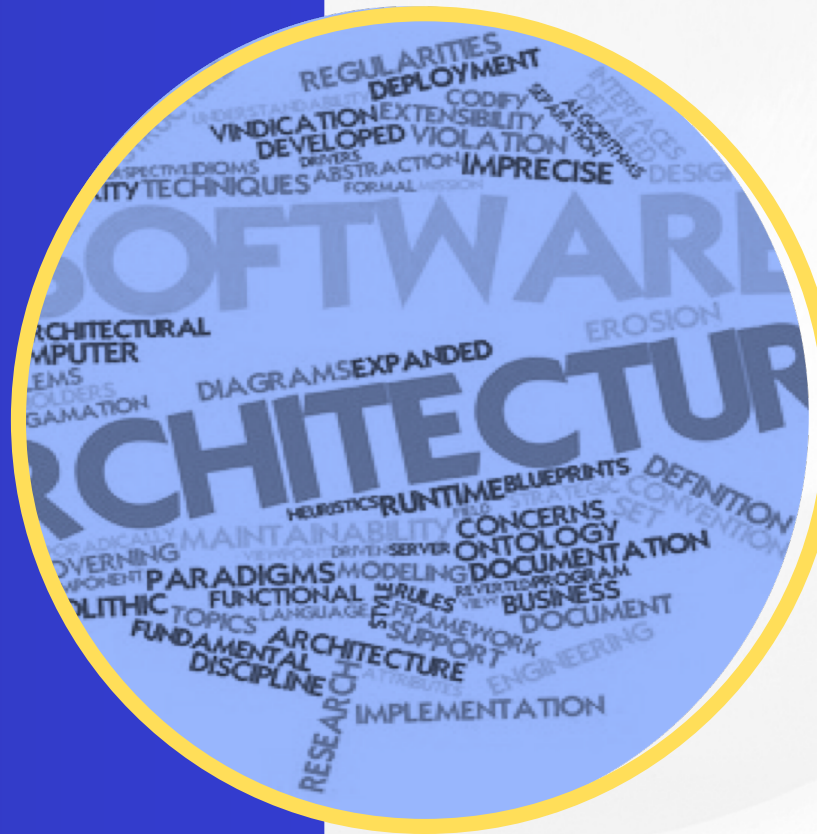
University of Tripoli – Faculty of Information Technology

Software Engineering Department

Software Architecture & Design

ITSE411



y Marwa Solla

Software architecture and design

for modern large-scale systems

Lecture 6:

Object-oriented design using the UML

What We Learn In This Lecture

- Introduction to Detailed design.
- Object–oriented design using the UML
- use case diagram

Software Design

Detailed design: the process of refining and expanding the preliminary design of a system or component to the extent that the design is sufficiently complete to be implemented .[ISO/IEC 24765]

Detailed Design

- Whereas the software architecture places a major emphasis on quality (nonfunctional requirements), the detailed design activity places a major focus on addressing functional requirements of the system.

- In object–oriented systems, the detailed design activity is where components are refined into one or more classes, interfaces are realized, relationships between classes are specified, class functions and variable names are created, design patterns are identified and applied.

Detailed Design

- Two major tasks of the detailed design activity are interface design and component design.
- Interface design refers to the design activity that deals with specification of interfaces between components in the design (Sommerville 2010).
- Component design refers to modeling the internal structure and behavior of components which includes the internal structure of both logical and physical components–identified during the software architecture phase.

Detailed Design

- Components are not limited to object–oriented systems; therefore, component designs can be realized in many ways. In object–oriented systems, the internal structure of components is typically modeled using UML through one or more diagrams, including class and sequence diagrams (Carlos E. Otero 2012).
- When modeling the internal structure of components, several design principles, heuristics, and patterns are used to create and evaluate component designs.

Unified Modeling Language (UML)

- Unified Modeling Language (UML) is a standardized visual language used in software engineering to represent, design, and communicate the structure and behavior of software systems. It provides a common framework for software developers, architects, and stakeholders to create and understand visual models of software systems
- UML supports both structural modeling, which focuses on the static structure of the system (e.g., classes, objects, relationships), and behavioral modeling, which captures the dynamic aspects of the system

Unified Modeling Language (UML)

- Unified Modeling Language (UML) diagrams are :

STRUCTURAL

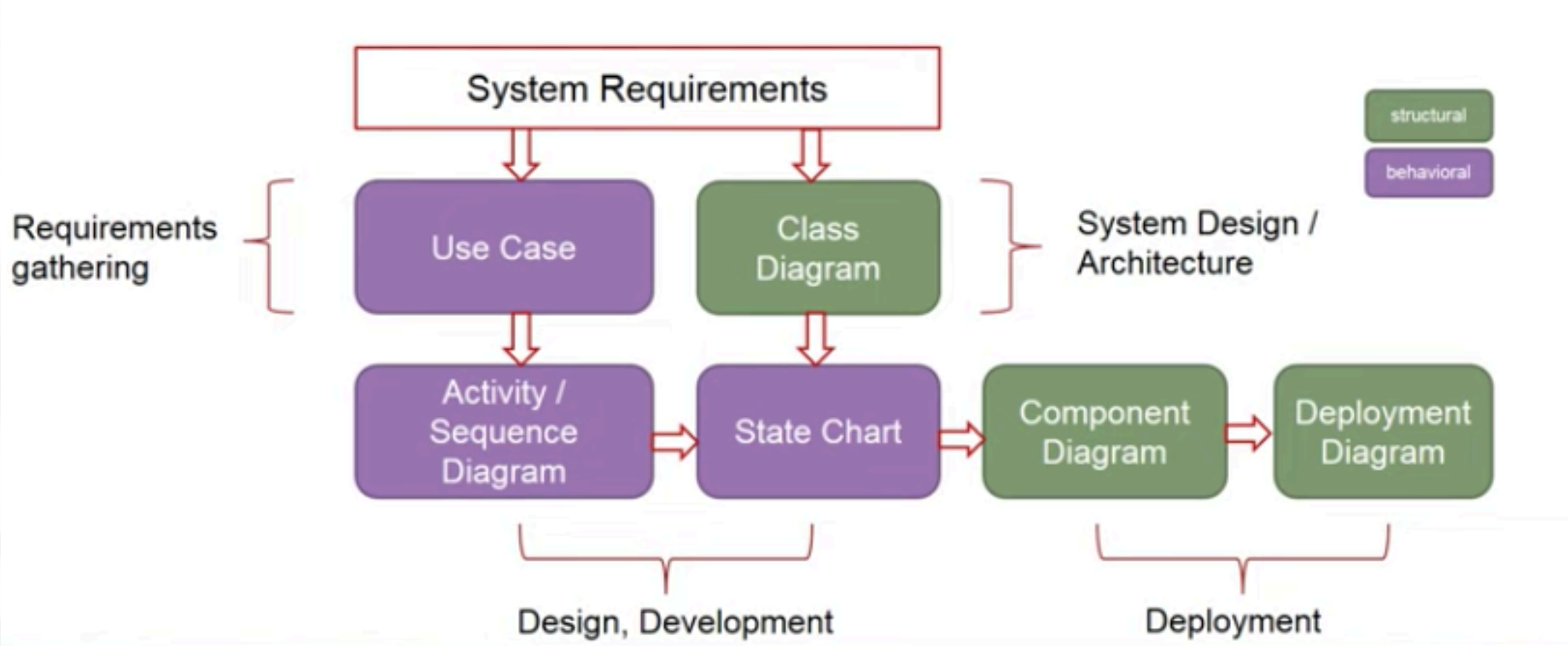
- CLASS DIAGRAM
- OBJECT DIAGRAM
- COMPONENT DIAGRAM
- DEPLOYMENT DIAGRAM

BEHAVIORAL

- USE CASE DIAGRAM
- SEQUENCE DIAGRAM
- COLLABORATION DIAGRAM
- STATE CHART DIAGRAM
- ACTIVITY DIAGRAM

Unified Modeling Language (UML)

- Unified Modeling Language (UML) diagrams are :



General Diagramming Guidelines

The guidelines presented are applicable to all types of diagrams, UML or otherwise.

❖ Readability Guidelines

1. Avoid Crossing Lines

When two lines cross on a diagram, such as two associations on a UML class diagram, the potential for misreading a diagram exists.

2. Depict Crossing Lines as a Jump

You can't always avoid crossing lines; for example, you cannot fully connect five symbols. When you need to have two lines cross, one of them should "hop" over the other as in Figure 1:

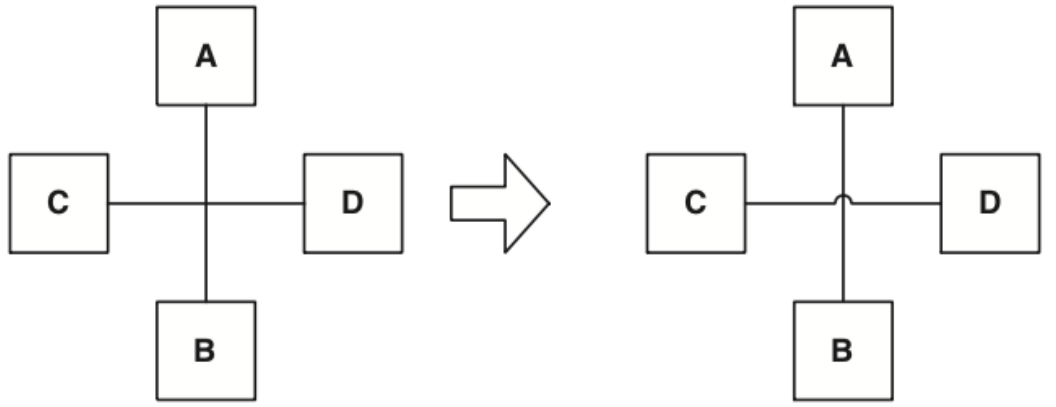


Figure 1

General Diagramming Guidelines

3. Avoid Diagonal or Curved Lines:

Straight lines, drawn either vertically or horizontally, are easier for your eyes to follow than diagonal or curved lines

4. Apply Consistently Sized Symbols

5. Align Labels Horizontally

In Figure 2 the two labels are easier to read in the second version of the diagram. Notice how Label 2 is horizontal even though the line it is associated with is vertical.

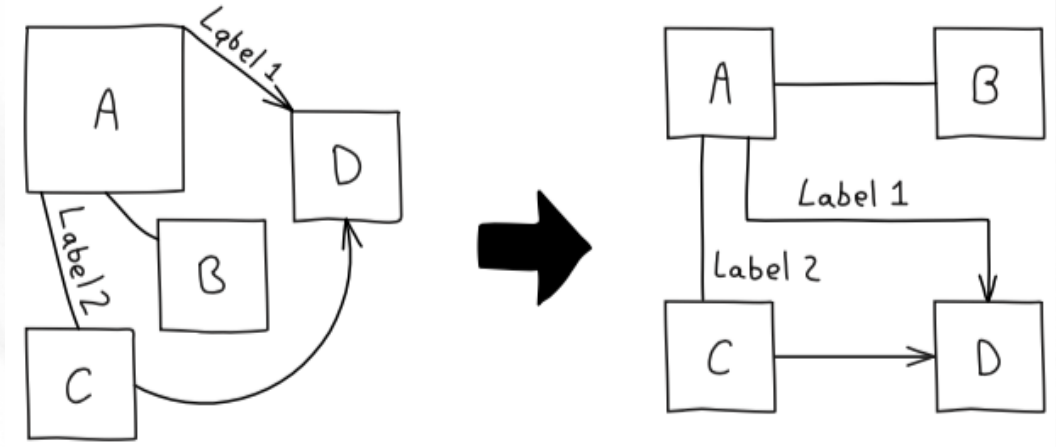


Figure 2

General Diagramming Guidelines

6. Organize Diagrams Left to Right, Top to Bottom

7. Avoid Many Close Lines:

Several lines close together are hard to follow.

❖Simplicity Guidelines

8. Reorganize Large Diagrams into Several Smaller Ones

A good rule of thumb is that a diagram shouldn't have more than nine symbols on it, based on the 7 ± 2 rule (Miller 1957),

❑ **Notes:**

•Symbols represent diagram elements such as class boxes, object boxes, use cases, and actors.
•Lines represent diagram elements such as associations, dependencies, and transitions between states.
•Labels represent diagram elements such as class names, as– sociation roles, and constraints.

General Diagramming Guidelines

9. Prefer Single-Page Diagrams

a diagram should be printable on a single sheet of paper to help reduce its scope as well as to prevent wasted time cutting and taping several pages together.

10. Focus on Content First, Appearance Second

11. Apply Consistent, Readable Fonts

Consistent, easy-to-read fonts improve the readability of your diagrams. Good ideas include fonts in the Courier, Arial, and Times families. Bad ideas include small fonts (less than 10 point), large fonts (greater than 18 point), and italics.

General Diagramming Guidelines

❖Naming Guidelines

12. Apply Common Domain Terminology in Names

Apply consistent and recognizable domain terminology, such as customer and order, whenever possible on your diagrams.

13. Name Common Elements Consistently Across Diagrams

A single modeling element, such as an actor or a class, will appear on several of your diagrams. For example, the same class will appear on several UML class diagrams, several UML sequence diagrams, several UML communication diagrams, and several UML activity diagrams. This class should have the same name on each diagram; otherwise your readers will become confused.

Guidelines for UML Notes

A UML note is a modeling construct for adding textual information—such as a comment, constraint definition, or method body—to UML diagrams. As you can see in Figure below, notes are depicted as rectangles with the top right corners folded over.

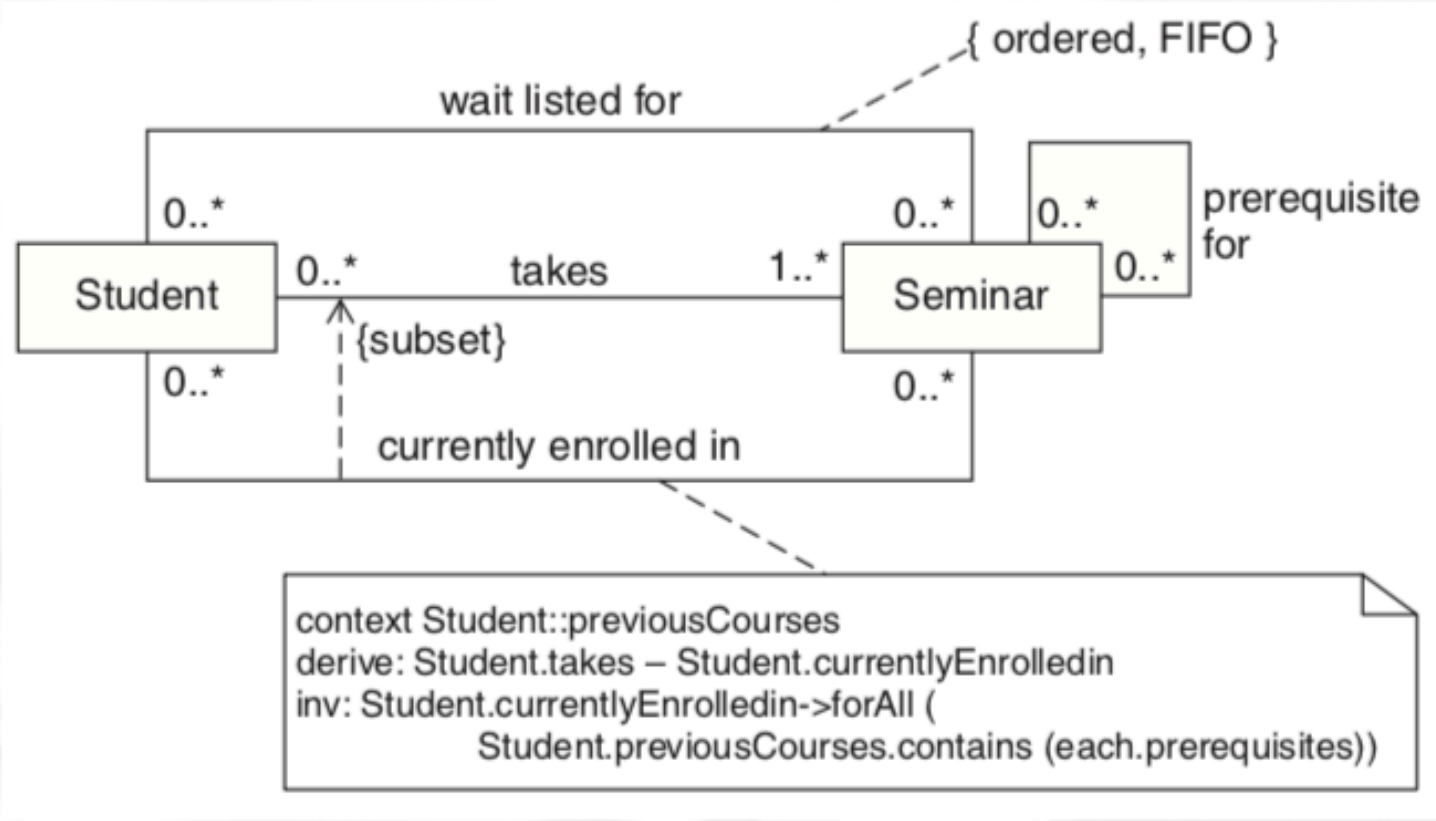
14. Left–Justify Text in Notes

It is common practice to left–justify text in UML notes, as you can see in Figure above.

Guidelines for UML Notes

15. Prefer Notes over OCL to Indicate Constraints:

In UML, constraints are modeled either by a UML note using free–form text or with Object Constraint Language (OCL).



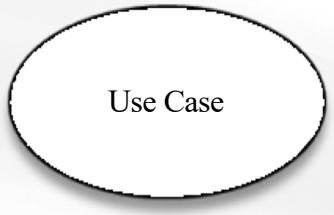
UML Use–Case Diagrams

A UML use–case diagram shows the relationships among actors and use cases within a system. They are often used to:

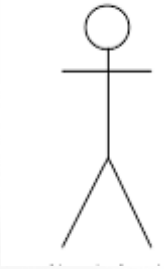
- provide an overview of all or part of the usage requirements for a system or organization in the form of an essential model.
- model the analysis of usage requirements in the form of a system use–case model.
- The use case model describes the functional requirements of the system in terms of the actors and use cases

COMPONENTS OF USE CASE DIAGRAMS

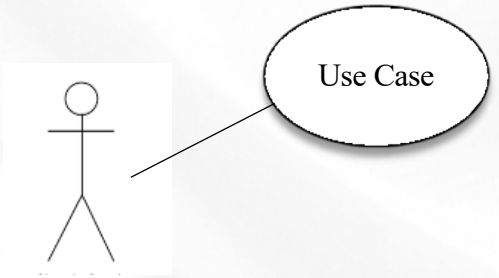
Use Case
Functionality Or Services Provided By The System



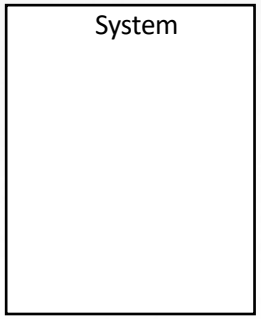
Actors
Who Interacts With The System



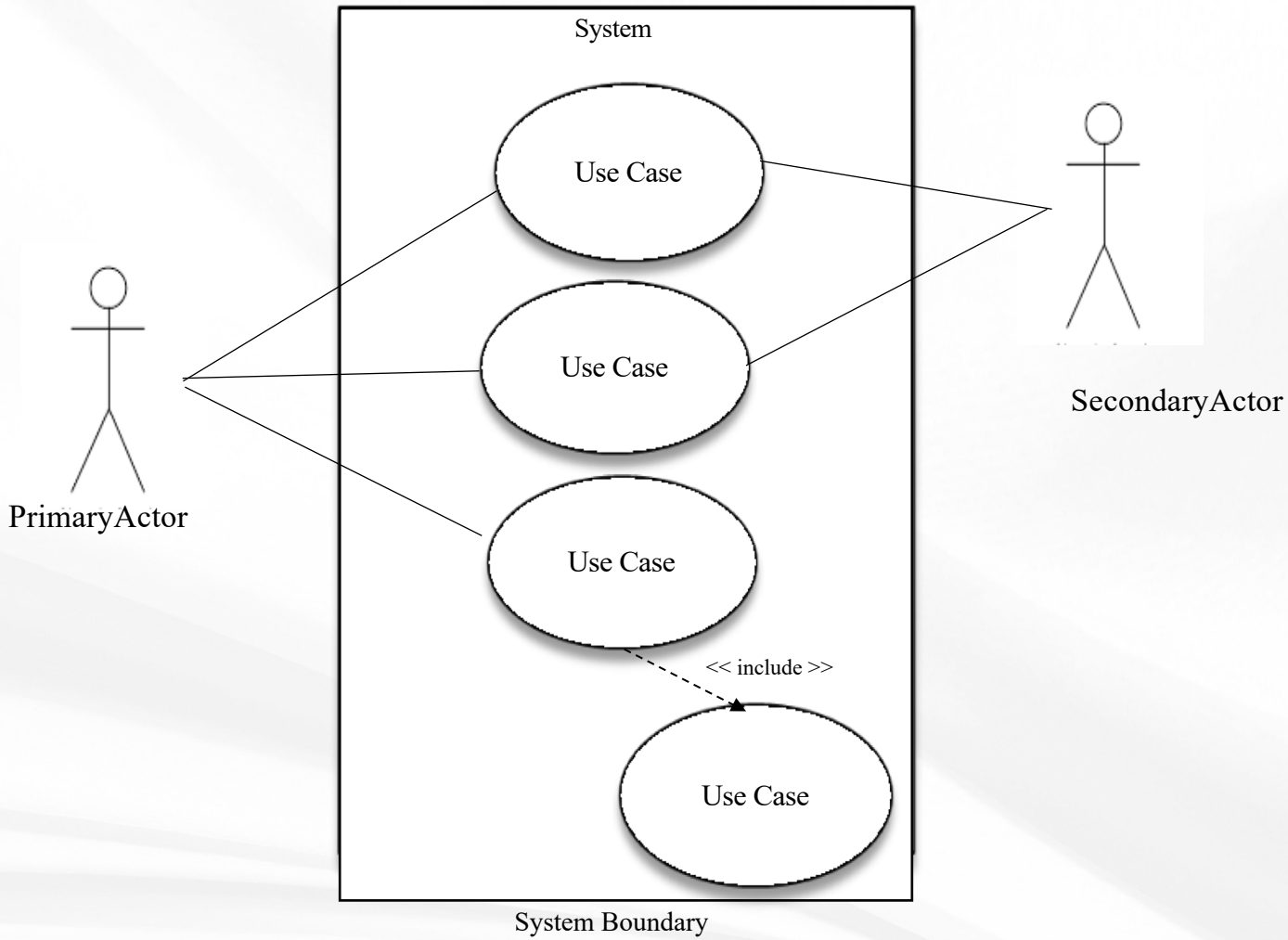
Relationship
Relation Between Actors and use cases



System Boundary
it indicates the scope of the system



COMPONENTS OF USE CASE DIAGRAMS

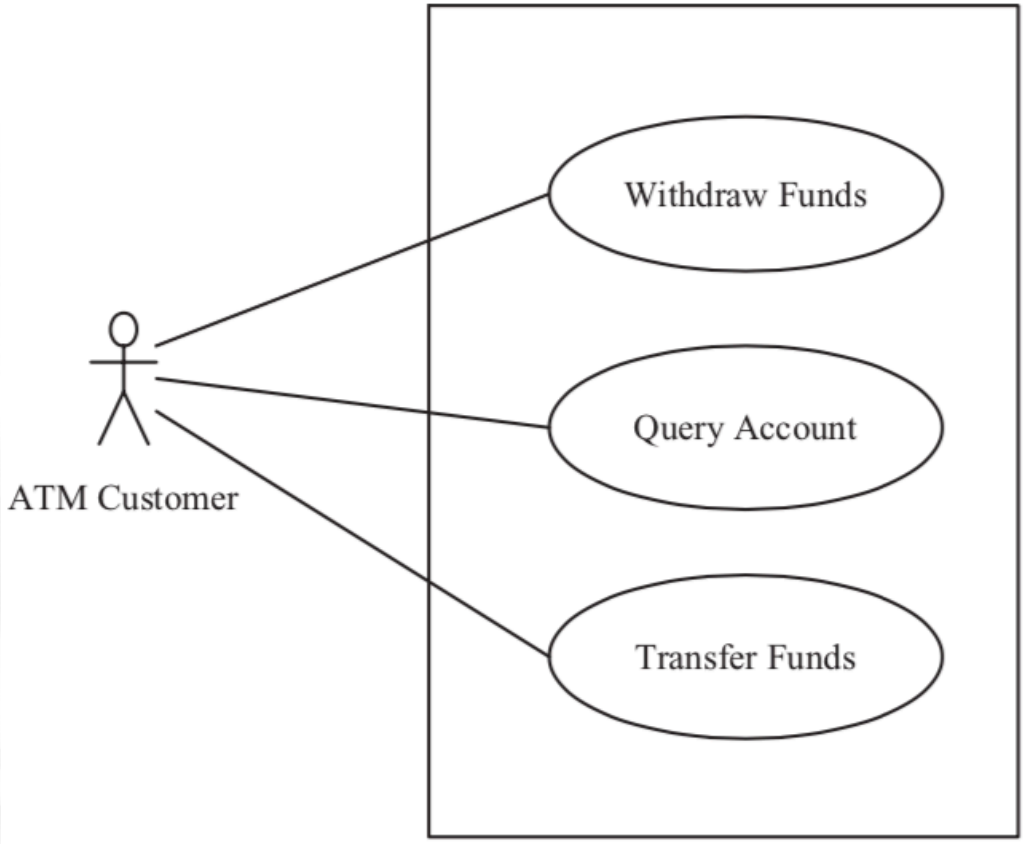


IDENTIFYING USE CASES

- A **use case** defines a sequence of interactions between one or more actors and the system.
- A use case always starts with input from an actor. A use case typically consists of a sequence of interactions between the actor and the system. Each interaction consists of an input from the actor followed by a response from the system.
- In this way, the functional requirements of the system are described in terms of the use cases

IDENTIFYING USE CASES

- Let us consider the banking example, The customer can initiate three use cases: Withdraw Funds, Query Account, and Transfer Funds



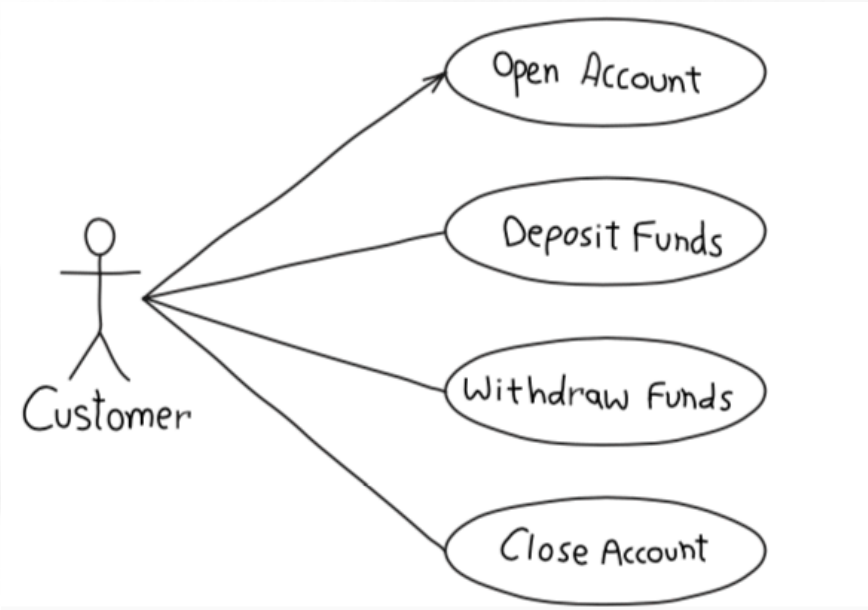
Banking System actor and use cases

Use–Case Guidelines

- Begin Use–Case Names with a Strong Verb
- Name Use Cases Using Domain Terminology
- Imply Timing Considerations by Stacking Use Cases

Withdraw Funds ✔

Process Withdrawal Transaction ✘



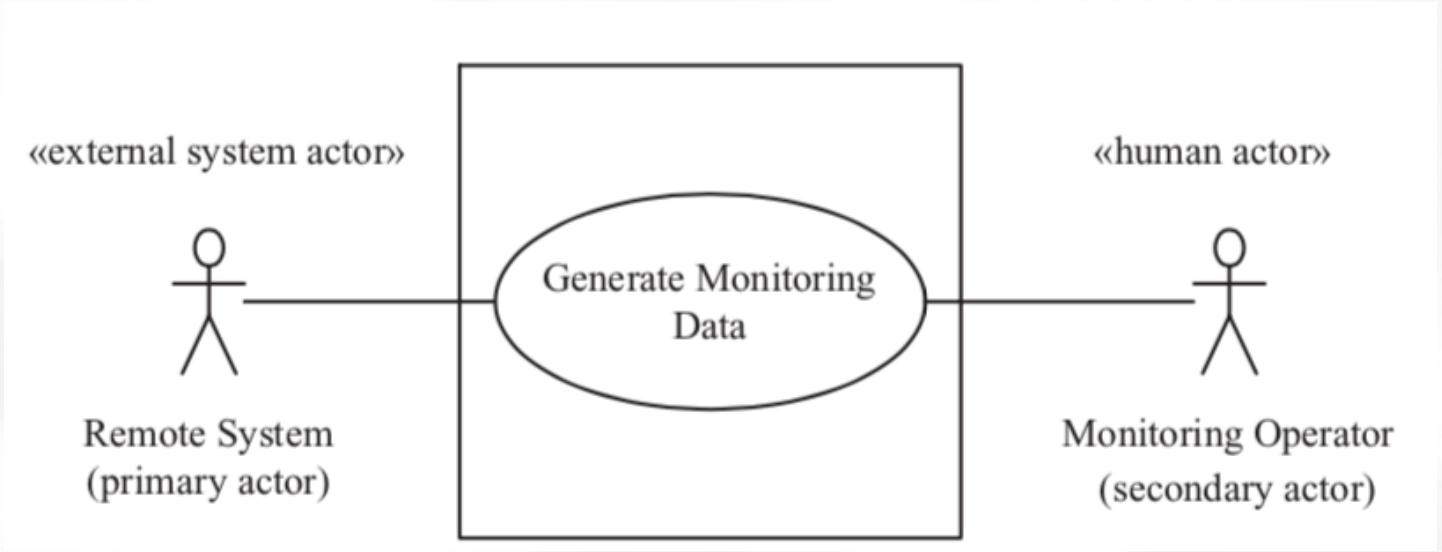
IDENTIFYING ACTORS

- An **actor** characterizes an external user (i.e., outside the system) that interacts with the system. in other words, actors are outside the system and not part of it.
- An actor is a person, organization, local process (e.g., system clock), or external system that plays a role in one or more interactions with your system. It is very often a human user. For this reason, in UML, an actor is depicted using a stick figure.
- It is possible for an actor to be an external system that interfaces to the system, in some applications, an actor can also be an external I/O device or a timer(in real–time embedded systems).

IDENTIFYING ACTORS

❑ Primary and Secondary Actors

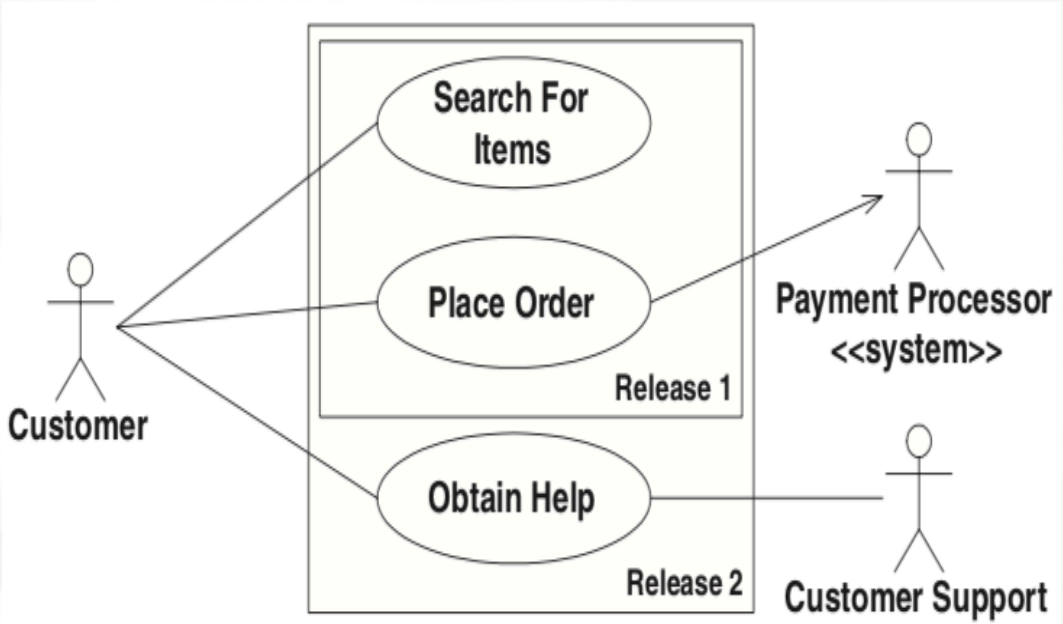
A **primary actor** initiates a use case. Thus, the use case starts with an input from the primary actor to which the system has to respond. Other actors, referred to as **secondary actors**, can participate in the use case.



Example of primary and secondary actors, as well as external system actor

Actor Guidelines

- Place Your Primary Actor(s) in the Top Left Corner of the Diagram.
- Draw Actors on the Outside Edges of a Use–Case Diagram
- Name Actors with Singular, Domain–Relevant Nouns
- Associate Each Actor with One or More Use Cases
- Use <<system>> to Indicate System Actors



Online shopping.

IDENTIFYING USE CASE RELATIONSHIP

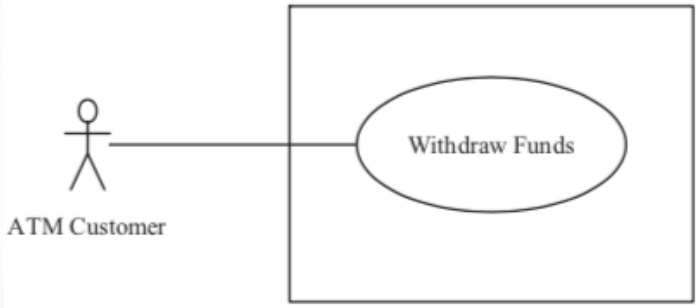
There are several types of relationships that may appear on a use–case diagram:

- Association Relationship
- Include Relationship
- Extend Relationship
- Generalization Relationship

IDENTIFYING USE CASE RELATIONSHIP

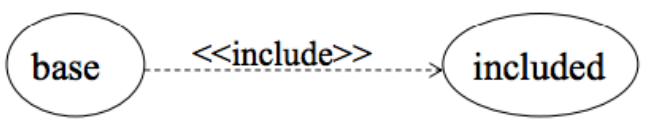
❖ Association Relationship

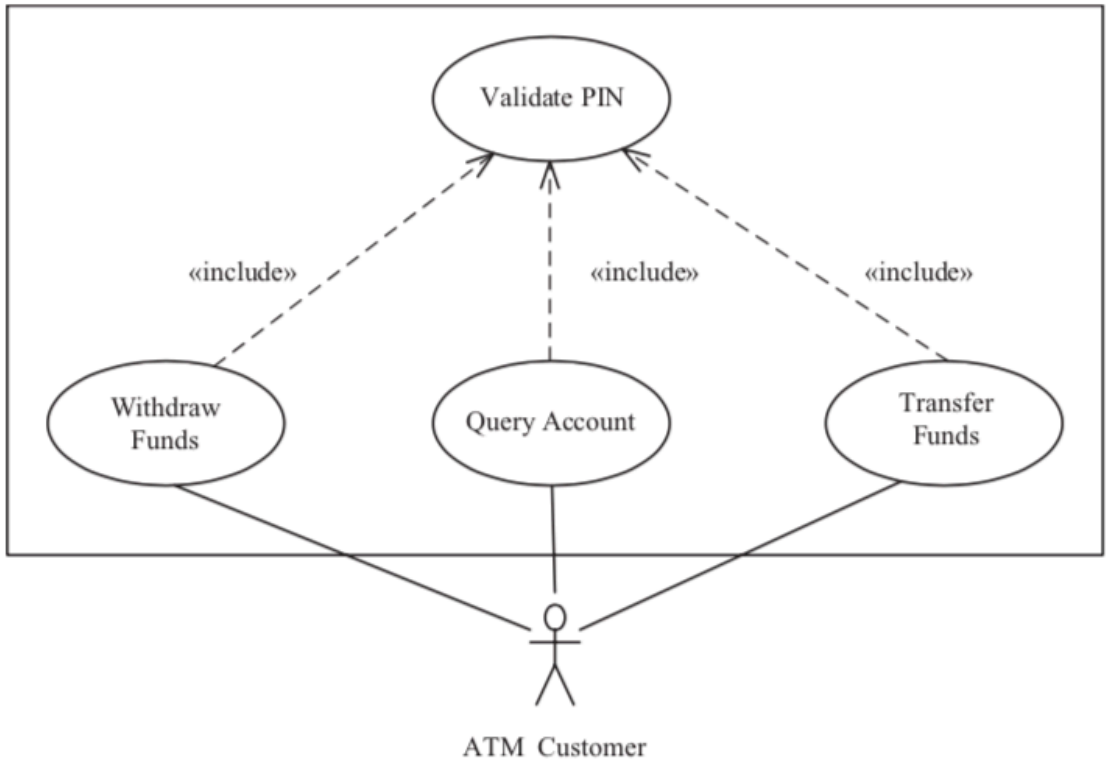
The Association Relationship represents a communication or interaction between an actor and a use case.



❖ Include Relationship

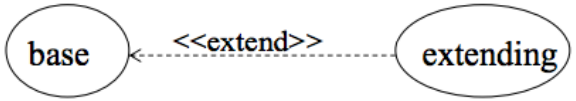
The Include Relationship indicates that a use case includes the functionality of another use case.





IDENTIFYING USE CASE RELATIONSHIP

❖ Extend Relationship

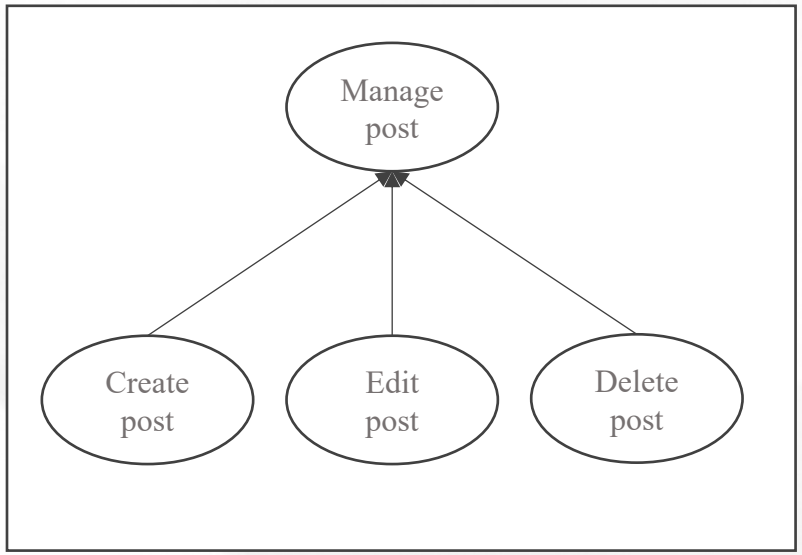


The Extend Relationship illustrates that a use case can be extended by another use case under specific conditions.

❖ Generalization Relationship

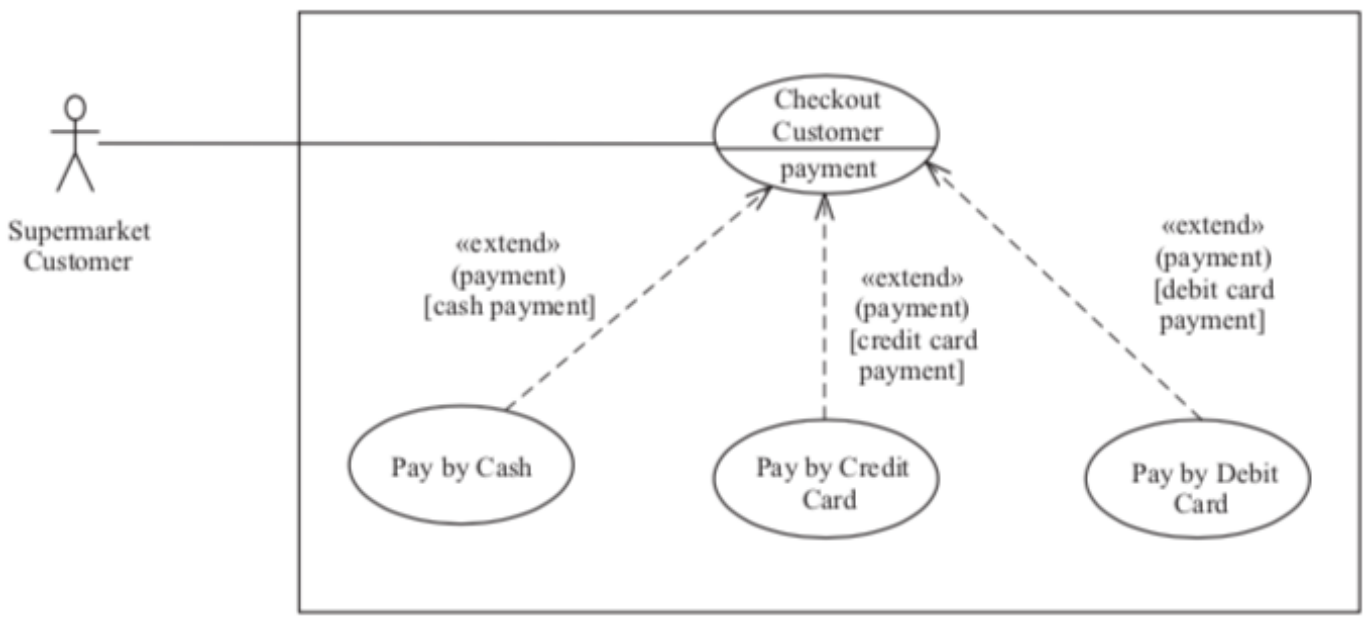
The Generalization Relationship establishes an “is–a” connection between two use cases, or two actors.

- 2 use case indicating that one use case is a specialized version of another.
- 2 actors represents a hierarchical relationship where one actor is a more specialized version of another actor.



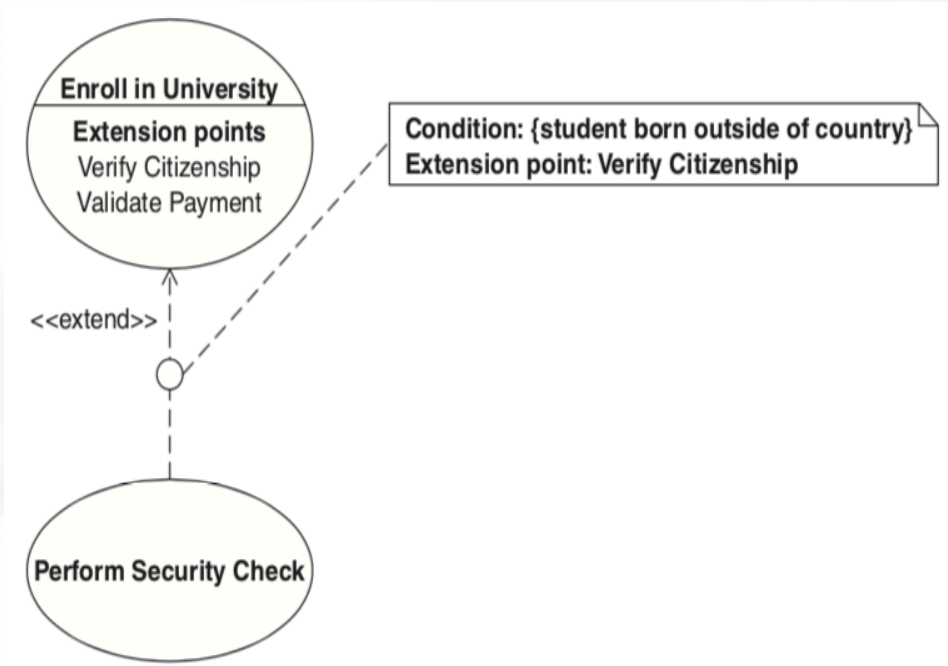
Extension Points

- Extension points are used to specify the precise locations in the base use case at which extensions can be added. An extension use case may extend the base use case only at these extension points.
- Each extension point is given a name.



Relationship Guidelines

- Avoid Arrowheads on Actor—Use-Case Relationships
- Do Not Apply <<uses>>, <<includes>>, or <<extends>>
- Place an Included Use Case to the Right of the Invoking Use Case
- Place an Extending Use Case Below the Parent Use Case
- Place an Inheriting Use Case Below the Base Use Case
- Avoid Modeling Extension Points
- Model Extension Conditions Only When They Aren't Clear



DOCUMENTING USE CASES IN THE USE CASE MODEL

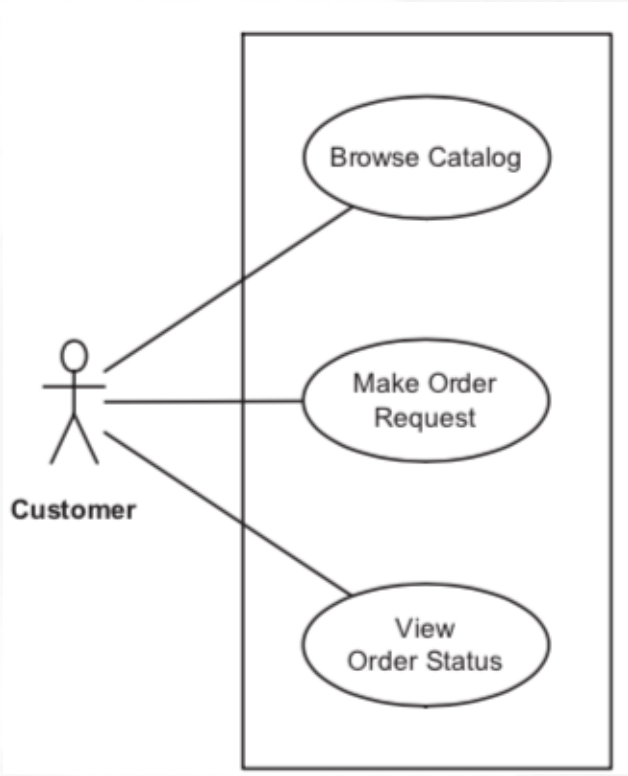
Each use case in the use case model is documented in a use case description, as follows:

Use case name:	Each use case is given a name.
Summary	A brief description of the use case, typically one or two sentences.
Dependency	This optional section describes whether the use case depends on other use cases — that is, whether it includes or extends another use case.
Actors	This section names the actors in the use case. There is always a primary actor that initiates the use case.
Preconditions	One or more conditions that must be true at the start of use case, from the perspective of this use case
Description of main sequence	description of the main sequence of the use case, which is the most usual sequence of interactions between the actor and the system.
Description of alternative sequences	Narrative description of alternative branches off the main sequence.
Postcondition	Condition that is always true at the end of the use case

DOCUMENTING USE CASES IN THE USE CASE MODEL

Example Of Use Case Description:

Use case name:	Make Order Request
Summary (Description)	Customer enters an order request to purchase items from the online shopping system. The customer's credit card is checked for sufficient credit to pay for the requested catalog items.
Actors	Customer
Preconditions	The customer has selected one or more catalog items.
Description of main sequence	1. Customer provides order request and customer account Id to pay for purchase. 2. System retrieves customer account information, including the customer's credit card details. 3. System checks the customer's credit card for the purchase amount and, if approved, creates a credit card purchase authorization number. 4. System creates a delivery order containing order details, customer Id, and credit card authorization number. 5. System confirms approval of purchase and displays order information to customer.

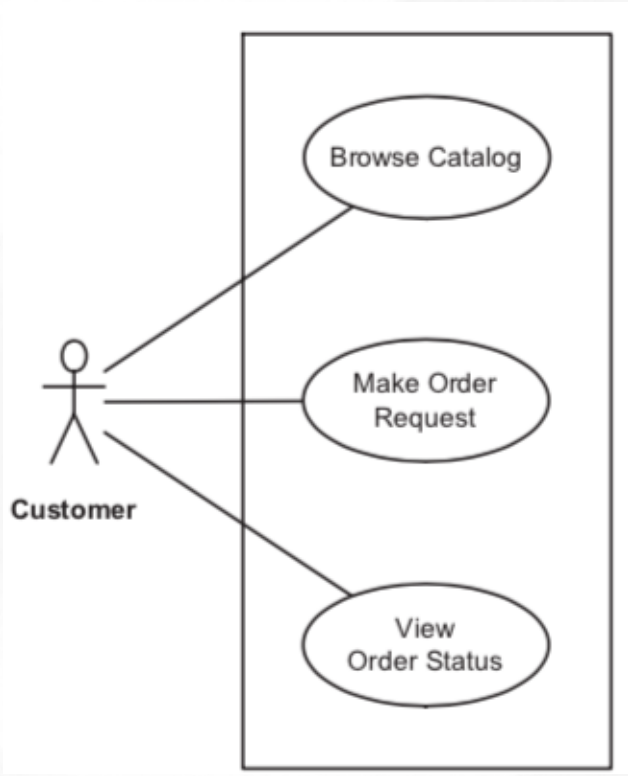


Online Shopping System

DOCUMENTING USE CASES IN THE USE CASE MODEL

Example Of Use Case Description:

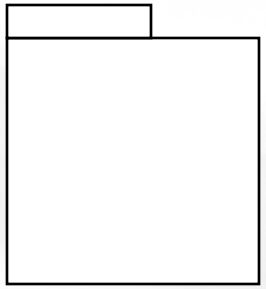
Description of Alternative sequences:	Step 2: If customer does not have account, the system creates an account. Step 3: If the customer's credit card request is denied, the system prompts the customer to enter a different credit card number. The customer can either enter a different credit card number or cancel the order.
Postcondition	1. System has created a delivery order for the customer.



Online Shopping System

USE CASE PACKAGES

For large systems, having to deal with a large number of use cases in the use case model often gets unwieldy. A good way to handle this scale–up issue is to introduce a **use case package** that groups together related use case



Example of use case package

The End