packages, class names, attributes, and method signatures (but not method bodies).



# Designing Objects

## How do developers design objects?

*Here are three ways:*

### 1. Code:

Design-while-coding (Java, C#, ...), ideally with power tools such as refactoring.

### 2. Draw, then code:

*Drawing some UML on a whiteboard or UML CASE tool, then switching to #1 with a text-strong IDE (e.g., Eclipse or Visual Studio).*

### 3. Only draw:

*Somehow, 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, as this still involves a text programming language attached to UML graphic elements.*

# *Object Design Techniques*

# Realization of Use case

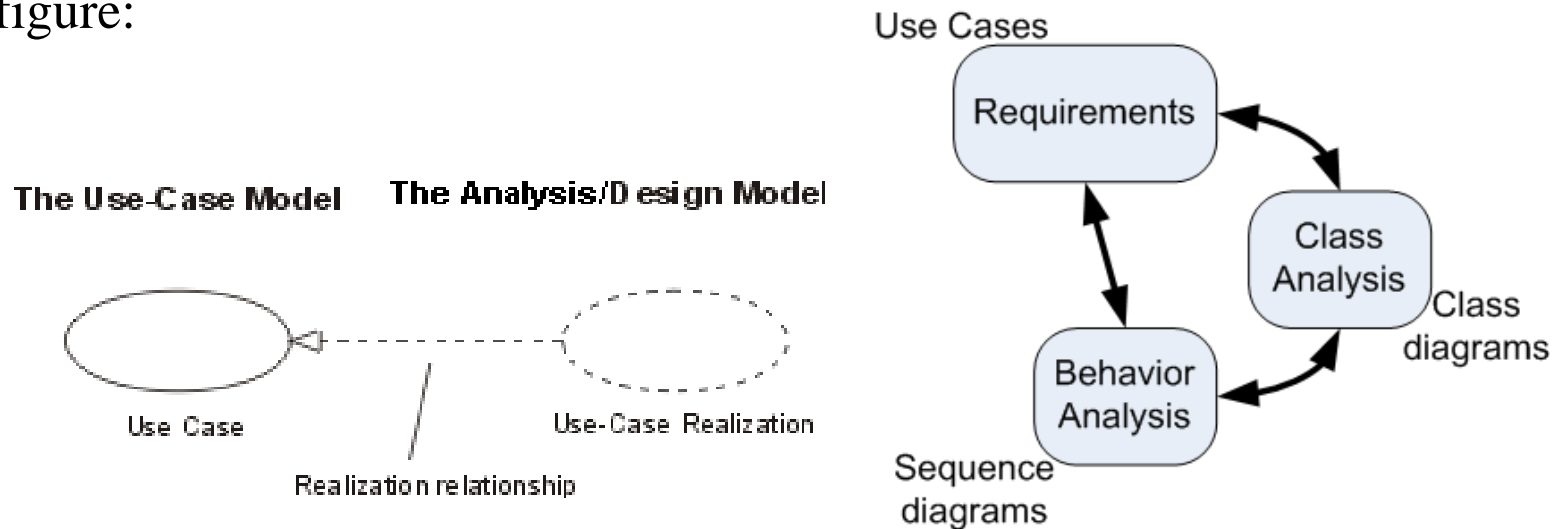
- ❖ A use-case realization represents how a use case will be implemented in terms of collaborating objects. This artifact can take various forms. It can include, for example, a textual description (a document), class diagrams of participating classes and subsystems, and interaction diagrams (communication and sequence diagrams) that illustrate the flow of interactions between class and subsystem instances.
- ❖ The reason for separating the use-case realization from its use case is that doing so allows the use cases to be managed separately from their realizations. This is particularly important for larger projects or families of systems where the same use cases can be designed differently in different products within the product family. Consider the case of a family of telephone switches which have many use cases in common, but which design and implement them differently according to product positioning, performance and price.
- ❖ For larger projects, separating the use case and its realization allows changes to the design of the use case without affecting the baseline use case itself.
- ❖ In a model, a use-case realization is represented as a UML collaboration that groups the diagrams and other information (such as textual descriptions) that form part of the use-case realization.

# Realization of Use case

- ❖ UML diagrams that support use-case realizations can be produced in an analysis context, a design context, or both, depending on the needs of the project. For each use case in the use-case model, there can be a use-case realization in the analysis/design model with a realization relationship to the use case. In UML this is shown as a dashed arrow, with an arrowhead like a generalization relationship, indicating that a realization is a kind of inheritance, as well as a dependency.

## Use case realization :

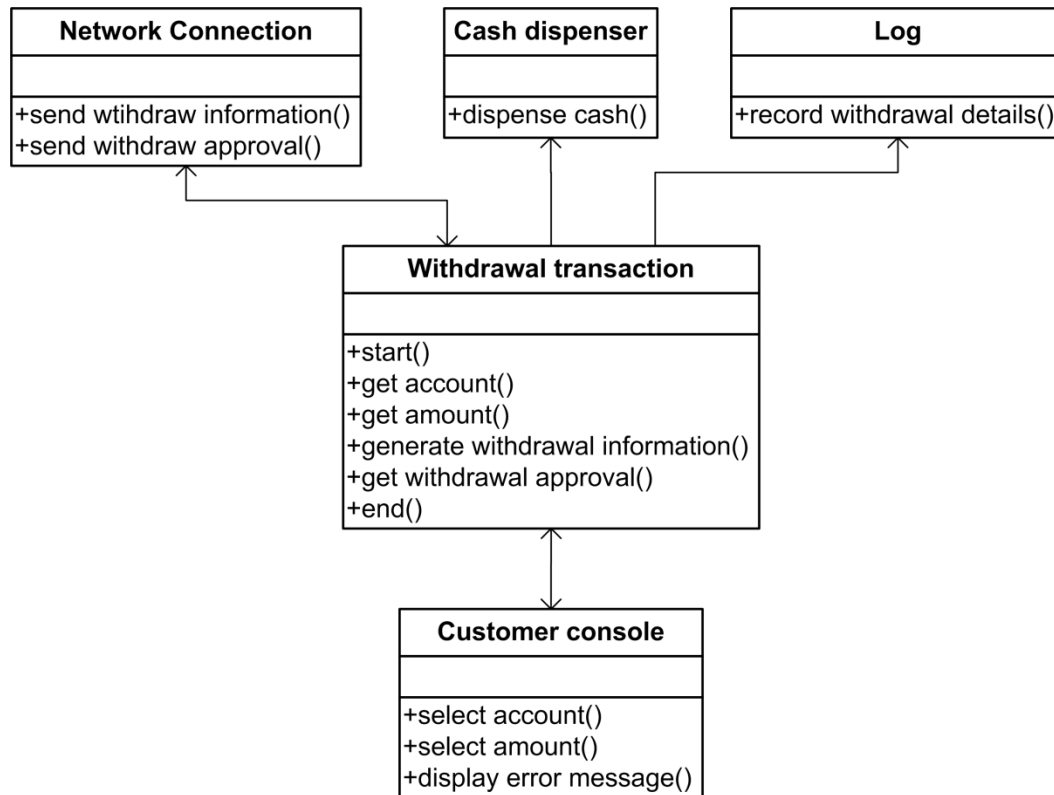
- ❖ Make sure that the sequence diagrams realize (show) the behavior outlined in the use cases and assign behavior to classes in the class diagram. Together, these three continue to evolve and sharpen each other. This can be shown as following figure:



**Fig:** Use case realization process

## Class Diagrams Owned by a Use-Case Realization

- ❖ For each use-case realization there can be one or more class diagrams depicting its participating classes. A class and its objects often participate in several use-case realizations. It is important while designing to coordinate all the requirements on a class and its objects that different use-case realizations can have.
- ❖ The figure below shows an analysis class diagram for the **realization of the Withdraw Cash Item use case.**



**Fig:** Use case realization of Cash Withdraw Use case

# Use case realization → Comn<sup>n</sup> & Sequence Diagrams

- ❖ For each use-case realization there can be one or more interaction diagrams depicting its participating objects and their interactions. There are two types of interaction diagrams: sequence diagrams and communication diagrams. They express similar information, but show it in different ways. Sequence diagrams show the explicit sequence of messages and are better when it is important to visualize the time ordering of messages, whereas communication diagrams show the communication links between objects and are better for understanding all of the effects on a given object and for algorithm design.

**Example, the simple use case for a automobile navigation system below.**

GPS Navigate to Address Use Case

1. Driver starts navigational system

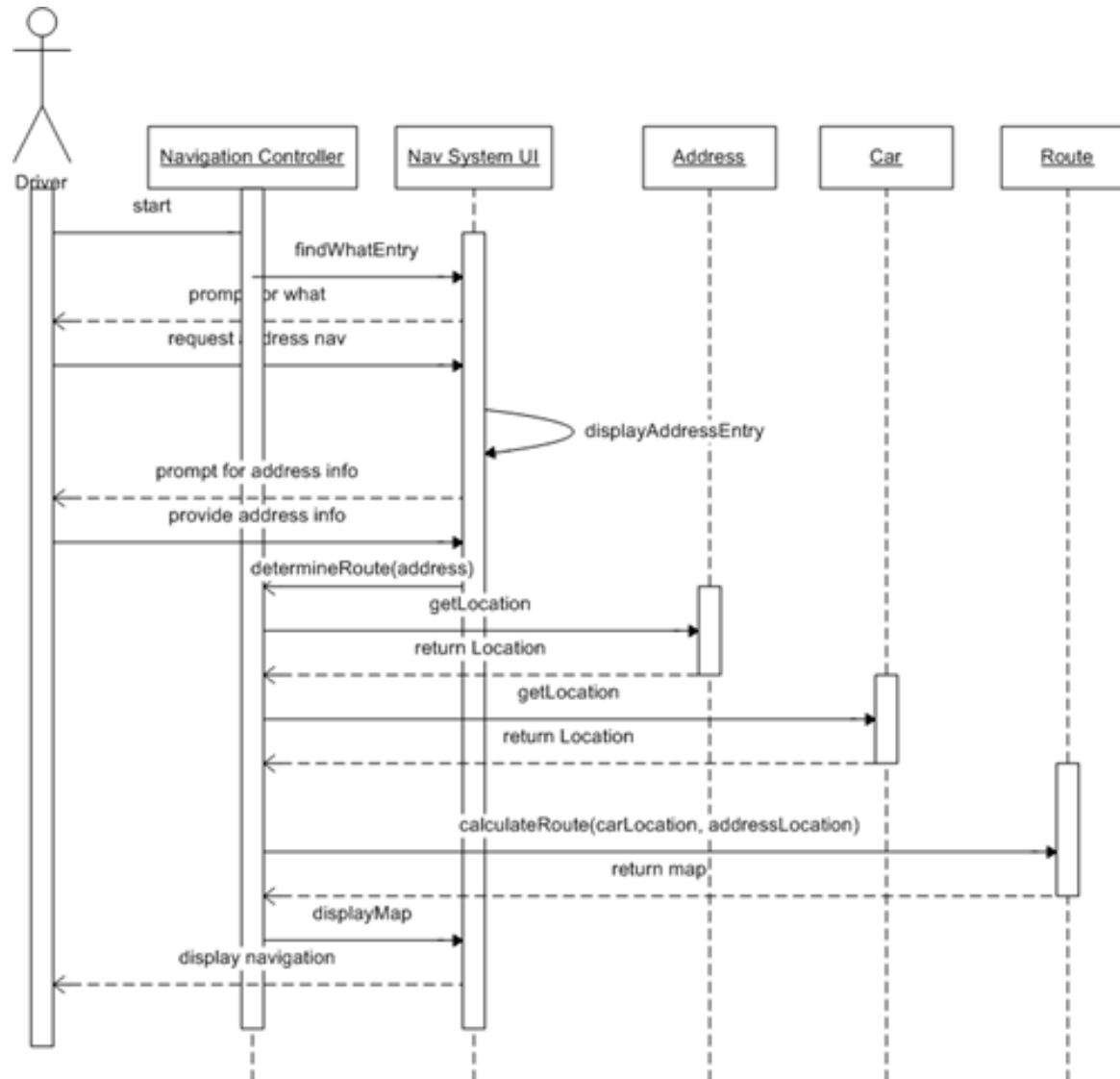
- ✓ System prompts for whether the driver needs help finding an address, intersection, or point of interest

2. Driver selects address

- ✓ System prompts for address (street, city)

3. Driver provides address info

- ✓ System computes location of address
- ✓ System computes car's current location
- ✓ System computes route from current location to address location
- ✓ System locates digital map based on address location
- ✓ System displays appropriate digital map with route information



**Fig:** Realizing Use Cases Automobile navigation System through sequence diagram

# CRC Cards

- ❖ A Class Responsibility Collaborator (CRC) model is a collection of standard index cards that have been divided into three sections:
  - ✓ **A class Name**  
Name of a class.
  - ✓ **A responsibility**  
It is something that a class knows or does.
  - ✓ **A collaborator**  
It is another class that a class interacts with to fulfill its responsibilities.
- ❖ Although not formally part of the UML, It is used to help assign responsibilities and indicate collaboration with other objects\class.
- ❖ The main motive of this model is encouraging objects designers to think more abstractly in terms of responsibility assignment and collaborations.
- ❖ CRC models are an incredibly effective tool for conceptual modeling.



# CRC Cards

## ❖ CRC Layout:

Class Name	
Responsibilities	Collaboration

## ❖ CRC Example:

Customer ← Class Name	
Places orders Knows name Knows address Knows order history  “Responsibilities”	Order ← Collaborating Class

# CRC Cards

## Class:

- ❖ A class represents a collection of similar objects. An object is a person, place, thing, event, or concept that is relevant to the system at hand.
- ❖ For example, in a **University Management System**, classes would represent **students**, **professors**, and **seminars** etc.
- ❖ The name of the class appears across the top of a CRC card and is typically a singular noun or singular noun phrase, such as *Student*, *Professor*, and *Seminar*.
- ❖ The information about a student describes a single person, not a group of people. Therefore, we use the name Student and not Students.
- ❖ Class names should also be simple and domain specific. For example, which name is better: Student or Person who takes seminars? [i.e. Domain specific naming as we already discussed in Domain modeling]

# CRC Cards

## Responsibility:

- ❖ A **responsibility** is anything that a class knows or does.
- ❖ For example, students have names, addresses, and phone number etc. These are the things a student knows.
- ❖ Students also enroll in seminars, drop seminars, and request transcripts. These are the things a student does.
- ❖ So we can conclude that, the things a class knows and does constitute its responsibilities.

# CRC Cards

## Collaborations:

- ❖ Sometimes a class has a responsibility to fulfill, but doesn't have enough information to do it.
- ❖ For example, as we know students enroll in seminars. To do this, a student needs to know if a spot is available in the seminar and, if so, then he needs to be added to the seminar.
- ❖ However, students only have information about themselves (their names and so forth), and not about seminars.
- ❖ What the student needs to do is collaborate/interact with the card labeled *Seminar* to sign up for a seminar. Therefore, *Seminar* is included in the list of collaborators of *Student*.

## Collaboration takes one of two forms:

- ✓ A request for information
- ✓ a request to do something.
- ❖ **A request for information:** *Student* requests an indication from the card *Seminar* whether a space is available.
- ❖ **A request to do something:** *Student* requests to be added to the *Seminar*.

# CRC Cards Example

**CRC card for Student:**

<b>Student</b> ← Class Name	
<b>Student number</b> <b>Name</b> <b>Address</b> <b>Phone Number</b> <b>Enroll in a seminar</b> <b>Drop a seminar</b> <b>Request transcripts</b>  “Responsibilities”	<b>Seminar</b> ← Collaborating Class

# CRC Cards

## Steps to create CRC cards

### ❖ Find classes:

- ✓ A good rule of thumb is that you should look for the three-to-five main classes right away, such as *Student*, *Seminar*, and *Professor* etc (in case of university management system).
- ✓ Sometimes you can include UI classes such as *Transcript* and *Student Schedule* etc.

### ❖ Find responsibilities.

To find responsibilities of a class, we must find out:

- ✓ what a class does.
- ✓ what information you wish to maintain about it.

# CRC Cards

## ❖ Define collaborators

- ✓ A class often does not have sufficient information to fulfill its responsibilities. Therefore, it must collaborate with other classes to get the job done.
- ✓ Collaboration will be in one of two forms: **a request for information or a request to perform a task.**
- ✓ To identify the collaborators of a class for each responsibility we must ask a question "does the class have the ability to fulfill this responsibility?". If not then look for a class that either has the ability to fulfill the missing functionality or the class which should fulfill it.

# CRC Cards Example → UMS

Student	
Name SID Address Phone number Email Address Average mark received List of seminars taken	Seminar

Professor	
Name PID Address Phone number Email Address Salary List of seminars instructing	Seminar

Seminar	
Name Seminar Number Fees Waiting list Enrolled students Instructor Add students Drop students	Student Professor



# *Design Class Diagram*

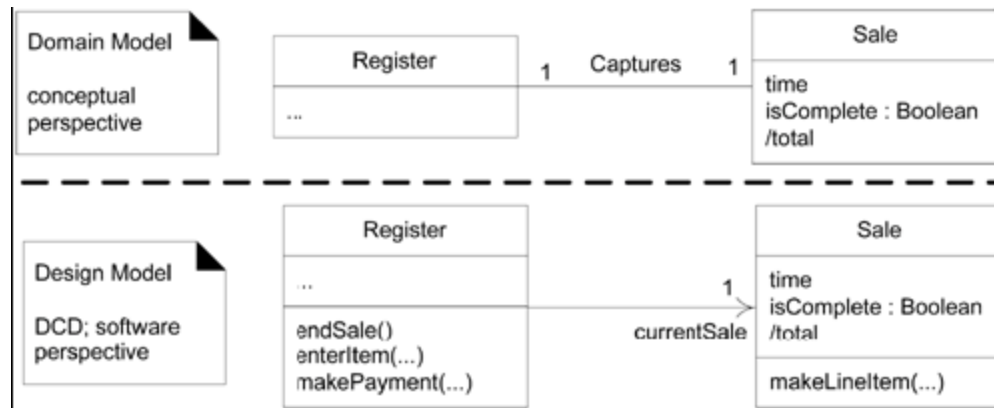
# Design Class Diagram

## Class Diagrams

- ❖ The UML includes class diagrams to illustrate classes, interfaces, and their associations. They are used for static object modeling. A class diagram describes the static structure of a system. It shows how a system is structured rather than how it behaves. The static structure of a system comprises of a number of class diagrams and their dependencies. The main constituents of a class diagram are classes and their relationships: generalization, aggregation, association, and various kinds of dependencies.

## Design Class Diagram(DCD)

- ❖ In a conceptual perspective the class diagram can be used to visualize a domain model. For discussion, we also need a unique term to clarify when the class diagram is used in a software or design perspective. A common modeling term for this purpose is design class diagram (DCD).



**Fig:** Different Perspectives in Class Diagram

## UML Class Diagram

- ❖ A class diagram is a **type of static structure diagram** that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

### A UML class diagram is made up of:

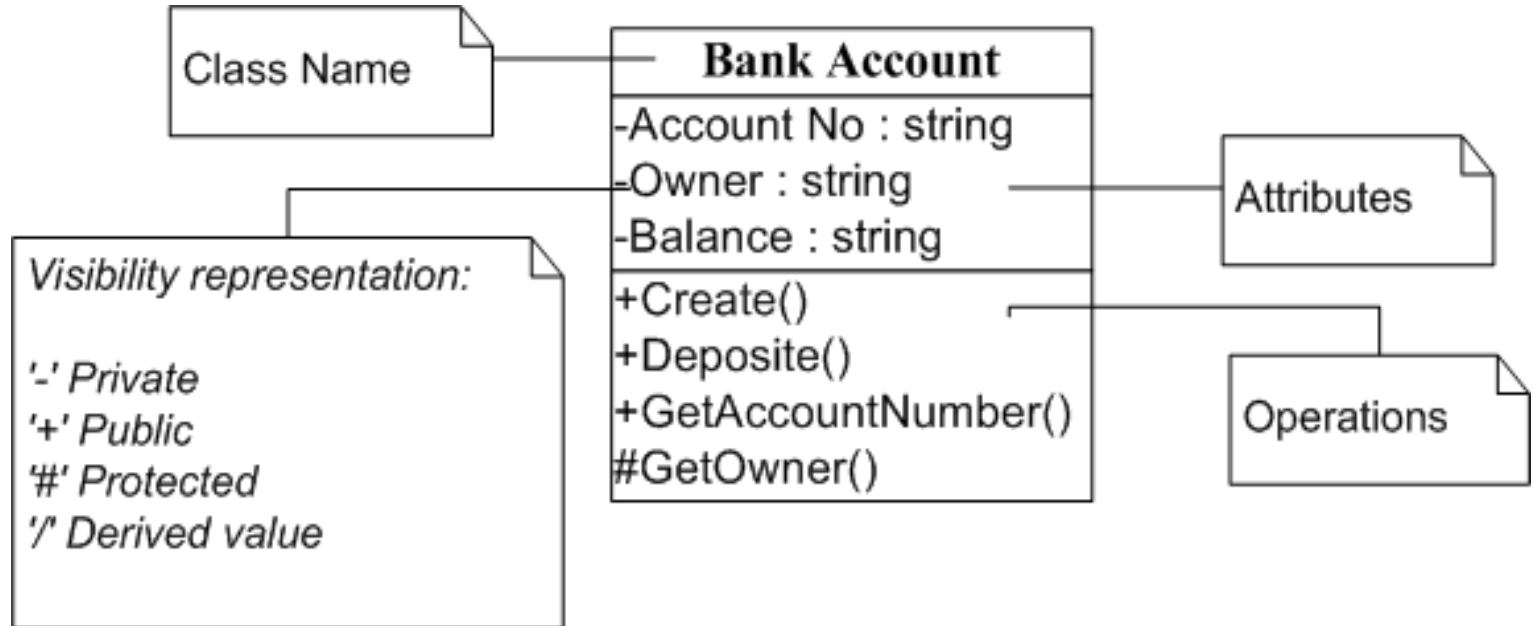
- ✓ A set of classes
- ✓ A set of relationships between classes

## What is a Class

- ❖ A description of a group of objects all with similar roles in the system.
- ❖ which consists of:
  - ✓ **Structural features** i.e. attributes
  - ✓ **Behavioral features** i.e. operations

# DCD

## Class Notations:



- ❖ The graphical representation of the class: **Bank Account** as shown above:
  - ✓ **Bank Account** has 3 attributes and 4 operations.
  - ✓ All 3 parameters/ attributes are of type string

## **A class notation consists of three parts:**

### **Class Name**

- ✓ The name of the class appears in the first partition.

### **Class Attributes**

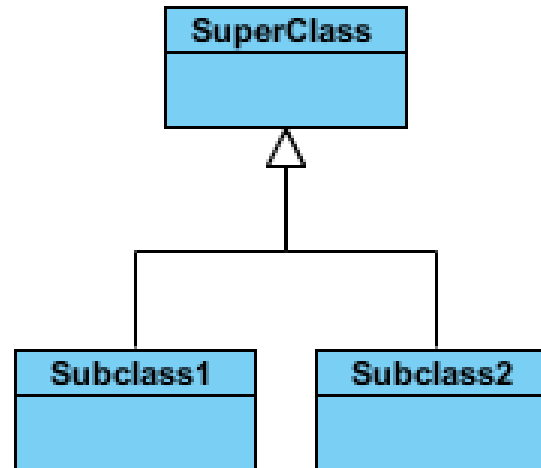
- ✓ Attributes are shown in the second partition.
- ✓ The attribute type is shown after the colon.
- ✓ Attributes map onto member variables (data members) in code.

### **Class Operations or Methods:**

- ✓ Operations are shown in the third partition. They are services the class provides.
- ✓ The return type of a method is shown after the colon at the end of the method signature.
- ✓ The return type of method parameters are shown after the colon following the parameter name.
- ✓ Operations map onto class methods in code

# Relationship Types in DCD

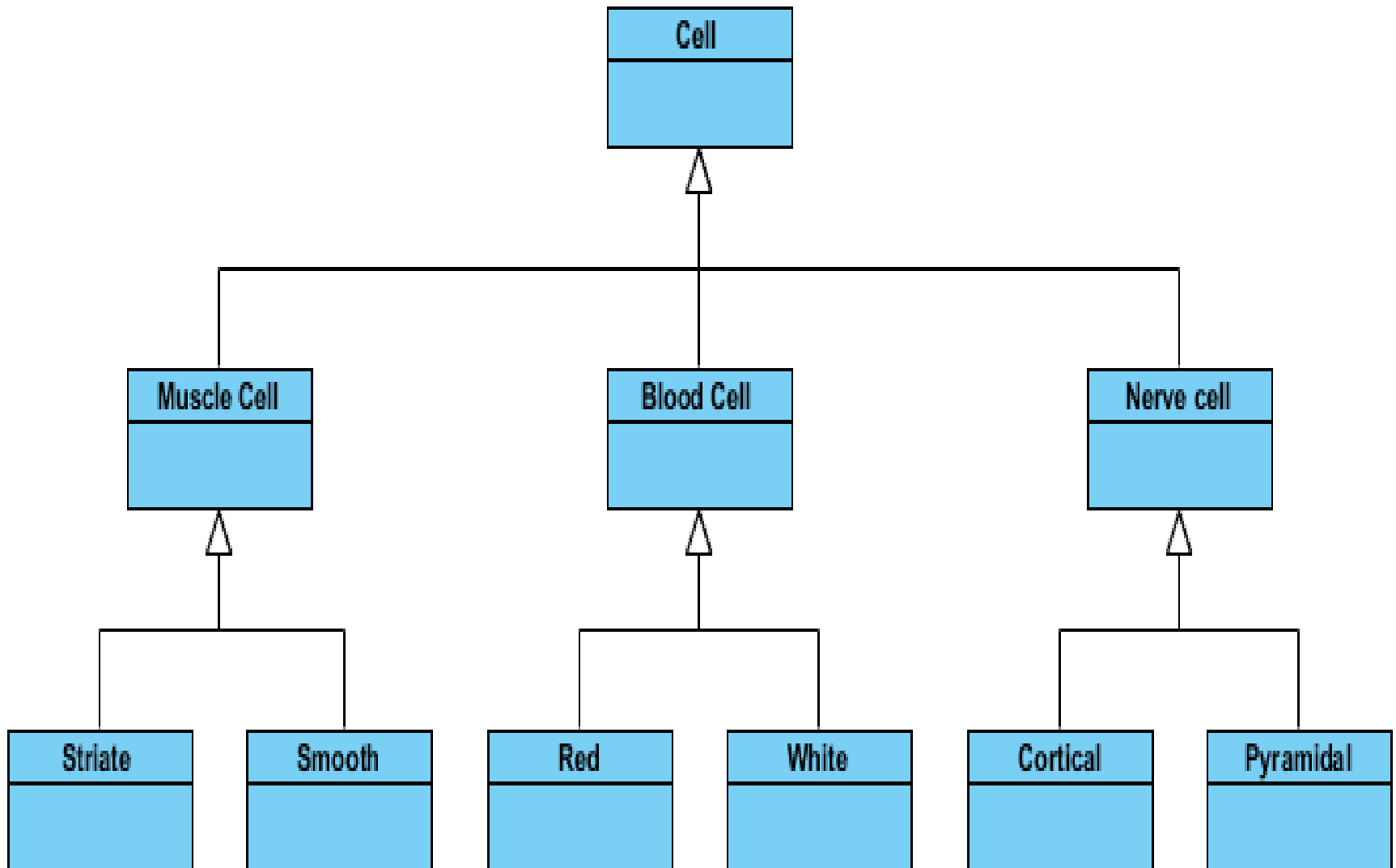
## Inheritance (or Generalization):



- ❖ Represents an "is-a" relationship.
- ❖ A solid line with a hollow arrowhead that point from the child to the parent class.
- ❖ **SubClass1** and **SubClass2** are Child class and **Super Class** is parent class.

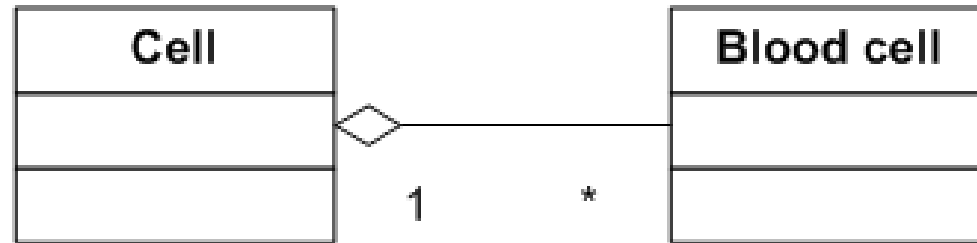
# Relationship Types in Class Diagram

**Inheritance example:**



# Relationship Types in Class Diagram

## Aggregation:

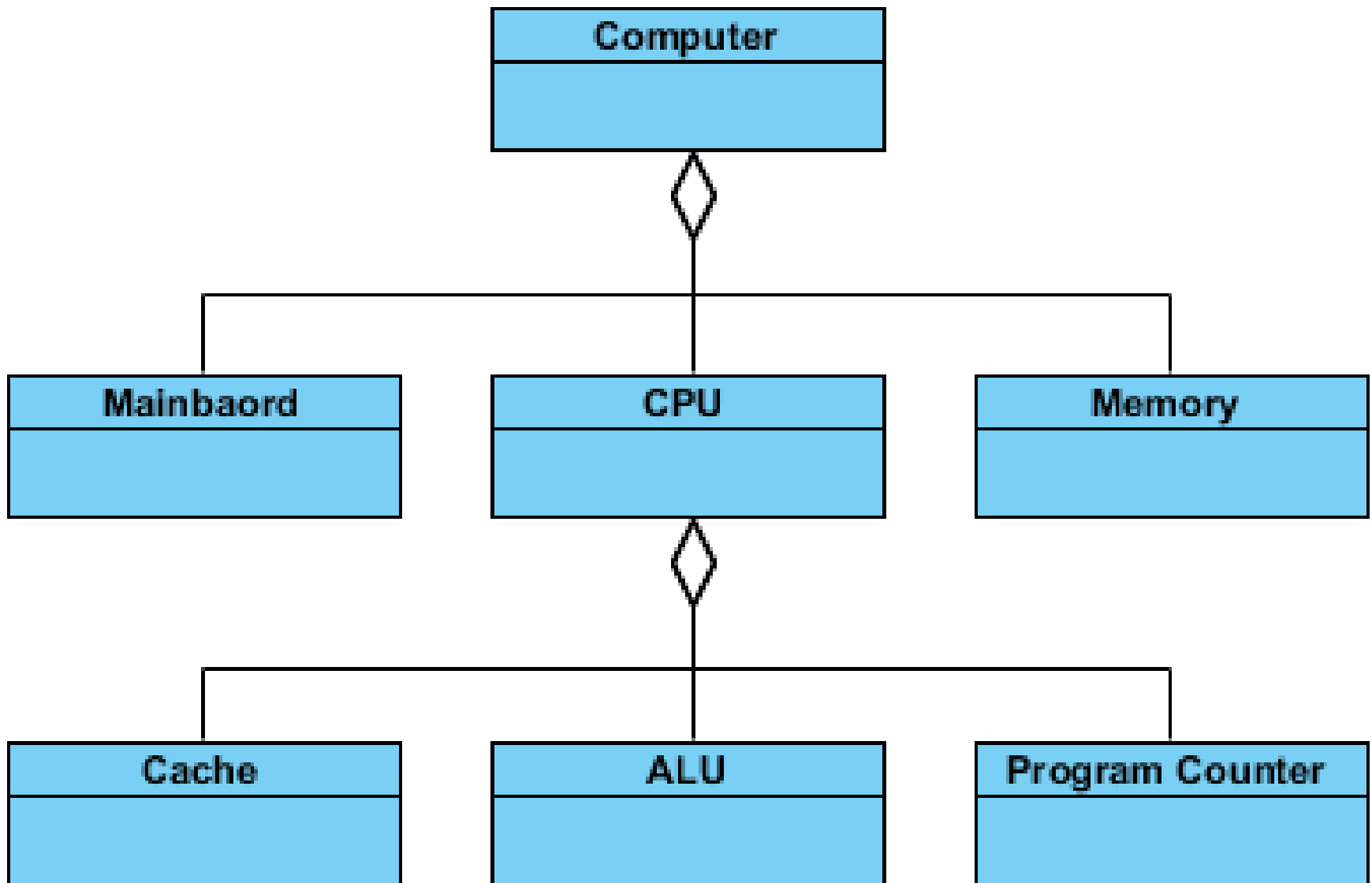


- ❖ It represents a "part of" relationship.
- ❖ Class2 is part of Class1.
- ❖ A solid line with a unfilled diamond at the association end connected to the class of composite.
- ❖ Many instances (denoted by the \*) of Class Blood cell can be associated with Class Cell.
- ❖ Objects of Class Blood cell and Class Cell have separate lifetimes.



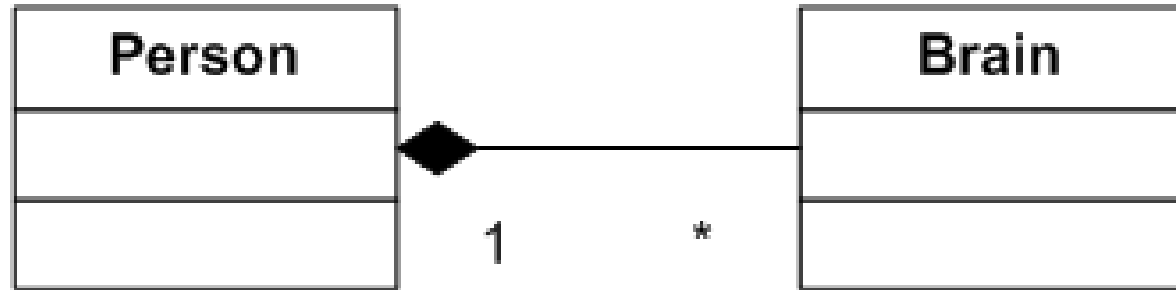
# Relationship Types in Class Diagram

Aggregation example:



# Relationship Types in Class Diagram

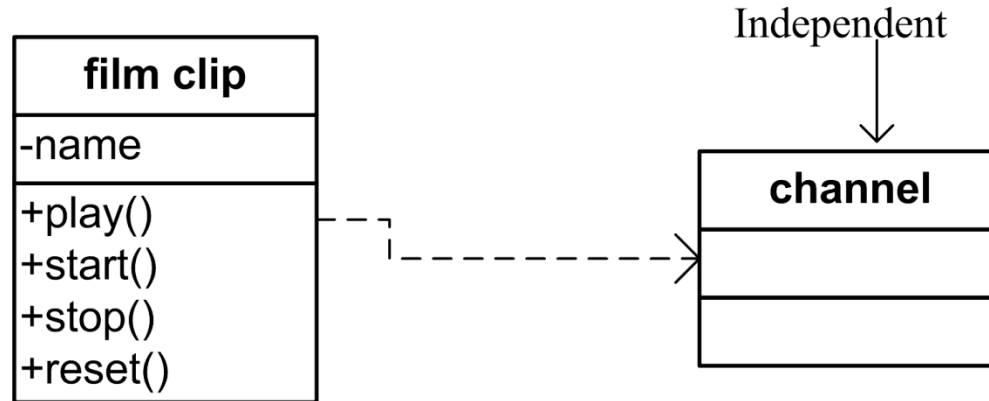
## Composition:



- ❖ A special type of aggregation where parts are destroyed when the whole is destroyed.
- ❖ Objects of Class Brain live and die with Class Person.
- ❖ Class Brain cannot stand by itself.
- ❖ A solid line with a filled diamond at the association connected to the class of composite

# Relationship Types in Class Diagram

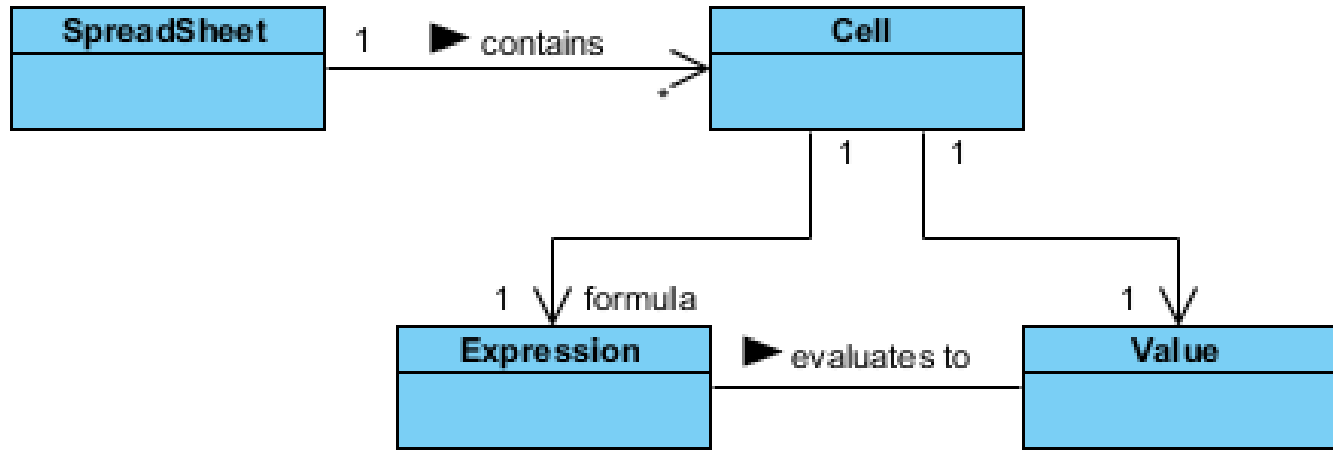
## Dependency:



- ❖ Exists between two classes if changes to the definition of one may cause changes to the other (but not the other way around).
- ❖ Class **film clip** depends on class **channel**.
- ❖ A dashed line with an open arrow

# Relationship Types in Class Diagram

## Relationship Name:



- ❖ Names of relationships are written in the middle of the association line.
- ❖ Good relation names make sense when you read them out loud:
  - ✓ "Every spreadsheet **contains** some number of cells",
  - ✓ "an expression **evaluates to** a value"
- ❖ They often have a **small arrowhead to show the direction** in which direction to read the relationship, e.g., expressions evaluate to values, but values do not evaluate to expressions.

# Relationship Types in Class Diagram

## Visibility of Class attributes and Operations

- ❖ [See Class Diagram in UML Notations](#)
- ❖ In object-oriented design, there is a notation of visibility for attributes and operations.
- ❖ UML identifies four types of visibility:
  - ❖ **public, protected, private, and package.**
- ❖ The +, -, # and ~ symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.
  - + denotes public attributes or operations
  - denotes private attributes or operations
  - # denotes protected attributes or operations
  - ~ denotes package attributes or operations

# Relationship Types in Class Diagram

## Multiplicity

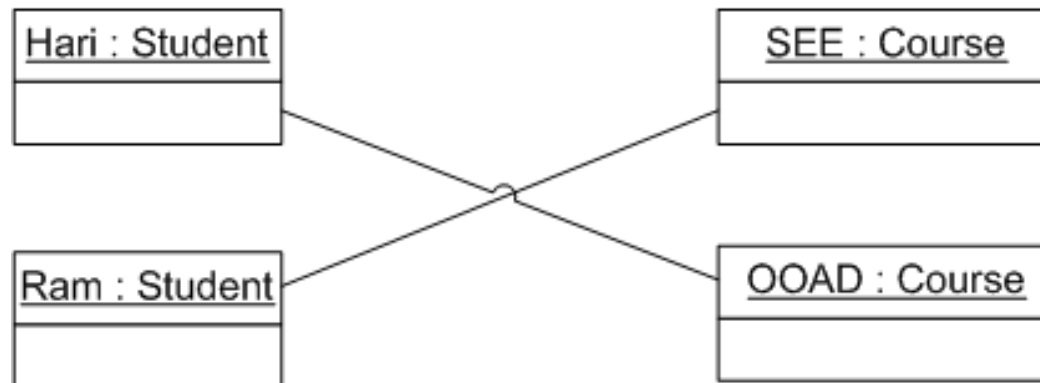
- ❖ How many objects of each class take part in the relationships and multiplicity can be expressed as:
  - ✓ 1- Exactly one.
  - ✓ 0..1- Zero or one.
  - ✓ 0..\* or \* - Many.
  - ✓ 1..\* - One or more.
  - ✓ 3..4 or 6- Exact Number { minimum 3 maximum 4 or exactly 6}
  - ✓ complex relationship - e.g. 0..1, 3..4, 6..\* would mean any number of objects other than 2 or 5

# Relationship Types in Class Diagram

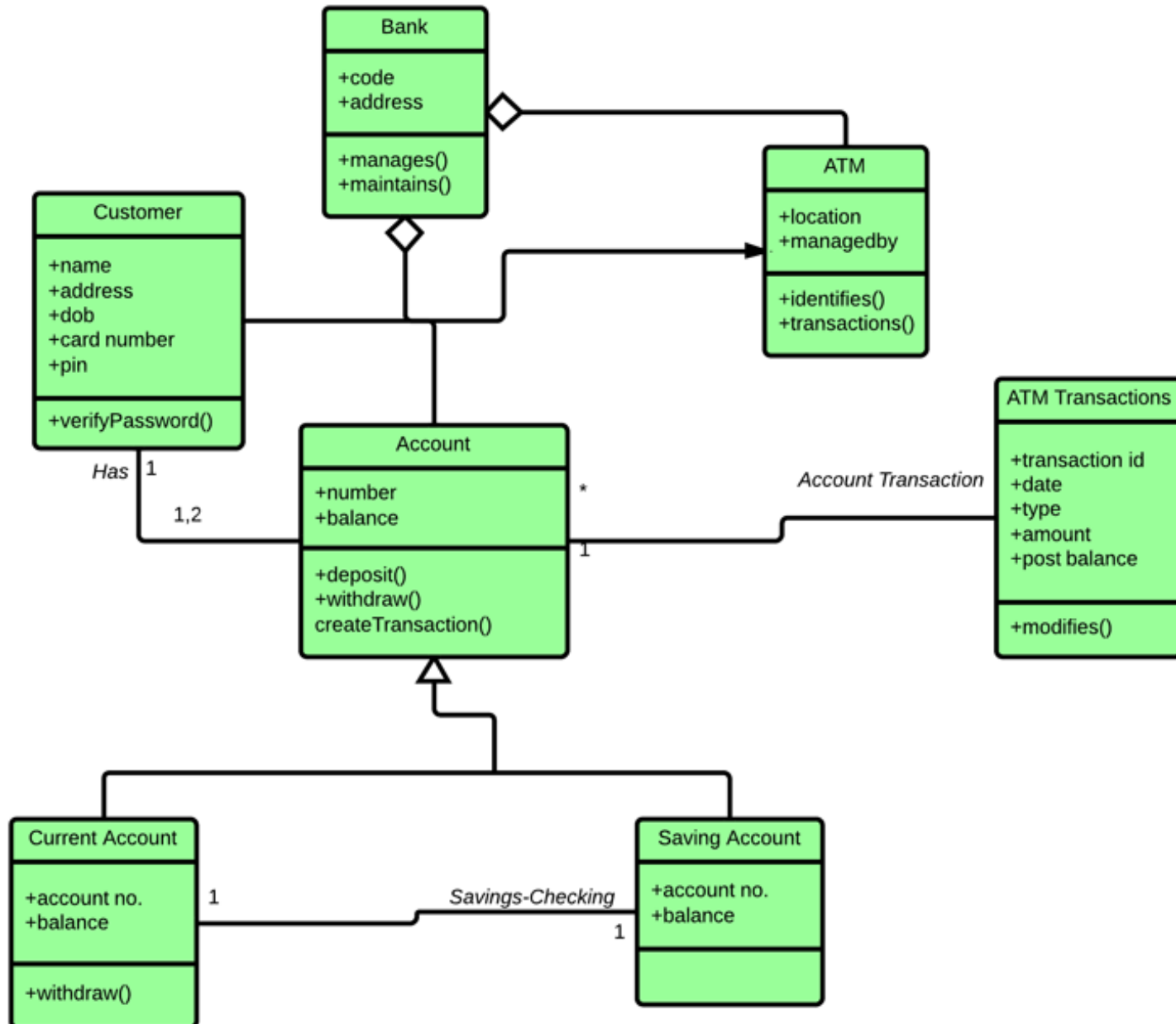
## Multiplicity:



A Student can take many Courses and many Students can be enrolled in one Course.

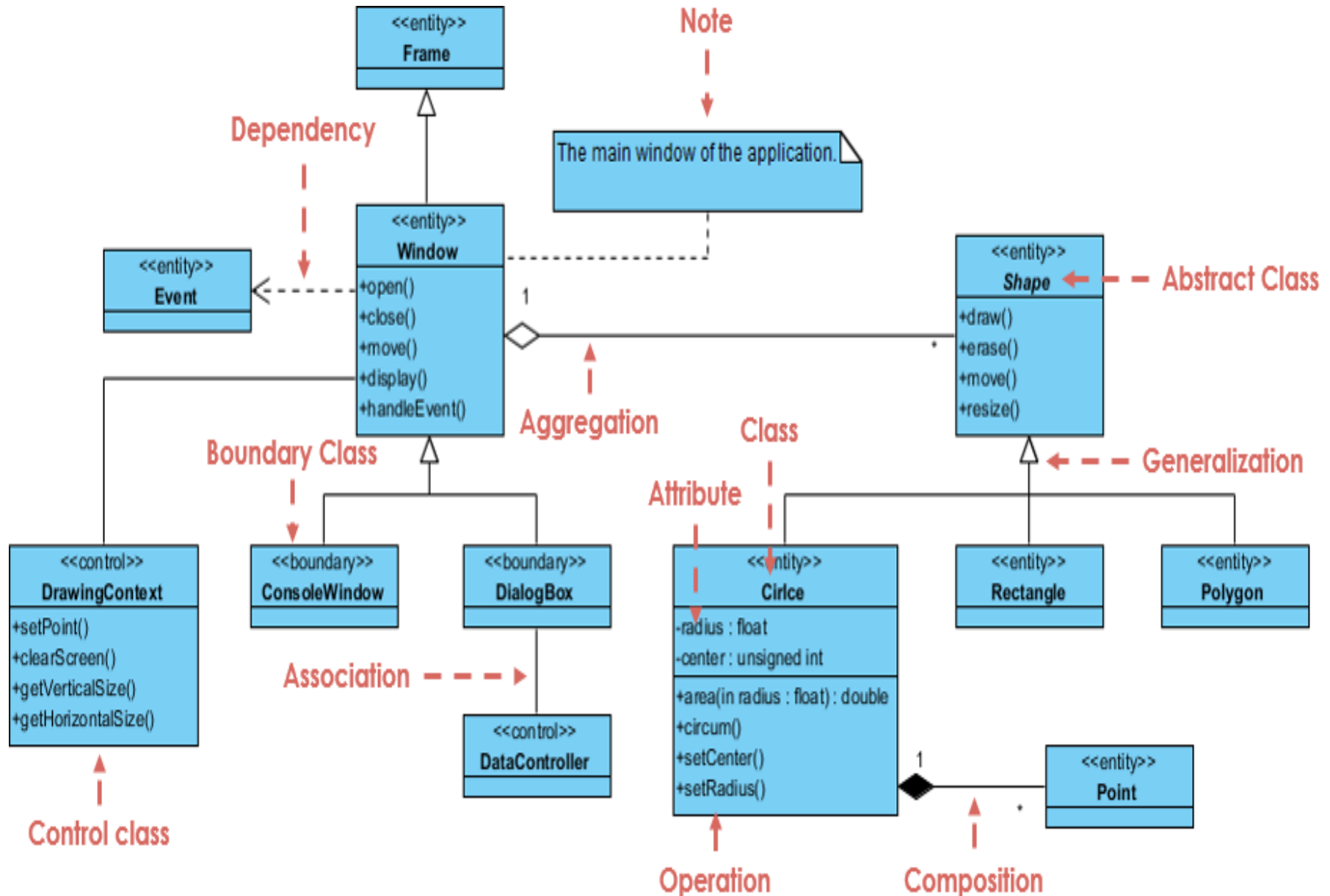


# Class diagram: ATM Machine





# Description: Assignment



# Design Class Diagram

## Description:

- ❖ Shape is an abstract class. It is shown in *Italics*.
- ❖ Shape is a superclass. Circle, Rectangle and Polygon are derived from Shape. In other words, a Circle is-a Shape. This is a generalization / inheritance relationship.
- ❖ There is an association between DialogBox and DataController.
- ❖ Shape is part-of Window. This is an aggregation relationship. Shape can exist without Window.
- ❖ Point is part-of Circle. This is a composition relationship. Point cannot exist without a Circle.
- ❖ Window is dependent on Event. However, Event is not dependent on Window.
- ❖ The attributes of Circle are radius and center. This is an entity class.
- ❖ The method names of Circle are area(), circum(), setCenter() and setRadius().
- ❖ The parameter radius in Circle is an in parameter of type float.
- ❖ The method area() of class Circle returns a value of type double.
- ❖ The attributes and method names of Rectangle are hidden. Some other classes in the diagram also have their attributes and method names hidden.

# *Object Diagram*

# Object Diagram

- ❖ Object is an instance of a class i.e. instance of a particular moment in runtime.
- ❖ A static UML object diagram is an instance of a class diagram.
- ❖ It shows a snapshot of the detailed state of a system at a point in time, thus an object diagram encompasses objects and their relationships at a point in time.
- ❖ The use of object diagrams is fairly limited, mainly to show examples of data structures.
- ❖ The best way to illustrate what an object diagram look like is to show the object diagram derived from the corresponding class diagram.

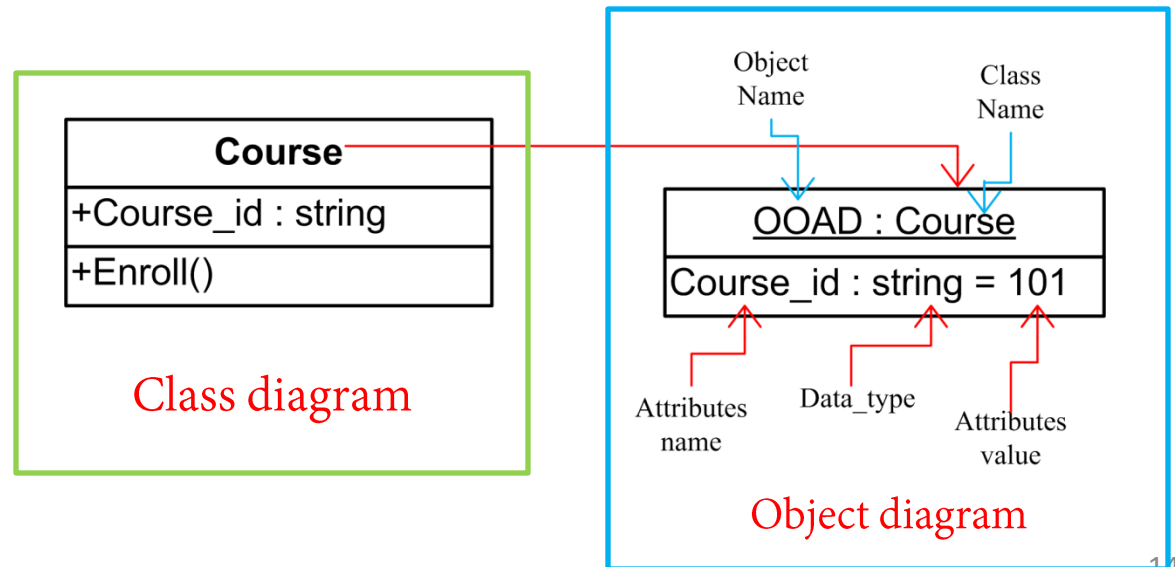
# Object Diagram Notations

## Object Names:

- ❖ Every object is actually symbolized like a rectangle, that offers the name from the object and **its class underlined as well as divided with a colon**.

## Object Attributes:

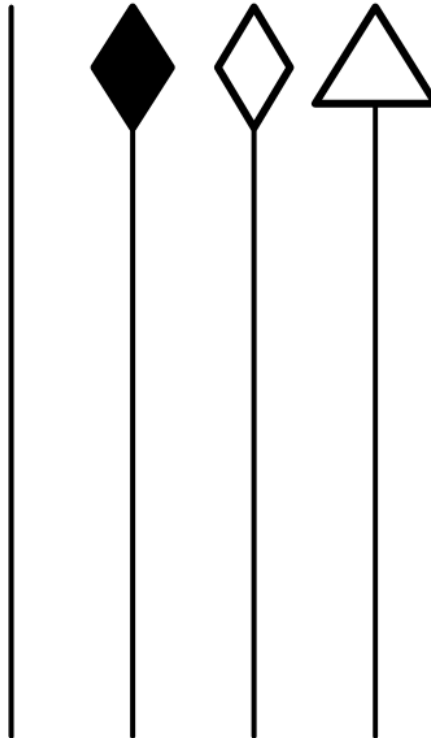
- ❖ Similar to classes, you are able to list object attributes inside a separate compartment. However, unlike classes, **object attributes should have values assigned for them**.



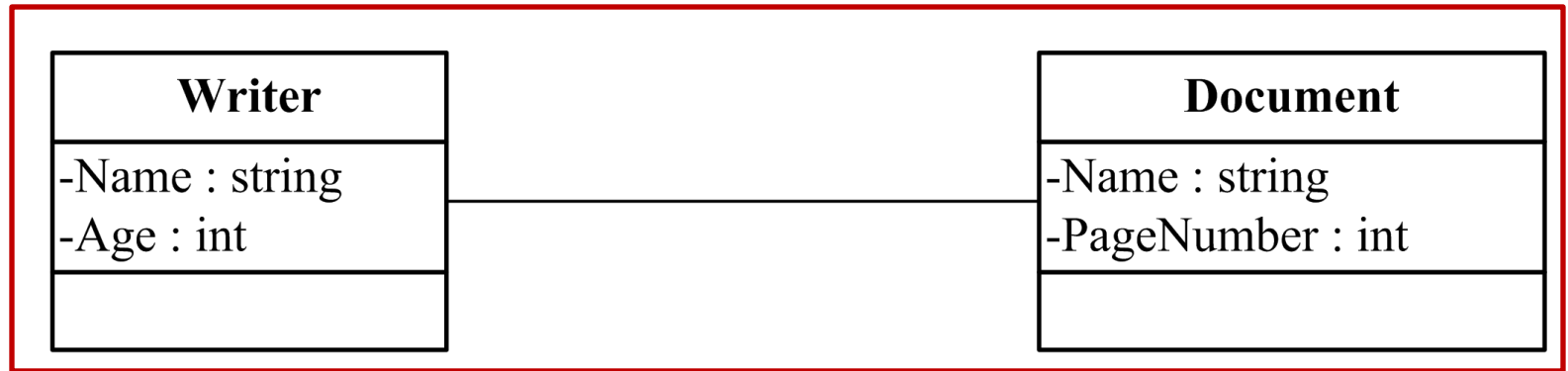
# Object Diagram Notations

## Links:

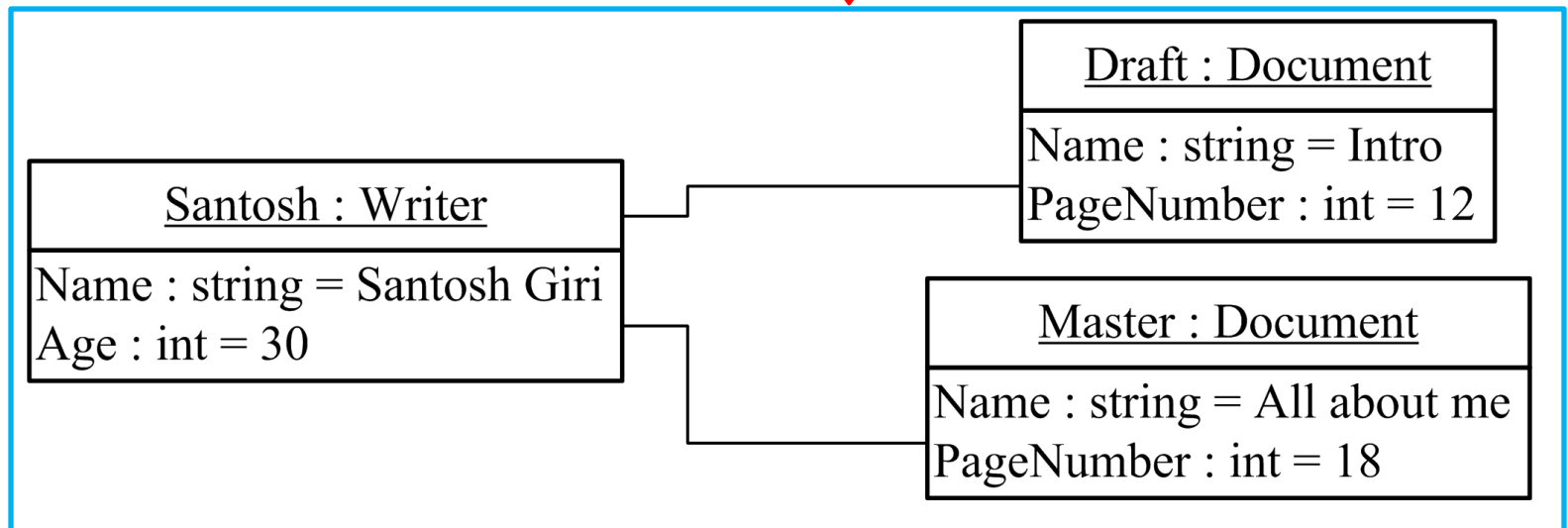
- ❖ Links tend to be instances associated with associations. You can draw a link while using the lines utilized in class diagrams.



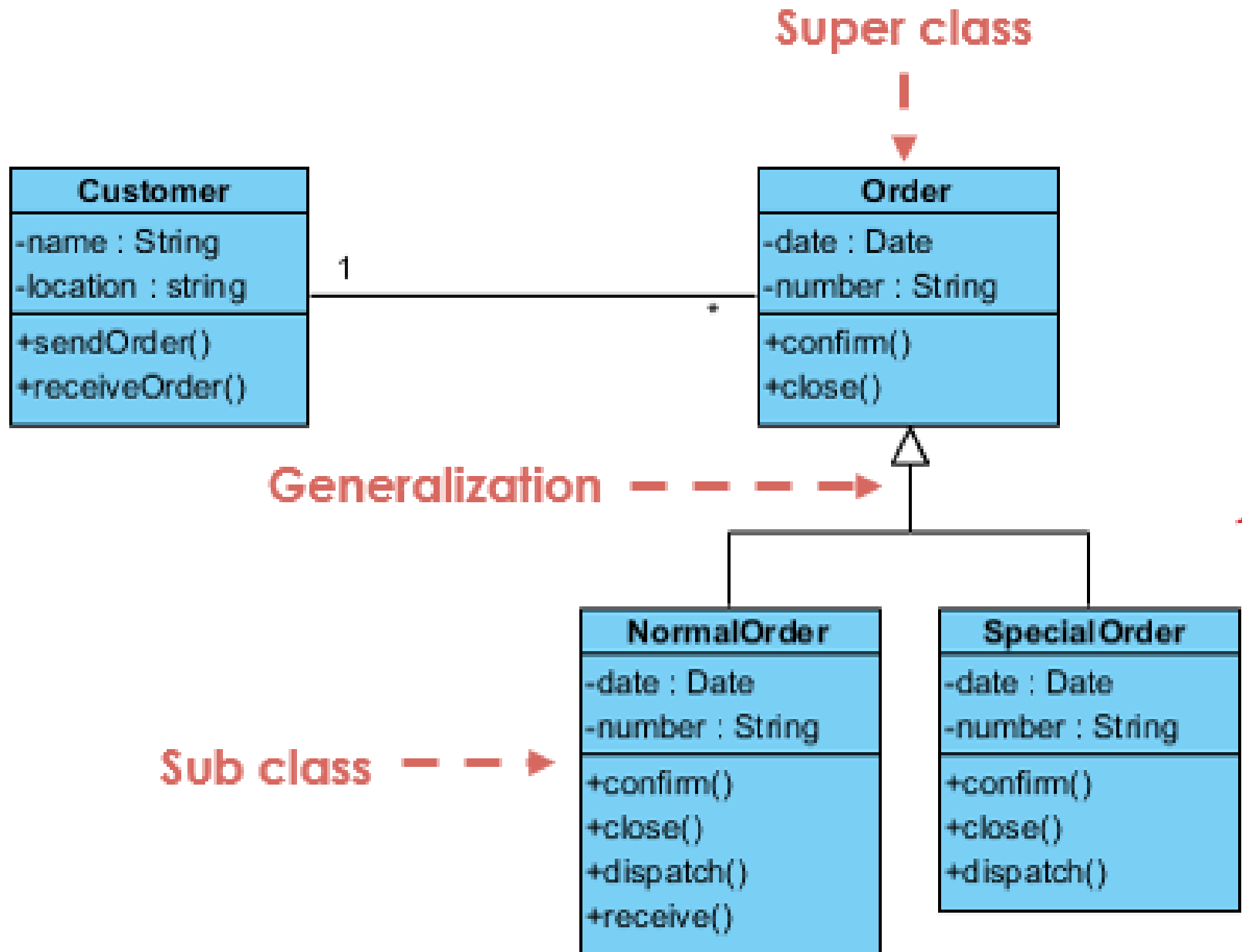
# Class Diagram example



Class to  
Object ↓

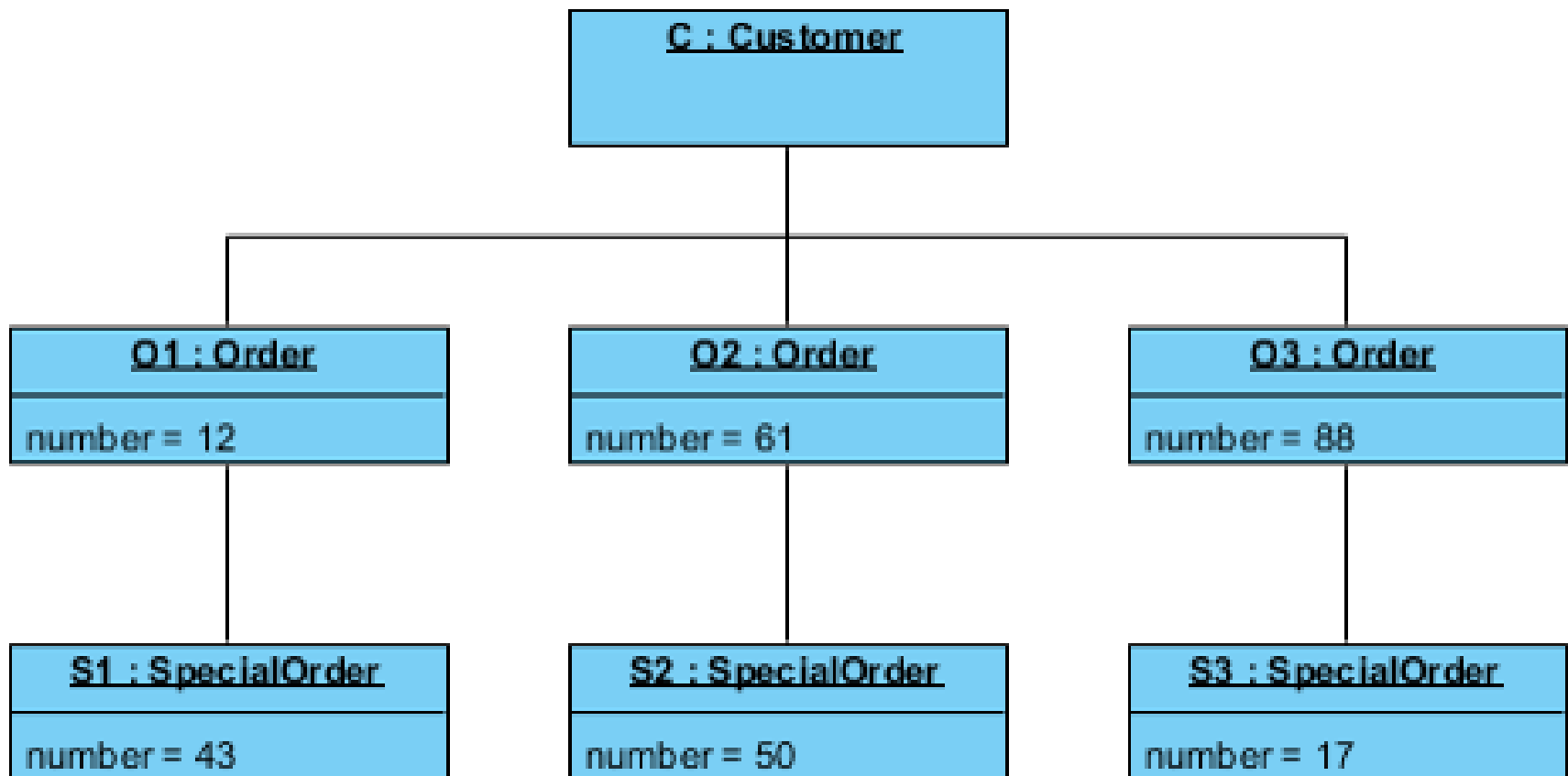


# Class Diagram: Order Management System

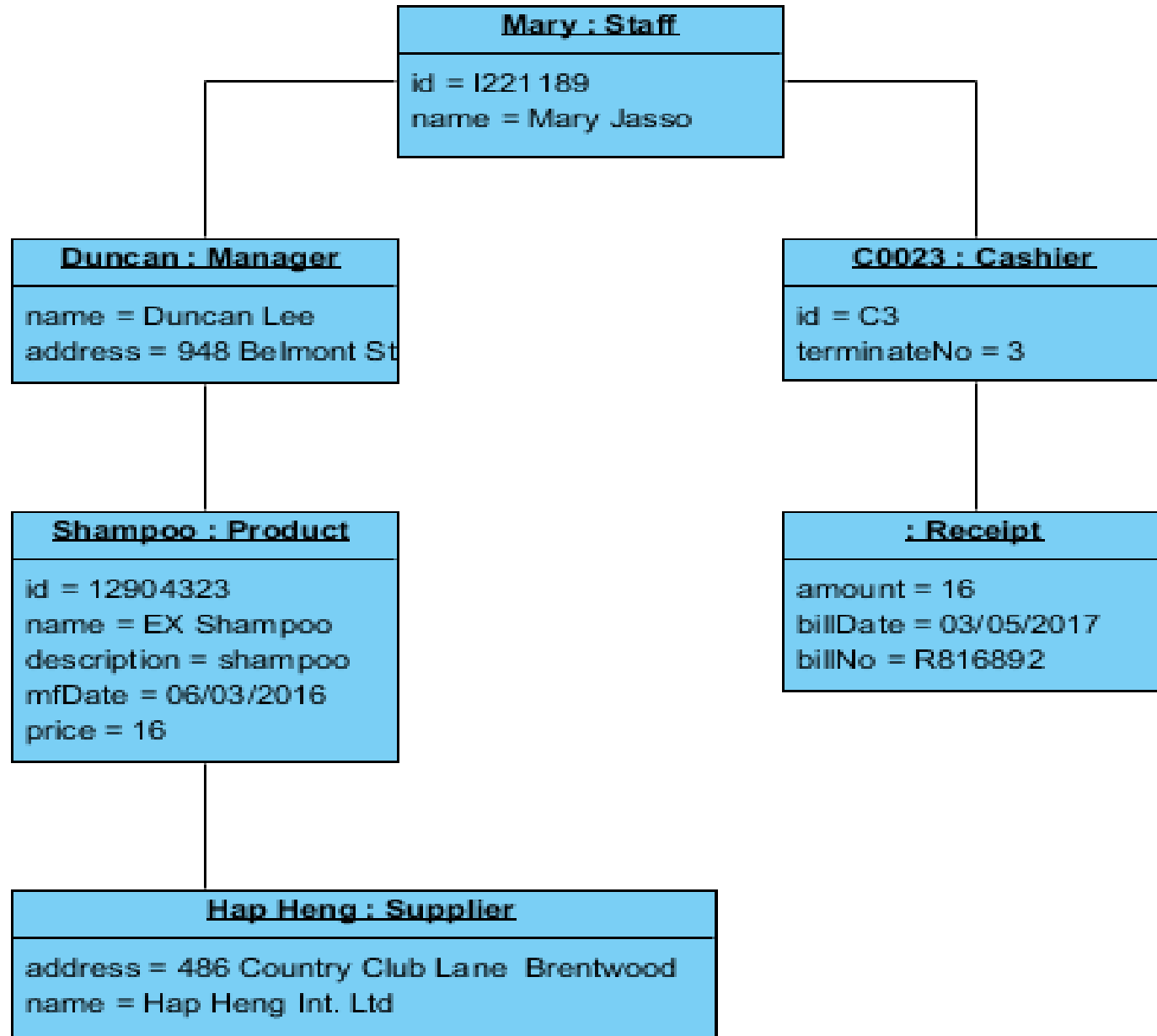




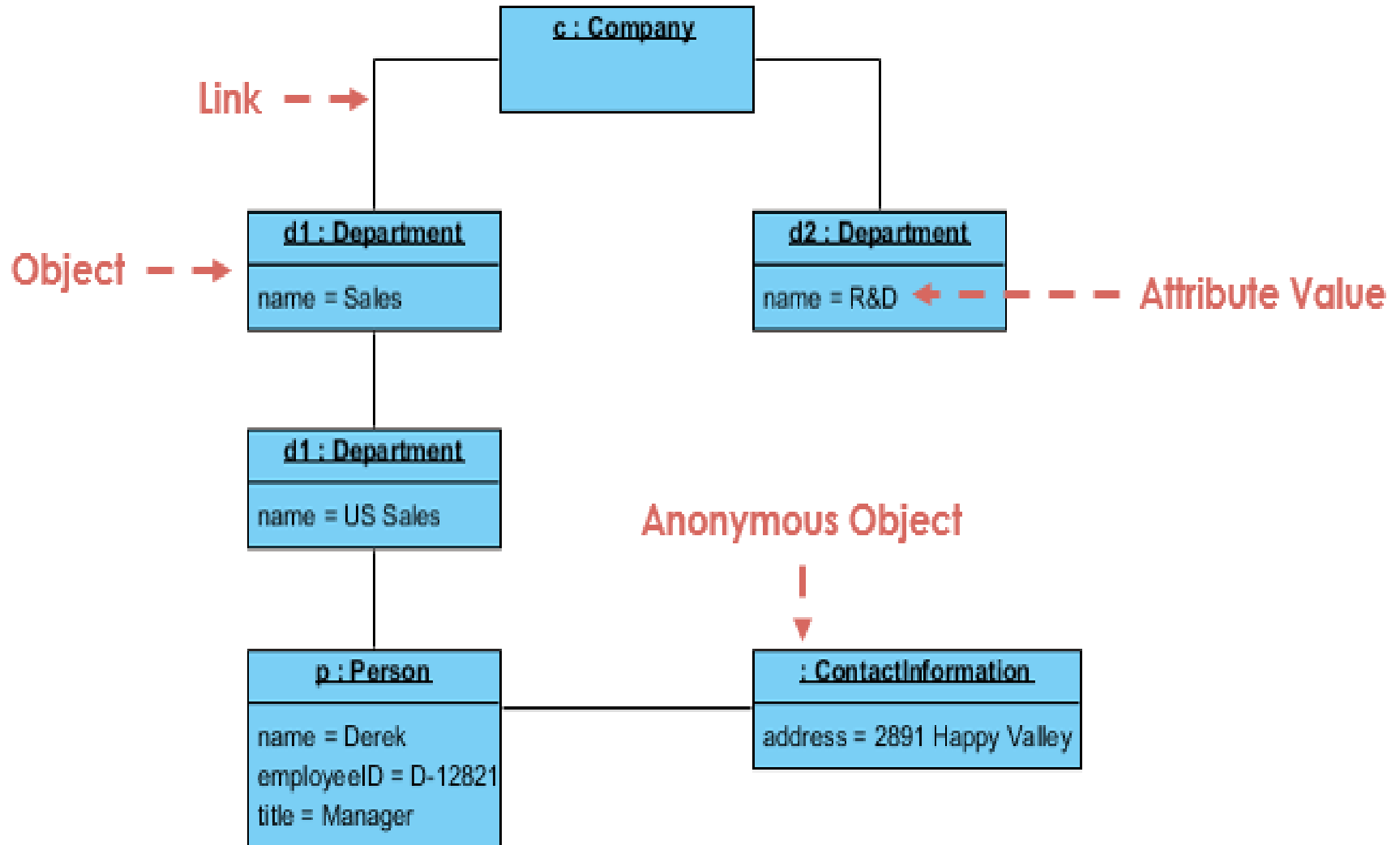
# Object Diagram: Order Management System



# Object Diagram: POS



# Object Diagram: Company Structure



# *Design Pattern*

# Design Patterns

- ❖ In software development, a pattern (or *design pattern*) is **a written document that describes a general solution to a design problem that recurs repeatedly in many projects.**
- ❖ Or a pattern is a named description of a problem & its solution that can be applied to new contexts.
- ❖ Software designers adapt the pattern solution to their specific project.
- ❖ A design pattern **isn't a finished design** that can be transformed directly into code. It is a **description or template** for how to solve a repeated problems.

# Design Patterns

## Pattern elements

There are four essential elements of the design patterns:

### ❖ Name

- ✓ A name that is a meaningful reference to the pattern

### ❖ Problem description

- ✓ Description of the problem.

### ❖ Solution description

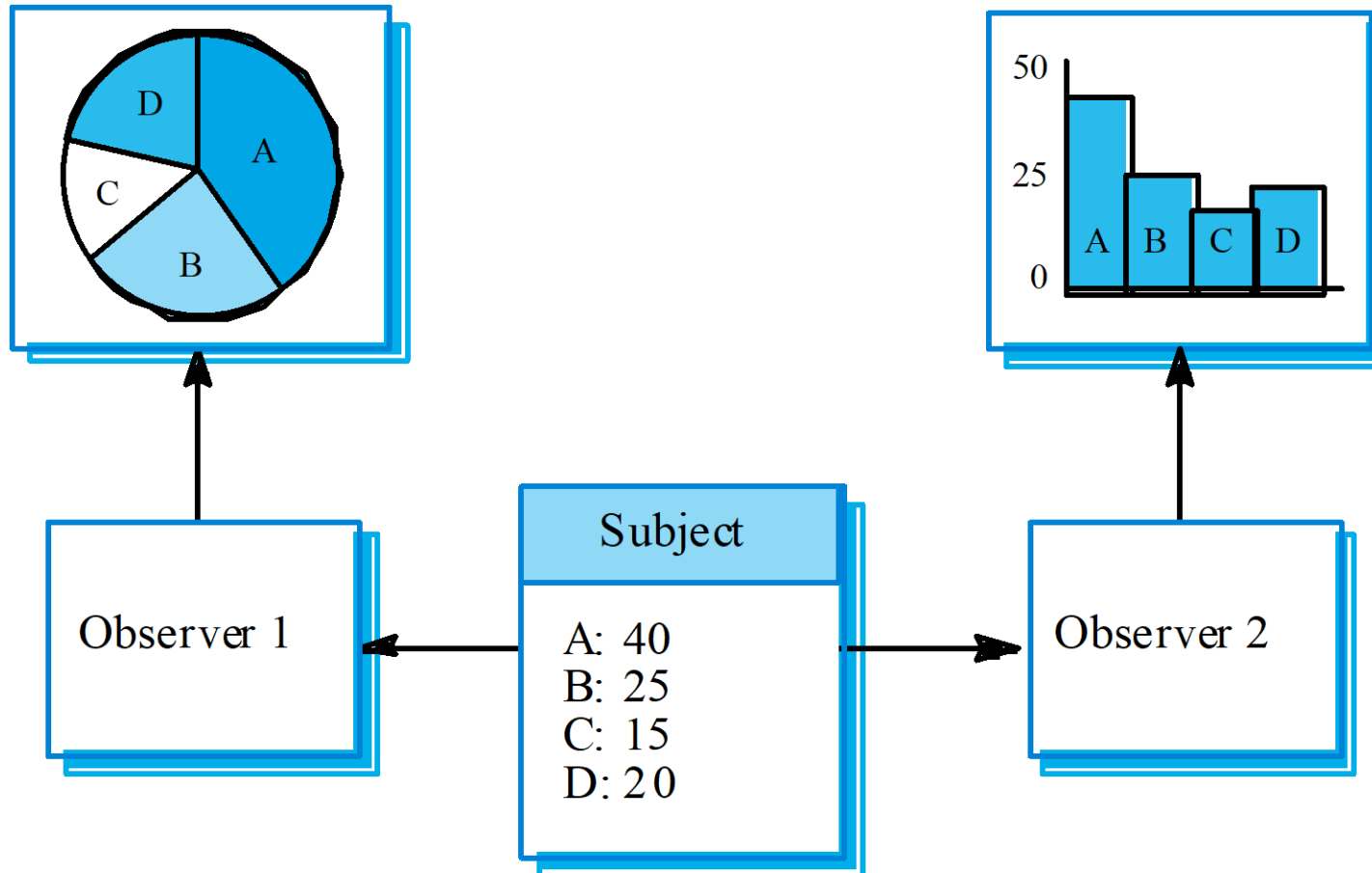
- ✓ Not a concrete design but a template for a design solution that can be implemented in different ways.

### ❖ Consequences

- ✓ The results and trade-offs of applying the pattern
- ✓ Helps the designer to understand whether a pattern can be effectively applied in particular situation

# Design Patterns

Two graphical representations of same data.



# Design Patterns

## Assignment

# Design Patterns and Its uses in OOAD



# GRASP

**GRASP (General Responsibility Assignment Software Principles) :**

- ❖ **GRASP helps us in deciding which responsibility should be assigned to which object/class** i.e. it helps to guide object oriented design by clearly outlining who does what.
- ❖ Which object or class is responsible for what actions.
- ❖ Helps us to define how classes work with one another.
- ❖ **9 GRASP Patterns:**
  - ✓ **Creator**
  - ✓ **Information Expert**
  - ✓ **Low Coupling**
  - ✓ **Controller**
  - ✓ **High Cohesion**
  - ✓ **Polymorphism**
  - ✓ **Indirection**
  - ✓ **Protected Variations**
  - ✓ **Pure Fabrication**

# GRASP

## Creator

- ❖ Who creates an Object? Or who should create a new instance of some class?
- ❖ “Container” object creates “contained” objects.
- ❖ Decide who can be creator based on the objects association and their interaction.

**Name:** Creator

**Problem:** Who creates A?

**Solution:** Assign class B the responsibility to create an instance of class A if one of these is true:

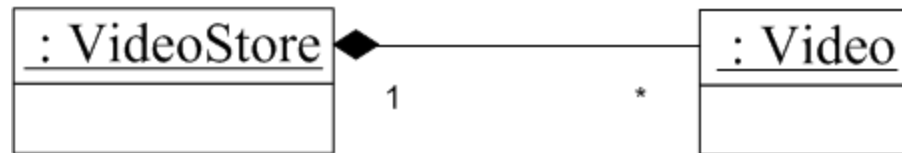
- ✓ B "contains" or compositely aggregates A.
- ✓ B records A.
- ✓ B closely uses A.
- ✓ B has the initializing data for A that will be passed to A when it is created.

# GRASP

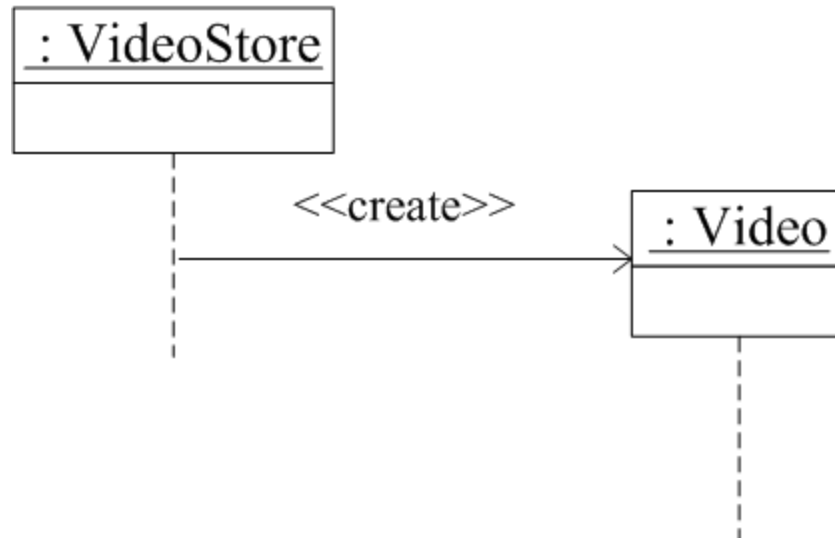
## Example:

Consider Video Store and Video in that store.

- ❖ Video Store has an Composite association with Video. i.e, Video Store is the container and the Video is the contained object or Video is part of VideoStore.



- ❖ So, we can instantiate video object in Video Store class



## Information Expert:

- ❖ Given an object A, **which responsibilities can be assigned to A?**
- ❖ Expert principle says – assign those responsibilities to A for which A has the information to fulfill that responsibility.
- ❖ They have all the information needed to perform operations, or in some cases they collaborate with others to fulfill their responsibilities.

**Name:** Information Expert

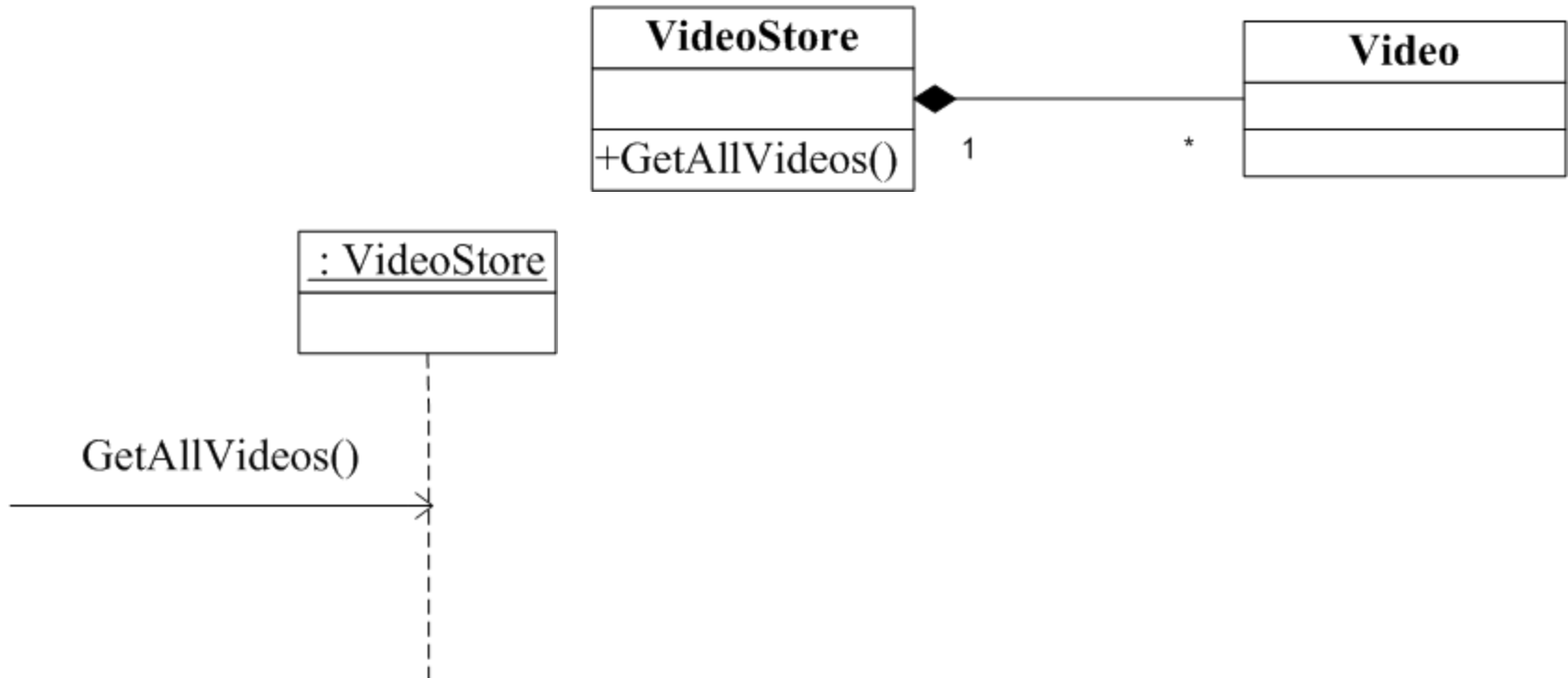
**Problem:** What is a general principle of assigning responsibilities to objects A?

**Solution:** Assign those responsibilities to A for which A has the information needed to fulfill it.

# GRASP

## Information expert Example:

- ❖ Assume we need to **get all the videos** of a Video Store.
- ❖ Since Video Store knows about all the videos, we can assign this responsibility of **giving all the videos** to Video Store class.
- ❖ So Video Store is the information expert.



# GRASP

## Low Coupling:

- ❖ How strongly the objects are connected to each other?
- ❖ Coupling – one object depending on other object.
- ❖ Low Coupling – How can we reduce the impact of change.
- ❖ Prefer low coupling – assign responsibilities so that coupling remain low.
- ❖ Minimizes the dependency hence making system maintainable and efficient

<b>Name:</b>	<b>Low Coupling</b>
<b>Problem:</b>	How to reduce the impact of change?
<b>Solution:</b>	Assign a responsibility so that coupling remains low.

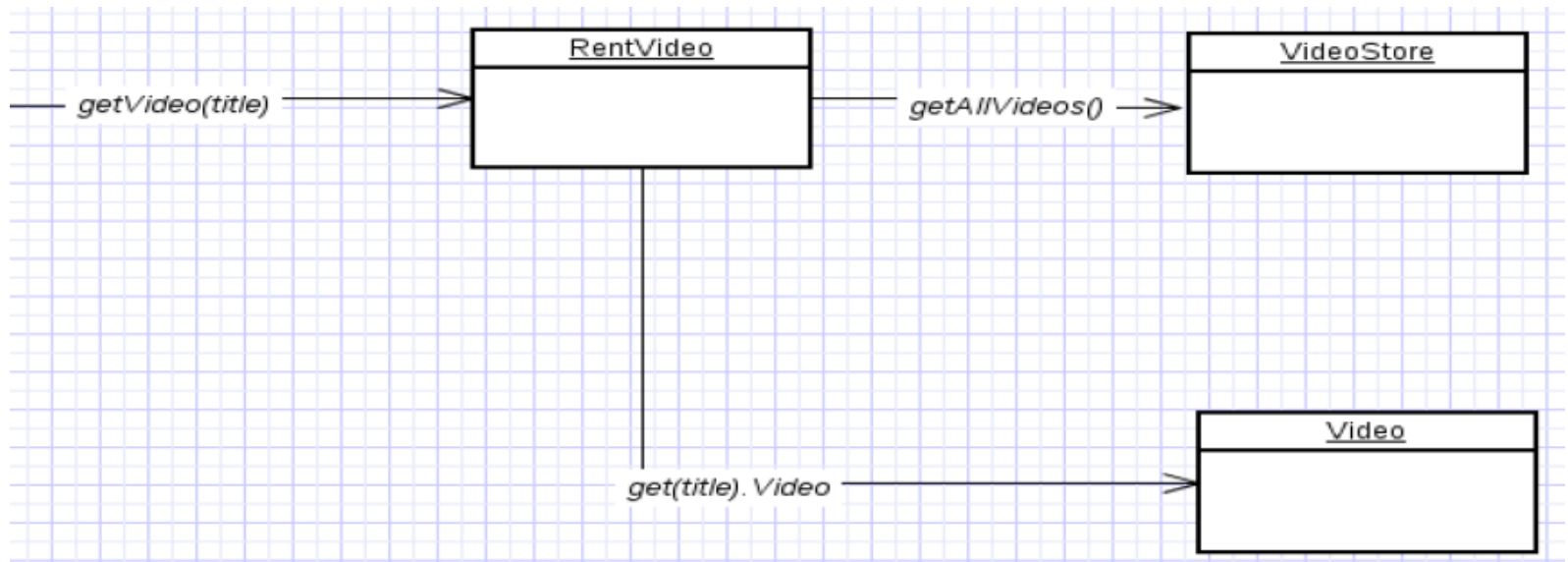
# GRASP

## Low coupling

Two elements are coupled, if

- ✓ One element **has aggregation/composition association** with another element.
- ✓ One element **implements/extends** other element.

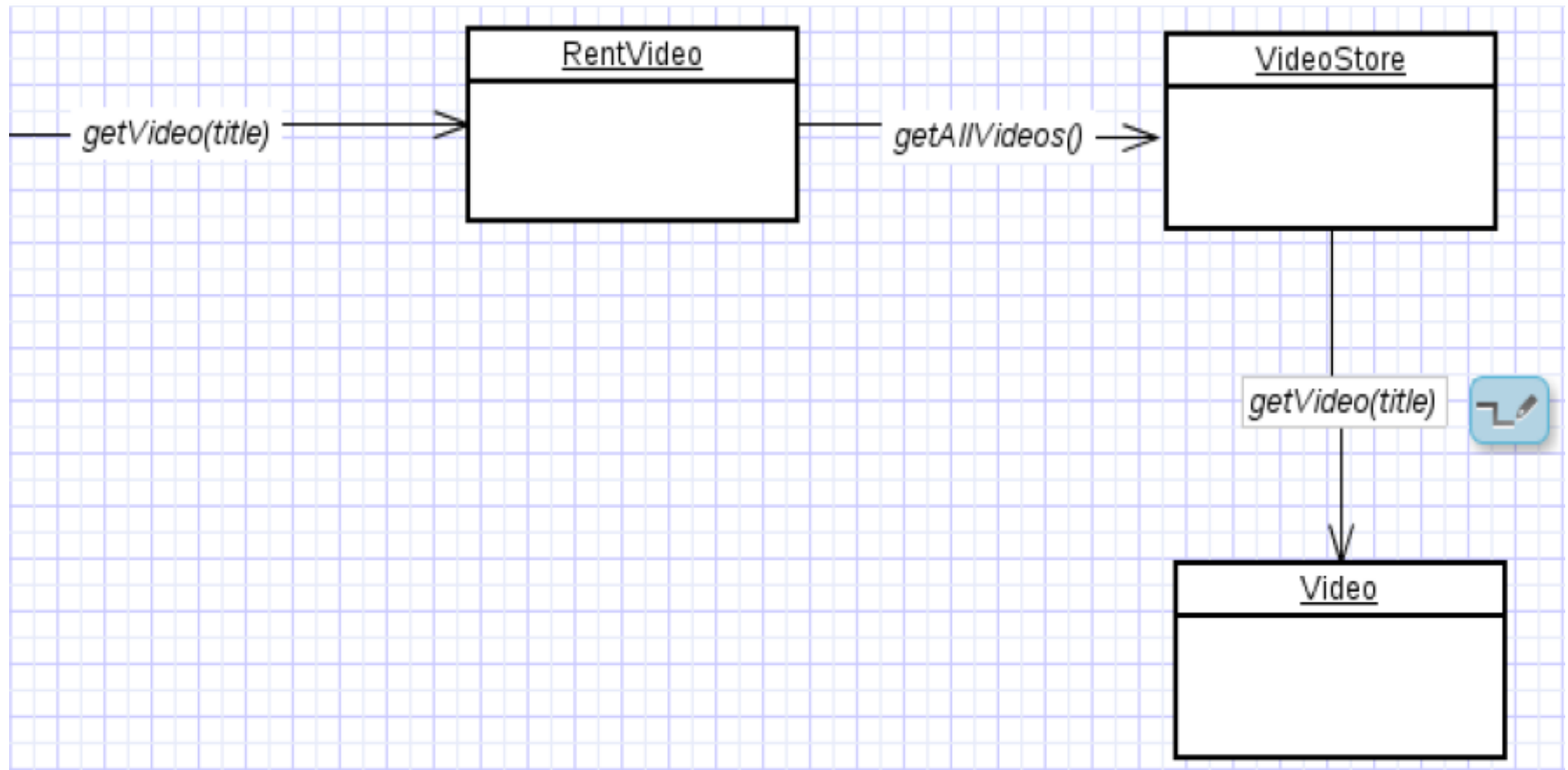
### Example of poor coupling



Here class Rent knows about both Video Store and Video objects. Rent is depending on both classes.

# GRASP

## Low coupling



- ❖ Video Store and Video class are coupled, and Rent is coupled with Video Store. Thus providing low coupling.



# GRASP

## Controller:

- ❖ A controller is the **first object beyond the UI layer** that is responsible for receiving or handling a system operation message.
- ❖ Deals with who should be responsible for handling event from external actors(UI).

**Name:**                      **Controller**

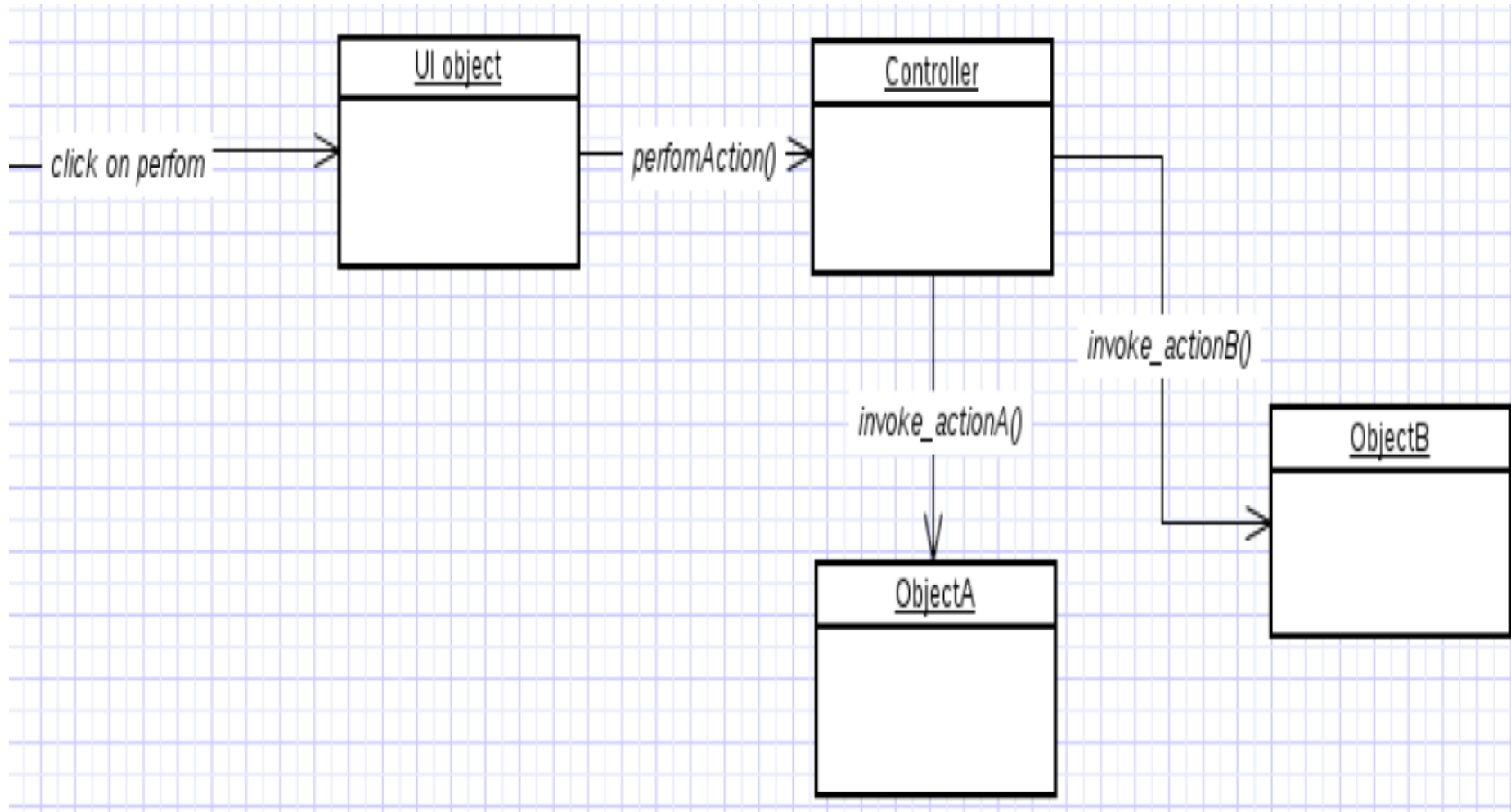
**Problem:**                What first object beyond the UI layer receives and coordinates ("controls") a system operation?

**Solution:**                Assign the responsibility to a class representing one of the following choices:

- ✓ Represents the overall "system.
- ✓ a root object
- ✓ a device that the software is running within
- ✓ a major sub system.

# GRASP

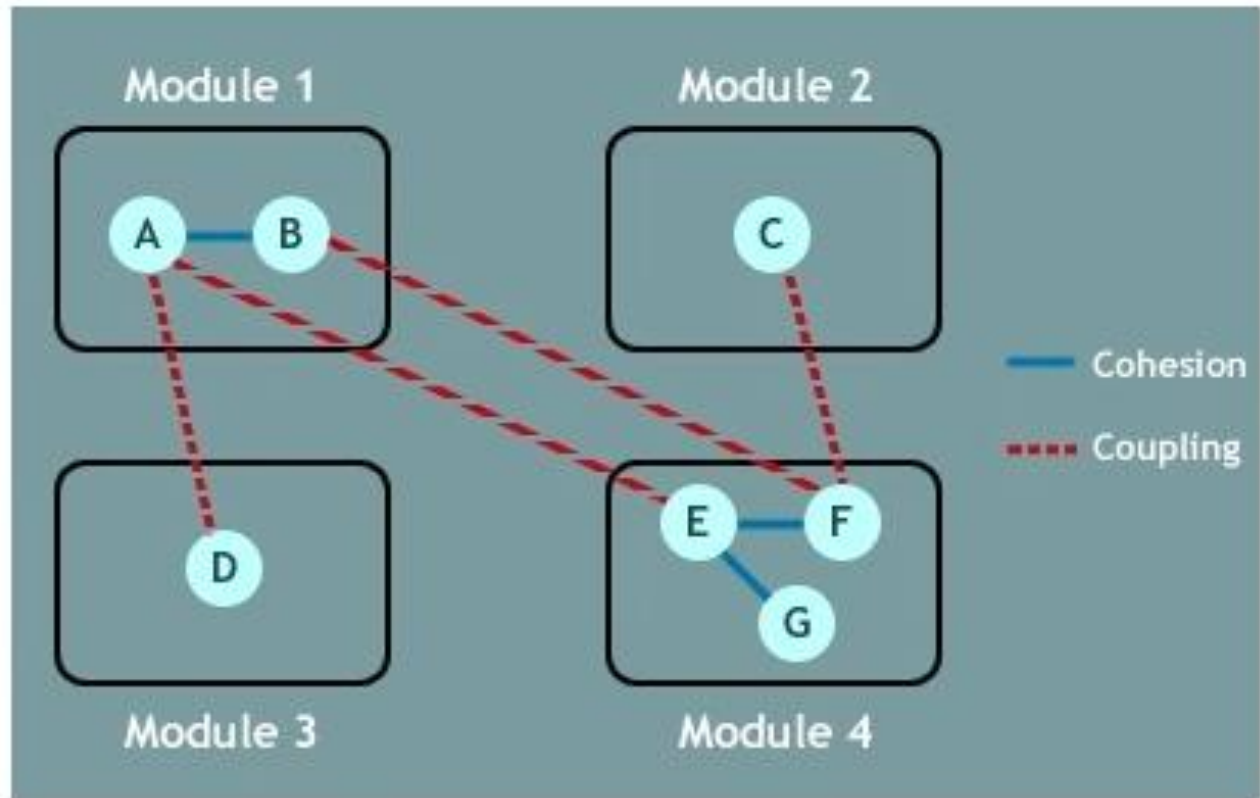
## Controller example:



# GRASP

## High cohesion

- ❖ **Cohesion** is the indication of the relationship **within** a module. Where as **Coupling** is the indication of the relationships **between** modules.
- ❖ We prefer high cohesion.



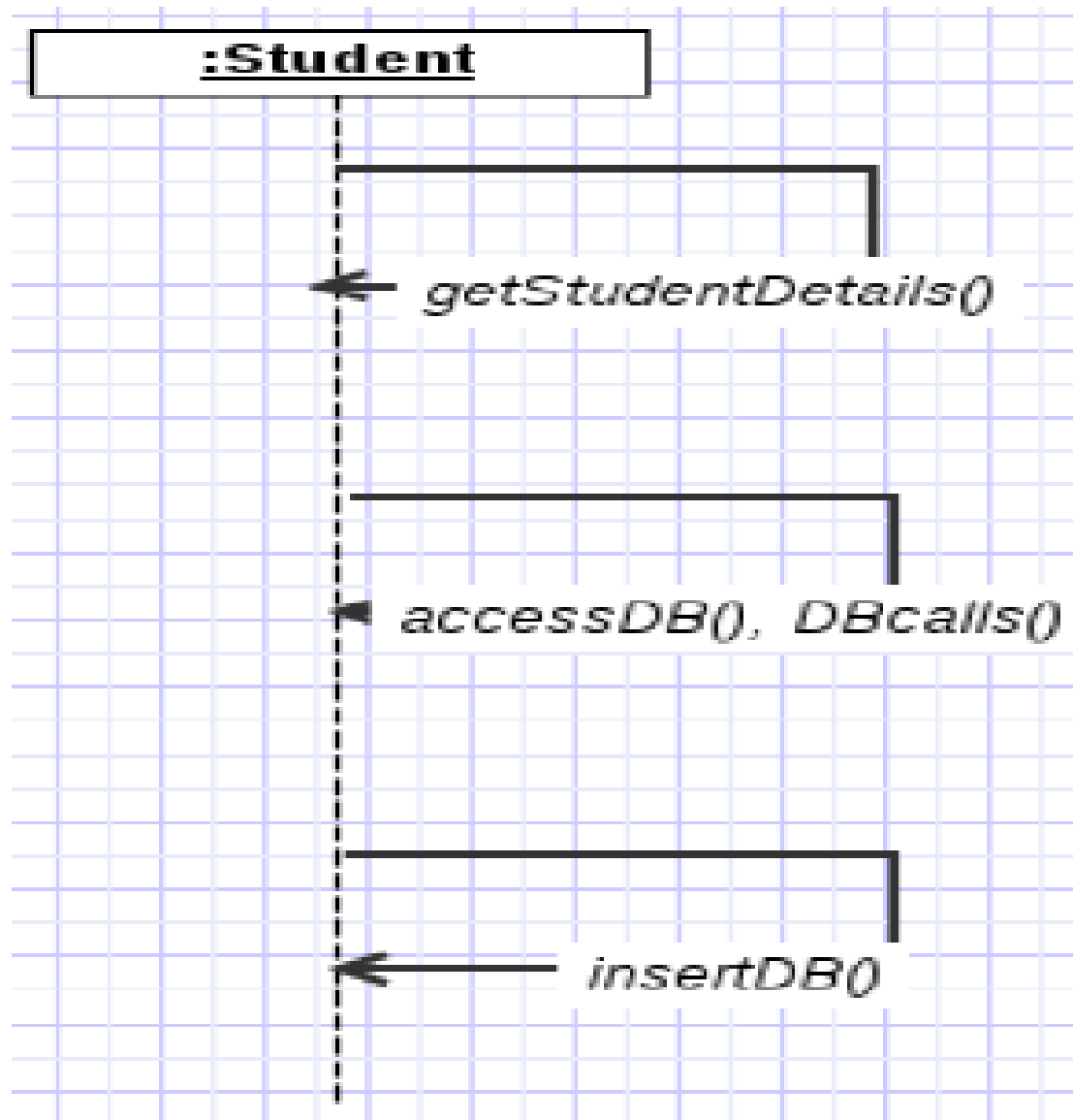
# GRASP

## High cohesion

<b>Name:</b>	<b>High Cohesion</b>
<b>Problem:</b>	How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?
<b>Solution:</b>	Assign a responsibility so that cohesion remains high.

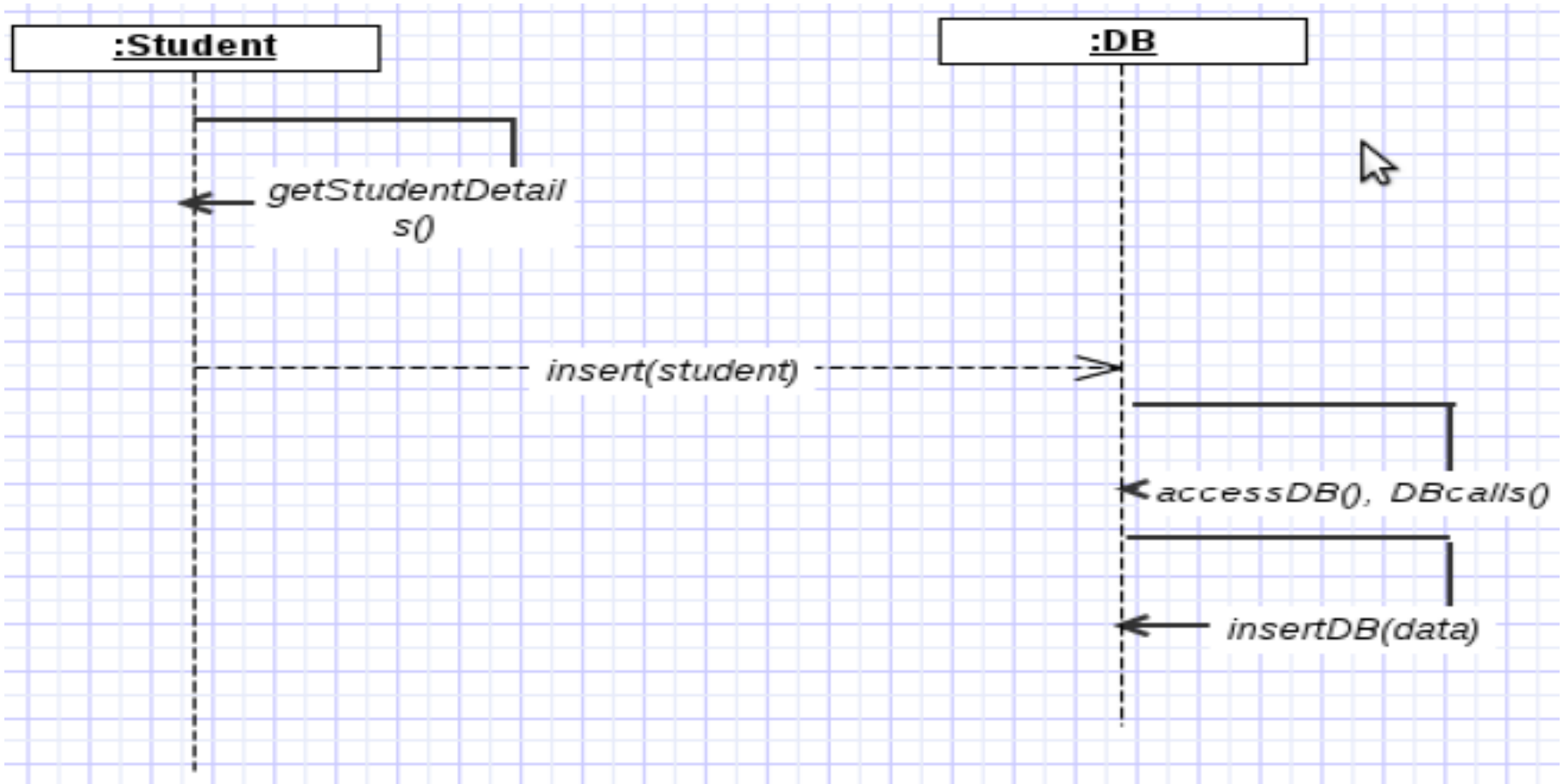
# GRASP

Low cohesion example:



# GRASP

## High cohesion example:



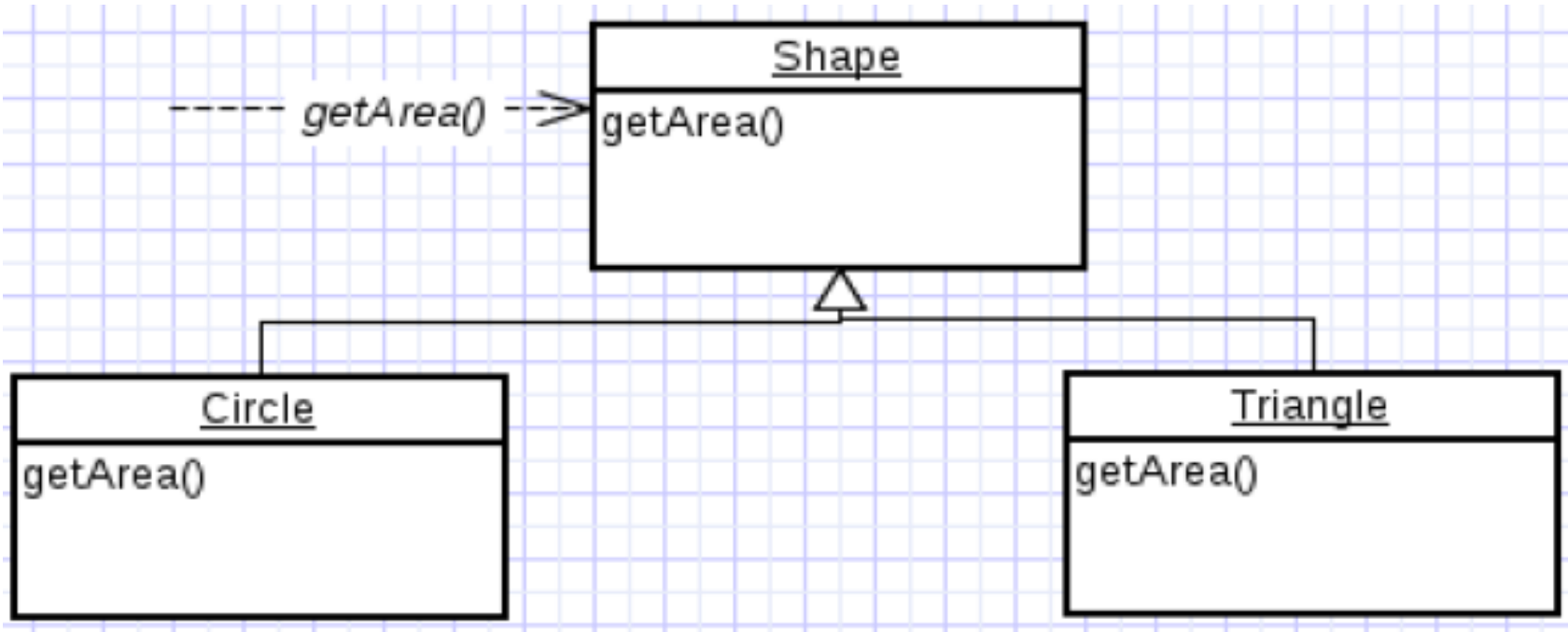
## Polymorphism

- ❖ **How to handle related but varying elements** based on element type?
- ❖ Polymorphism guides us in deciding which object is responsible for handling those varying elements.
- ❖ **Benefits:** handling new variations will become easy.

# GRASP

## Example for Polymorphism

- ❖ the `getArea()` varies by the type of shape, so we assign that responsibility to the subclasses.



- ❖ By sending message to the Shape object, a call will be made to the corresponding sub class object – Circle or Triangle.



# GRASP

**Indirection**  
**Protected Variations**  
**Pure Fabrication**

**See by yourself**

✓ **(Reference: Applying UML and Patterns, Craig Larman)**

# *Mapping Design to Code*

# Mapping Design to Code

Implementations of an object oriented language requires writing source code for:

- ✓ Class and interface definitions
- ✓ Method definitions

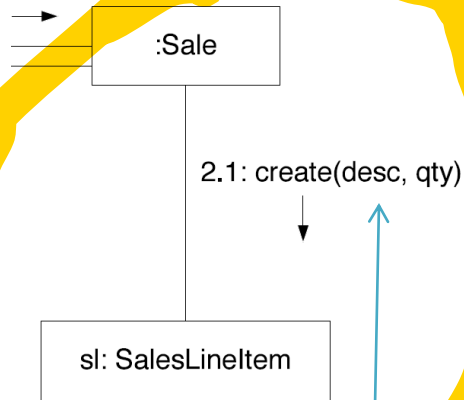
## **Creating Class Definitions from DCD (Design Class Diagram):**

- ❖ At the very least, DCDs depict the class or interface name, super classes, operation signatures, and attributes of a class. This is sufficient to create a basic class definition in an OO language.
- ❖ If the DCD was drawn in a UML tool, it can generate the basic class definition from the diagrams.

## **Defining a Class with Method Signatures and Attributes**

- ❖ From the DCD, a mapping to the attribute definitions (Java fields) and method signatures for the Java definition of SalesLineItem is straightforward, as shown in Figure ( next page).

# Mapping Design to Code

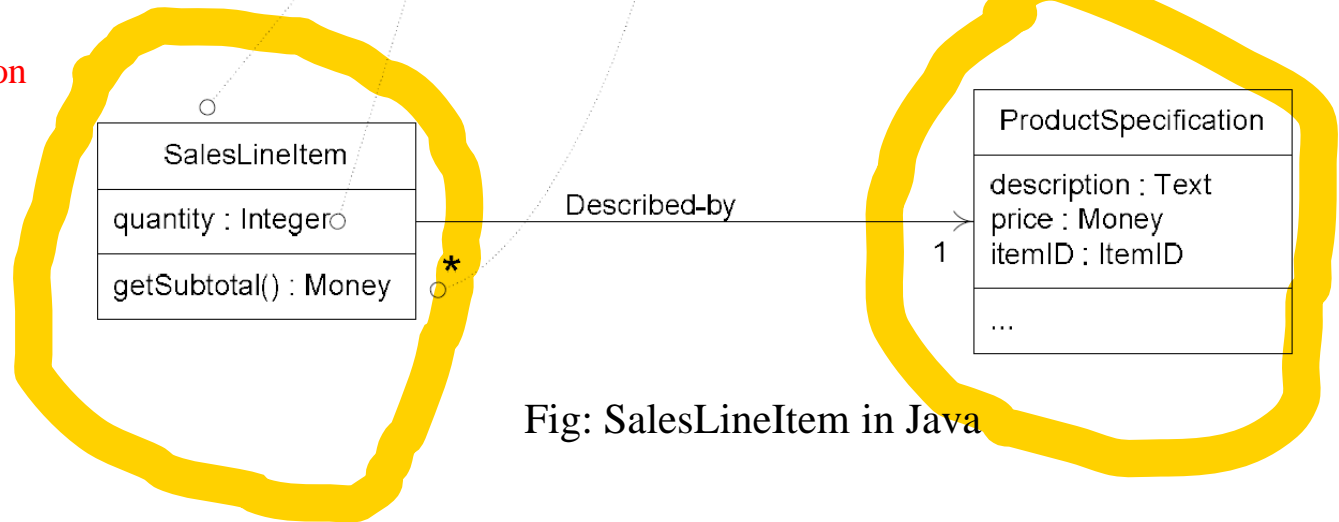


**Fig: Constructor from interaction diagram**

```
public class SalesLineItem
{
    private int quantity;

    public SalesLineItem(ProductSpecification spec, int qty) { ... }

    public Money getSubtotal() { ... }
}
```



**Fig: SalesLineItem in Java**

## Note:

The addition in the source code of the Java constructor `SalesLineItem(...)`. It is derived from the `create(desc, qty)` message sent to a `SalesLineItem` in the `enterItem` interaction diagram. This indicates, in Java, that a constructor supporting these parameters is required. The `create` method is often excluded from the class diagram because of its commonality and multiple interpretations, depending on the target language

# Mapping Design to Code

## Creating Methods from Interaction Diagrams:

- ❖ The sequence of the messages in an interaction diagram translates to a series of statements in the method definitions. The enterItem interaction diagram in Figure below illustrates the Java definition of the enterItem method. For this example, we will explore the implementation of the Register and its enterItem method.

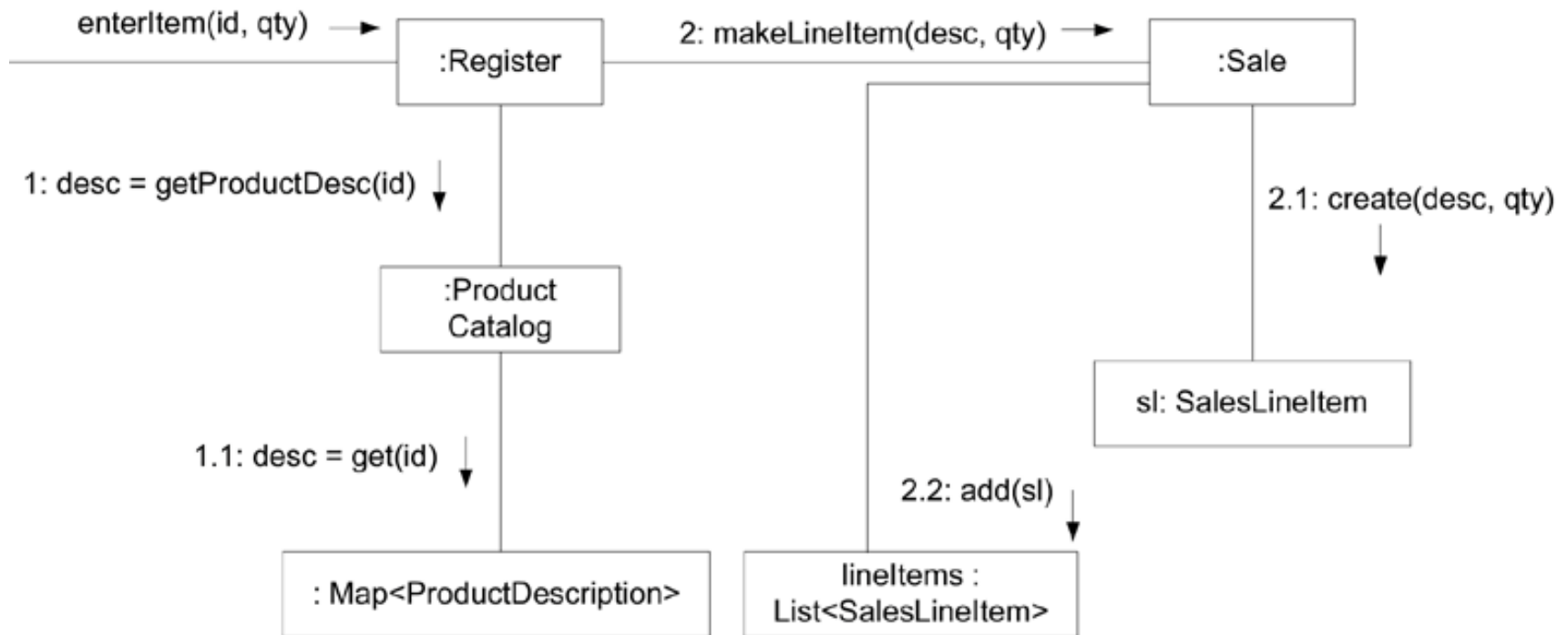


Figure: EnterItem Interaction Diagram

# Mapping Design to Code

- ❖ The enterItem message is sent to a Register instance therefore, the enterItem method is defined in class Register.

*public void enterItem(**ItemID** itemID, int qty)// Parameter Visibility(**ItemId**)*

## Message 1:

A getProductDescription message is sent to the ProductCatalog to retrieve a ProductDescription.

*ProductDescription desc = catalog.getProductDescription(itemID);*

## Message 2:

The makeLineItem message is sent to the Sale.

*currentSale.makeLineItem(desc,qty);*

- ❖ In summary, each sequenced message within a method, as shown on the interaction diagram, is mapped to a statement in the Java method.
- ❖ The complete enterItem method and its relationship to the interaction diagram is shown in Figure (Next page)

# Mapping Design to Code

- ❖ A java Definition of the Register class is shown in figure below.

## *The Register.enterItem Method*

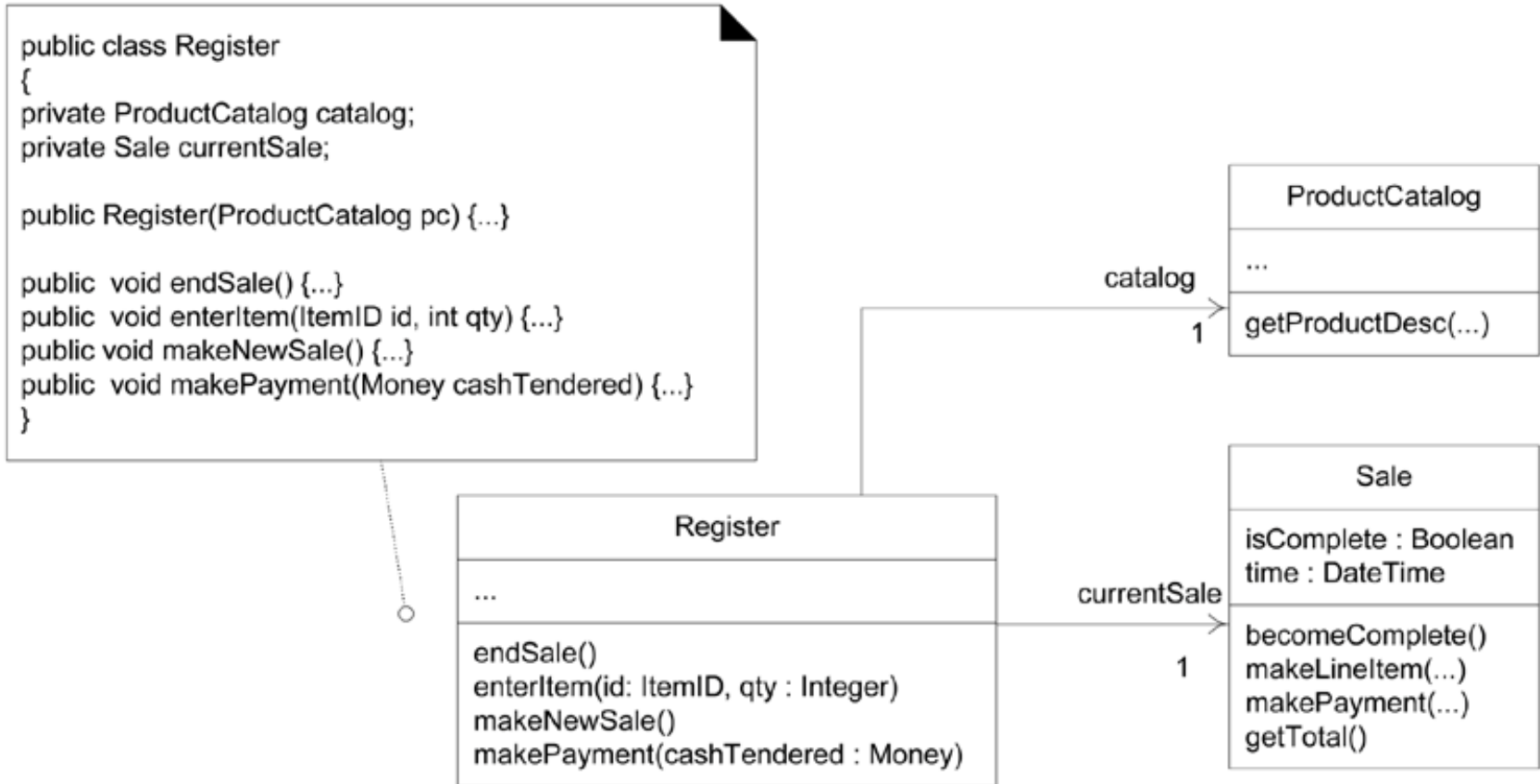


Fig: The Register Class

## Exception and Error Handling

**See by yourself**