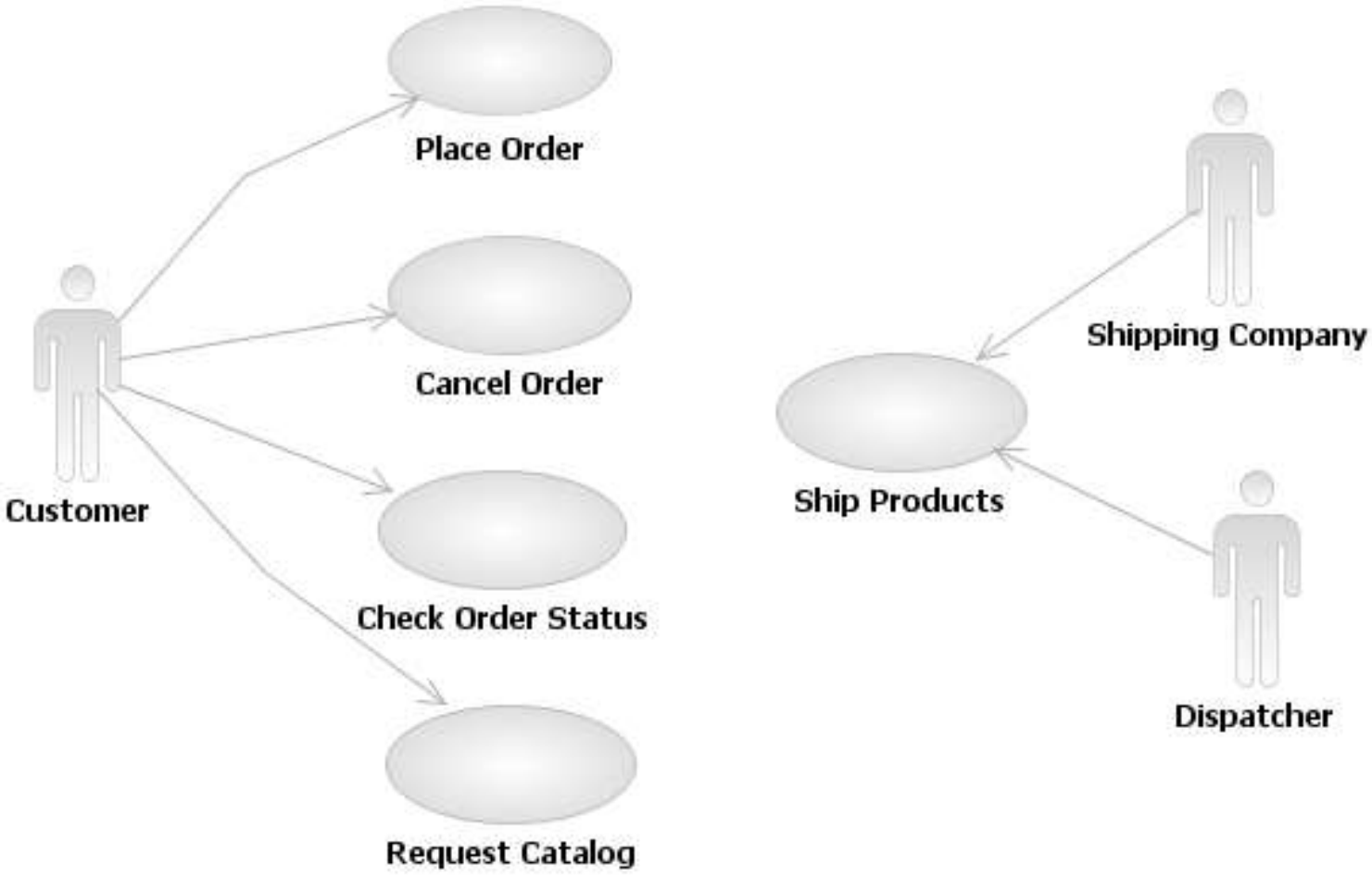
Use Case Modeling

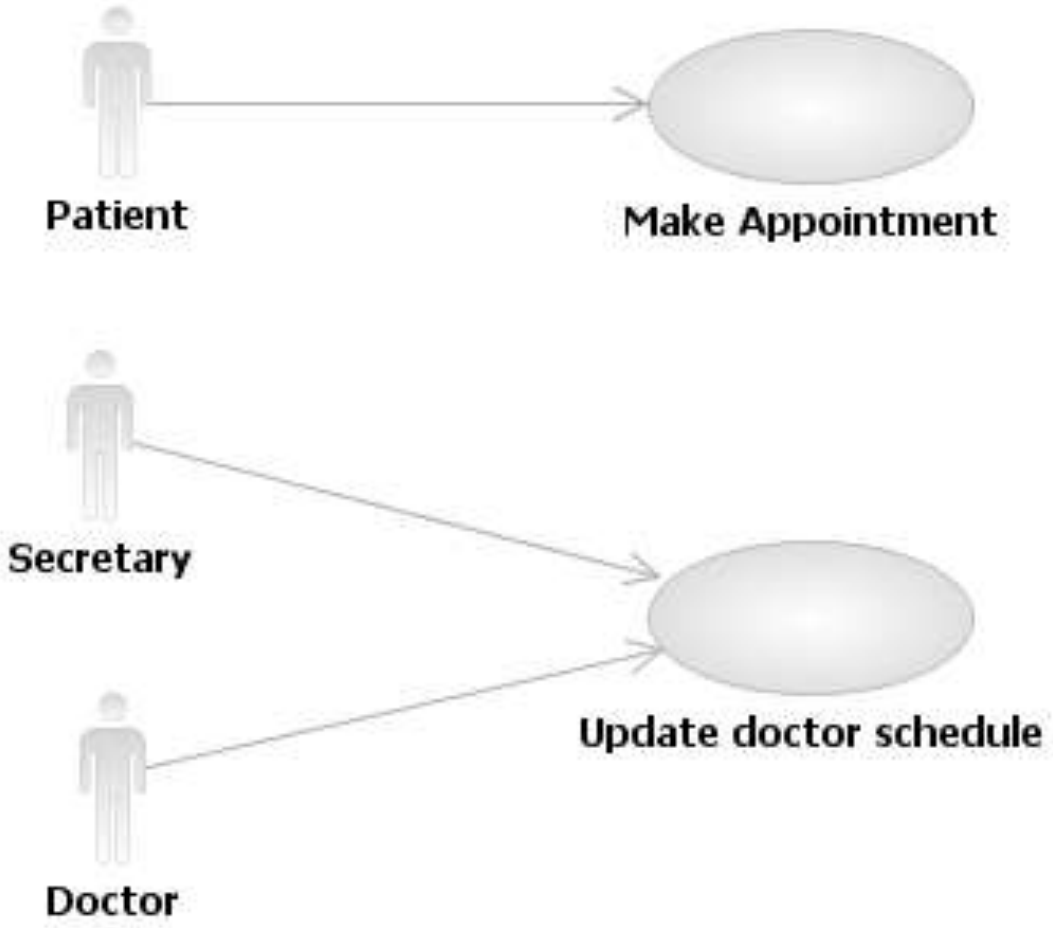
Learning Objectives

- What is a use case diagram?
- What is its purpose?
- How do we create use case diagrams?

Use Case Diagram Example



Use Case Diagram Example



Use Case Modeling - Overview

- A form of requirements engineering
- Lets us identify the system boundary, who or what uses the system, and what functions the system should offer
- Helps us identify and define all processes that our system must support

Use Case Modeling

- Find the system/automation boundary
- Find actors
- Find and specify use cases
- Describe scenarios
- Iterate until elements are stable

Elements in a Use Case Model

- System boundary/subject – the edge of the system – what part of the system is going to be automated?
- Actors – who or what uses the system?
- Use Cases – what do the actors do with the system?
- Relationships - between actors and use cases

Actors

- A role that someone or something (an external entity) in the environment plays in relation to our system
- Everything that interacts with the system
- Note that someone/something can play different roles simultaneously or over time
- Example – Customer and Administrator (John Doe might play both these roles)

Actors

- Could be a person, another software application, or hardware device
  - E.g.: Customer, Supply chain management system, printer
- Same actor can be involved in several use cases
- Actors are external to the system
- Do not confuse the role that someone plays with the person itself

Actors

- Who / which
  - Uses the main functionality of the system (primary actors)?
  - Needs system support for daily tasks?
  - Maintains, administers, operates system (secondary)?
  - Has interest in the results of the system?
  - Other systems interact with ours?

Actors

- Do things happen in your system at specific points in time that may not be triggered by any actor?
- Time as an actor
- Example?
  - Automatic system backup
  - Automatic triggering of emails to customers on some occasions

Use Cases

- A specification of *sequences of actions* that a system performs
- Yields an observable result of value to a particular actor
- Provides a functional description of the system and its processes
- A “case of use” of the system by a specific actor

Use Cases

- Not a class or object, but a process that satisfies some user need
- Always initiated by an actor
- Provides value to an actor
- Written from the point of view of the actors
- Describe basic functions of the system
  - What the user can do
  - How the system responds

Finding Use Cases

- Start with the list of actors that interact with the system.
  - What function does the actor require from the system?
  - What does the actor need to do?
  - Does the actor need to read, create, destroy, modify, or store some kind of information in the system?
  - Does the system interact with any external system?
  - Does the system generate any reports?

Describing Use Cases

- Use cases connected to actors through associations
- Named according to what they performs
  - E.g.: Purchase insurance policy, Update account
- Usually starts with a verb

Scenario

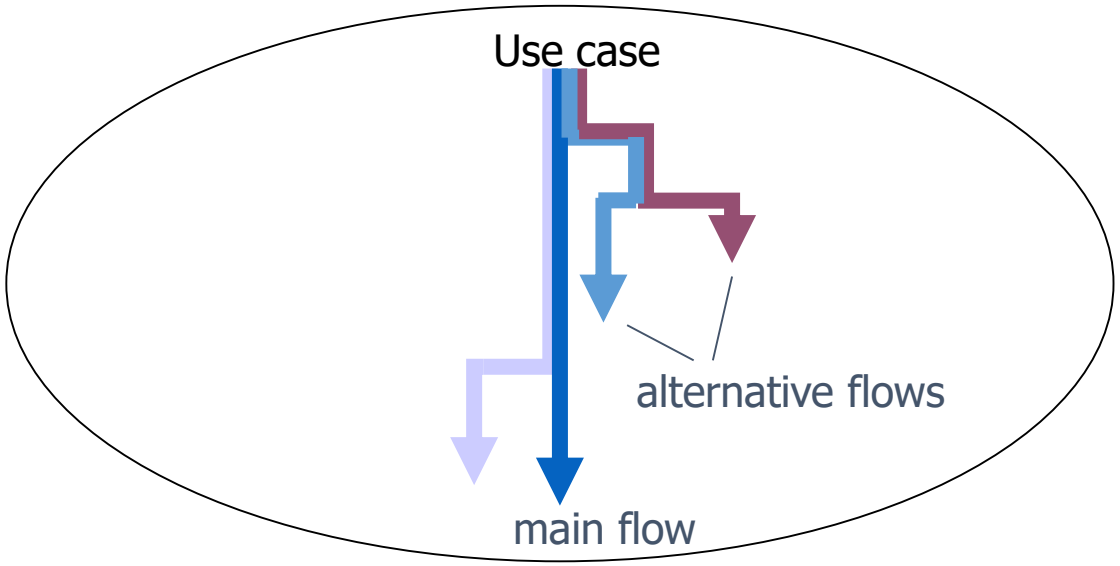
- An instance of a use case
  - E.g.: “Customer purchases insurance” is instantiated as:
  - *“John Doe contacts the system through the web and purchases insurance for his new car”*

Use Case Specification

Use case: Place Order	
ID: 1	
Brief Description: This use case describes the process to follow when a customer places a new order.	
Primary actors: Customer	
Secondary actors: None	
Pre-conditions: The customer must have an account with us.	
Main flow:	
Actor action	**System response**
1. Customer searches for item.	2. System provides a list of items that matches the customers search keywords.
3. Customer selects an item.	4. System provides more information about the selected item.
5. Customer chooses to add item to cart.	6. System adds item to cart.
7. Customer chooses to proceed to checkout.	8. System asks for payment and shipping details.
9. Customer provides details and confirms order.	10. System processes order.
Post-conditions: Order must be processed and confirmation provided to customer.	
Alternate flows: Item not available Customer account ineligible for placing new orders	

Branching: Alternative flows

- Alternative flows capture errors, branches, and interrupts
- Alternative flows usually do not return to the main flow
- Potentially very many alternative flows
  - Pick the most important ones
  - If there are groups of similar alternative flows - document one member of the group as an exemplar



Referencing Alternative Flows

- List alternative flows at the end of the use case
- Examine each step in the main flow and look for:
  - Alternatives
  - Exceptions
  - Interrupts

Use case: CreateNewCustomerAccount
ID: 5
Brief description: The system creates a new account for the Customer.
Primary actors: Customer
Secondary actors: None.
Preconditions: None.
Main flow: 1. The use case begins when the Customer selects "create new customer account". 2. While the Customer details are invalid 2.1 The system asks the Customer to enter his or her details comprising e-mail address, password, and password again for confirmation. 2.2 The system validates the Customer details. 3. The system creates a new account for the Customer.
Postconditions: 1. A new account has been created for the Customer.
Alternative flows: InvalidEmailAddress InvalidPassword Cancel

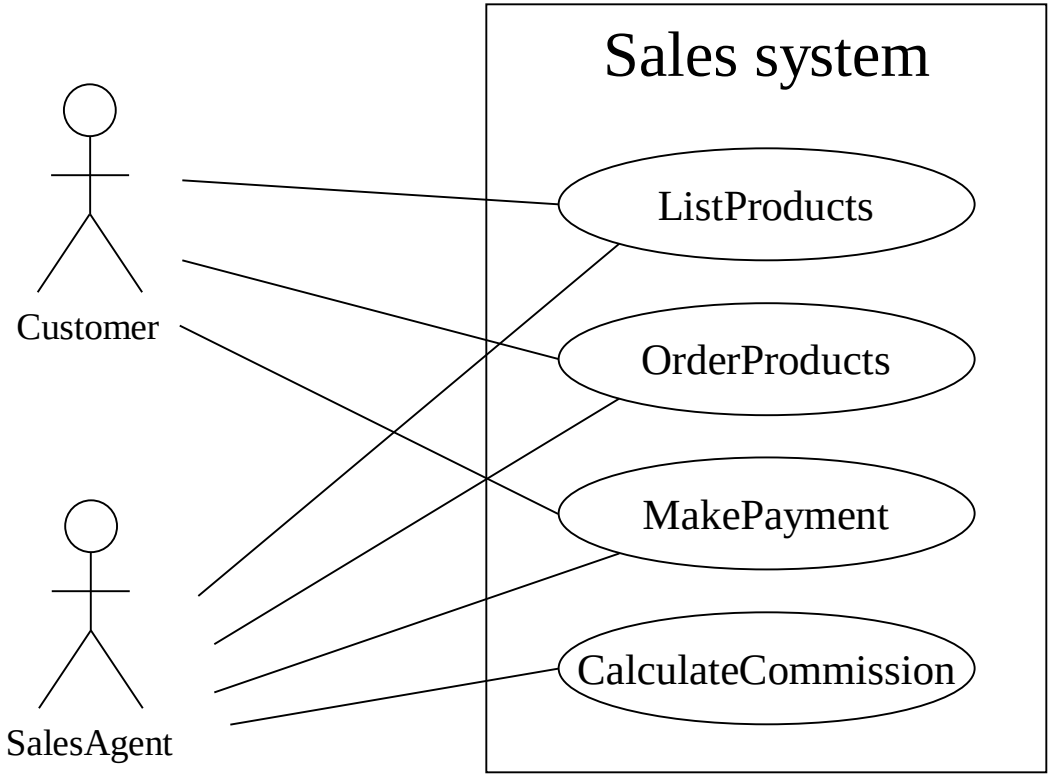
An Alternative Flow Example

- May be triggered
  - *instead* of the main flow - started by an actor
  - *after a particular step* in the main flow
  - *at any time* during the main flow - at any time

Alternative flow: CreateNewCustomerAccount:InvalidEmailAddress
ID: 5.1
Brief description: The system informs the Customer that they have entered an invalid email address.
Primary actors: Customer
Secondary actors: None.
Preconditions: 1. The Customer has entered an invalid email address
Alternative flow: 1. The alternative flow begins after step 2.2. of the main flow. 2. The system informs the Customer that he or she entered an invalid email address.
Postconditions: None.

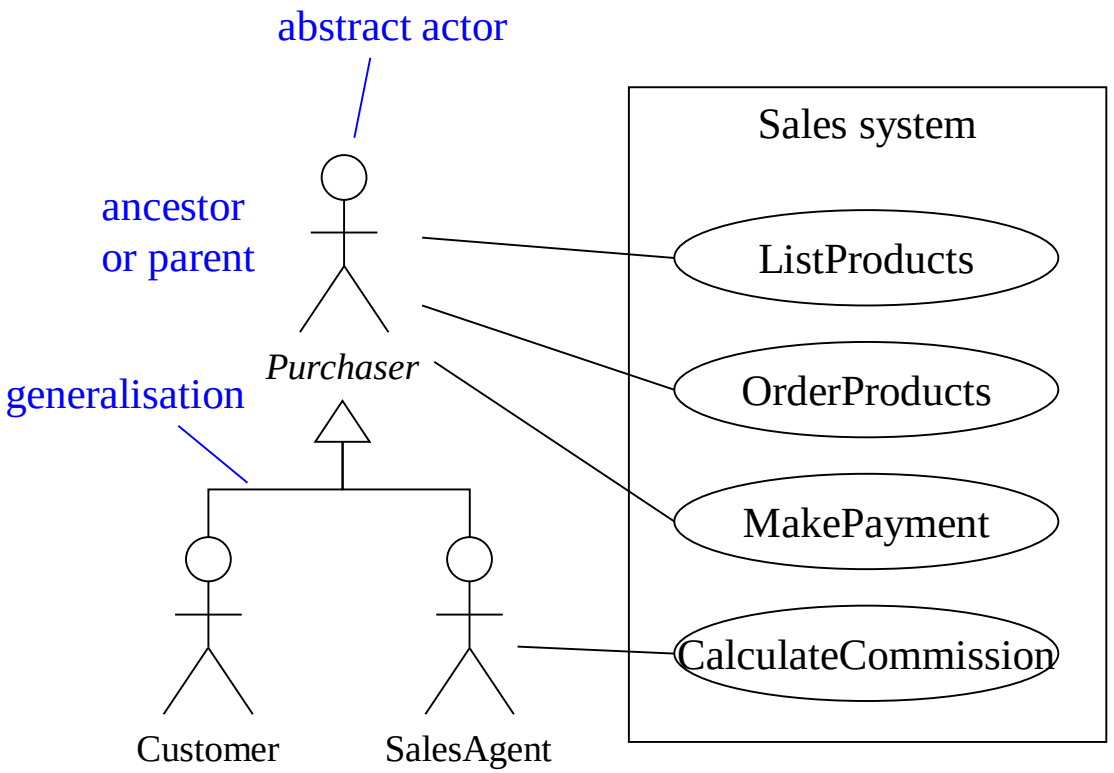
Actor Generalization

- The Customer and the Sales Agent actors are *very* similar
- Similar (almost) interactions
- Simplifying the diagram



Actor Generalization

- Inheritance
- Substitutability
- Parent actor – abstract?

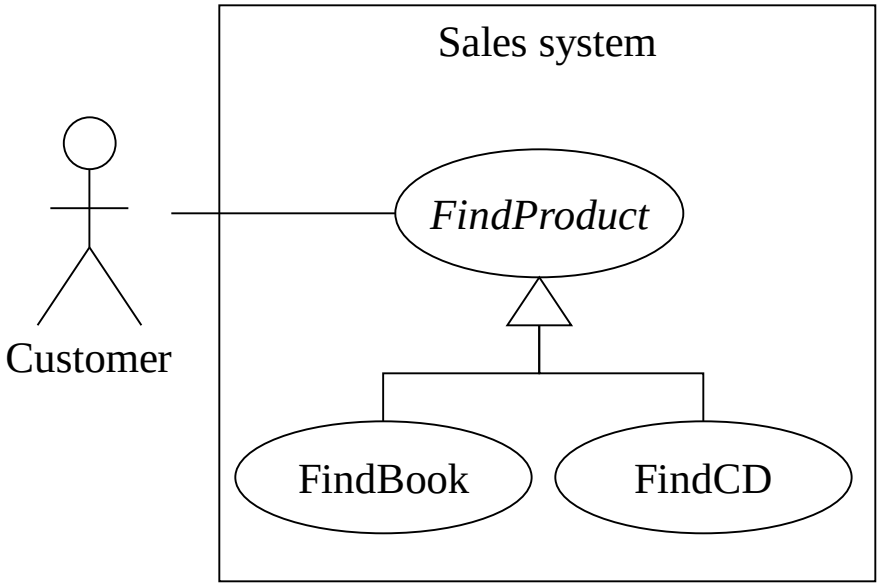


descendents or children

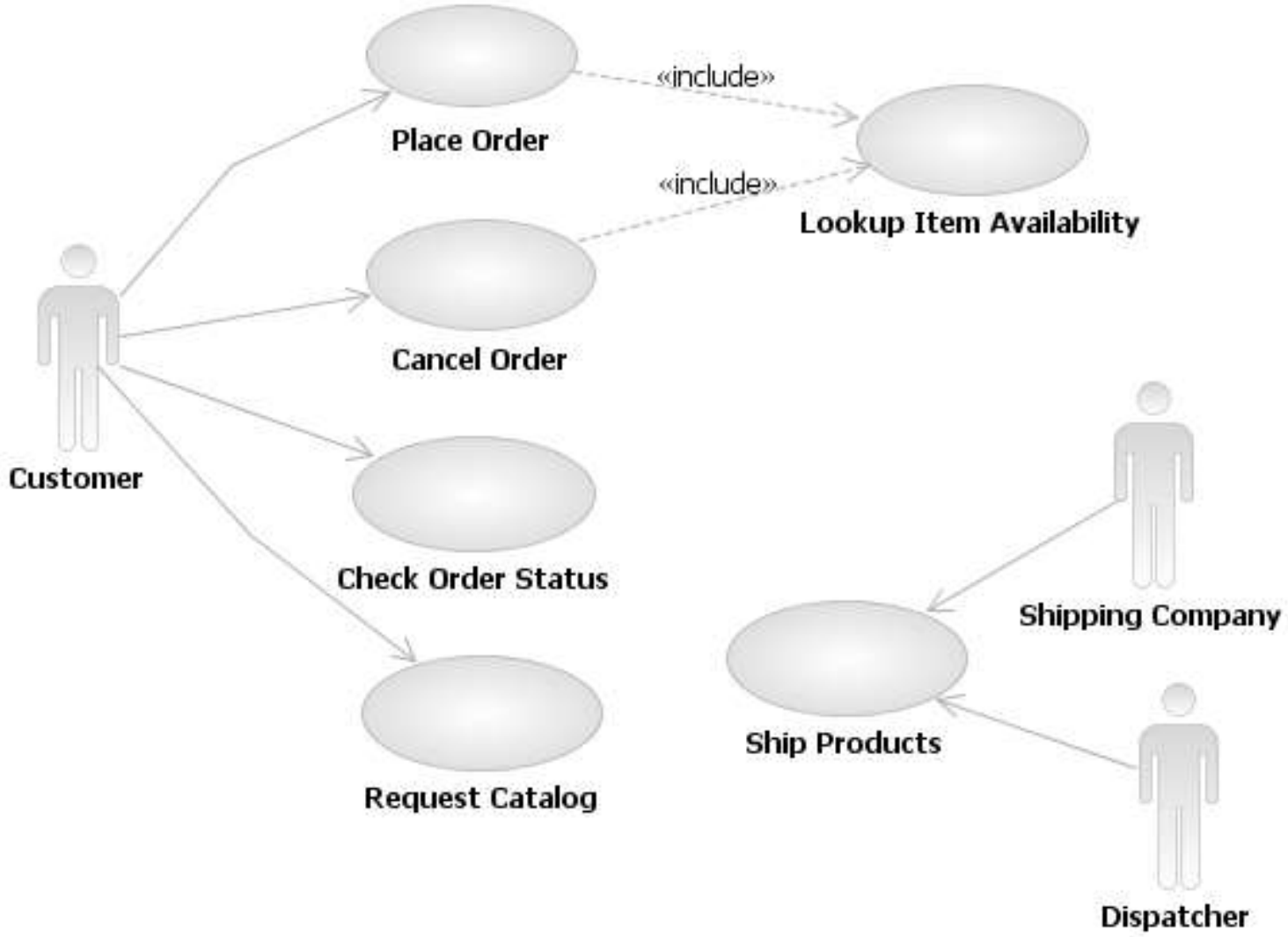
Use Case Generalization

- The ancestor use case must be a more general case of one or more descendant use cases
- Child use cases are more specific forms of their parent
- They can inherit, add and override features of their parent

Use case generalization semantics			
Use case element	Inherit	Add	Override
Relationship	Yes	Yes	No
Extension point	Yes	Yes	No
Precondition	Yes	Yes	Yes
Postcondition	Yes	Yes	Yes
Step in main flow	Yes	Yes	Yes
Alternative flow	Yes	Yes	Yes



Example with «includes»



Extend versus include

I often use this to remember the two:

My use case: I am going to the city.

includes -> drive the car

extends -> fill the petrol

"Fill the petrol" may not be required at all times, but may optionally be required based on the amount of petrol left in the car. "Drive the car" is a prerequisite hence I am including.

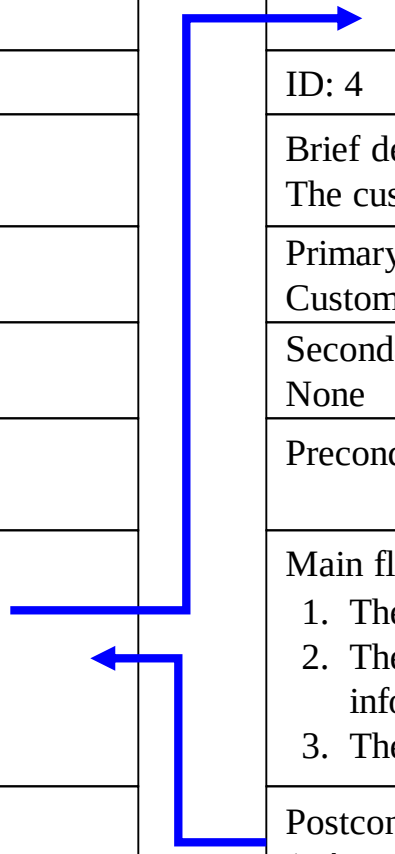
«include»

- When use cases share common behavior…
- The base use case executes until the point of inclusion: include (InclusionUseCase)
  - Control passes to the inclusion use case which executes
  - When the inclusion use case is finished, control passes back to the base use case which finishes execution
- Base use cases are *not complete* without the included use cases
- Inclusion use cases may be complete use cases, or they may just specify a fragment of behaviour for inclusion elsewhere

Example with «include»

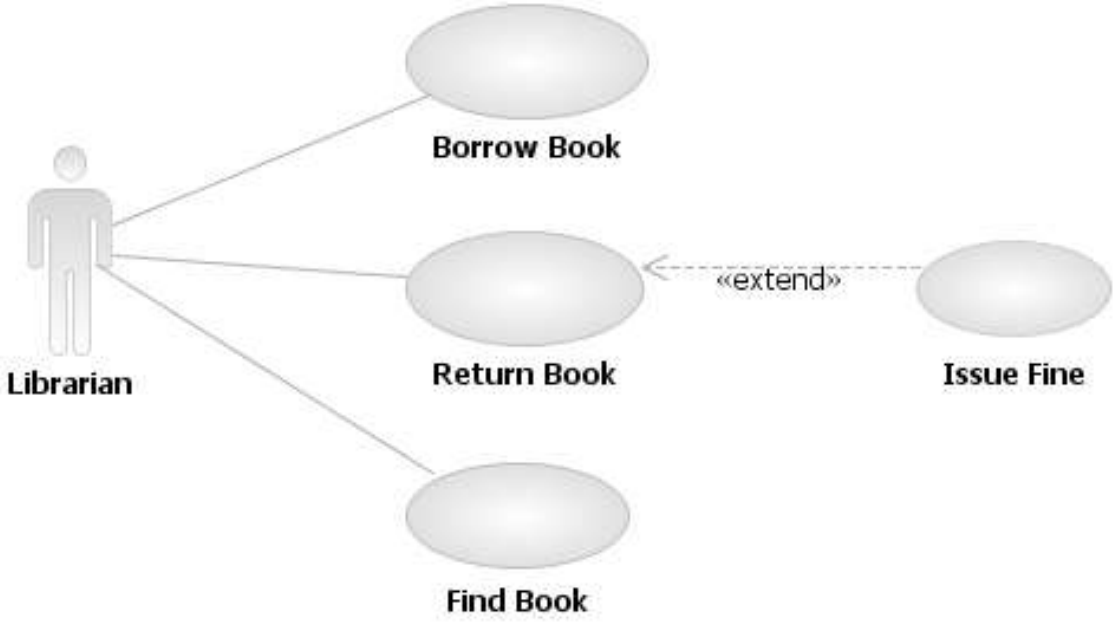
Use case: CreateNewOrder
ID: 1
Brief description: Customer/Order clerk places new order.
Primary actors: Customer and Order Clerk
Seconday actors: None
Preconditions:
Main flow: 1. include(ValidateCustomerAccount). 2. Include(LookupItemAvailability). 3. The system displays item and order details. …
Postconditions: 1. A new order is created.
Alternative flows: None.

Use case: ValidateCustomerAccount
ID: 4
Brief description: The customer/order clerk logs in.
Primary actors: Customer and Order Clerk
Seconday actors: None
Preconditions:
Main flow: 1. The system asks for login information. 2. The customer/order clerk enters authentication information. 3. The system verifies account
Postconditions: 1. Account has been verified.
Alternative flows: None.

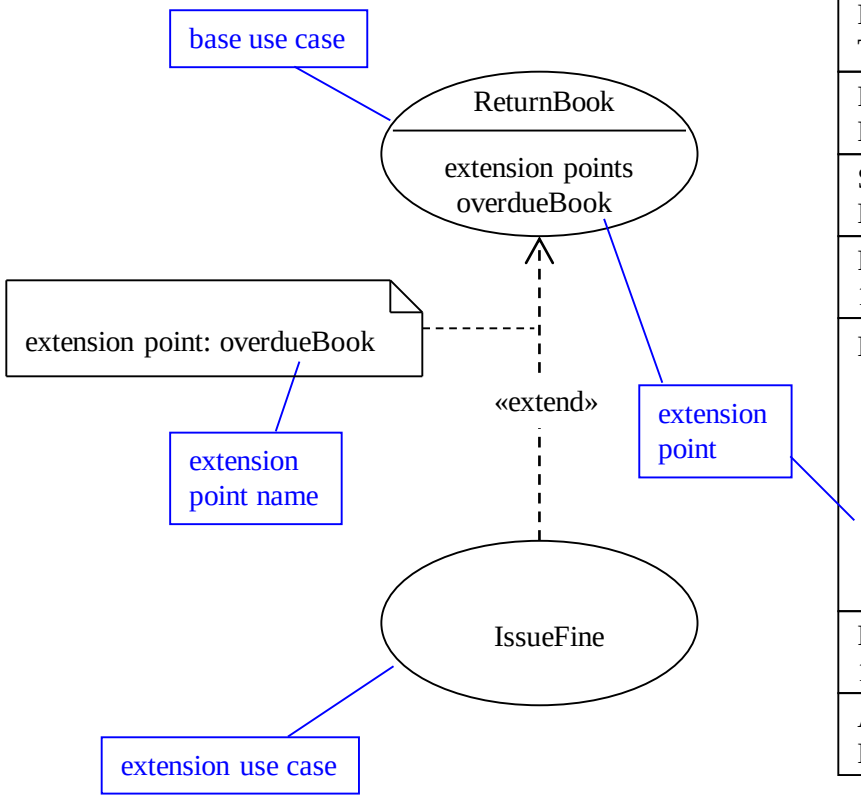


«extend»

- Adding new behaviour into the base use case by inserting behavior from one or more extension use cases
- The base use case specifies one or more extension points in its flow of events
- The base use case *does not know* about the extensions



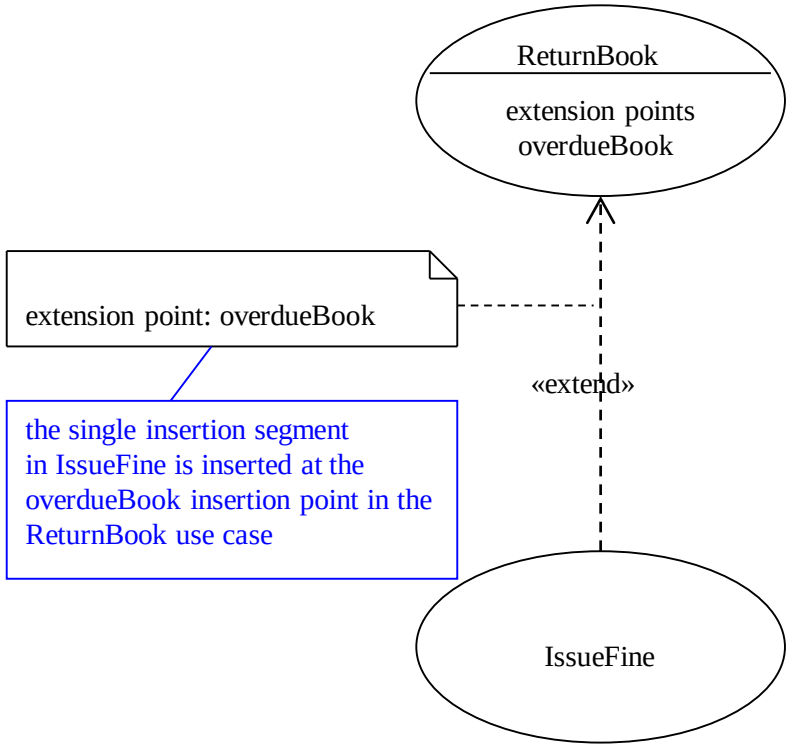
Base Use Case



Use case: ReturnBook
ID: 9
Brief description: The Librarian returns a borrowed book.
Primary actors: Librarian
Secondary actors: None.
Preconditions: 1. The Librarian is logged on to the system.
Main flow: 1. The Librarian enters the borrower's ID number. 2. The system displays the borrower's details including the list of borrowed books. 3. The Librarian finds the book to be returned in the list of books. extension point: overdueBook 4. The Librarian returns the book. …
Postconditions: 1. The book has been returned.
Alternative flows: None.

- Extension points are *not* numbered, as they are *not* part of the flow

Extension Use Case



Extension Use case: IssueFine
ID: 10
Brief description: Segment 1: The Librarian records and prints out a fine.
Primary actors: Librarian
Secondary actors: None.
Segment 1 preconditions: 1. The returned book is overdue.
Segment 1 flow: 1. The Librarian enters details of the fine into the system. 2. The system prints out the fine.
Segment 1 postconditions: 1. The fine has been recorded in the system. 2. The system has printed out the fine.

- Extension use cases have one or more *insertion segments* which are behaviour fragments that will be inserted at the specified extension points in the base use case

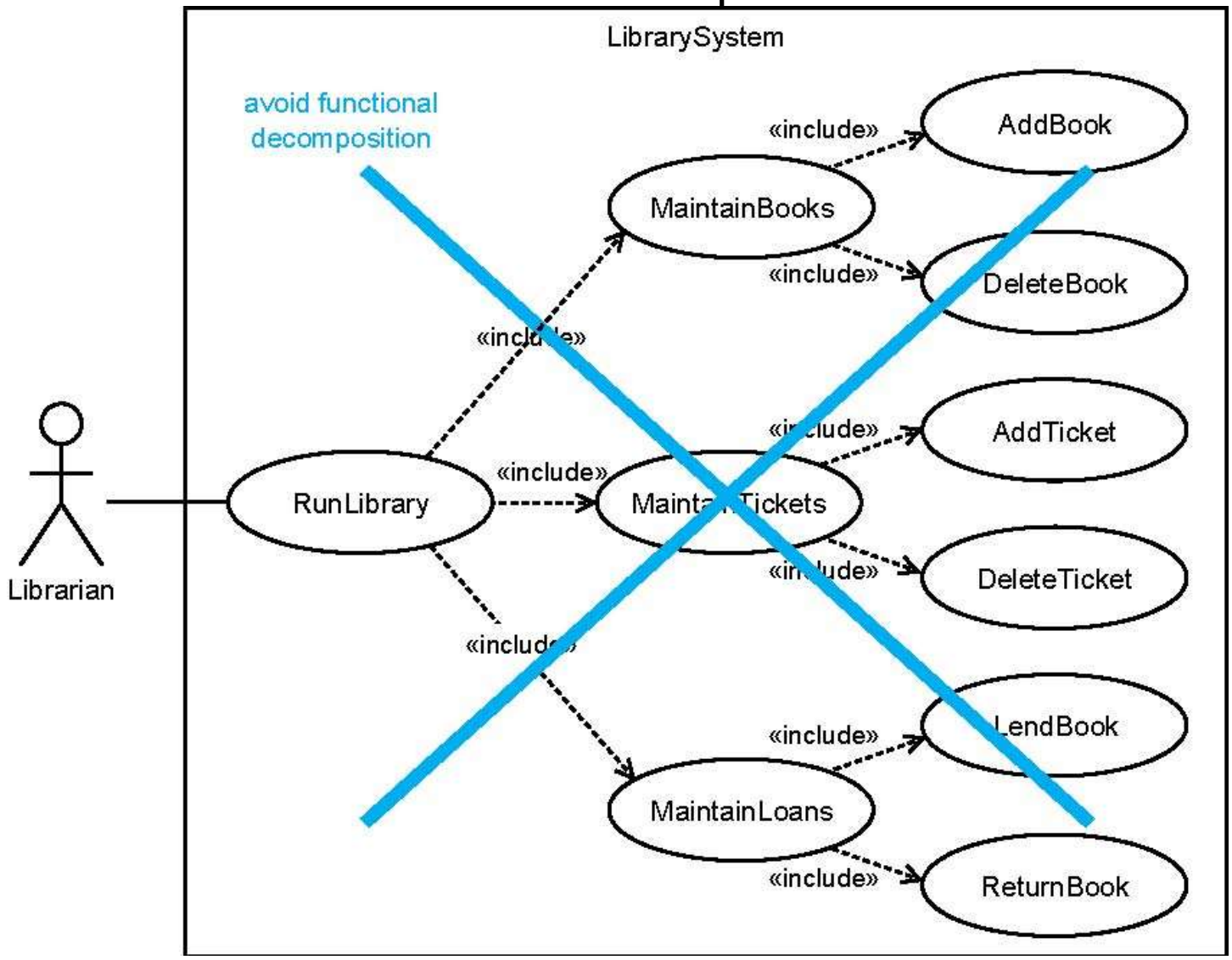
When to use advanced features?

- When it simplifies your use case model
- Use sparingly
- Remember the purpose – communication with stakeholders
  - Stakeholders may not understand –
    - Inheritance
    - Extension

Writing Use Cases

- Short and simple
- Focus on *what,* not *how*
  - *Customer presses the OK button vs.*
  - *Customer accepts the order*
- Avoid functional decomposition
  - Often used in traditional/structured approach

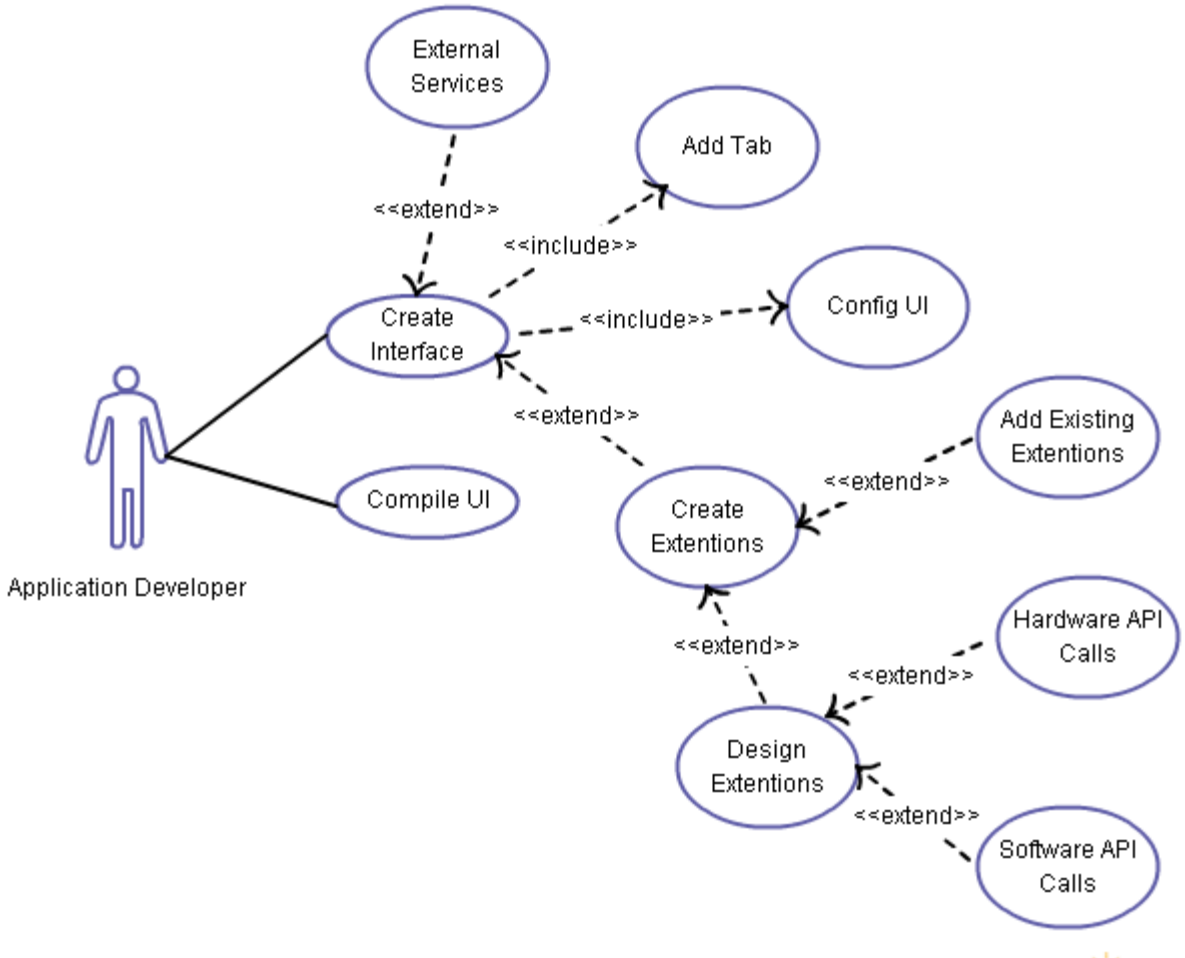
Avoid Functional Decomposition



When to Use ‘Use Case Analysis’

- A good choice when the system:
  - is dominated by functional requirements
  - has many types of user to which it delivers different functionality
  - has many interfaces
- A poor choice when the system:
  - is dominated by non-functional requirements
  - has few users
  - has few interfaces

example2


External Services
Add Tab
<<extend>>
<<include>>
Create Interface
<<include>>
Config UI
<<extend>>
Add Existing Extentions
<<extend>>
Compile UI
Create Extentions
Application Developer
<<extend>>
Hardware API Calls
<<extend>>
Design Extentions
<<extend>>
Software API Calls

Case Study 2

A player is in Jail. The player clicks the “Get out of Jail” button. $50 is decremented from their money. The player can then roll the dice and continue with the game.

A player is in Jail. The player clicks the “Get out of Jail” button. The player has less than $50. The player becomes bankrupt and all the tradable cells he or she owns becomes available in the game. The player is out of the game.

Requirements tracing

- Relating requirements and use cases
- Many-to-many relationship
  - One use case covers many individual functional requirements
  - One functional requirement may be realised by many use cases
- CASE support for requirements tracing:
  - UML tagged values - assign numbered requirements to use cases
  - Capture use case names in our Requirements Database
- Requirements Traceability matrix

		Use cases			
		U1	U2	U3	U4
Requirements	R1	■			
	R2		■	■	
	R3			■	
	R4				■
	R5	■			

Summary

- What are actors and use cases?
- What is the system boundary?
- How do you determine the above for a given system?
- What are use case diagrams comprised of?
- What are the various parts of use case specifications?
- What is requirements tracing?

Summary

- What is actor generalization?
- What is use case generalization?
- What are different types of relationships between use cases?
  - Explain <<include>> and <<extends>> relationships.
- What are the general dos and don’ts in writing use cases?

Introduction to Modeling

Dr. Issa Atoum

Faculty of Information Technology

The World Islamic Sciences & Education University

Part of these slides are from Richard N. Taylor

Software Engineering

Software engineering is an engineering discipline concerned with practical problems of developing large software systems.

Software engineering involves:

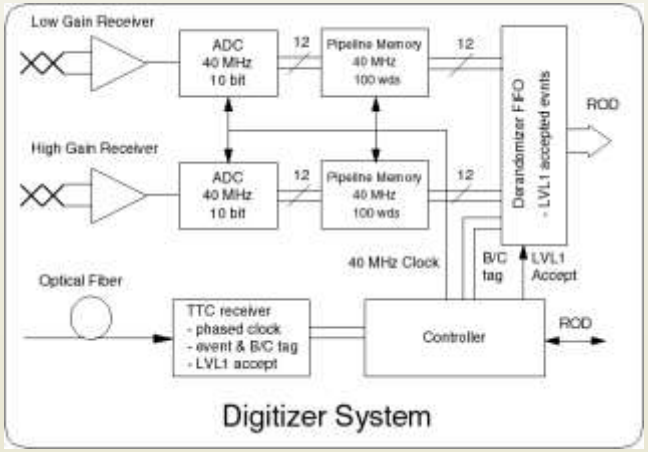
- technical and non-technical issues
- knowledge of specification, design and implementation techniques
- human factors
- software management

Engineering Models

- Engineering model:

A reduced representation of some system that highlights the properties of interest from a given viewpoint

Modeled system

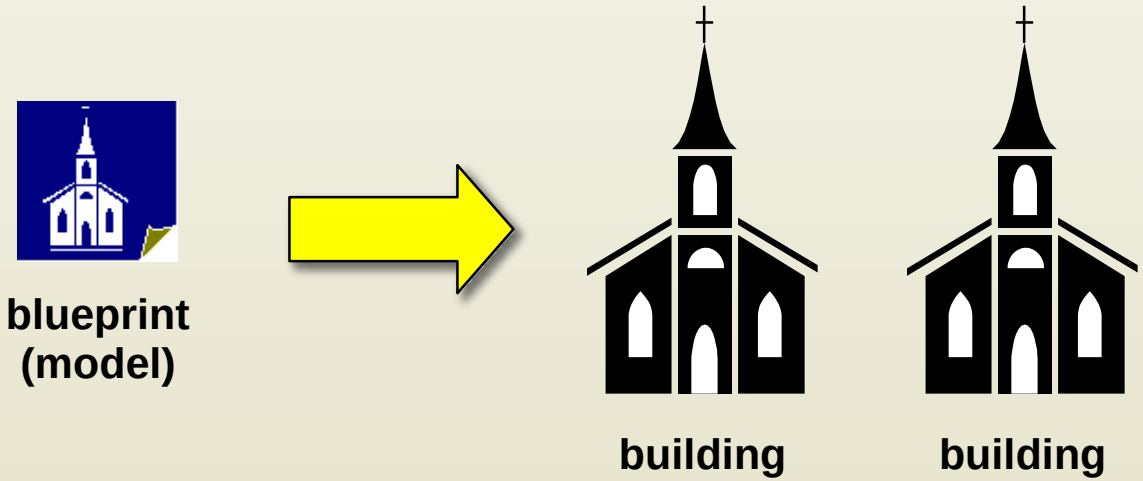


Functional Model

- We don't see everything at once
- We use a representation (notation) that is easily understood for the purpose on hand

Models

- A *model* is a description of something
  - "*a pattern for something to be made*" (Merriam-Webster)



- model ≠ thing that's modeled
  - The Map is Not The Territory

What is a Model?

- Software Modeling plays an important role in software engineering.
- A model is an abstract, simplified or incomplete description/representation of a part of the real word:
  - system under construction
  - or system under study.

Modeling

- It is the process of creating models.
- Modeling is often practiced in all engineering disciplines.
- Modeling require a power of **abstraction:**
  - Persons that cannot THINK ABSTRACTLY fail to learn modeling skills.

Abstraction

- Selective examination of certain aspects of a problem or system.
- Goal:
  - Isolate those aspects that are important for some purpose and suppress those aspects that are not important: **essential** versus **details.**
- Abstraction must always be for **some purpose**.
- Many different abstractions of the same system are possible depending on their purposes.

Abstraction

- Abstractions are incomplete and inaccurate.
- Dont search for absolute truth but for the adequacy for some purpose.
- A good model captures the crucial aspects of a problem and ignore the rest.

Why Modeling?

- **Testing a physical entity before building it**
  - Engineers test scale models of airplane, cars in wind tunnels and water tanks to improve their dynamics
  - Recent advances in computation permits the simulation of any physical structures without the need to build physical models.
- **Communication with customers**
  - Products engineers/architects build models to show to customers.
  - Prototypes/Mock ups: demonstration that imitate some of or all of the external behavior of a system

Why Modeling?

- **Reduction of complexity:**
  - The main reason for modeling is to deal with systems that are too complex to understand directly.
  - The human mind cope with only a small amount of information at one time.
  - Models reduce complexity by separating out a small number of important things to deal with at a time.

What to model?

- **Application Model:**
  - The most common reason to use modeling.
  - Helps developers understanding and analyzing requirements
  - Provides a basis for building the corresponding software
- **Enterprises/Business/Domain Model:**
  - Describes an entire organization or some major aspect of it
  - Enterprise models are not used for building software.
  - Enterprise models are used to understand an enterprise, to detect its weaknesses and eventually reengineer it for improvement.
  - Domain models allow to reconcile concepts across different applications belonging to the same domain.

Modeling in SE

- It is useful to model a system from different but related viewpoints.
- Each viewpoint captures important aspects of the system.
- A combination of these viewpoints must capture a complete description of the system.

- Each model is intended to describe one aspect/viewpoint of a system but contains references to the other models.

Uses of Models in SE

- **Models as sketches**: Developers find it useful to sketch descriptions of requirements, design or deployment concepts on whiteboards or paper when discussing their ideas with other developers or customer representatives
- **Models as analysis artifacts**: **Developers build analyzable models to check specified properties (e.g., consistency and satisfiability properties), to predict implementation qualities (e.g., performance), or to simulate implemented behavior**
- **Models as the basis for code generation or synthesis of software artifacts**: **Models can be built for the purpose of generating implementations, test cases, deployment or software configuration scripts, or other software artifacts.**

Which viewpoints to model?

- Functional/process aspect.
- Structural/Data aspect
- Behavioral/Dynamic aspect
- Interaction/collaboration aspect

How to express Models?

- **Need of modeling languages** to express models.
- Modeling languages should be simple enough to be comprehensible and usable by modelers.
- Modeling languages should be **expressive** and **abstract.**

Typology of modeling languages

- Various criteria may be used to make a classification of modeling languages:
  - Textual versus diagram based syntax.
  - General purpose or Domain based modeling languages

The Unified Modeling Language

- The Unified Modeling Language (UML) is a standard language for writing software blueprints.
- The UML may be used to **visualize**, specify, **construct** and document the artifacts of a software-intensive system.
  - **Visualizing** means graphical language
  - Specifying means building precise, unambiguous, and complete models
  - **Constructing** means that models can be directly connected to a variety of programming languages

System modeling

- System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.
- System modeling has now come to mean representing a system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML).
- System modelling helps the analyst to understand the functionality of the system and models are used to communicate with customers.

Existing and planned system models

- Models of the existing system are used during requirements engineering. They help clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system.
- Models of the new system are used during requirements engineering to help explain the proposed requirements to other system stakeholders. Engineers use these models to discuss design proposals and to document the system for implementation.
- In a model-driven engineering process, it is possible to generate a complete or partial system implementation from the system model.

System perspectives

- An external perspective, where you model the context or environment of the system.
- An interaction perspective, where you model the interactions between a system and its environment, or between the components of a system.
- A structural perspective, where you model the organization of a system or the structure of the data that is processed by the system.
- A behavioral perspective, where you model the dynamic behavior of the system and how it responds to events.

UML diagram types

- Activity diagrams, which show the activities involved in a process or in data processing .
- Use case diagrams, which show the interactions between a system and its environment.
- Sequence diagrams, which show interactions between actors and the system and between system components.
- Class diagrams, which show the object classes in the system and the associations between these classes.
- State diagrams, which show how the system reacts to internal and external events.

Use of graphical models

- As a means of facilitating discussion about an existing or proposed system
  - Incomplete and incorrect models are OK as their role is to support discussion.
- As a way of documenting an existing system
  - Models should be an accurate representation of the system but need not be complete.
- As a detailed system description that can be used to generate a system implementation
  - Models have to be both correct and complete.

Context models

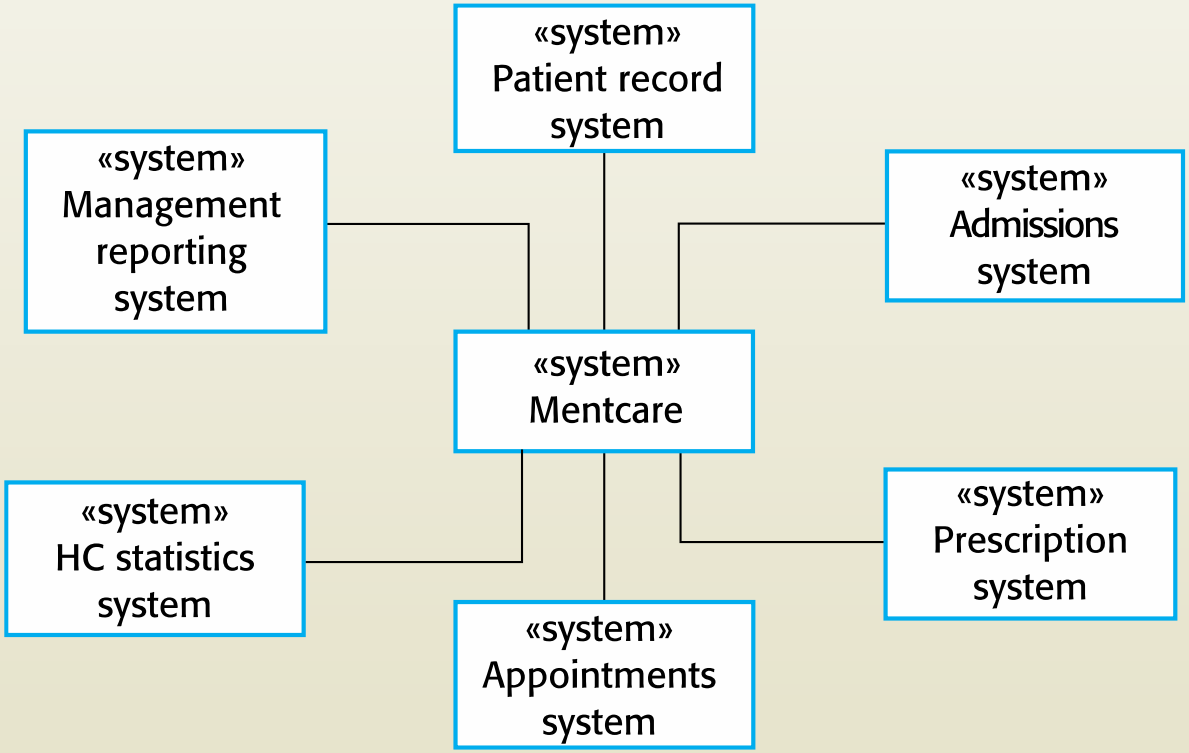
Context models

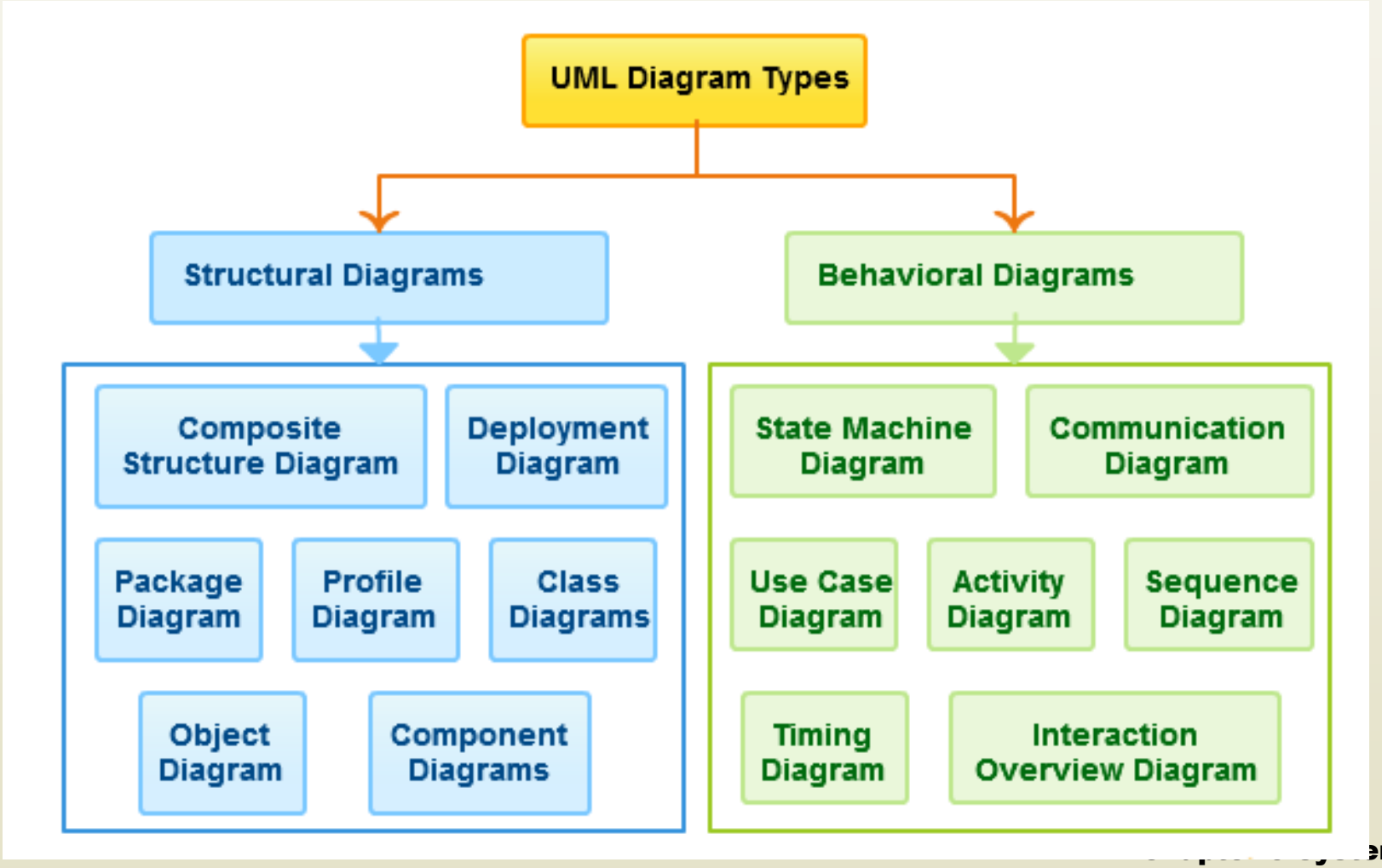
- Context models are used to illustrate the operational context of a system - they show what lies outside the system boundaries.
- Social and organisational concerns may affect the decision on where to position system boundaries.
- Architectural models show the system and its relationship with other systems.

System boundaries

- System boundaries are established to define what is inside and what is outside the system.
  - They show other systems that are used or depend on the system being developed.
- The position of the system boundary has a profound effect on the system requirements.
- Defining a system boundary is a political judgment
  - There may be pressures to develop system boundaries that increase / decrease the influence or workload of different parts of an organization.

The context of the Mentcare system




UML Diagram Types
Structural Diagrams
Behavioral Diagrams
Composite Structure Diagram
Deployment Diagram
Package Diagram
Profile Diagram
Class Diagrams
Object Diagram
Component Diagrams
State Machine Diagram
Communication Diagram
Use Case Diagram
Activity Diagram
Sequence Diagram
Timing Diagram
Interaction Overview Diagram

Behavioral Modeling:
Sequence and Communication Diagrams

Learning Objectives

- Develop interaction diagrams based on the principles of object responsibility and use case controllers
- Develop sequence diagrams to model scenarios
- Develop communication diagrams and understand the difference between sequence and communication diagrams
- Explain the relationship between the behavioral models and the structural and functional models.

Behavioral Models

- Systems have static &dynamic characteristics
  - Structural models describe the static aspects of the system
  - Behavioral models describe the dynamics and interactions of the system and its components
- Behavioral models describe how the classes described in the structural models interact in support of the ***use cases***.

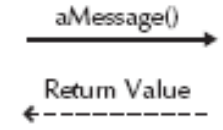
Sequence Diagrams

- Illustrate the objects that participate in a use-case
- Show the messages that pass between objects for a particular use-case

Sequence Diagram Syntax

An actor: ■ Is a person or system that derives benefit from and is external to the system. ■ Participates in a sequence by sending and/or receiving messages. ■ Is placed across the top of the diagram. ■ Is depicted either as a stick figure (default) or, if a nonhuman actor is involved, as a rectangle with <<actor>> in it (alternative).	 anActor <<actor>> Actor/Role
An object: ■ Participates in a sequence by sending and/or receiving messages. ■ Is placed across the top of the diagram.	anObject : aClass
A lifeline: ■ Denotes the life of an object during a sequence. ■ Contains an X at the point at which the class no longer interacts.	
An execution occurrence: ■ Is a long narrow rectangle placed atop a lifeline. ■ Denotes when an object is sending or receiving messages.	

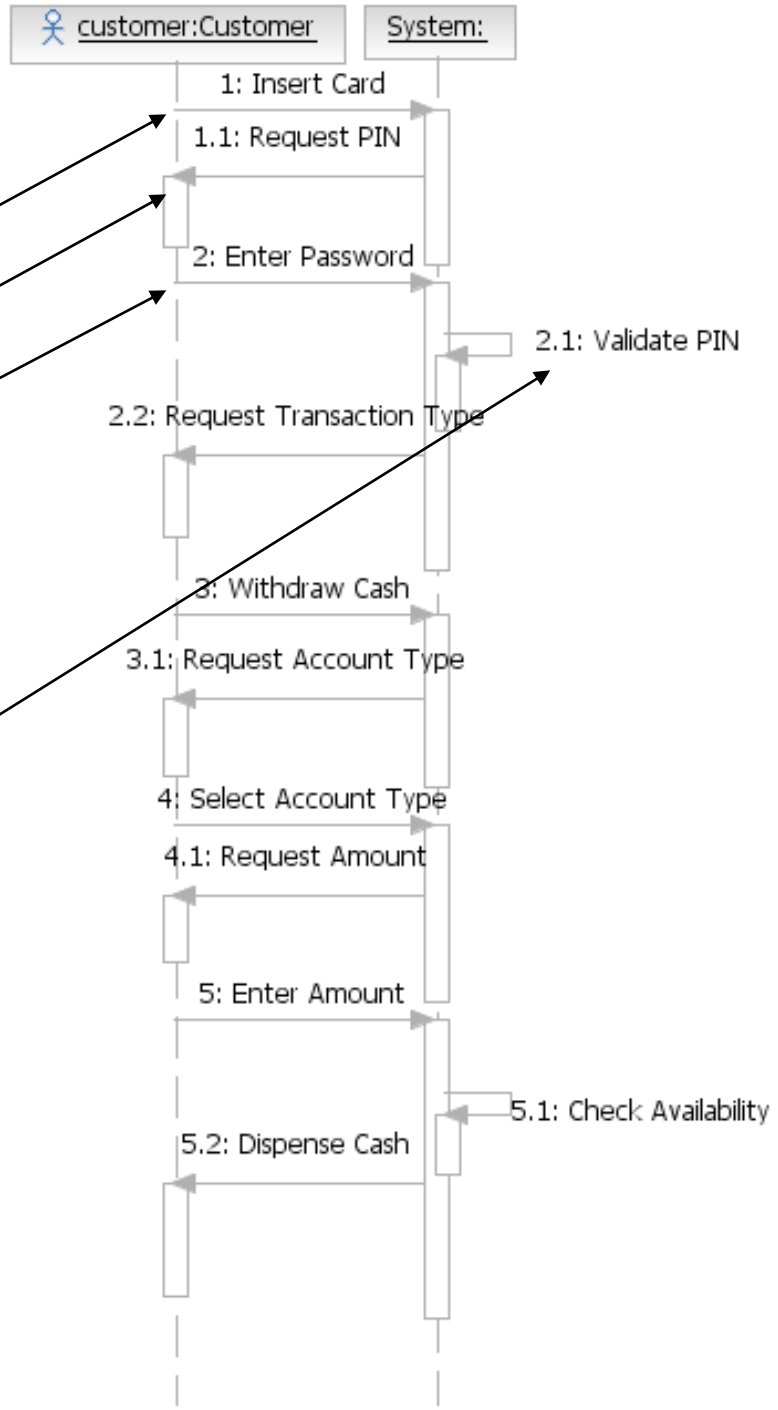
More Sequence Diagram Syntax

A message: ■ Conveys information from one object to another one. ■ A operation call is labeled with the message being sent and a solid arrow, whereas a return is labeled with the value being returned and shown as a dashed arrow.	
A guard condition: ■ Represents a test that must be met for the message to be sent.	
For object destruction: ■ An X is placed at the end of an object's lifeline to show that it is going out of existence.	X
A frame: ■ Indicates the context of the sequence diagram.	

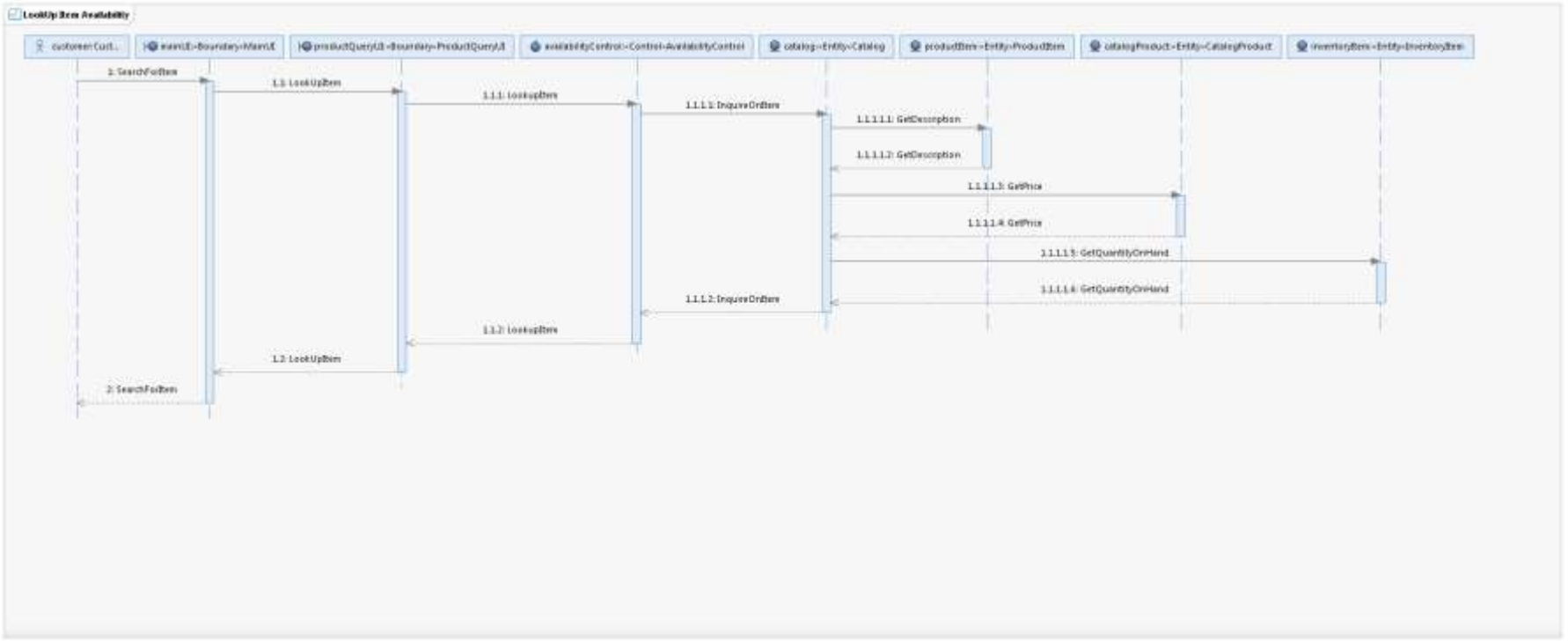
Use case to System Sequence diagram (SSD)

- Typical course of events:

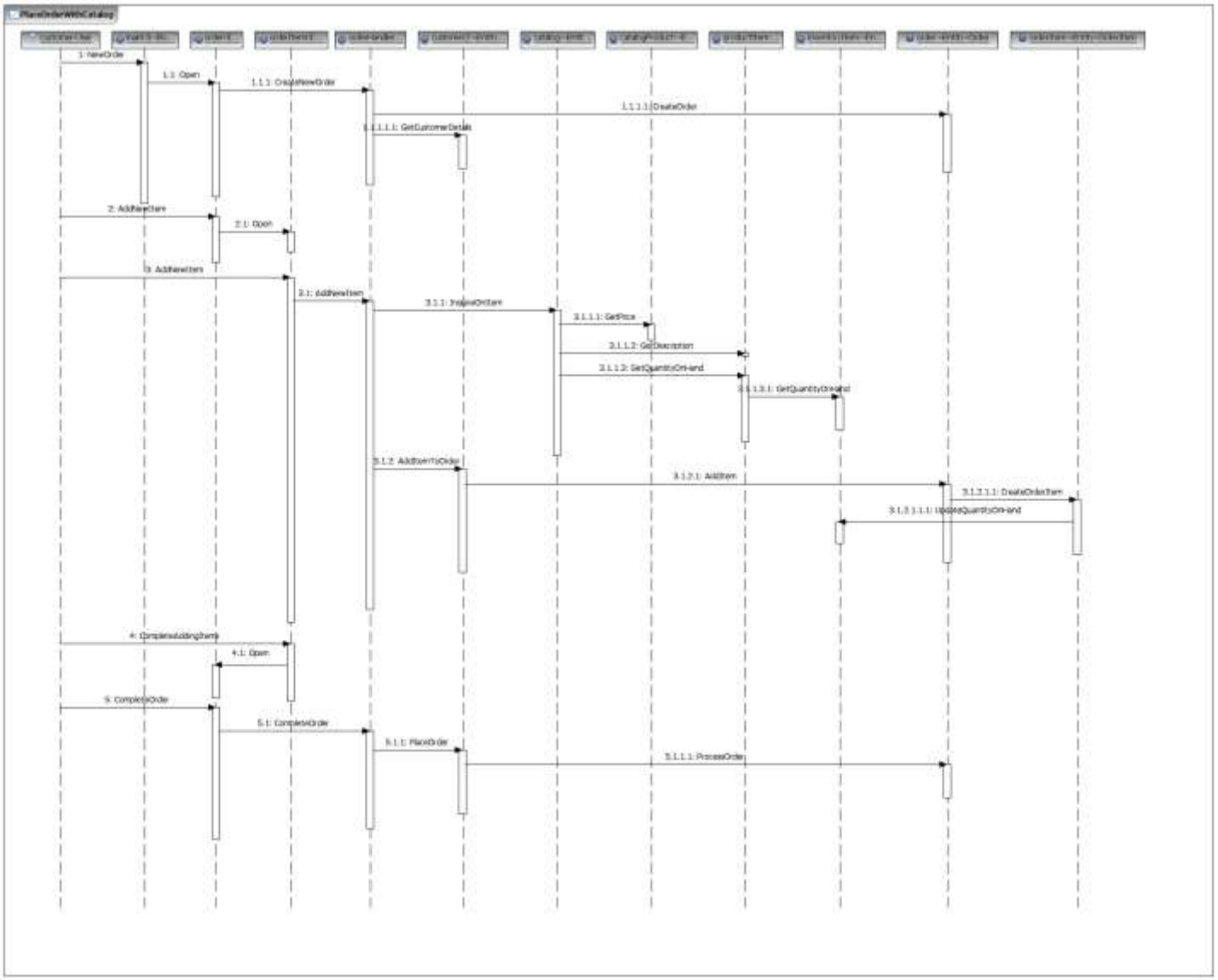
Actor Action	**System Response**
1. Customer inserts card	1.1 System asks for PIN
2. Customer enters PIN	2.1 System validates PIN (validate PIN use case is used here)
	2.2 If PIN correct, system asks for type of transaction
3. Customer selects 'Withdraw cash'	3.1 System asks for account type
4. Select Ac Type	4.1 Request Amount
5. Enter Amount	5.1 Check Availability
	5.2 Dispense Cash



Sequence for *Look Up Item Availability*

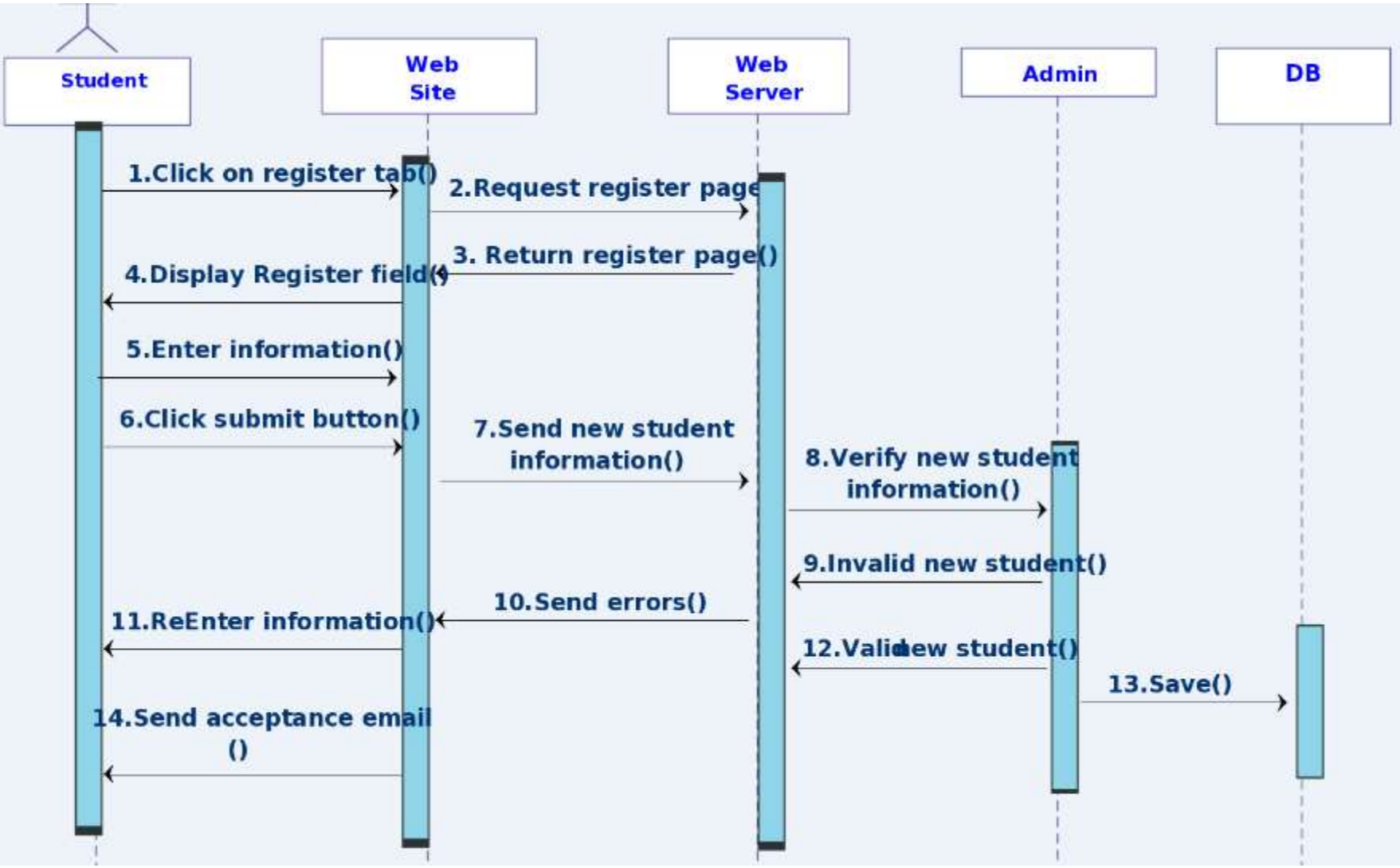


Place Order with Catalog



Example

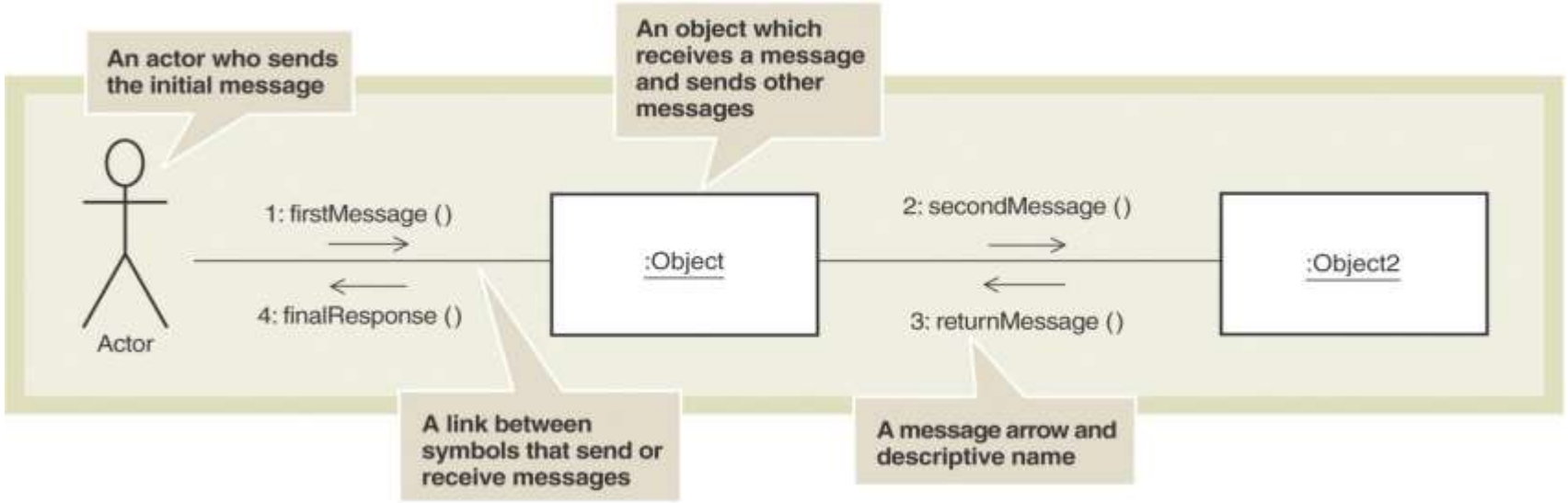
- Let us draw student registration in a sequence diagram


Student
Web Site
Web Server
Admin
DB
1.Click on register tab()
2.Request register page
3. Return register page()
4.Display Register field()
5.Enter information()
6.Click submit button()
7.Send new student information()
8.Verify new student information()
9.Invalid new student()
10.Send errors()
11.ReEnter information()
12.Valid new student()
13.Save()
14.Send acceptance email ()

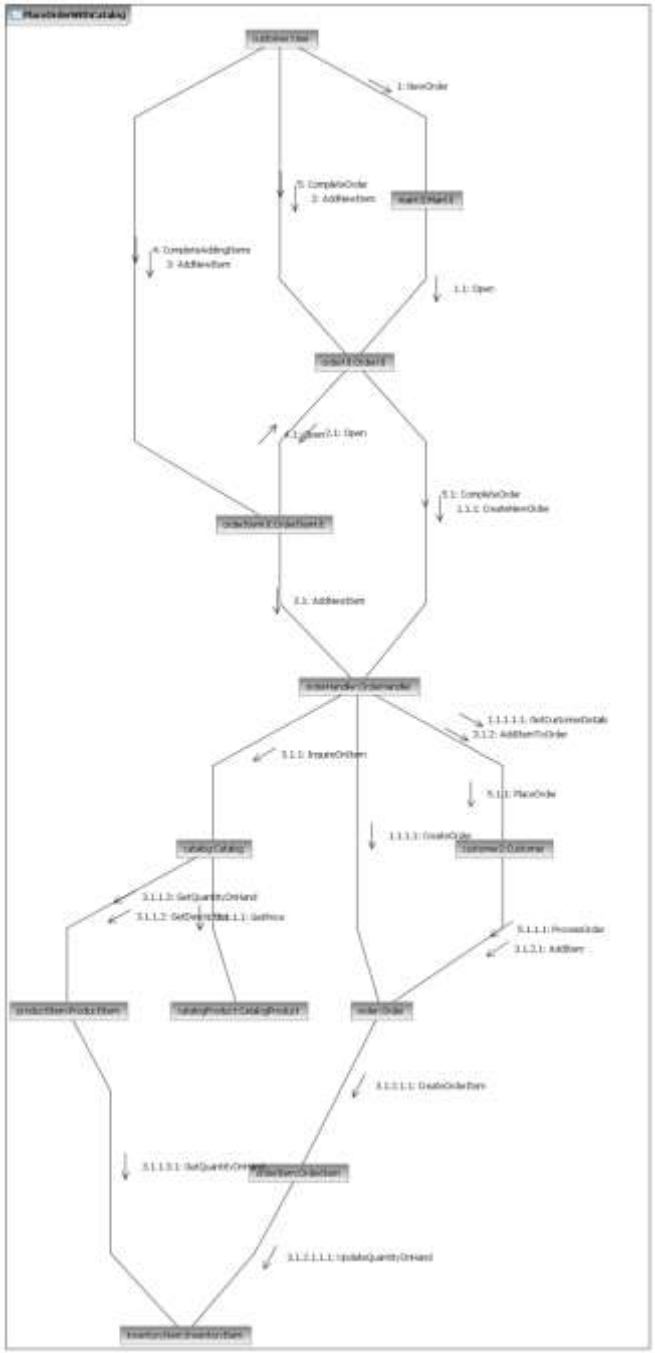
Designing with Communication Diagrams

- Communication diagrams and sequence diagrams
  - Both are interaction diagrams
  - Both capture same information
- Which one to create first is designer's personal preference
  - Sequence diagram – because use case descriptions and dialog follow sequence of steps
  - Communication diagram – highlights coupling

Communication Diagram Notation



Communication Diagram – Place Order



Updating the Design Class Diagram

- Based on what you add in your sequence diagrams, you should update your class diagram
- New objects, messages, etc. that are used in sequence diagram should match their corresponding classes and methods in class diagram

Summary

- What is a use case realization?
- What are interaction diagrams?
- What are sequence diagrams?
- What are communication diagrams?
- Develop sequence and communication diagrams for a given scenario.

Activity Diagrams

Learning Objectives

- Describe the various purposes for which we can use activity diagrams
- Describe the different elements of an activity diagram
- Develop activity diagrams to model specific processes

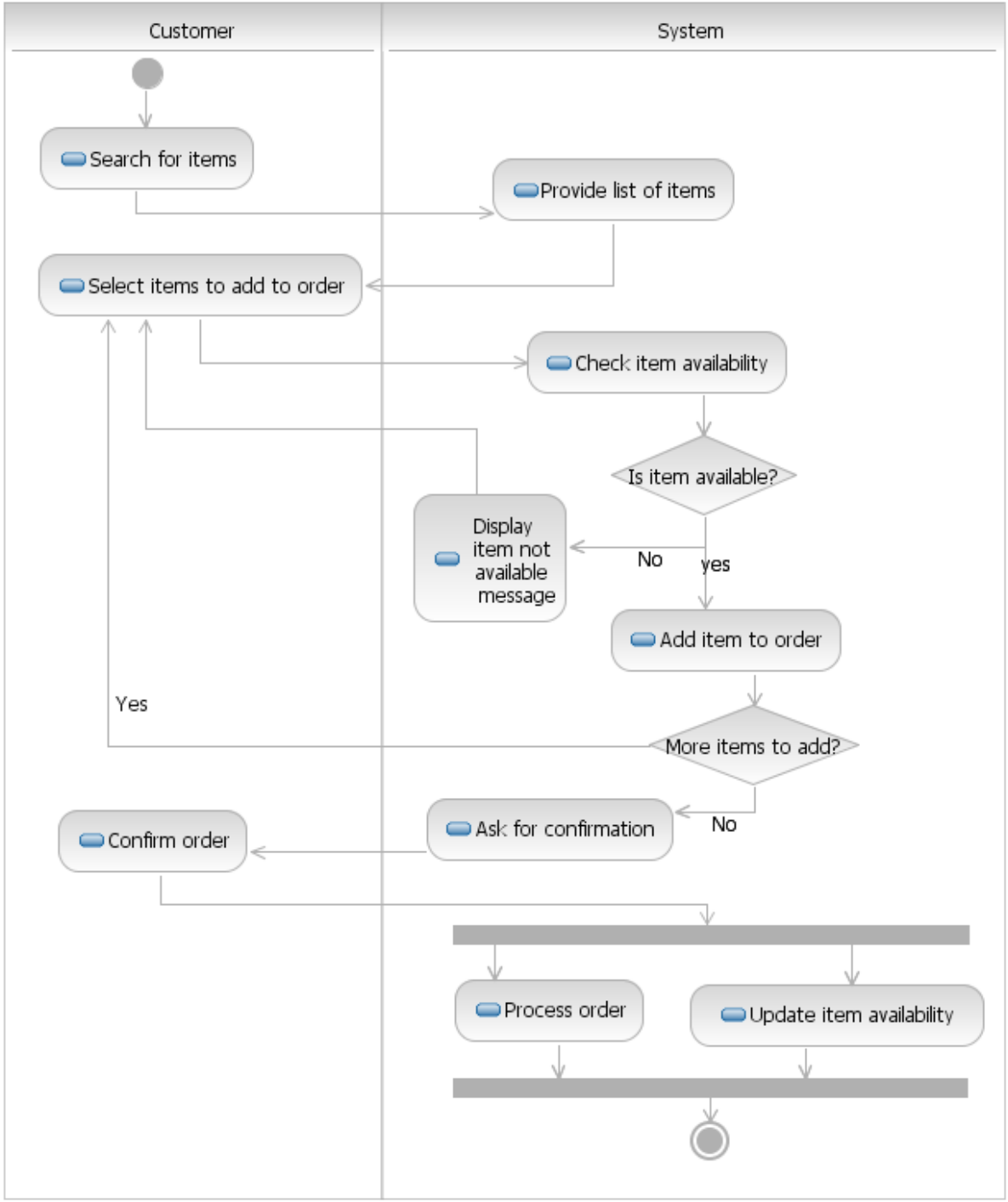
Activity Diagrams

- OO flowcharts
- Can be used for:
  - Business process modeling
  - Use case sequence of action
  - Modeling methods in classes

Business Process Modeling

- Business process models describe the activities that collectively support a business process
- A very powerful tool for communicating the analyst's current understanding of the requirements with the user
- Activity diagrams are used to model the behavior in a business process

Sample Activity Diagram



Activity Diagram Syntax

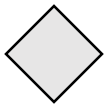
- Action or Activity

- Control Flow

- Initial Node

- Final Node

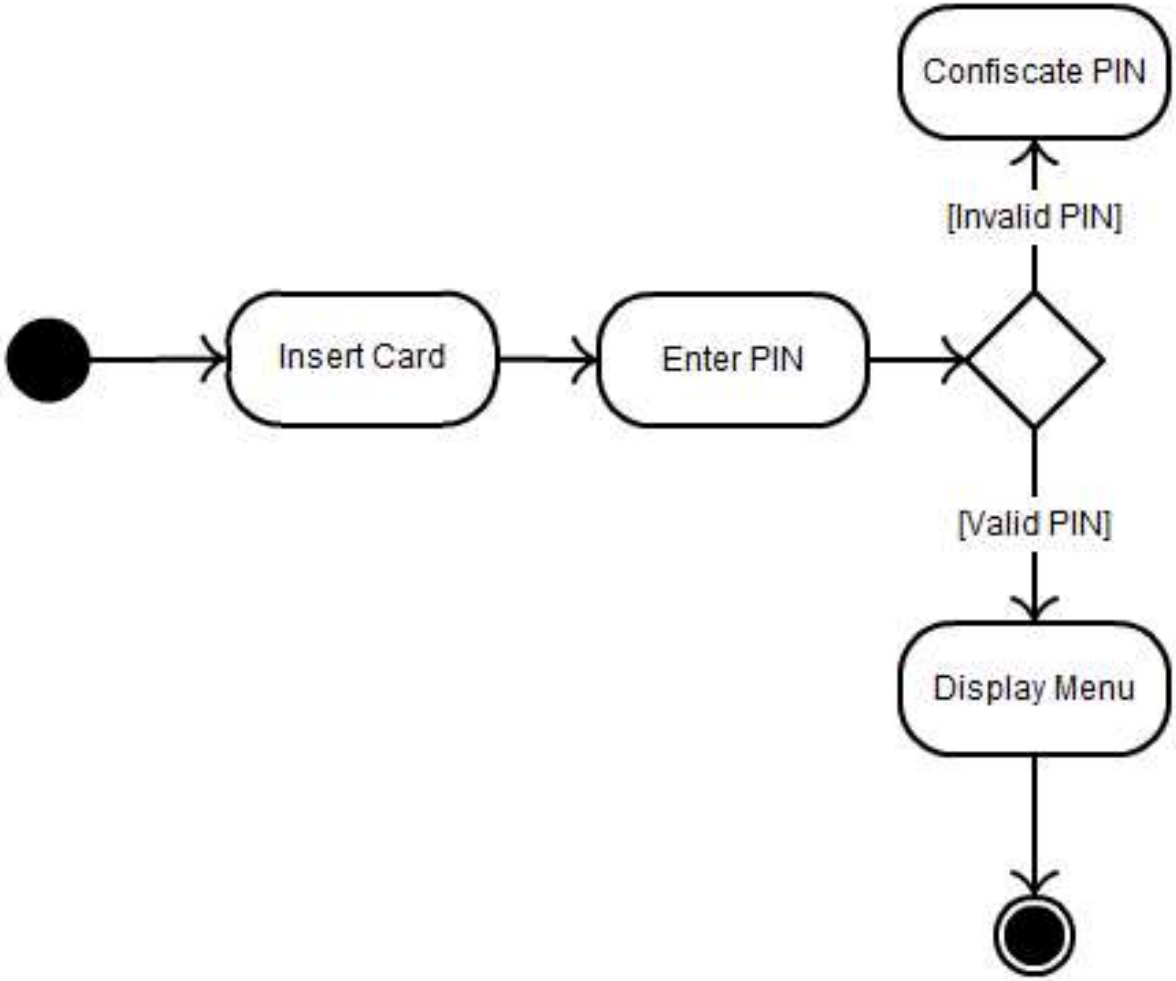
- Decision Node



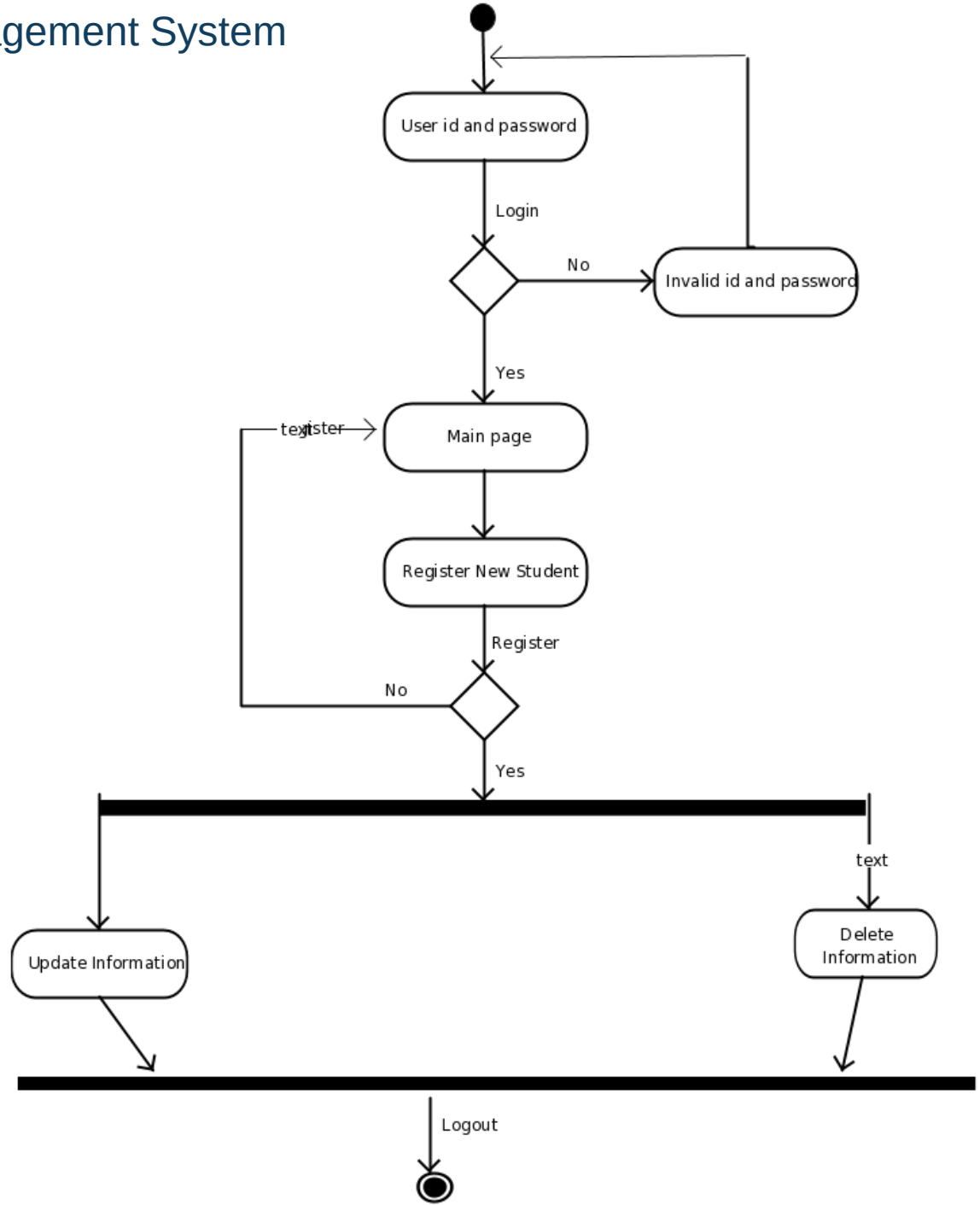
Guidelines for Activity Diagrams

1. Set the scope of the activity being modeled
2. Identify the activities, control flows, and object flows that occur between the activities
3. Identify any decisions that are part of the process being modeled
4. Identify potential concurrencies in the process

Example (1)


Confiscate PIN
[Invalid PIN]
Insert Card
Enter PIN
[Valid PIN]
Display Menu

Student Management System



Activity diagrams can replace use case diagrams.

1. Yes
2. No

How do you show conditions in an activity diagram?

A. Decision node (diamond)
B. Activity (rectangle)
C. Fork and join
D. Extends
E. Includes

Which one of the following is an activity diagram useful for?

A. Flowchart
B. To depict a business process
C. To show main flow
D. A, B, & C
E. To show includes
F. D and E
G. To show inheritance

How will preconditions show up in an activity diagram?

A. Not show up at all
B. As an activity
C. As a decision node
D. As transitions

Summary

- What is an activity diagram?
- What can we model using activity diagrams?
- What are the different elements in an activity diagram?
- Develop activity diagrams for a given process.

Behavioral Modeling: State Diagrams

Learning Objectives

- Explain how statecharts can be used to describe system behaviors
- Use statecharts to model object and system behaviors
- Explain how state charts are different from other diagrams that we have seen so far

Analysis and Design so far…

- Use case diagrams
  - What? Shows functionalities of the system that provides measurable value to actors
  - Why? Must understand functionality of the system from the perspective of one or more actors that interact with the system
- Class diagrams
  - What? Shows elements in the system, static structure, things and relations
  - Why? Must represent elements that are to be remembered by the system; Used to understand the system structure; Interaction of classes fulfill functionalities

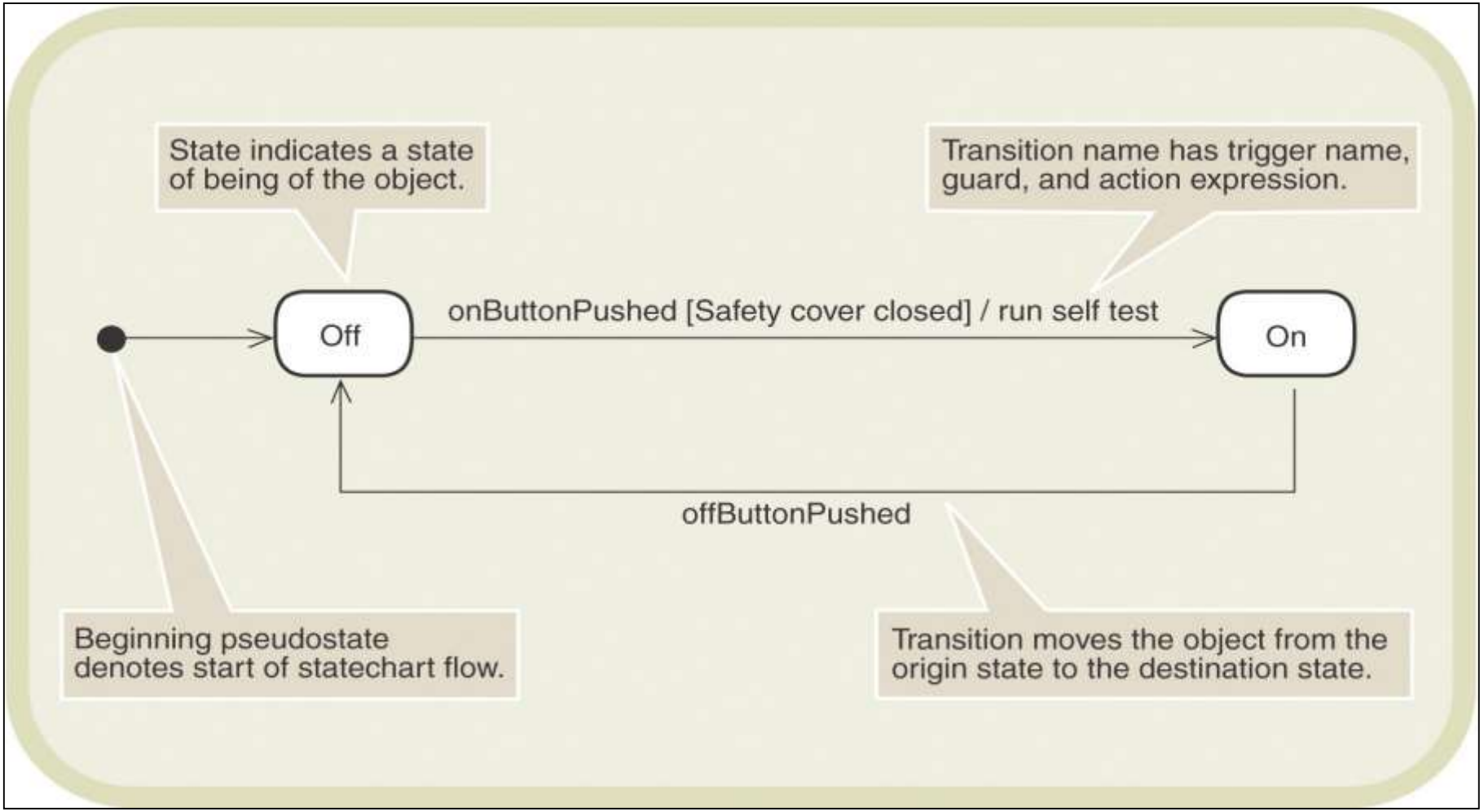
Analysis and Design so far…

- Interaction diagrams (Sequence and communication)
  - What? Represents sequence of interactions among a set of objects
  - Why? Must capture interactions among objects in a chronological order – this represents how a use case is executed in a particular scenario; Use case realization
- State diagrams
  - What? Show the possible interesting states that objects of a class can have and the events that cause the transitions to and from those states
  - Why? Must capture ‘temporal’ evolution of an object in response to interactions

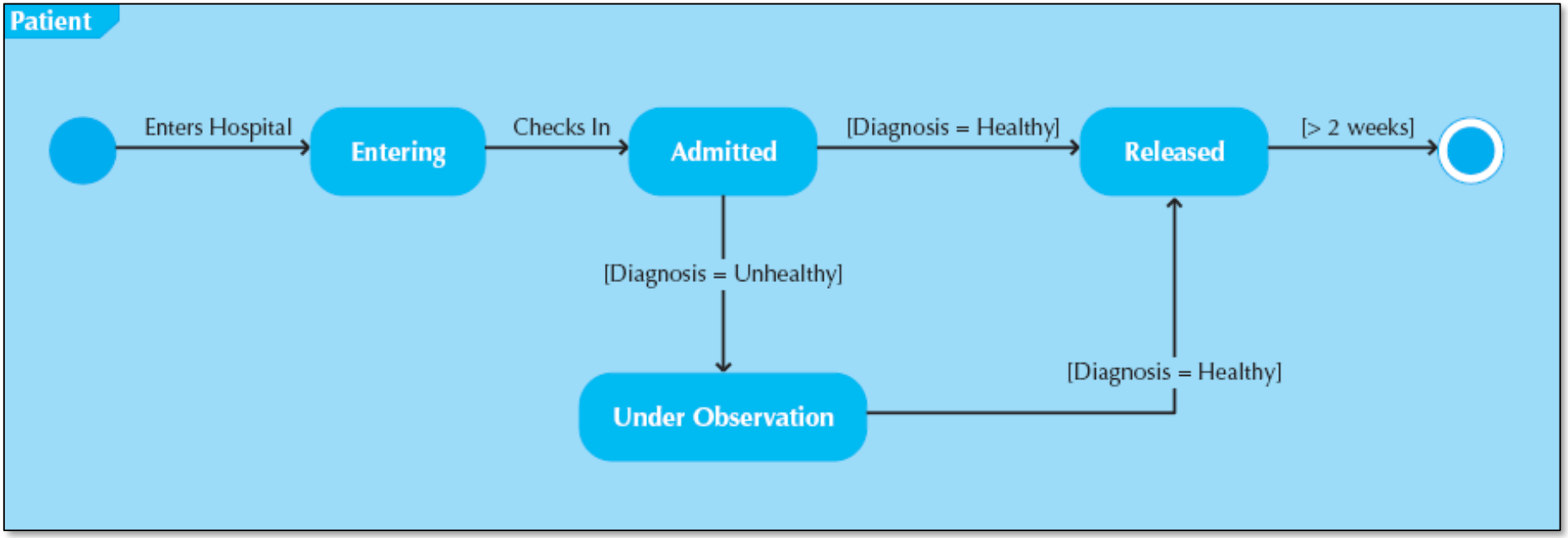
State Diagrams

- A dynamic model that shows the different states through which a single object passes during its life in response to events, along with its responses and actions
- Typically not used for all objects
  - Just for complex ones
- Does an object behave differently to stimuli from environment and other objects when it is in different states?

Simple Statechart for a Printer



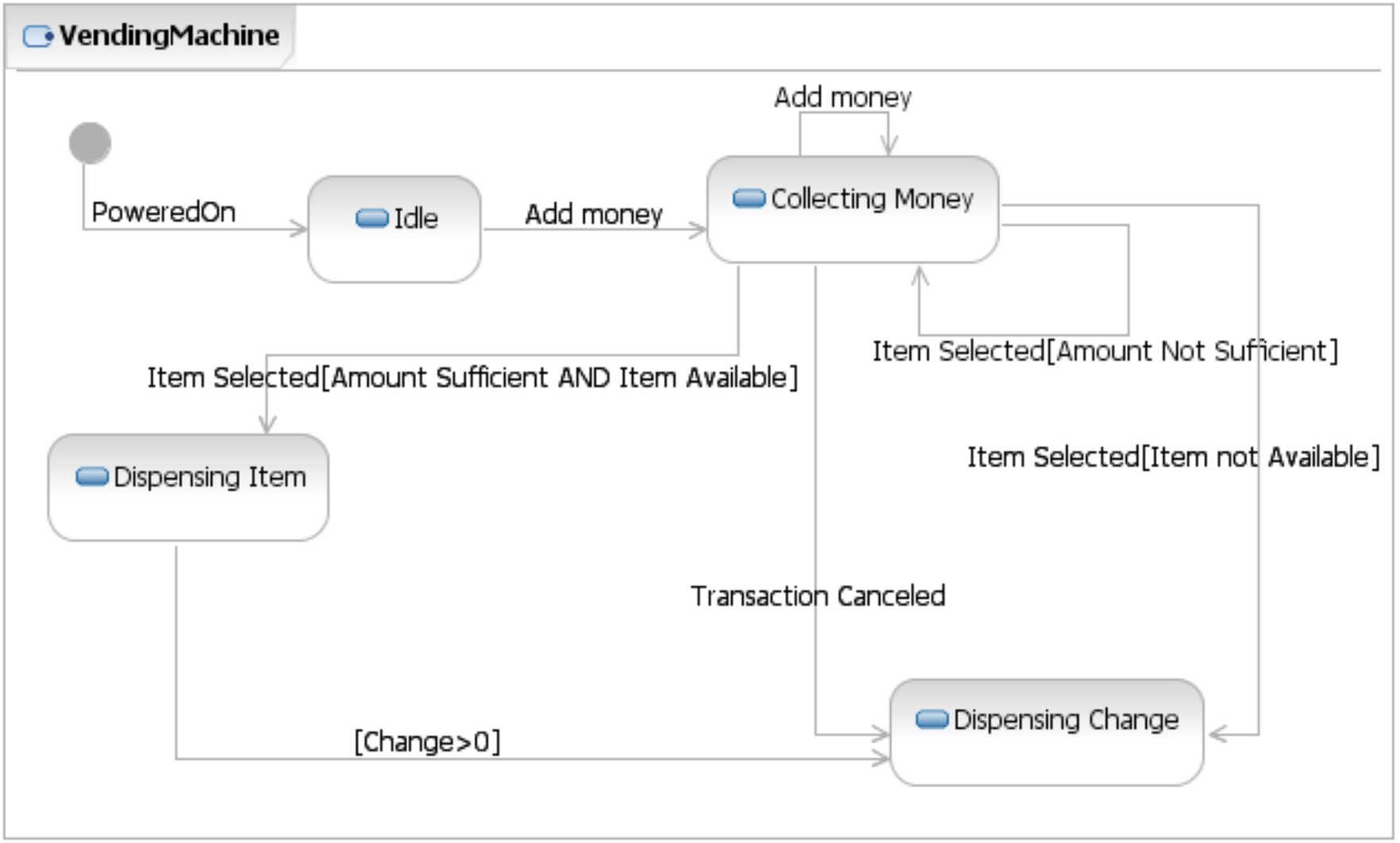
Sample State Diagram for Patient Class



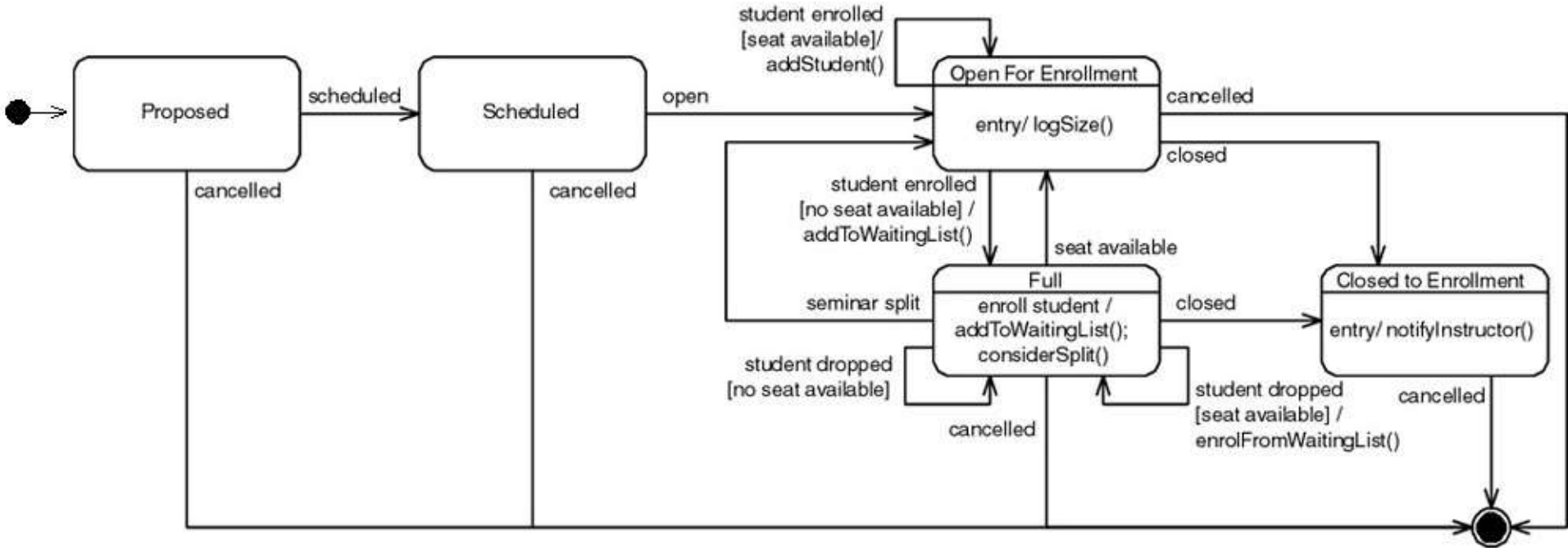
Parts of a State Chart

- Pseudo state – starting point of a statechart
- State – condition that occurs during an object's life when it satisfies some criteria, performs some action, or waits for an event
- Transition – movement of object from one state to another state
- Message event – trigger for the transition

Vending Machine Example

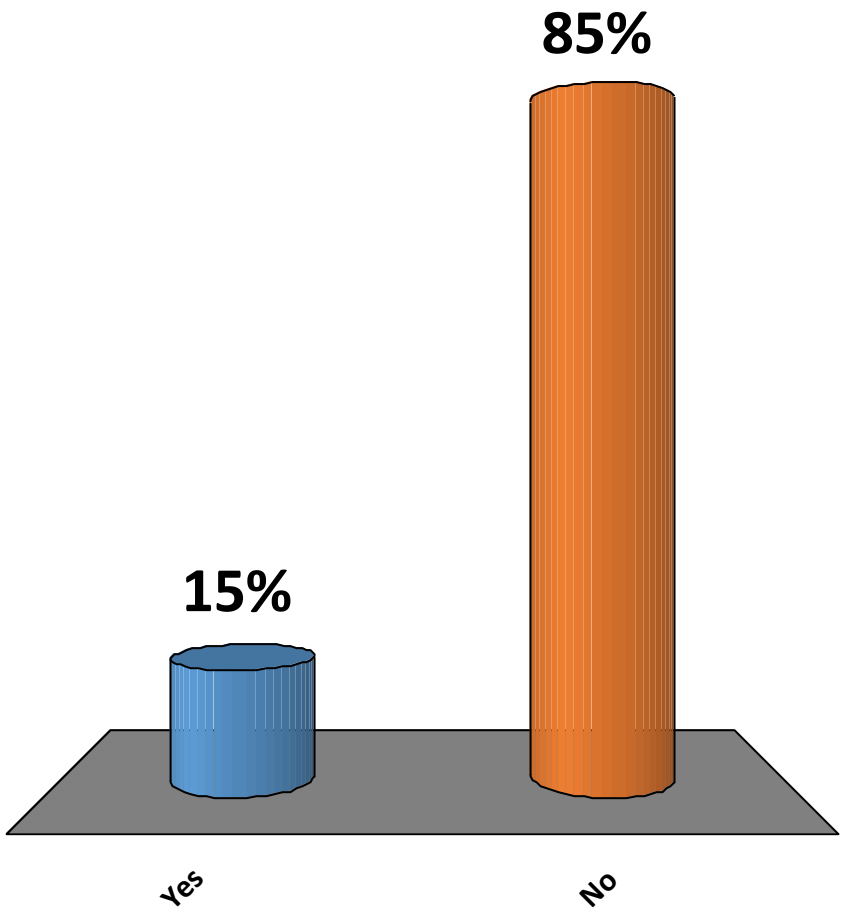


Seminar


Proposed
scheduled
Scheduled
open
Open For Enrollment
entry/ logSize()
student enrolled
[seat available]/
addStudent()
cancelled
closed
cancelled
cancelled
student enrolled
[no seat available] /
addToWaitingList()
seat available
Full
enroll student /
addToWaitingList();
considerSplit()
seminar split
closed
Closed to Enrollment
entry/ notifyInstructor()
student dropped
[no seat available]
cancelled
student dropped
[seat available] /
enrolFromWaitingList()
cancelled

Can a seminar object exist in more than one state at the same time?

A. Yes
B. No



State Details

- Based on attribute values
- Specifies the response of an object to input events
  - Response qualitatively different in different states
  - E.g.: Solvent customer places order

State Details

- Response to input events may be an action or change of state
  - E.g.: Place order, become insolvent
- Often associated with value of an object satisfying some condition
  - Customer credit limit balance – change in the value of this attribute might dictate state of solvency

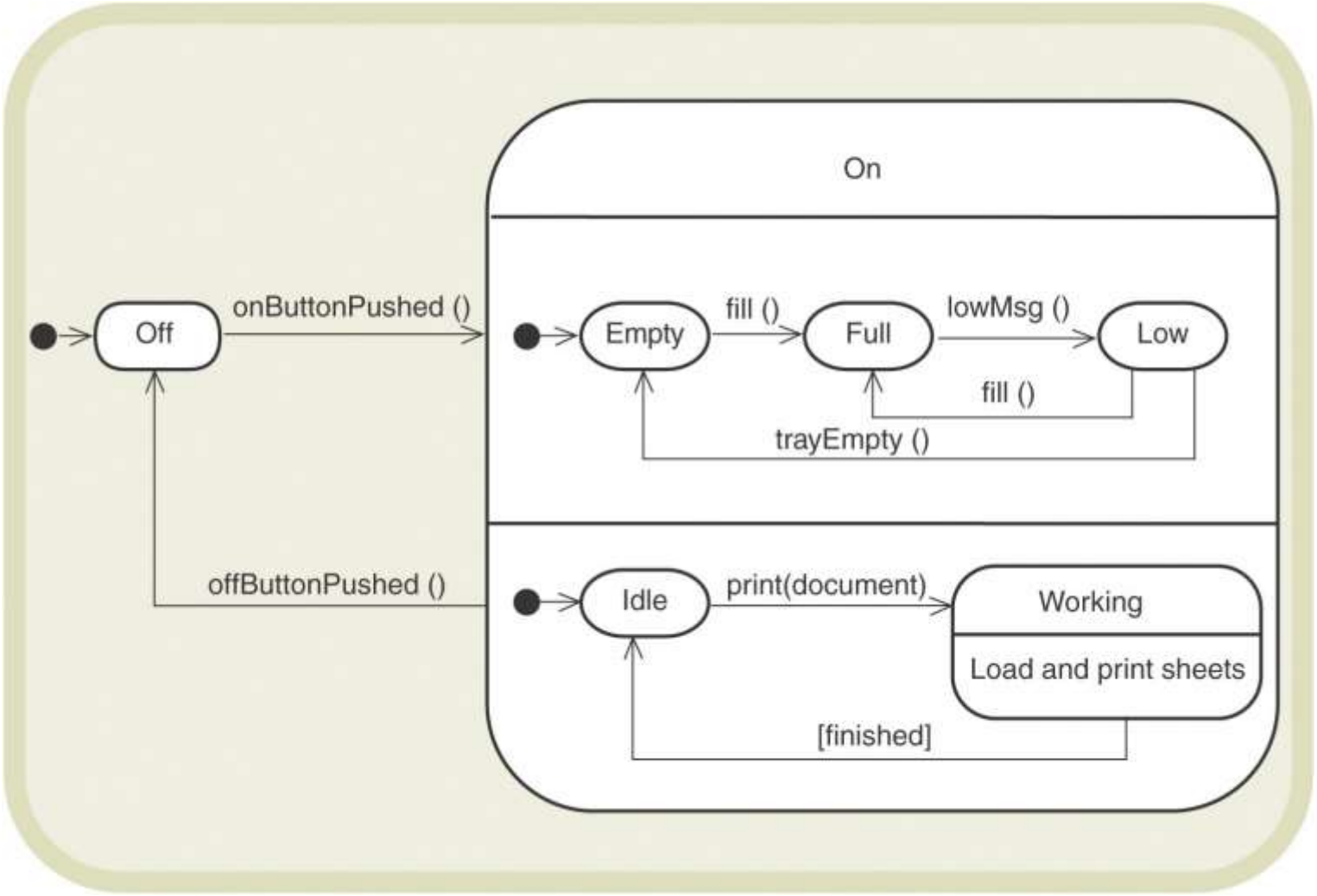
State Transition Details

- If an event occurs, the transition is fired
- A transition can lead to the same state
- A state transition may have an action and/or a guard condition associated with it
- An ACTION is behavior that occurs when the state transition occurs
- A GUARD CONDITION is a Boolean expression of attribute values that allows a state transition only if the condition is true

Concurrency

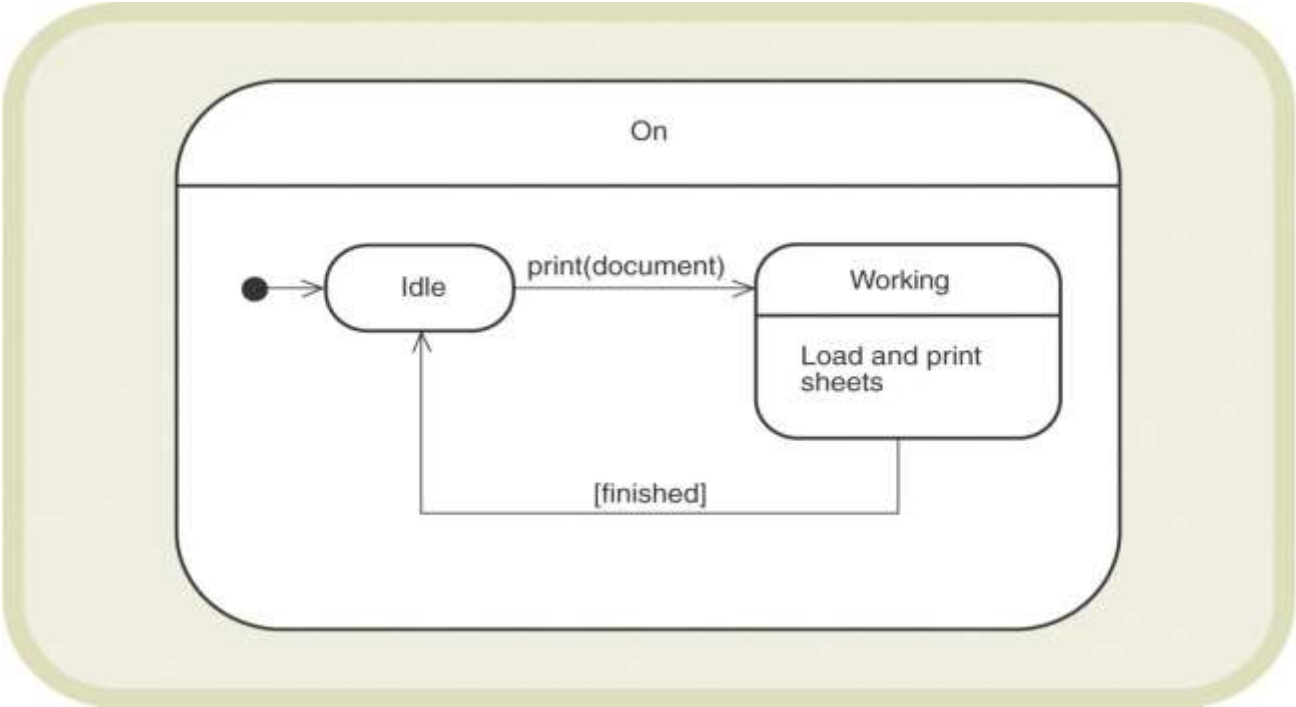
- Concurrency – condition of being in more than one state at a time
- Objects in a system are inherently concurrent
- Concurrency within an object can be represented by multiple subsets of states, each with their own initial state
- Differentiated by different state variables

Concurrent States for a Printer in the *On* State



Nested States

- Composite states – state containing multiple levels and transitions
  - Can contain nested states and transition paths



Developing State charts

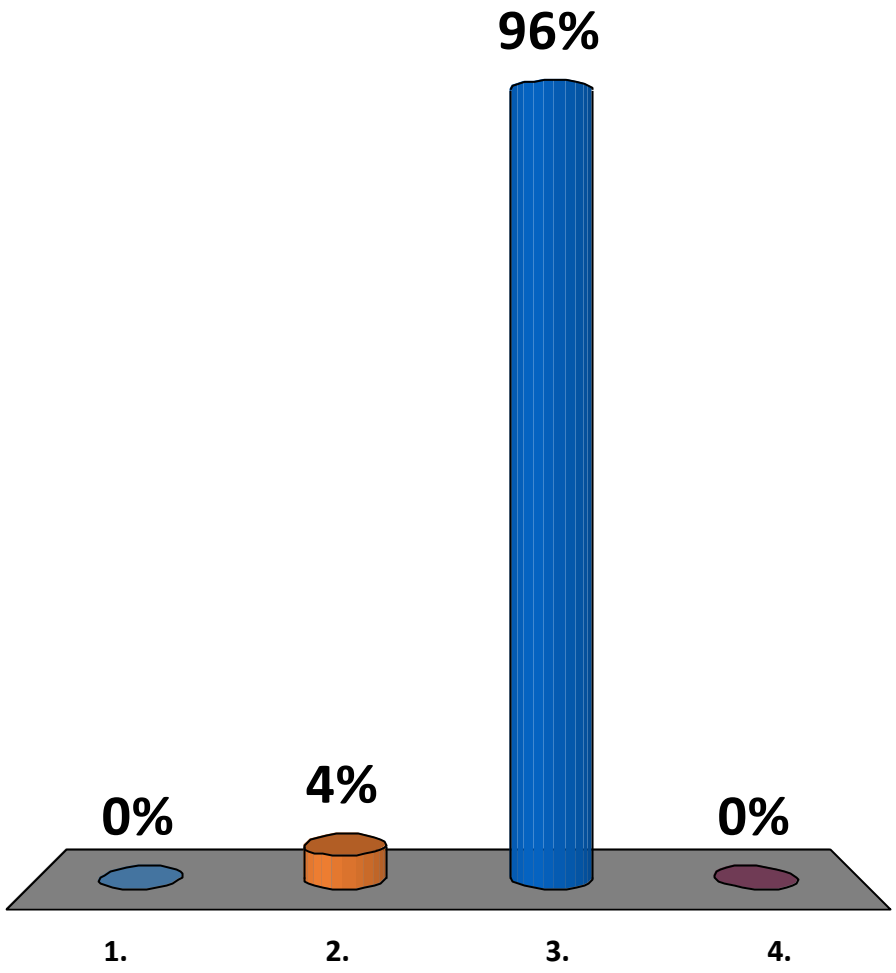
- Review class diagram and select classes that require statecharts
- For each selected class in group, brainstorm to list all status conditions you can identify
- Begin building statechart fragments by identifying transitions that cause object to leave identifying state
- Sequence these state-transition combinations in correct order

Developing State charts (contd.)

- Review paths and look for independent, concurrent paths
- Look for additional transitions
  - Take every pairwise combination of states and look for valid transition between states
- Expand each transition with the appropriate message event, guard-condition, and action-expression
- Review and test each statechart

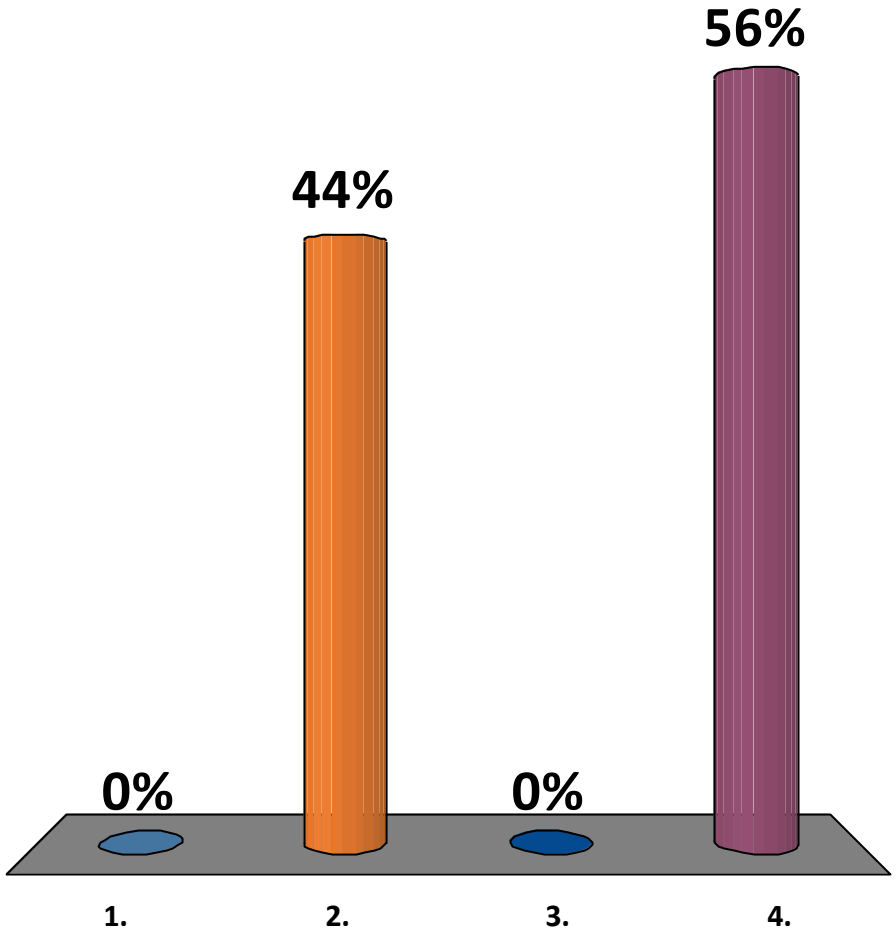
A state is related to a…

1. Use case
2. Sequence
3. Class
4. Main flow



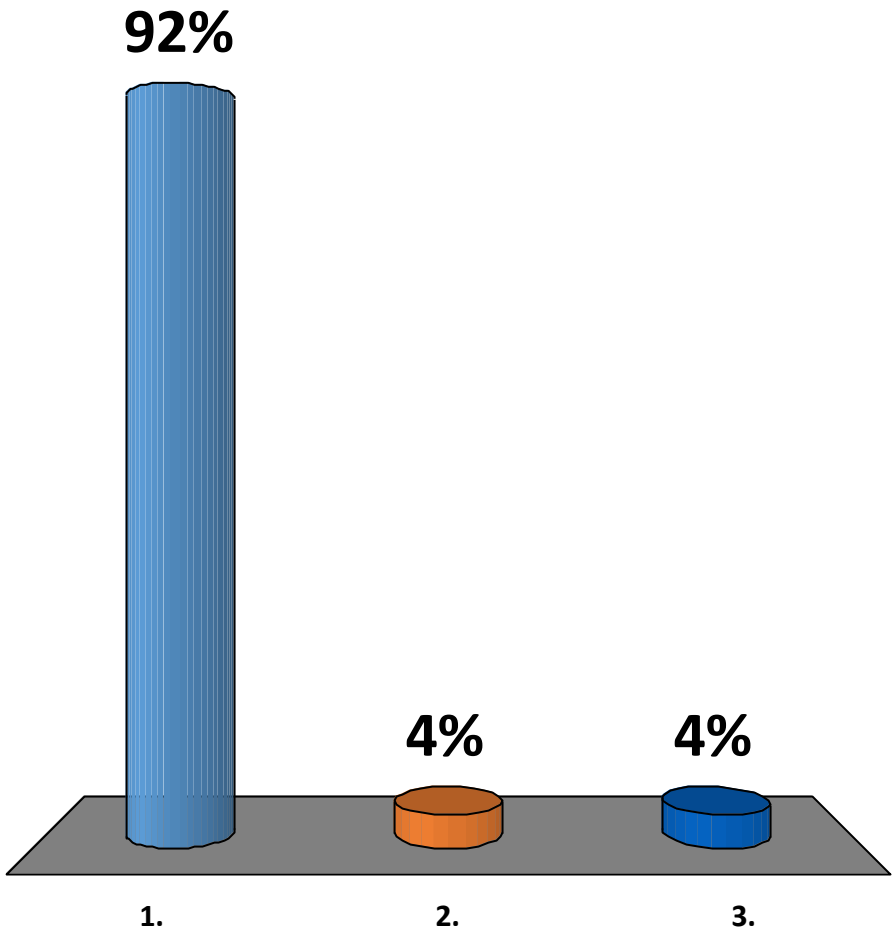
Which of the following is a valid state for the class employee?

1. Manager
2. Retired
3. Executive
4. Full-time employee



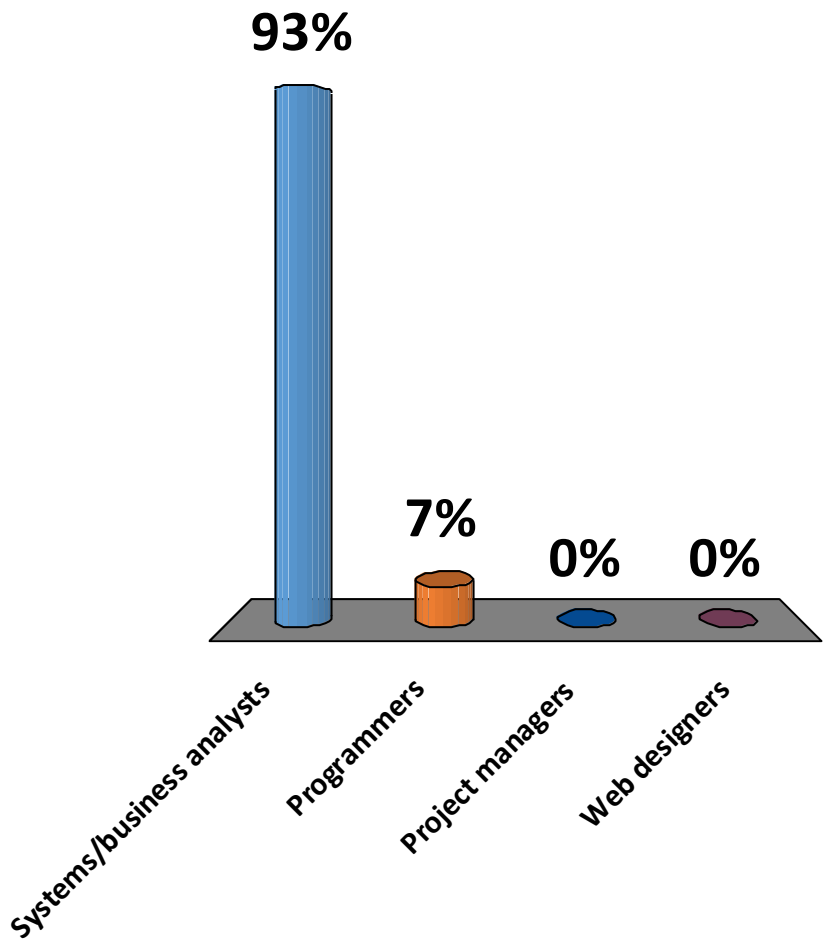
In a state diagram, can an object be in more than one state at the same time?

1. Yes, if it is a concurrent state diagram
2. Yes
3. No



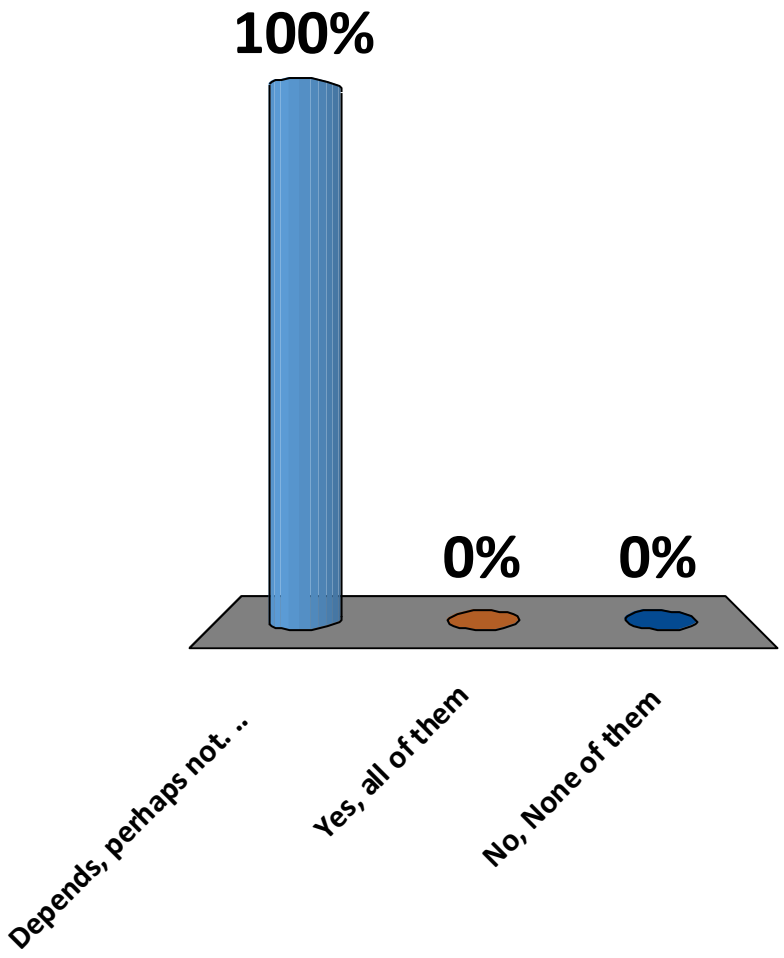
Who is this course the most useful for?

A. Systems/business analysts
B. Programmers
C. Project managers
D. Web designers



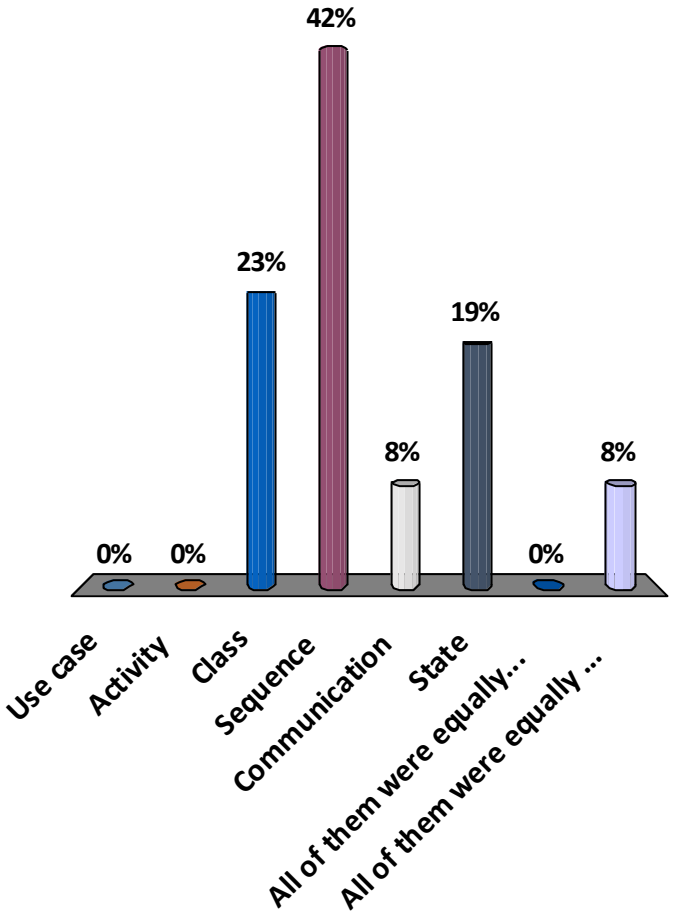
Would you use all the diagrams we learnt in a single project?

A. Depends, perhaps not. We have to pick and choose what's best for that project
B. Yes, all of them
C. No, None of them



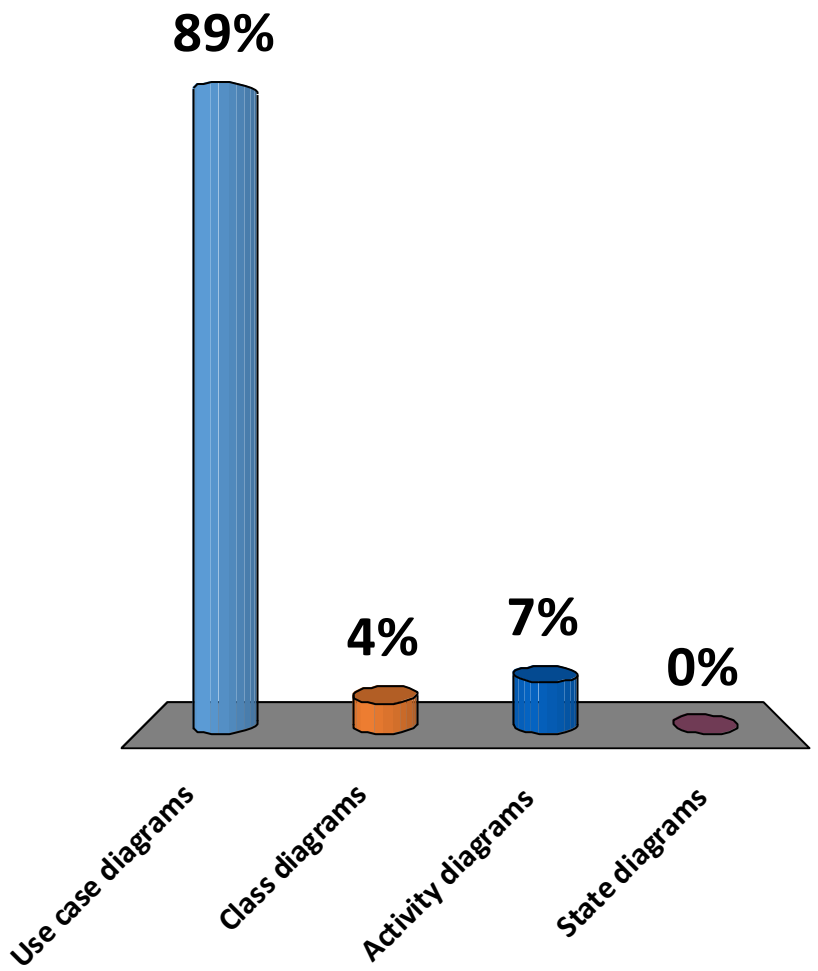
Which diagram was the most difficult?

A. Use case
B. Activity
C. Class
D. Sequence
E. Communication
F. State
G. All of them were equally easy
H. All of them were equally difficult



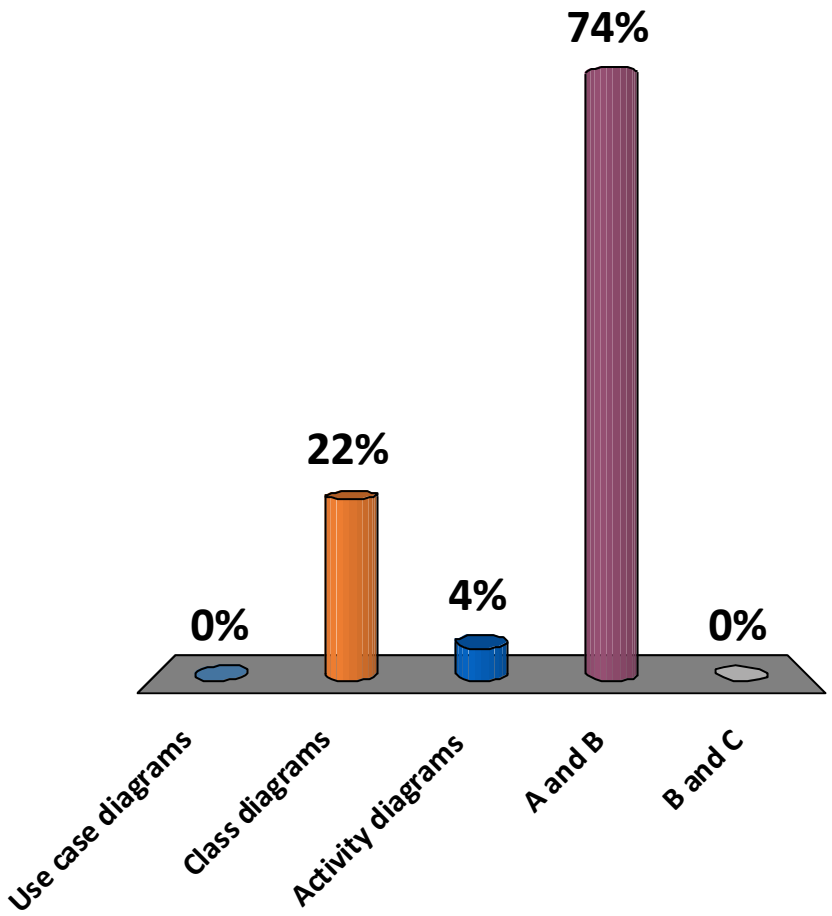
Includes and extends can be used in…

A. Use case diagrams
B. Class diagrams
C. Activity diagrams
D. State diagrams



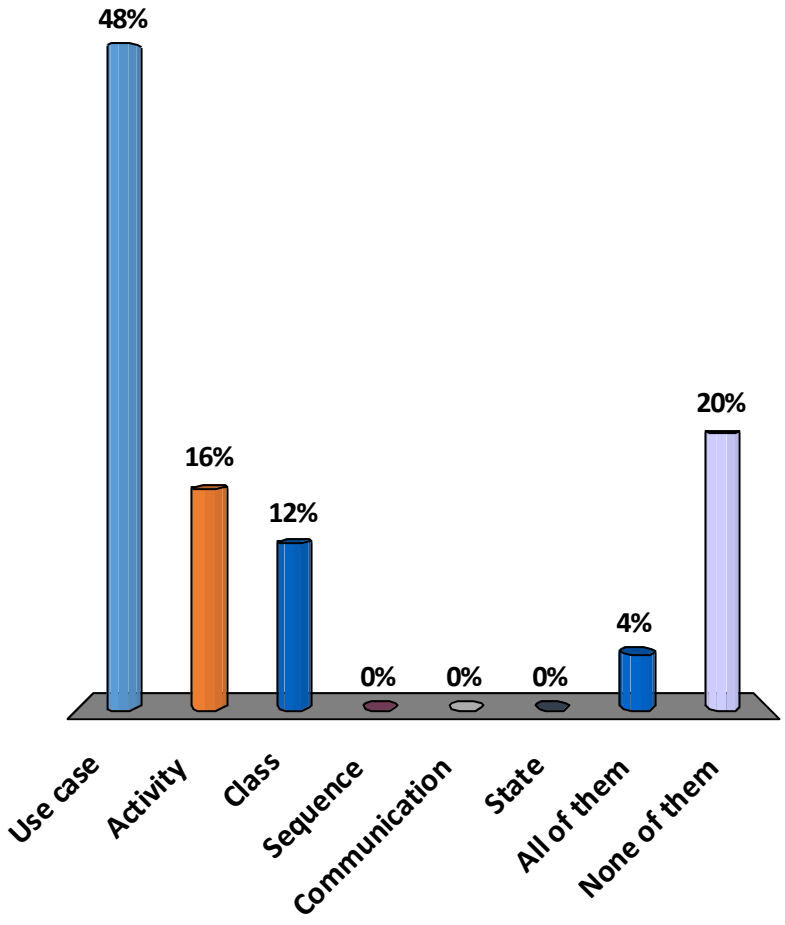
Inheritance can be used in…

A. Use case diagrams
B. Class diagrams
C. Activity diagrams
D. A and B
E. B and C



Which diagram do you enjoy creating?

A. Use case
B. Activity
C. Class
D. Sequence
E. Communication
F. State
G. All of them
H. None of them

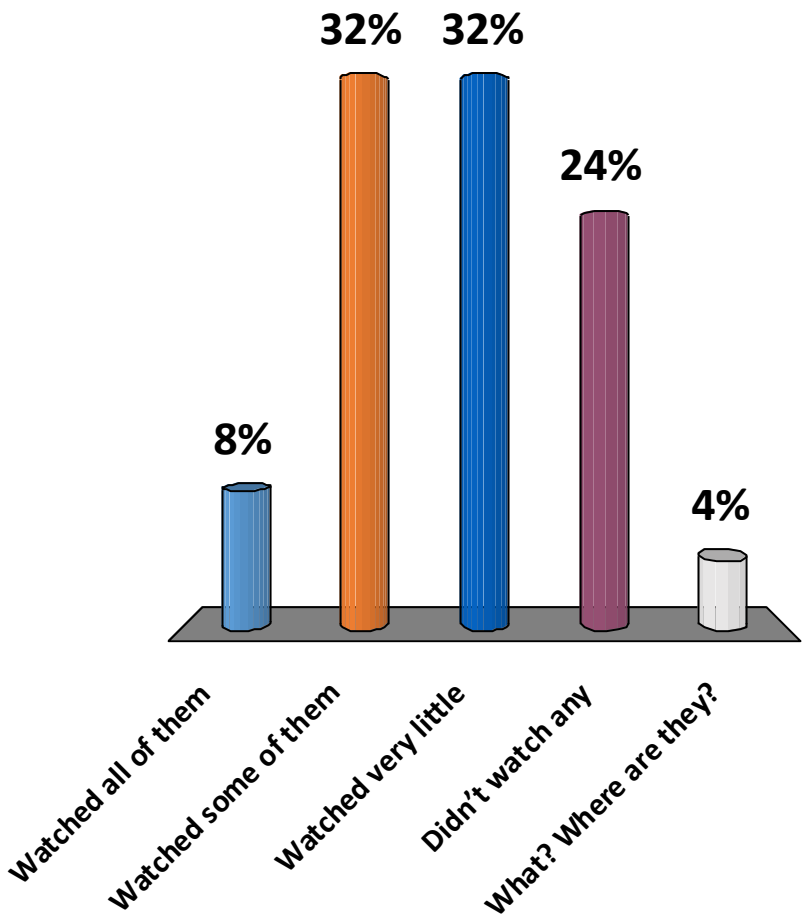


Summary

- What is the purpose of state diagrams?
- Develop state diagrams for specific classes in a given case.
- What are concurrent and nested state diagrams?
- What are the relationships between the diagrams that we have seen so far?
  - For example, how is a transition in a state chart related to a message in a sequence diagram?

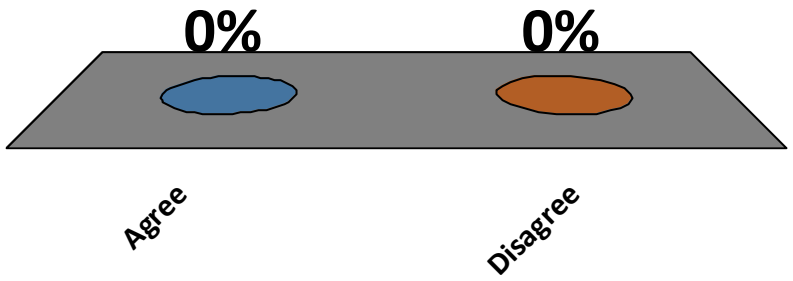
IBM RSA tutorials?

A. Watched all of them
B. Watched some of them
C. Watched very little
D. Didn’t watch any
E. What? Where are they?



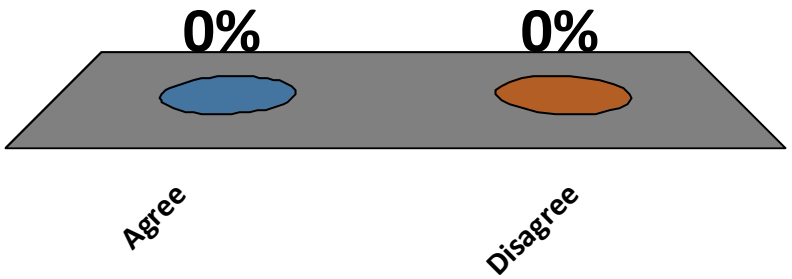
Don’t have to create both activity and sequence diagram

A. Agree
B. Disagree



Multi-tier architecture (or Model-View-Controller architecture) is better than single tier architecture. Do you agree with this assertion?

A. Agree
B. Disagree

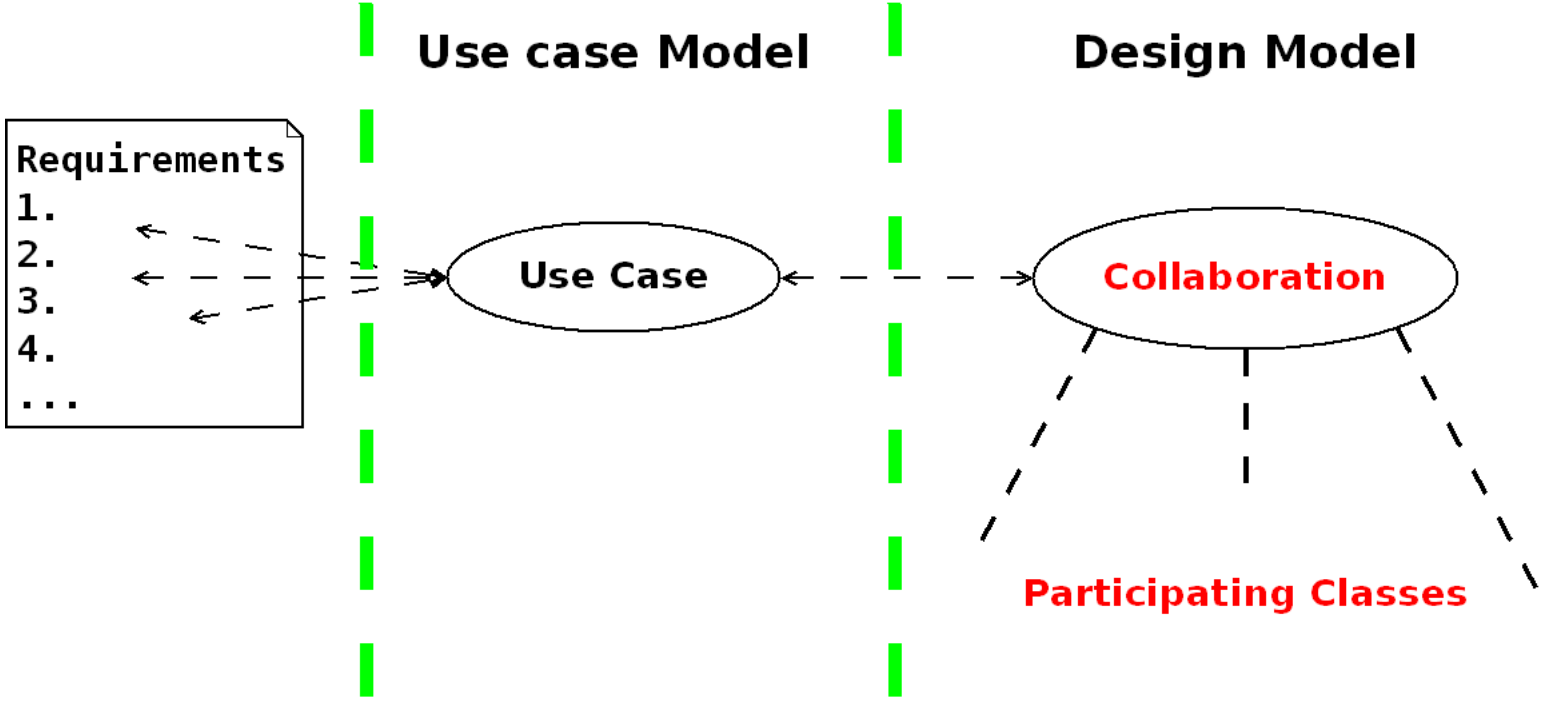


Communication Diagrams

Massimo Felici

School of informatics

Realizing Use cases in the Design Model



Slide 1: Realizing Use cases in the Design Model

- Use-case driven design is a key theme in a variety of software processes based on the UML
- UML supports specific modelling constructs that realize use cases in the implementation
- Collaborations (Communications) enhance the systematic and aggregate behavioural aspects of the system
- Collaborations support traceability from requirements expressed in use cases into the design

Communication Diagrams

- Model collaborations between objects or roles that deliver the functionalities of use cases and operations
- Model mechanisms within the architectural design of the system
- Capture interactions that show the passed messages between objects and roles within the collaboration
- Model alternative scenarios within use cases or operations that involve the collaboration of different objects and interactions
- Support the identification of objects (hence classes) that participate in use cases

Communication Diagrams

- The communication is implicit in a Sequence Diagram, rather than explicitly represented as in a Communication Diagram

- There is some redundancy between Communication and Sequence Diagrams
  - They differently show how elements interact over time
  - They document in detail how classes realize user cases
  - Communication Diagrams show relationship between objects
  - Sequence Diagrams focus on the time in which events occur

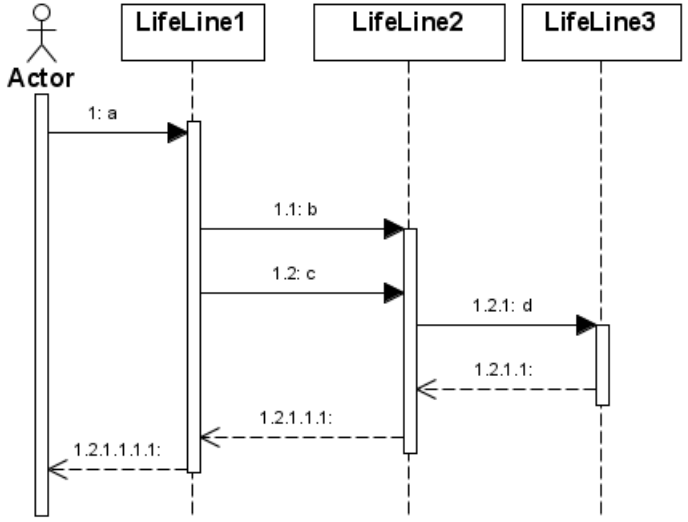
Slide 3: Communication Diagrams

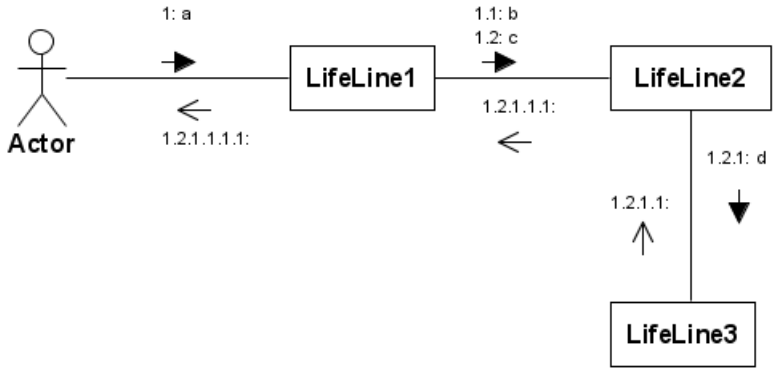
- Communication Diagrams, formerly called Collaboration Diagrams.

- UML Interaction Diagrams refine the kind of activity undertaken in checking with CRC cards.

School of informatics

Sequence and Communication Diagrams





Communication Diagrams

- A Collaboration is a collection of named objects and actors with links connecting them
- A Collaboration defines a set of participants and relationships that are meaningful for a given set of purposes
- A Collaboration between objects working together provides emergent desirable functionalities in Object-Oriented systems
- Objects collaborate by communicating (passing messages) with one another in order to work together

Slide 5: Communication Diagrams

- Objects and actors collaborate in performing some task. Each object (responsibility) partially supports emergent functionalities.

- Objects are able to produce (usable) high-level functionalities by working together.

Collaborations

Actors

- Each Actor is named and has a role
- One actor will be the initiator of the use case

Objects

- Each object in the collaboration is named and has its class specified
- Not all classes need to appear
- There may be more than one object of a class

Links

- Links connect objects and actors and are instances of associations
- Each link corresponds to an association in the class diagram

Interactions

- Use Cases and Class Diagrams constrain interactions
- Associations and Links in a Collaboration Diagram show the paths along which messages can be sent from one instance to another
- A message is the specification of a stimulus
- A stimulus represents a specific instance of sending the message, with particular arguments

Communication Diagrams

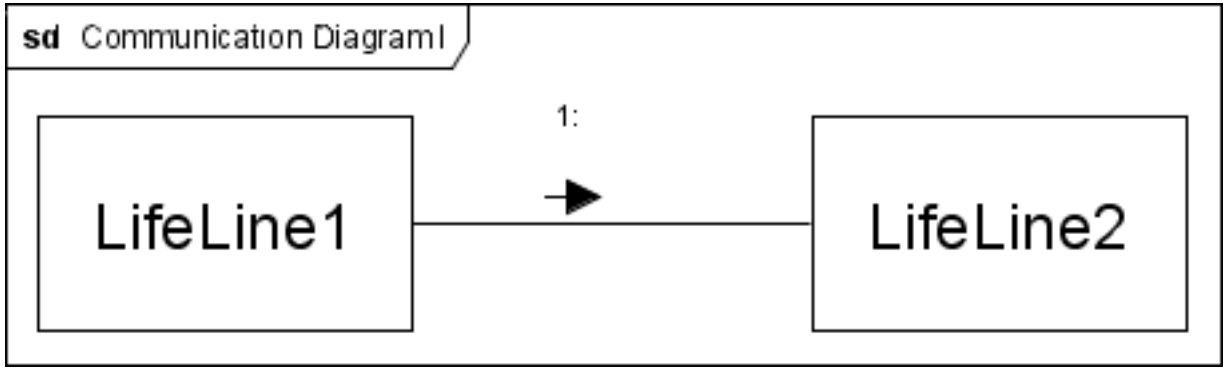
- Specification level shows generic cases of collaborations (communications)

  Generic form captures a collaboration among class roles and association roles and their interactions

- Instance level shows a specific instance of an interaction taking place and involving specific object instances

  Instance form captures a scenario among objects conforming to class roles and links conforming to association roles

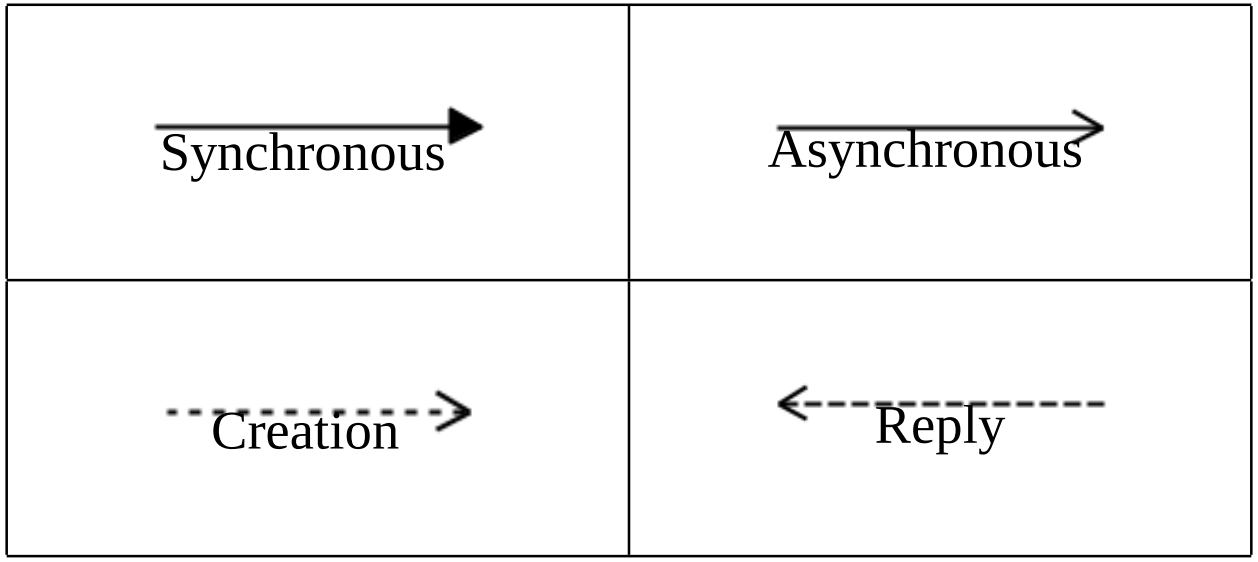
Communication Diagrams



Messages

- A message on a communication diagram is shown using an arrow from the message sender to the message receiver
- Message Signature: return-value, message-name, argument-list
- Each message in a collaboration diagram has a sequence number. The top-level message is numbered 1. Messages sent during the same call have the same decimal prefix but suffixes of 1, 2, etc. according to when they occur.

Messages



Synchronous Messages

Synchronous messaging involves a client that waits for the server to respond to a message. Messages are able to flow in both directions, to and from. Essentially it means that synchronous messaging is a two way communication. i.e. Sender sends a message to receiver and receiver receives this message and gives reply to the sender. Sender will not send another message until get reply from receiver.

Asynchronous Messages

Asynchronous messaging involves a client that does not wait for a message from the server. An event is used to trigger a message from a server. So even if the client is down , the messaging will complete successfully. Asynchronous Messaging means that, it is a one way communication and the flow of communication is one way only.

Messages

- Procedural or Synchronous: A message is sent by one object to another and the first object waits until the resulting action has completed.
- Asynchronous: A message is sent by one object to another, but the first object does not wait until the resulting action has completed.
- Flat: Each arrow shows a progression from one step to the next in a sequence. Normally the message is asynchronous.
- Return: the explicit return of control from the object to which the message was sent.

Messages

- Messages occurring at the same time: Adding a number-and-letter notation to indicate that a message happens at the same time as another message
- Invoking a message multiple times: Looping constraint, e.g., `*[i=0..9]`
- Sending a message based on a condition: A guardian condition is made up of a logical boolean statement, e.g., `[condition=true]`
- When a participant sends a message to itself

Messages

- The message is directed from sender to receiver
- The receiver must understand the message
- The association must be navigable in that direction
- Law of Demeter
- Dealing with a message m an Object O can send messages to:
  - Itself
  - Objects sent as argument in the message m
  - Objects O creates in responding to m
  - Objects that are directly accessible from O, using attribute values

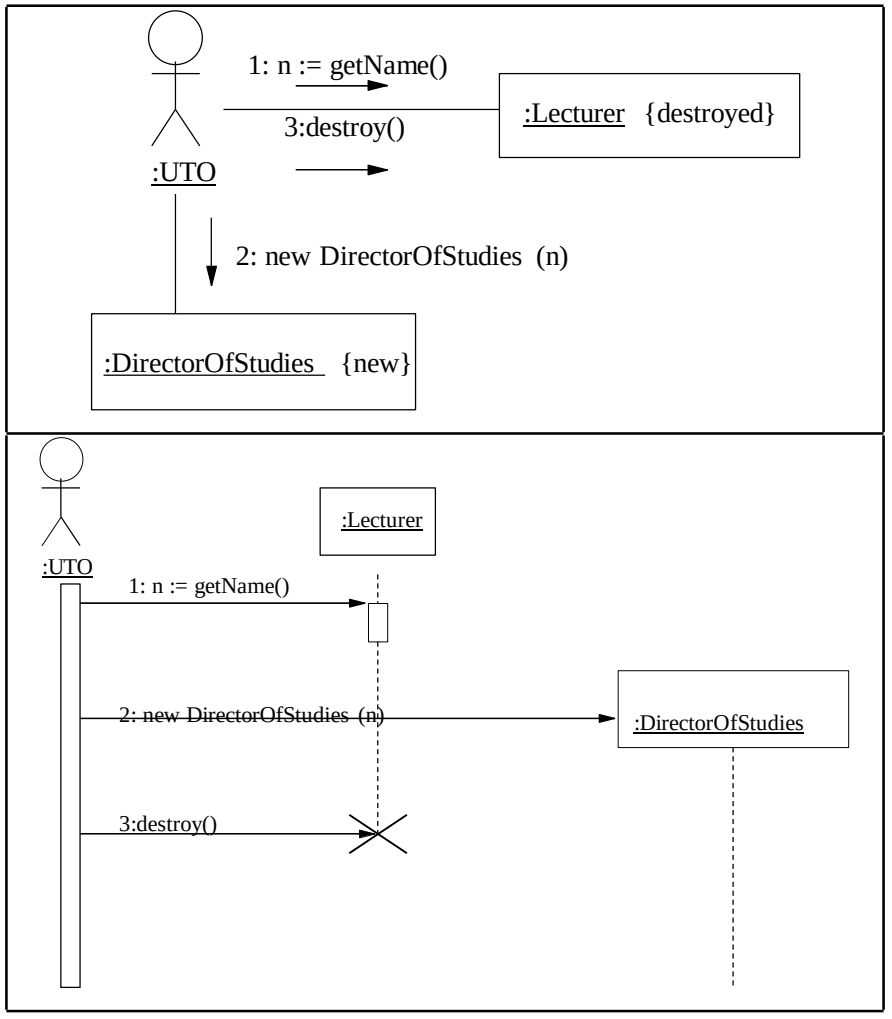
Flow of Control

- Procedural interactions
  At most one object is computing at any time
- Activation
  An object has a live activation from when it receives a message until it responds to the message
- Waiting for response
  Synchronous messages on sending a message to another object, an object will wait until it receives a response
- Activation task
  Activations are stacked and the top activation has control. When the top action responds the next to top regains control and so on...

Creation and Deletion

- In Sequence Diagrams, It is possible to use the lifelines
  - New objects have their icon inserted when they are created
  - Destroyed objects have their lifeline terminated with ×
- In Communication Diagrams the objects are labelled:
  - New for objects created in the collaboration
  - Destroyed for objects destroyed during the collaboration

Slide 17: Example



Communication vs. Sequence Diagrams

	Communication Diagrams	Sequence Diagrams
Participants	✦	✦
Links	✦	
Message Signature	✦	✦
Parallel Messages	✦	✦
Asynchronous messages		✦
Message Ordering		✦
Create & Maintain	✦	

Slide 18: Communication vs. Sequence Diagrams

- Shows participants effectively: Both Communication and Sequence diagrams show participants effectively
- Showing the links between participants: Communication diagrams explicitly and clearly show the links between participants
- Showing message signatures: Both Communication and Sequence diagrams show messages effectively
- Support parallel messages: Both Communication and Sequence diagrams show parallel messages effectively
- Support asynchronous messages: Sequence diagrams explicitly and clearly show the links between participants
- Easy to read message ordering: Sequence diagrams explicitly and clearly show message ordering
- Easy to create and maintain: Communication diagrams do have the edge on the ease-of-maintenance

Constructing Communication Diagrams

1. Identify behaviour
2. Identify the structural elements
3. Model structural relationships
4. Consider the alternative scenarios

Slide 19: Constructing Communication Diagrams

1. Identify behaviour whose realization and implementation is specified

2. Identify the structural elements (class roles, objects, subsystems) necessary to carry out the functionality of the collaboration; Decide on the context of interaction: system, subsystem, use case and operation

3. Model structural relationships between those elements to produce a diagram showing the context of the interaction

4. Consider the alternative scenarios that may be required; Draw instance level collaboration diagrams, if required; Optionally, draw a specification level collaboration diagram to summarise the alternative scenarios in the instance level sequence diagrams

Readings

Required Readings

- UML course textbook, Chapter 10 on More on Interaction Diagrams

Suggested Readings

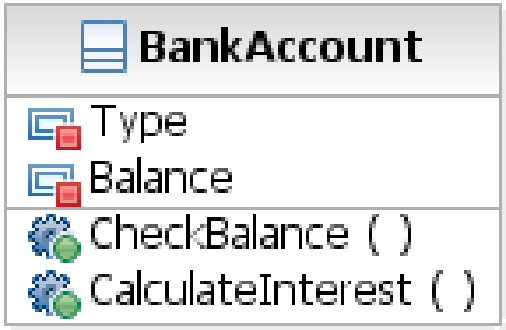
- K.J. Lieberherr, I.M. Holland. Assuring good style for object-oriented programs. IEEE Software 6(5):38-48, 1989.

Summary

- Interaction Diagrams
  - Sequence Diagrams
  - Communication Diagrams
- Communication Diagrams Rationale
- Communication Diagrams
  - Collaborations
  - Interactions
  - Messages
- Constructing Communication Diagrams

Structural Modeling: Class Diagrams

Classes, Attributes, & Operations



- Classes

  Templates for instances of people, places, or things

- Attributes

  Properties that describe the state of an instance of a class (an object)

- Operations

  Actions or functions that a class can perform

Rules of Thumb

- Limited responsibilities and collaboration
  - About 3 to 5 responsibilities
- No class stands alone
- Watch out for –
  - Too many small classes or too few large classes
  - Deep inheritance trees – 3 or more levels
  - *Functoids* – Too many classes with ‘DoIt’ methods – normal procedural function disguised as a class
  - *Omnipotent* classes – such as ‘system’ or ‘controller’

Finding Analysis Classes

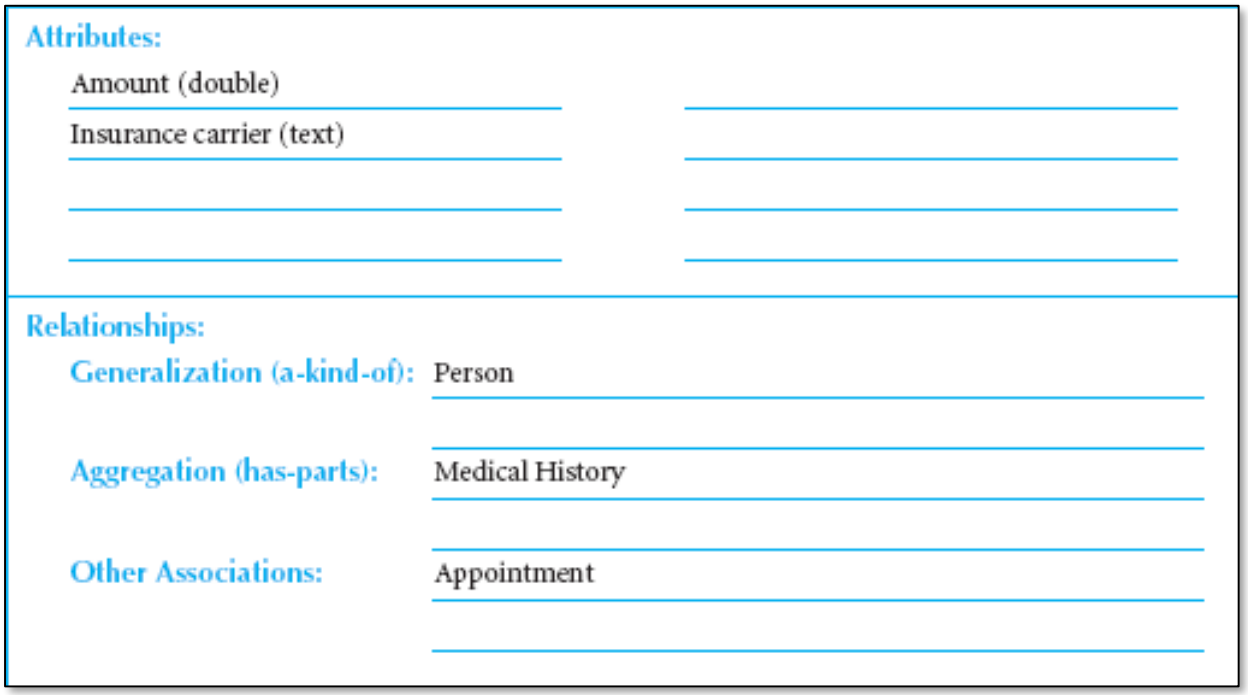
- Noun/verb analysis on documents
- CRC card analysis

Front-Side of a CRC Card

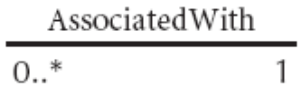
Class Name: Patient	ID: 3	Type: Concrete, Domain
Description: An individual that needs to receive or has received medical attention		Associated Use Cases: 2

Responsibilities	Collaborators
Make appointment	Appointment
Calculate last visit	
Change status	
Provide medical history	Medical history

Back-Side of a CRC Card



More Elements of Class Diagrams

An association: • Represents a relationship between multiple classes or a class and itself. • Is labeled using a verb phrase or a role name, whichever better represents the relationship. • Can exist between one or more classes. • Contains multiplicity symbols, which represent the minimum and maximum times a class instance can be associated with the related class instance.	
A generalization: • Represents a-kind-of relationship between multiple classes.	
An aggregation: • Represents a logical a-part-of relationship between multiple classes or a class and itself. • Is a special form of an association.	
A composition: • Represents a physical a-part-of relationship between multiple classes or a class and itself • Is a special form of an association.	

Association Example

Multiplicity

- How many instances of one class may relate to a single instance of an associated class at any point in time?
- 1..1, 1..n, 0..n, etc.

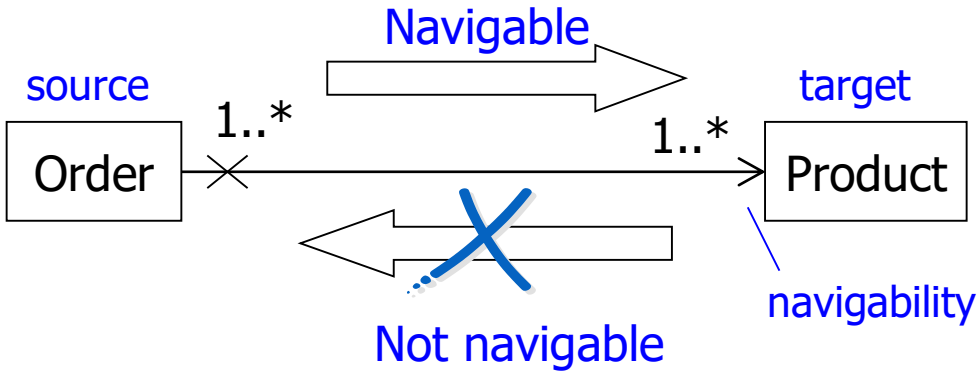
Multiplicity

- Company employs many persons
- A person is employed by one company
- Are these reasonable constraints? Depends on the system and requirements
- Documents business rules

Navigability

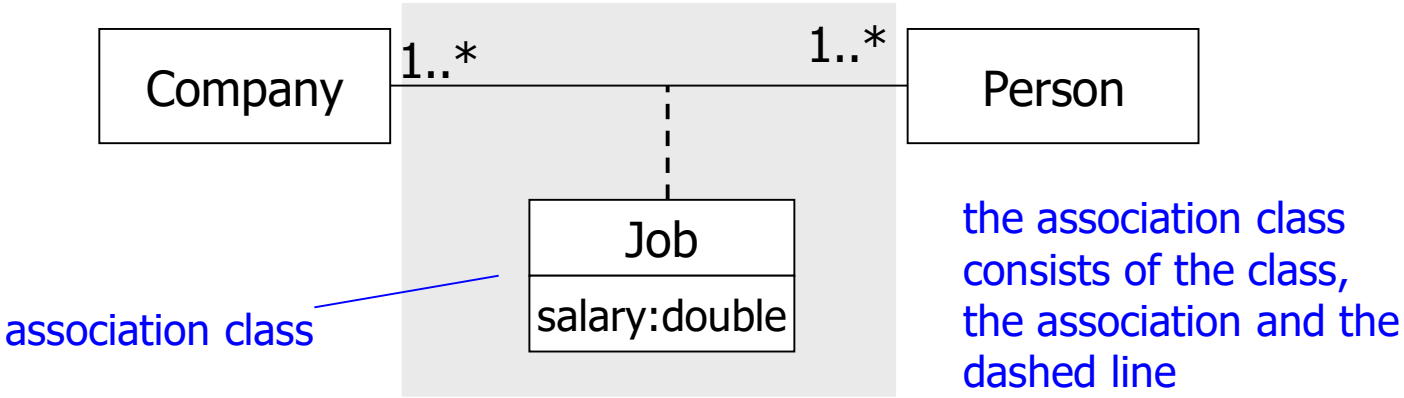
- Messages sent in the direction of the arrow
- Company sends messages to Person

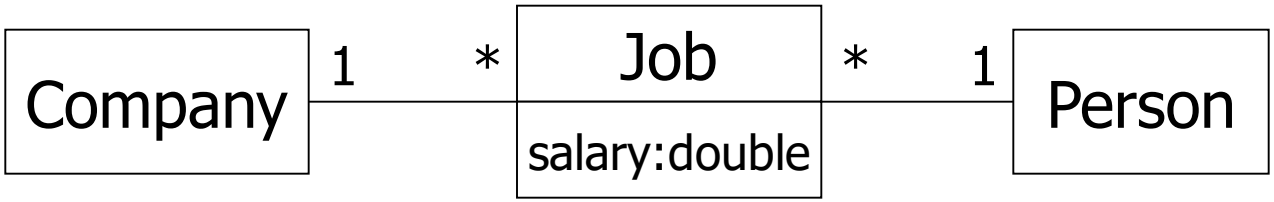
An Order object stores a list of Products



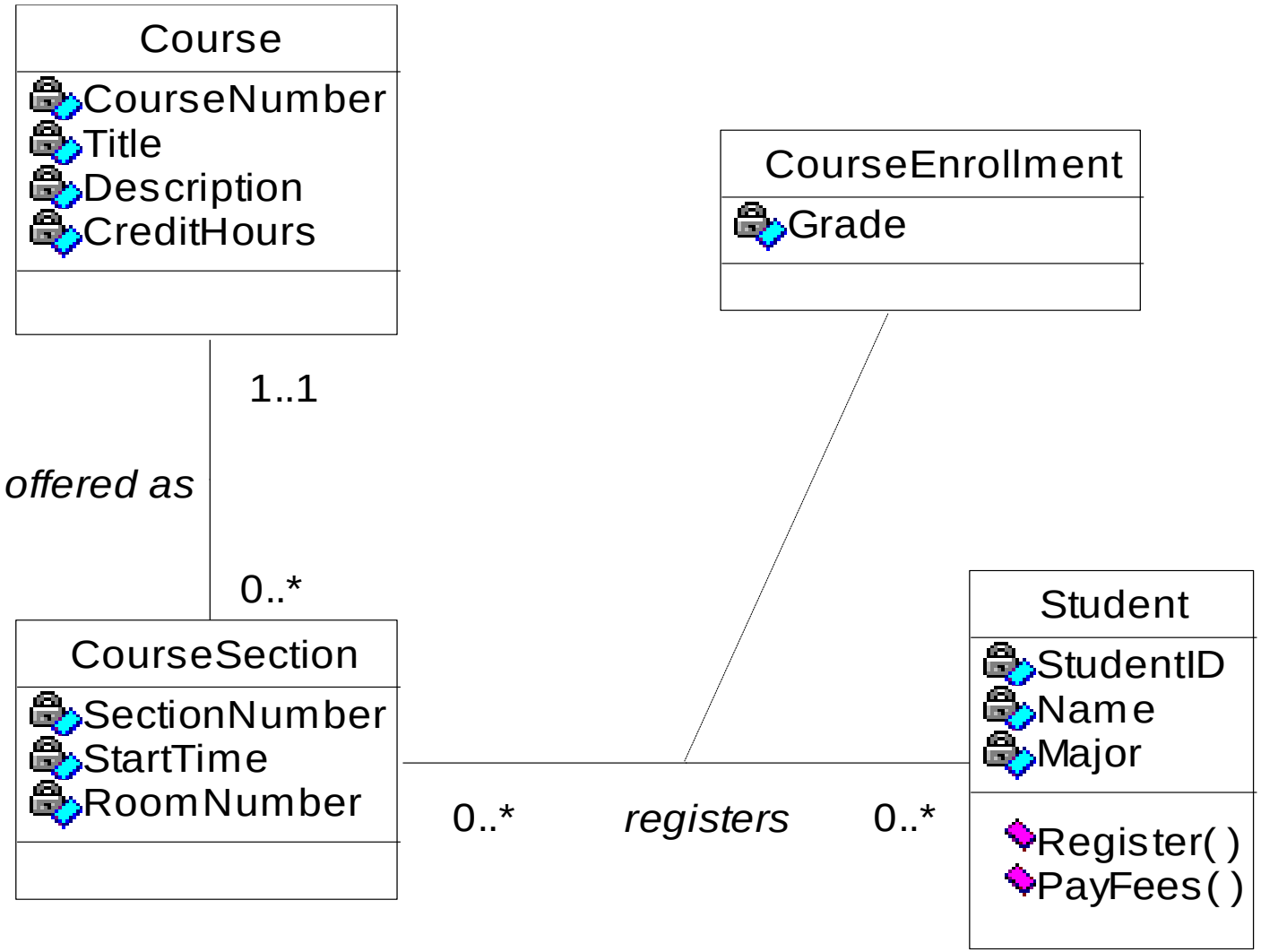
A Product object does not store a list of Orders

Association Class



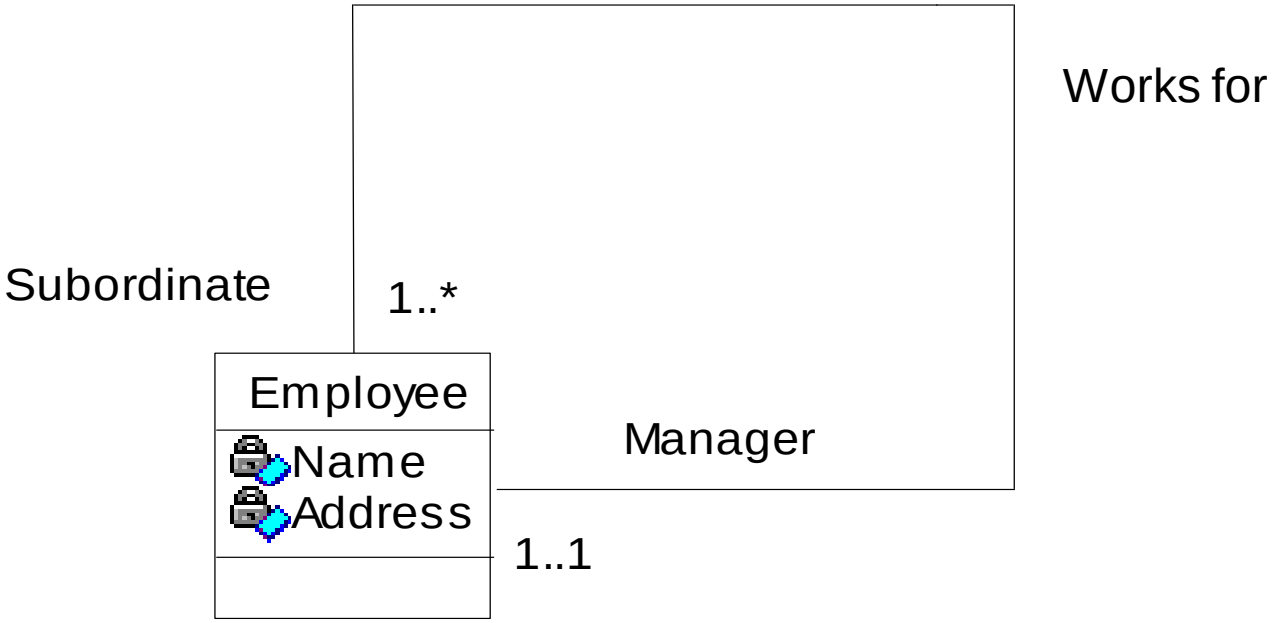


Association Class Example



Reflexive/Recursive Associations

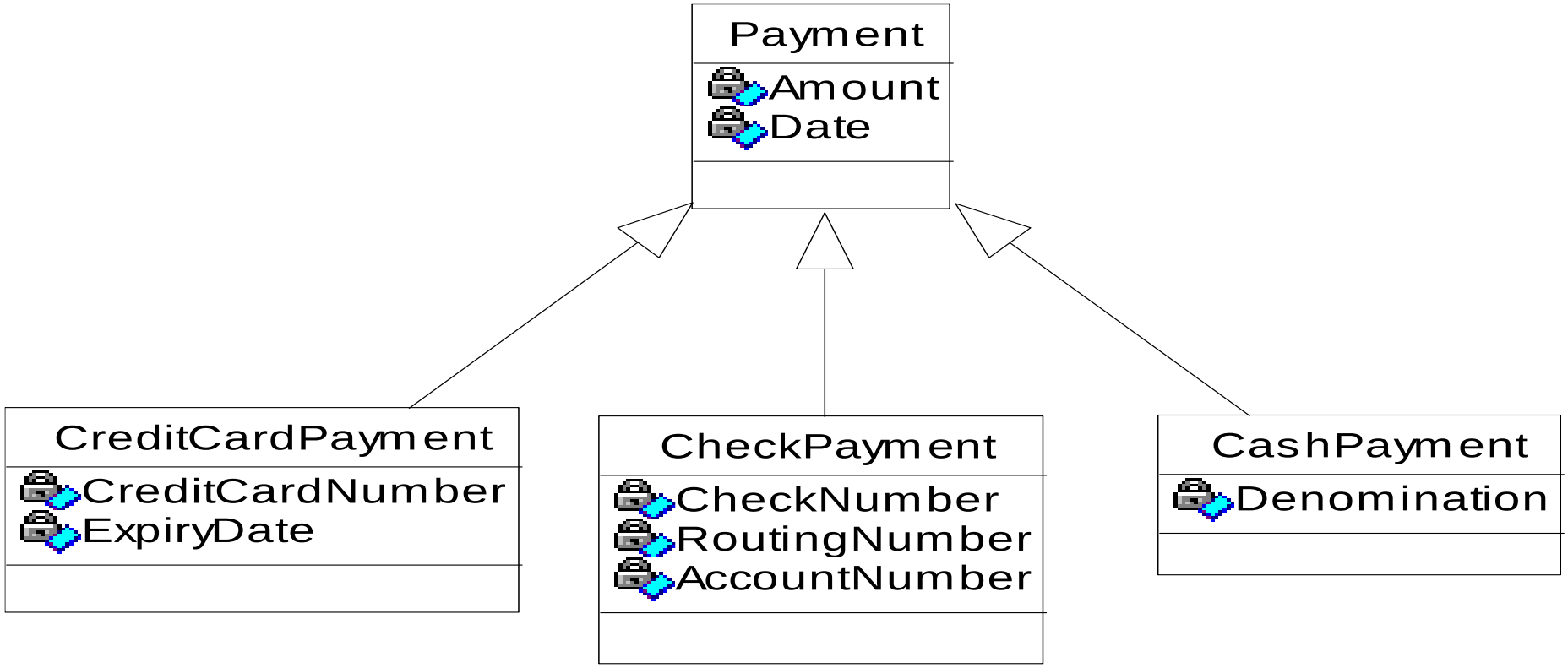
- A class is associated with itself
- Manager/Subordinate – “Role” names



Aggregation

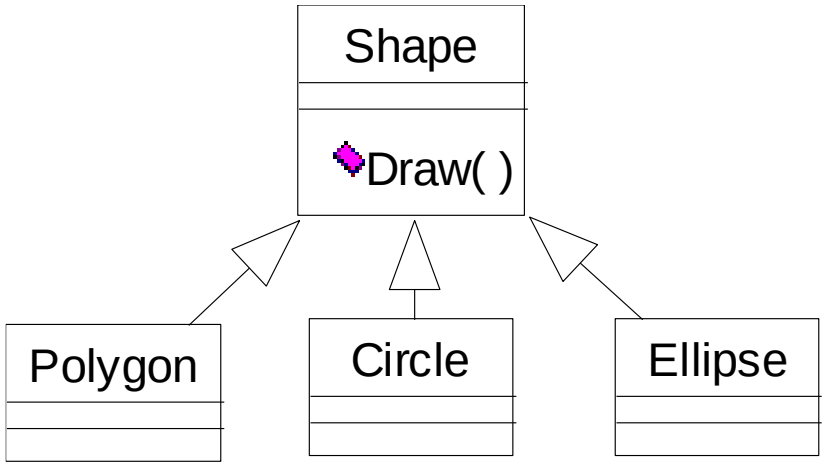
- Part-whole' relationship
  - Engine is 'a part of' car
  - Chassis is 'a part of' car
- A special form of association
- Composition
  - Parts live inside the whole and are destroyed if the whole is destroyed
- Shared aggregation
  - Parts can become parts of different wholes (Team - persons)

Generalization / Specialization



Abstract Classes

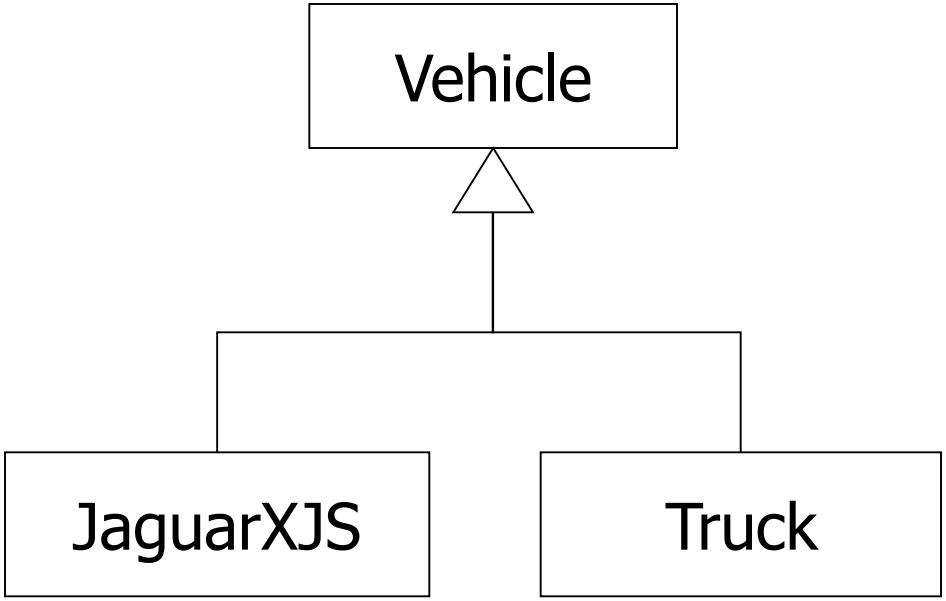
- Methods are not implemented, only signatures are specified
- They are implemented in specialized sub-classes
- Cannot create instances of abstract classes
- Those classes to which you can create instances of, are called as concrete classes



- Draw method does not have an implementation in the super class
- Why do we need this? Interface uniformity among sub-classes

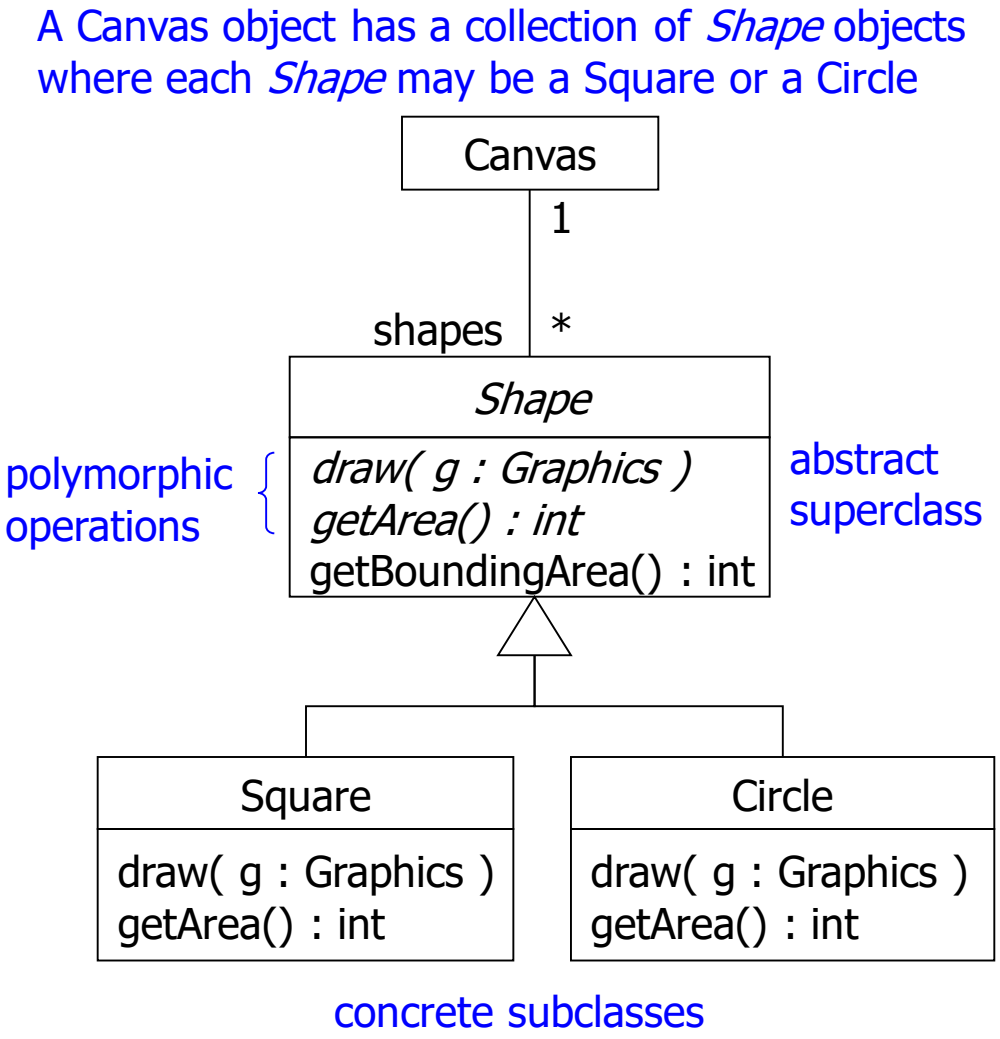
Levels of Abstraction

- What is wrong with this model?



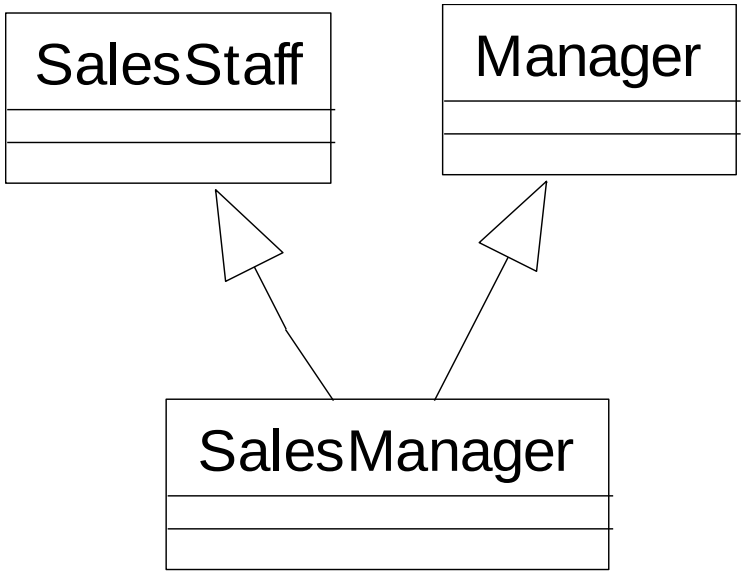
Polymorphism

- Many forms
- A polymorphic operation has many implementations
- What about overriding concrete operations?



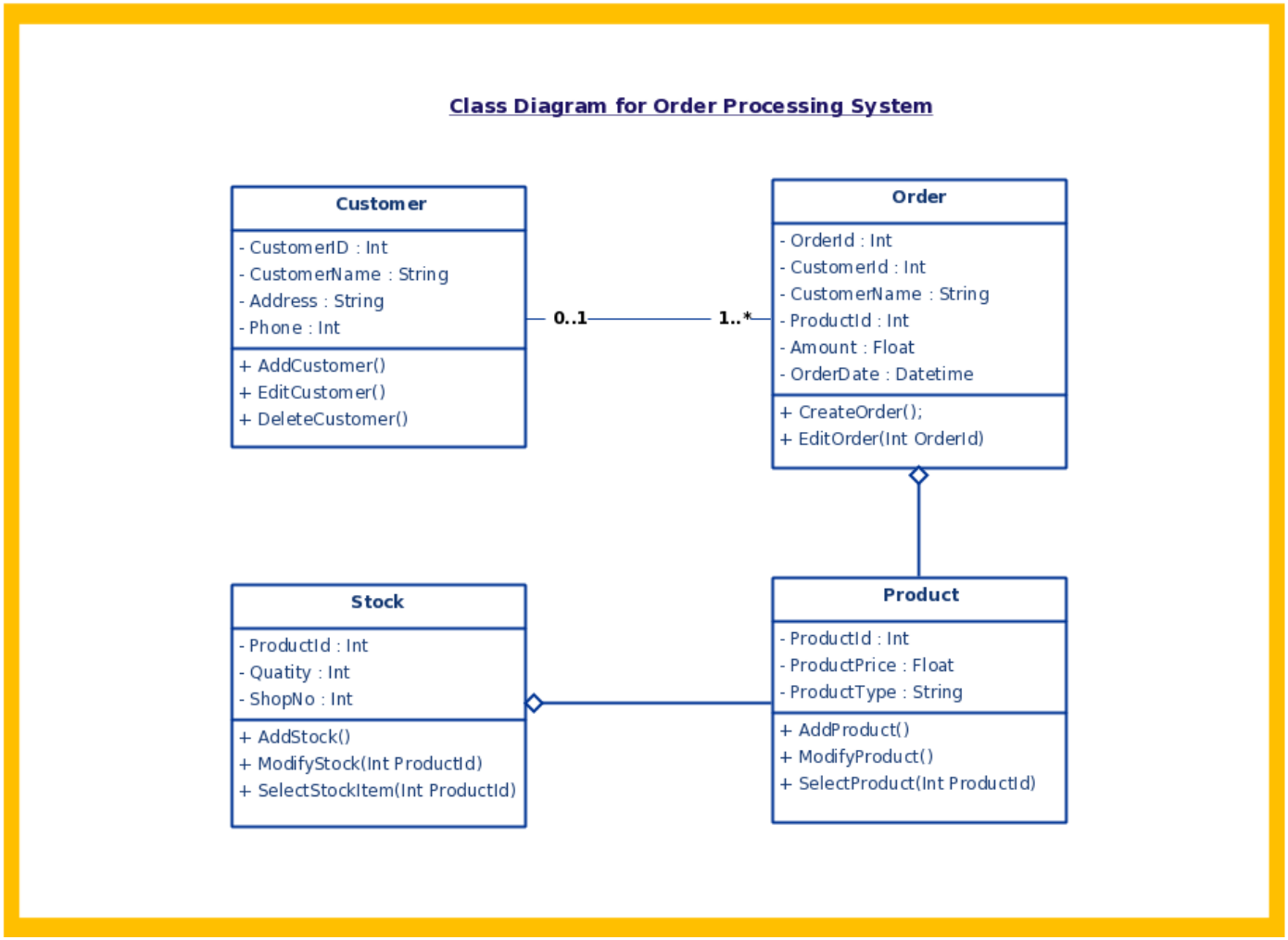
Multiple Inheritance

- In single inheritance we discriminated based on one attribute (say type of transaction – cash, credit and check)
- In Multiple inheritance we discriminate based on more than one attribute



- Sales Manager inherits properties and methods from both Sales staff and Manager

example1

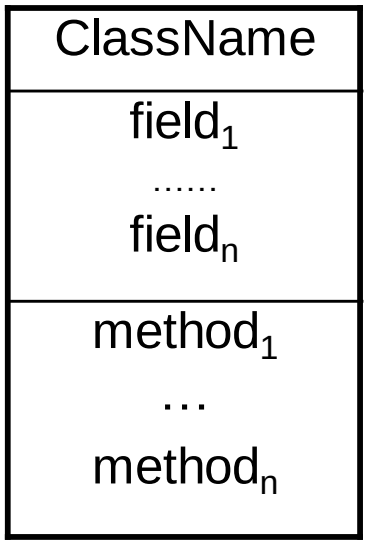

Class Diagram for Order Processing System
Customer
- CustomerID : Int
- CustomerName : String
- Address : String
- Phone : Int
+ AddCustomer()
+ EditCustomer()
+ DeleteCustomer()
0..1
1..*
Order
- OrderId : Int
- CustomerId : Int
- CustomerName : String
- ProductId : Int
- Amount : Float
- OrderDate : Datetime
+ CreateOrder();
+ EditOrder(Int OrderId)
Stock
- ProductId : Int
- Quatity : Int
- ShopNo : Int
+ AddStock()
+ ModifyStock(Int ProductId)
+ SelectStockItem(Int ProductId)
Product
- ProductId : Int
- ProductPrice : Float
- ProductType : String
+ AddProduct()
+ ModifyProduct()
+ SelectProduct(Int ProductId)

UML Class Diagram

- Most common diagram in OO modeling
- Describes the static structure of a system
- Consist of:
  - Nodes representing classes
  - Links representing of relationships among classes
    - Inheritance
    - Association, including aggregation and composition
    - Dependency

Notation for Classes

- The UML notation for classes is a rectangular box with as many as three compartments.

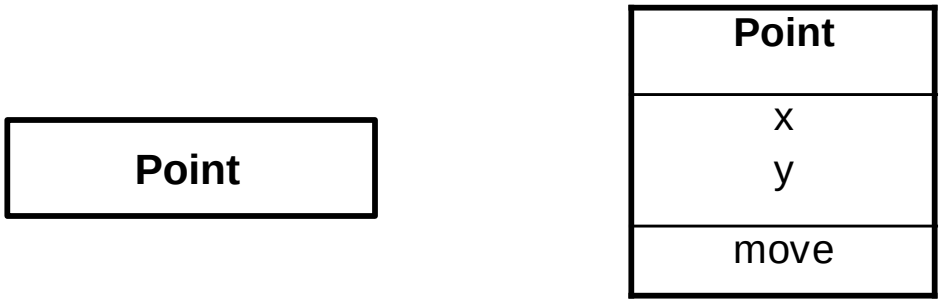


The top compartment show the class name.

The middle compartment contains the declarations of the fields, or *attributes,* of the class.

The bottom compartment contains the declarations of the methods of the class.

Example



Field and Method Declarations in UML

- Field declarations
  - birthday: Date
  - +duration: int = 100
  - -students[1..MAX_SIZE]: Student
- Method declarations
  - +move(dx: int, dy: int): void
  - +getSize(): int

Visibility	Notation
public	+
protected	#
package	~
private	-

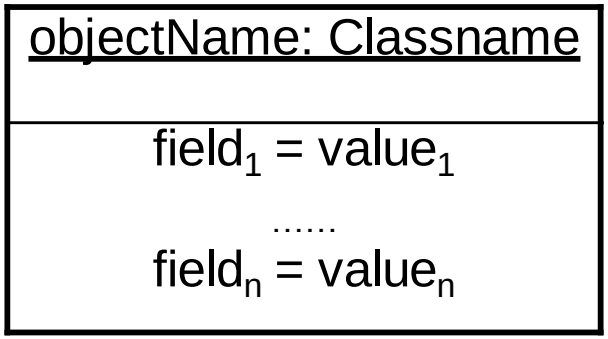
Exercise

- Draw a UML class diagram for the following Java code.

```
class Person {
  private String name;
  private Date birthday;
  public String getName() {
    // …
  }
  public Date getBirthday() {
    // …
  }
}
```

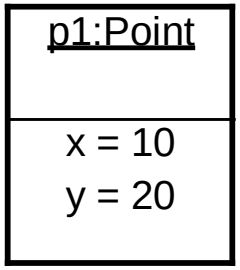
Notation for Objects

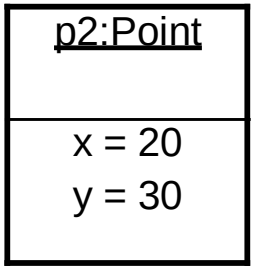
- Rectangular box with one or two compartments



The top compartment shows the name of the object and its class.

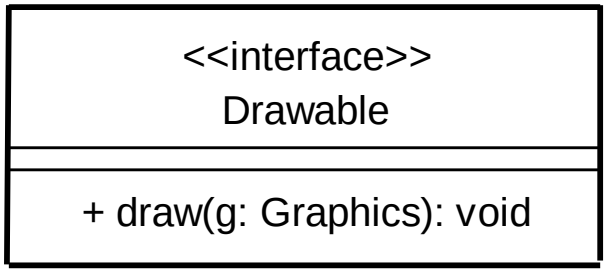
The bottom compartment contains a list of the fields and their values.

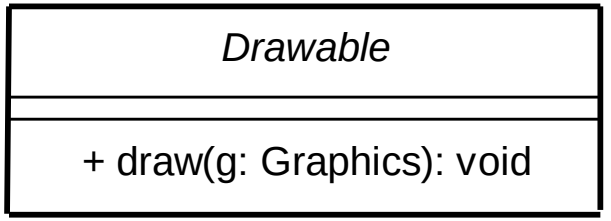




UML Notation for Interfaces

```
interface Drawable {
   void draw(Graphics g);
}
```



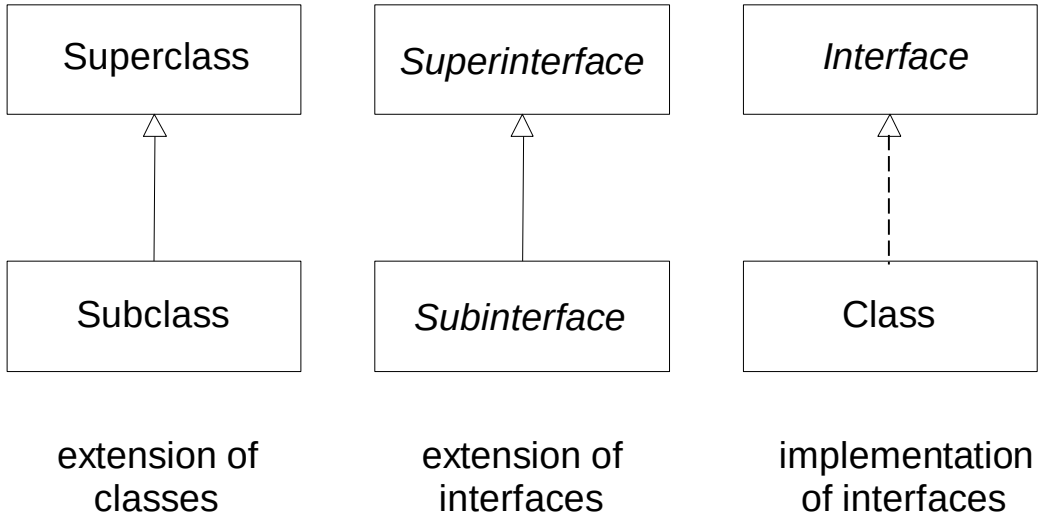


Inheritance in Java

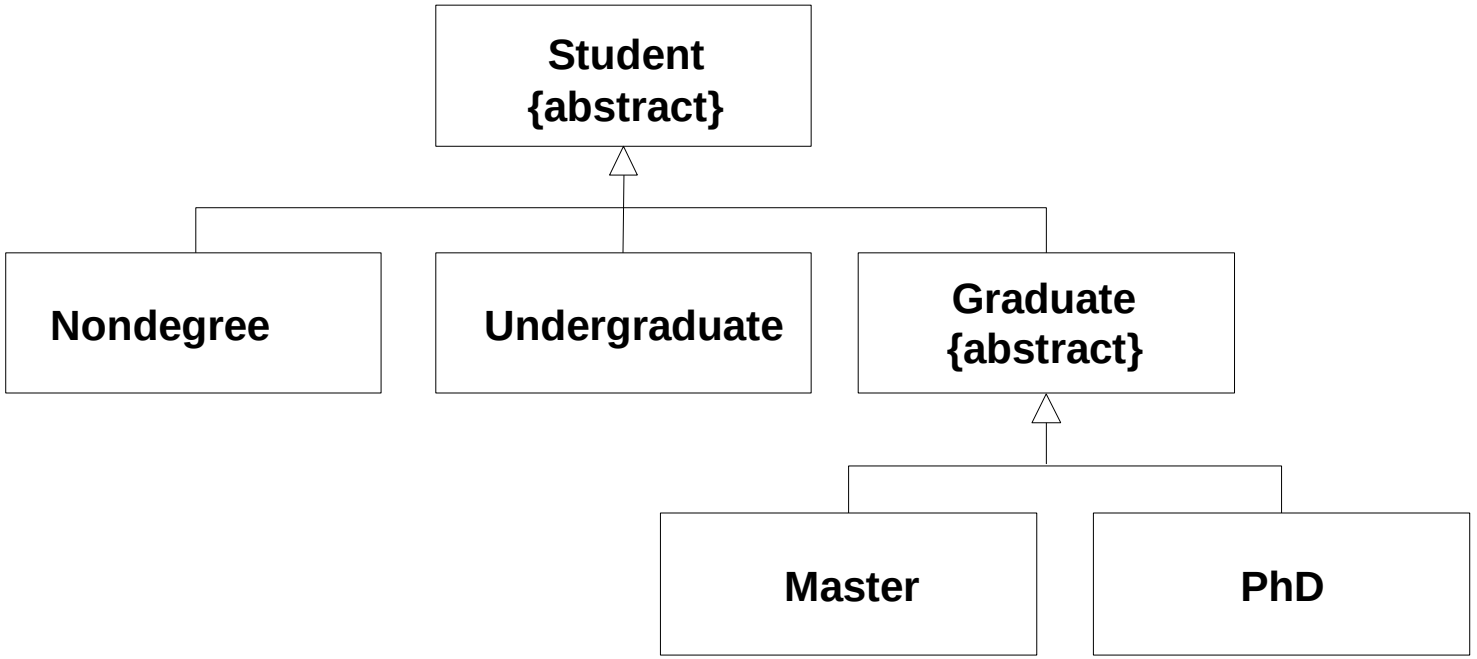
- Important relationship in OO modeling
- Defines a relationship among classes and interfaces.
- Three kinds of inheritances
  - *extension* relation between two classes (*subclasses* and *superclasses*)
  - *extension* relation between two interfaces (*subinterfaces* and *superinterfaces*)
  - relation between a class and an interface *implementation*

Inheritance in UML

- An extension relation is called *specialization* and *generalization*.
- An implementation relation is called *realization*.



Example

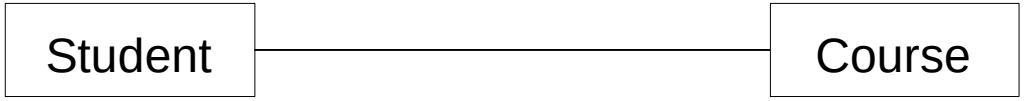

Student
{abstract}
Nondegree
Undergraduate
Graduate
{abstract}
Master
PhD

Exercise

- Draw a UML class diagram showing possible inheritance relationships among classes Person, Employee, and Manager

Association

- General binary relationships between classes.
- Commonly represented as direct or indirect references between classes



Association (Cont.)

- May have an optional label consisting of a name
- and a direction drawn as a solid arrowhead with no tail.
- The direction arrow indicates the direction of
- association with respect to the name.

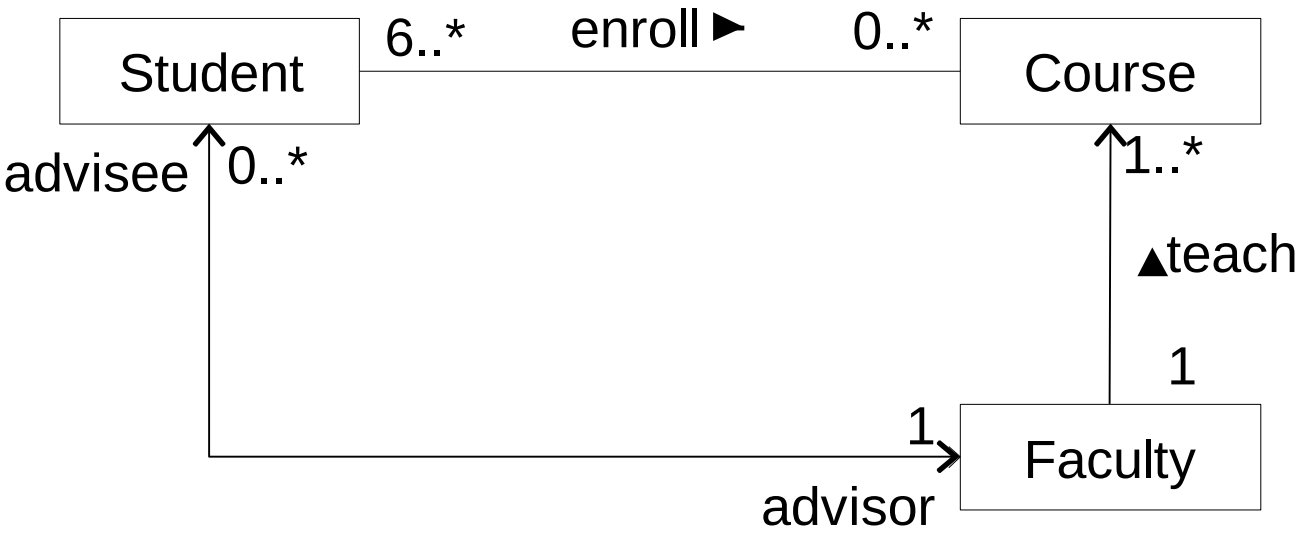
Association (Cont.)

- An arrow may be attached to the end of path to indicate that navigation is supported in that direction
- What if omitted?

Association (Cont.)

- May have an optional *role name* and an optional *multiplicity specification.*
- The multiplicity specifies an integer interval, e.g.,
  - *l..u* closed (inclusive) range of integers
  - i singleton range
  - 0..* entire nonnegative integer, i.e., 0, 1, 2, …

Example

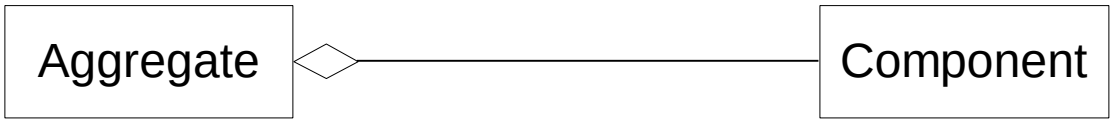

Student
6..*
enroll ►
0..*
Course
advisee
0..*
1..*
▲teach
1
1
Faculty
advisor

Exercise

- Identify possible relationships among the following classes and draw a class diagram
  - Employee
  - Manager
  - Department

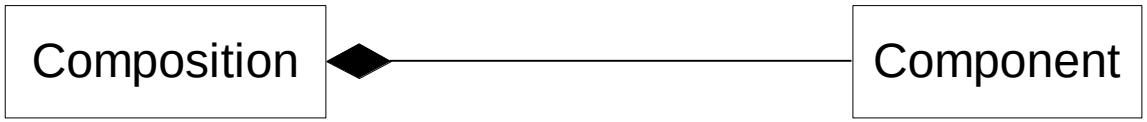
Aggregation

- Special form of association representing has-a or part-whole relationship.
- Distinguishes the whole (aggregate class) from its parts (component class).
- No relationship in the lifetime of the aggregate and the components (can exist separately).

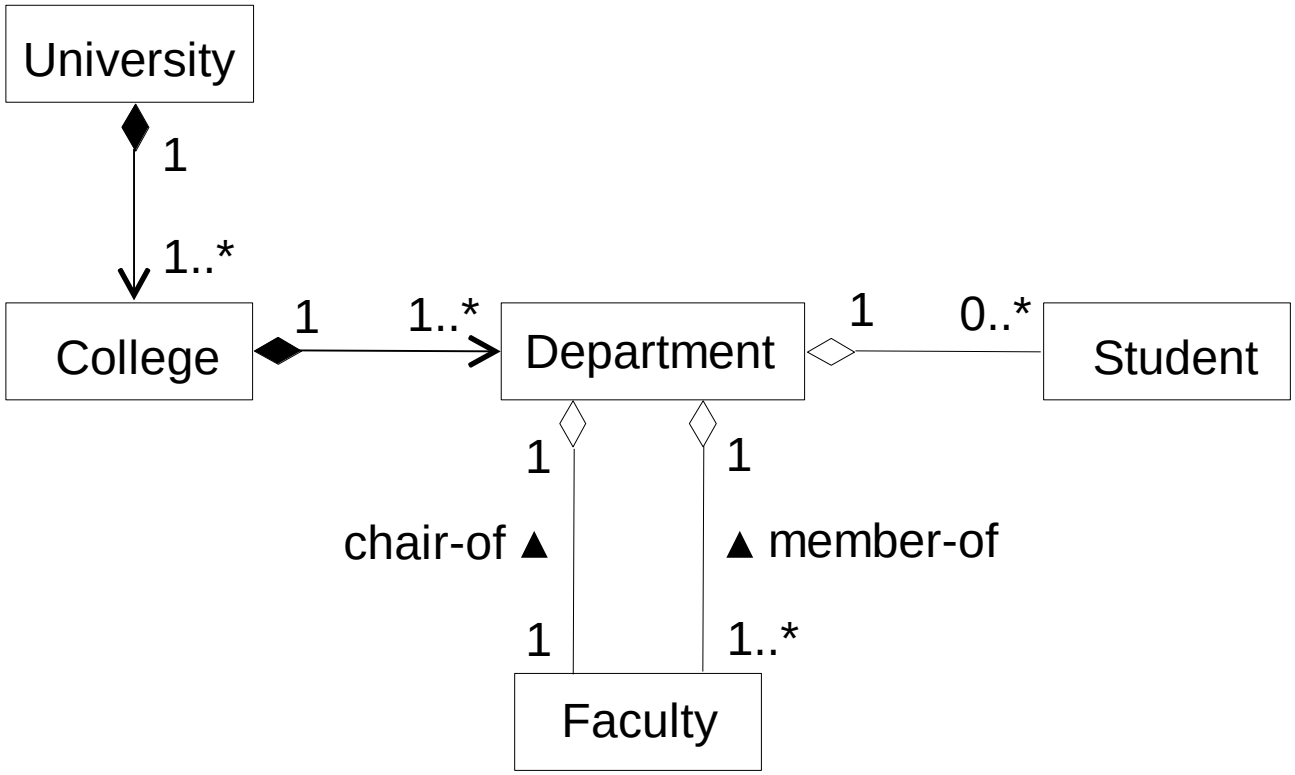


Composition

- Stronger form of aggregation.
- Implies exclusive ownership of the component class by the aggregate class.
- The lifetime of the components is entirely included in the lifetime of the aggregate (a component can not exist without its aggregate).

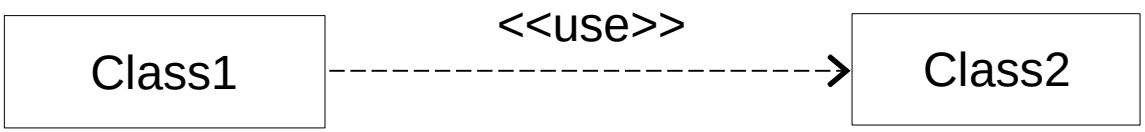


Example

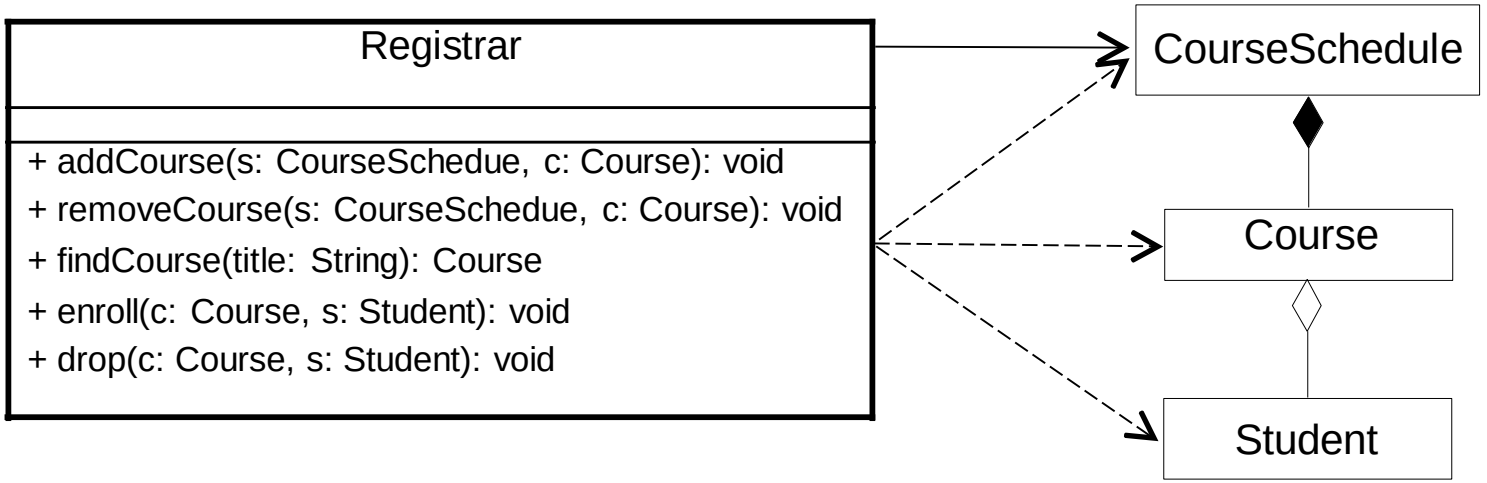

University
1
1..*
College
1
1..*
Department
1
0..*
Student
1
1
chair-of ▲
▲ member-of
1
1..*
Faculty

Dependency

- Stronger form of aggregation
- Implies exclusive ownership of the component class by the aggregate class
- The lifetime of the components is entirely included in the lifetime of the aggregate (a component can not exist without its aggregate).



Example



Dependencies are most often omitted from the diagram unless they convey some significant information.

Group Exercise: E-book Store

Develop an OO model for an e-bookstore. The core requirements of the e-bookstore are to allow its customers to browse and order books, music CDs, and computer software through the Internet. The main functionalities of the system are to provide information about the titles it carries to help customers make purchasing decisions; handle customer registration, order processing, and shipping; and support management of the system, such as adding, deleting, and updating titles and customer information.

1. Identify classes. Classes can represent physical objects, people, organizations places, events, or concepts. Class names should be noun phrases.
2. Identify relevant fields and methods of the classes. Actions are modeled as the methods of classes. Method names should be verb phrases.
3. Identify any inheritance relationships among the classes and draw the class diagram representing inheritance relationships.
4. Identify any association relationships among the classes and draw the class diagram representing association relationships.
5. Identify any aggregation and composition relationships among the classes and draw the class diagram representing dependency relationships.

Best Practices

- Favor object composition over class inheritance
  - White box reuse/inheritance vs. black box reuse/composition
  - Encapsulation and interfaces respected in composition
- Design to an interface, not to an implementation
  - Commit to an interface defined by the abstract class

(Gamma et al, 1994)

Comparing Traditional and OO: Traditional Viewpoint

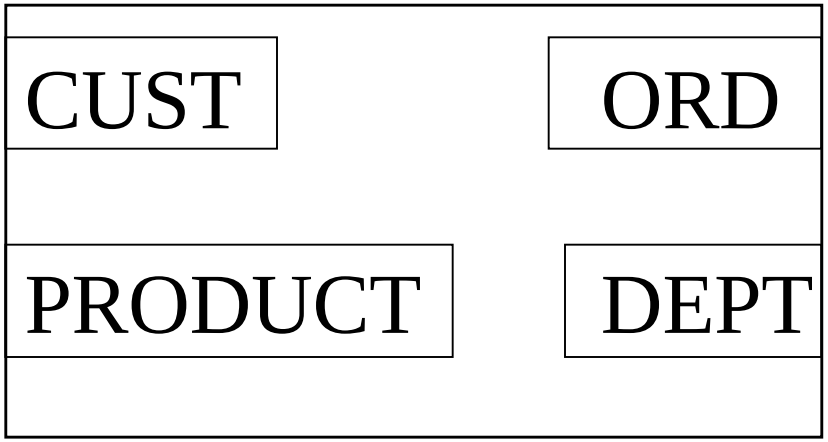
- Focus on procedures
- Functionality is vested in procedures (modules)
- Data and Procedures separated
- Data exist to be operated upon by procedures
- Procedures know about the structure of data
- Responsibility of what can be done to a piece of data is implicit

Traditional View – An Example

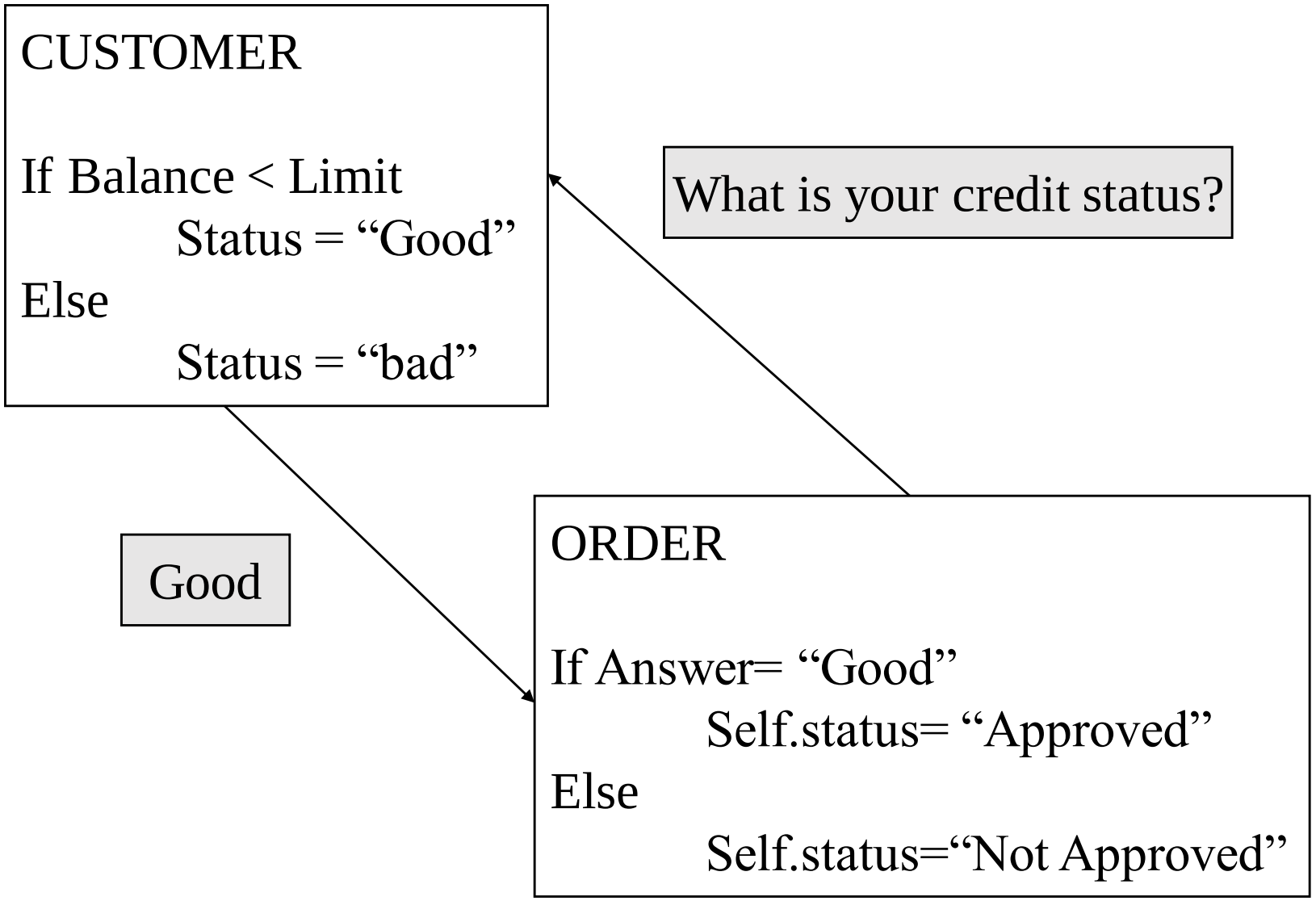
```
If CUST.credit > CUST.balance
        ORD.status = “Approved”
else
        ORD.status=“Not Approved”
```

Procedure

Data



Object-Oriented View – An Example



How many classes?

Summary

- How do we identify classes, relationships, attributes, and behavior for a given case?
- What are different types of relationships?
- What is multiplicity and how do we specify it?
- How do we create a domain class diagram using a CASE tool?
- Explain the following:
  - Class, object, message, method
- What is abstraction?
- What is encapsulation and why is it important?