What is UML?

- UML stands for “Unified Modeling Language”
- It is a industry-standard graphical language for specifying, visualizing, constructing, and documenting the artifacts of software systems
- The UML uses mostly graphical notations to express the OO analysis and design of software projects.
- Simplifies the complex process of software design

Why UML for Modeling

- Use graphical notation to communicate more clearly than natural language (imprecise) and code(too detailed).
- Help acquire an overall view of a system.
- UML is *not* dependent on any one language or technology.
- UML moves us from fragmentation to standardization.

Types of UML Diagrams

- **Use Case Diagram**
- **Class Diagram**
- **Sequence Diagram**
- **Collaboration Diagram**
- **State Diagram**

This is only a subset of diagrams … but are most widely used

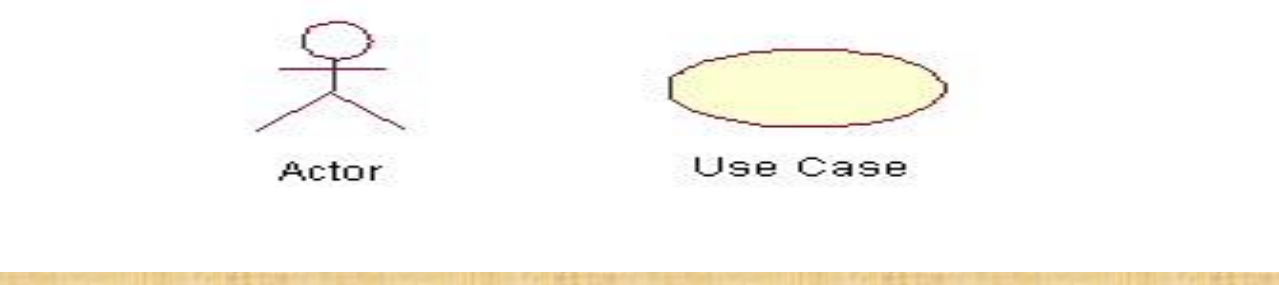
Use Case Diagram

- Used for describing a set of user **scenarios**
- Mainly used for capturing user requirements
- Work like a **contract** between end user and software developers

Use Case Diagram (core components)

Actors: A role that a user plays with respect to the system,including human users and other systems. e.g.,inanimate physical objects (e.g. robot); an external system that needs some information from the current system.

Use case: A set of scenarios that describes an interaction between a user and a system, including alternatives.



System boundary: rectangle diagram representing the boundary between the actors and the system.

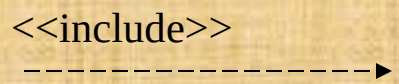
Use Case Diagram(core relationship)

Association: communication between an actor and a use case; Represented by a solid line.

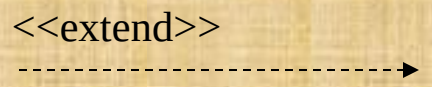
Generalization: relationship between one general use case and a special use case (used for defining special alternatives)
Represented by a line with a triangular arrow head toward the parent use case.

Use Case Diagram(core relationship)

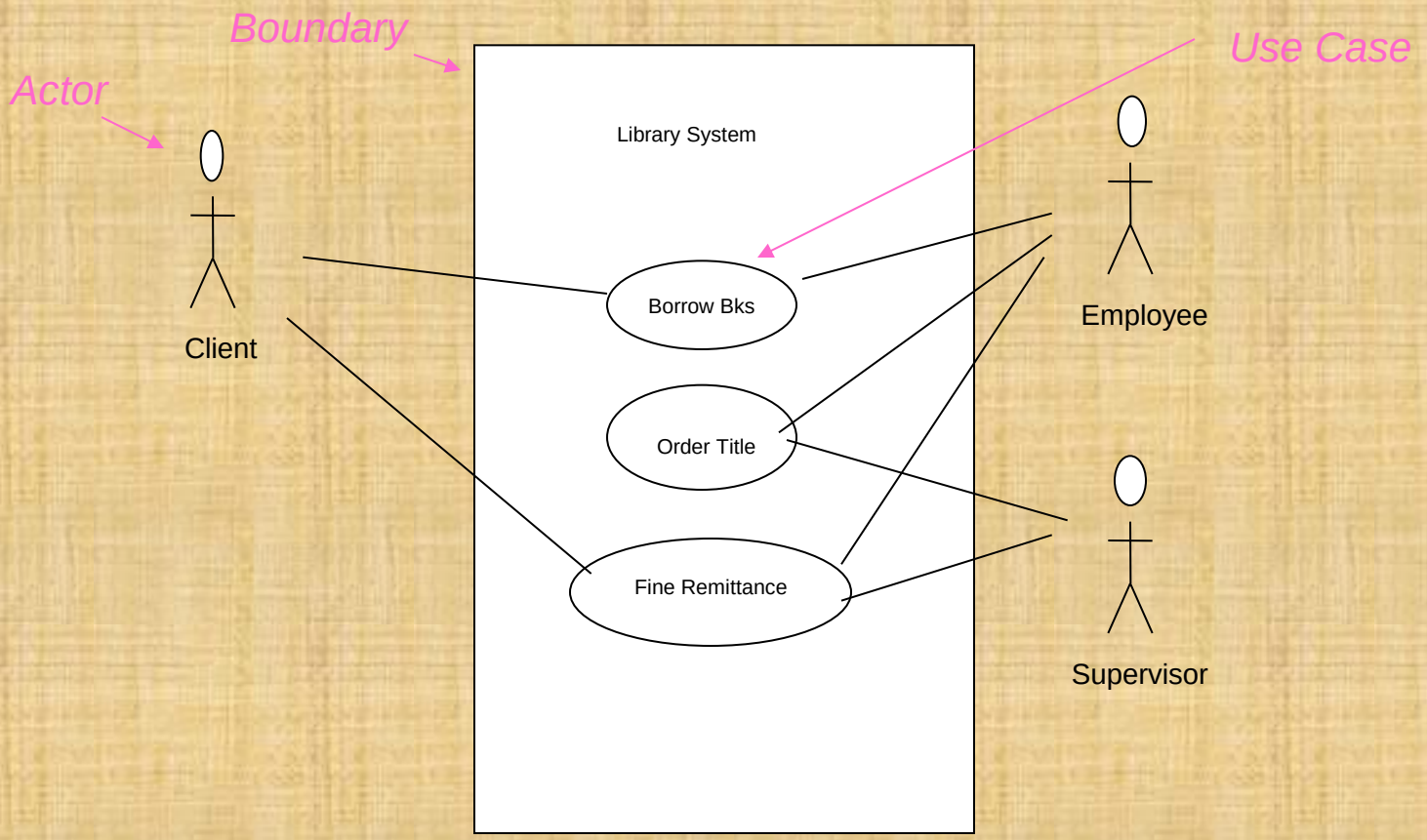
Include: a dotted line labeled <<include>> beginning at base use case and ending with an arrow pointing to the include use case. The include relationship occurs when a chunk of behavior is similar across more than one use case. Use "include" instead of copying the description of that behavior.



Extend: a dotted line labeled <<extend>> with an arrow toward the base case. The extending use case may add behavior to the base use case. The base class declares "extension points".

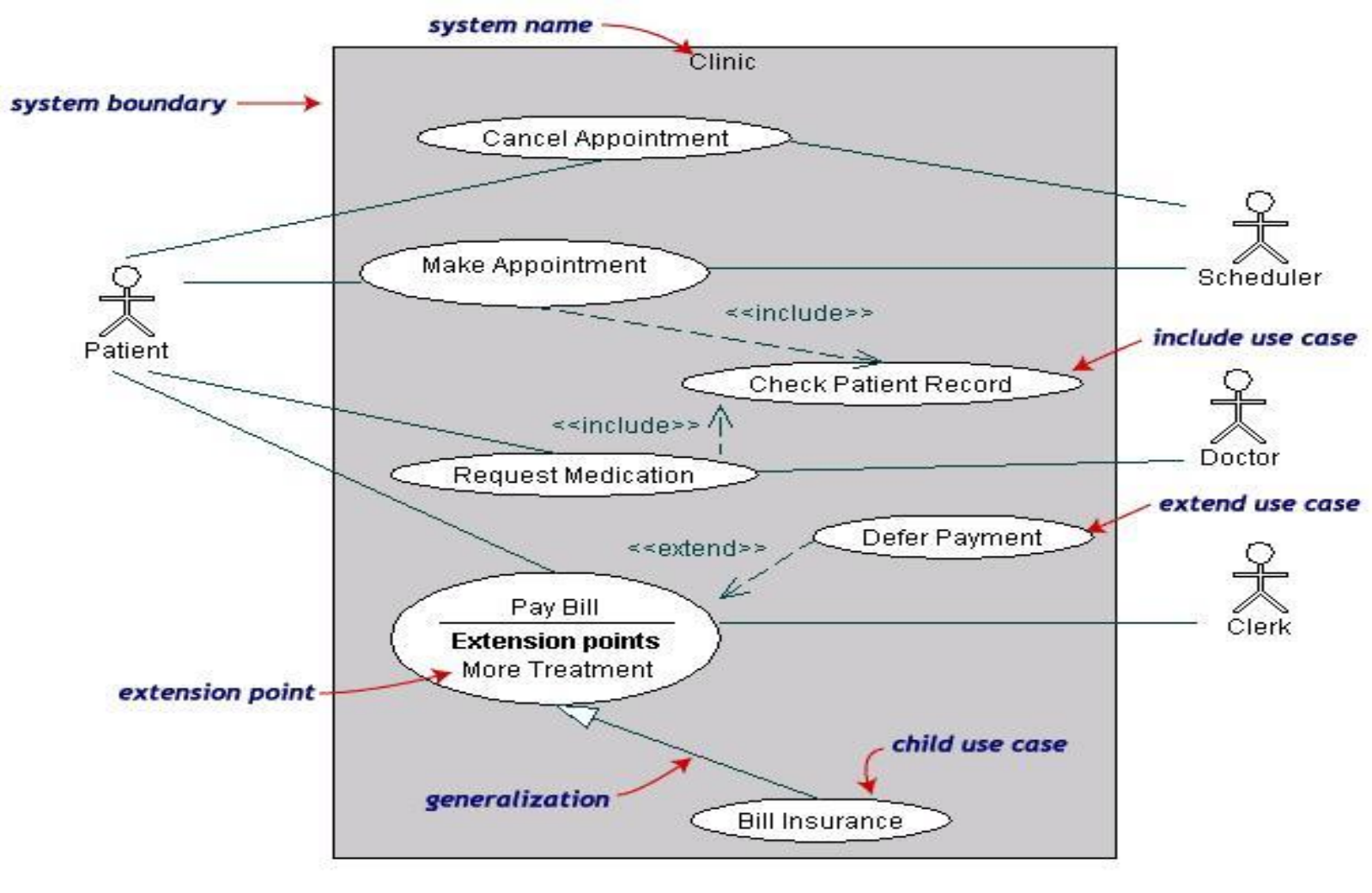


Use Case Diagrams



- A generalized description of how a system will be used.
- Provides an overview of the intended functionality of the system

Use Case Diagrams(cont.)



Use Case Diagrams(cont.)

•**Pay Bill** is a parent use case and **Bill Insurance** is the child use case. (generalization)

•Both **Make Appointment** and **Request Medication** include **Check Patient Record** as a subtask.(include)

•The **extension point** is written inside the base case **Pay bill**; the extending class **Defer payment** adds the behavior of this extension point. (extend)

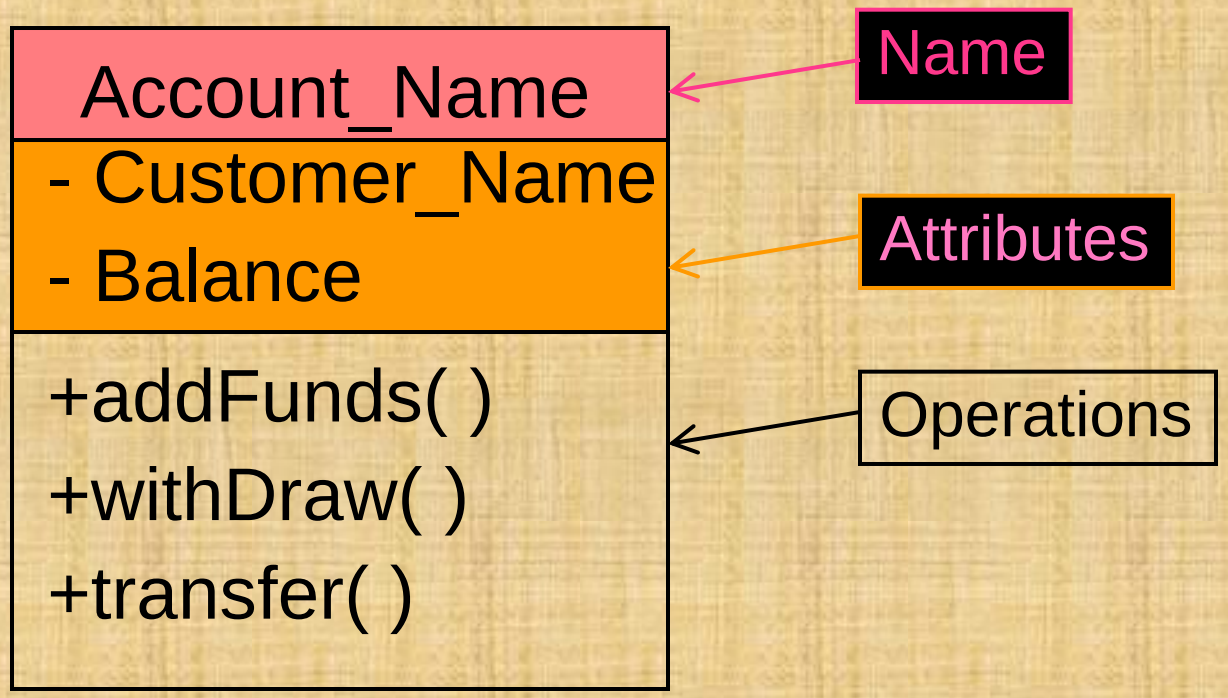
Class diagram

- Used for describing structure and behavior in the use cases
- Provide a conceptual model of the system in terms of entities and their relationships
- Used for requirement capture, end-user interaction
- Detailed class diagrams are used for developers

Class representation

- Each class is represented by a rectangle subdivided into three compartments
  - Name
  - Attributes
  - Operations
- Modifiers are used to indicate visibility of attributes and operations.
  - '+' is used to denote *Public* visibility (everyone)
  - '#' is used to denote *Protected* visibility (friends and derived)
  - '-' is used to denote *Private* visibility (no one)
- By default, attributes are hidden and operations are visible.

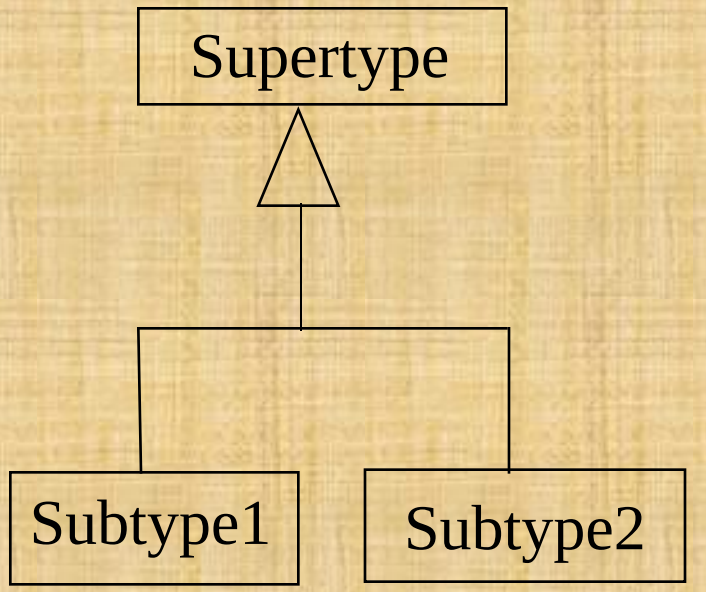
An example of Class



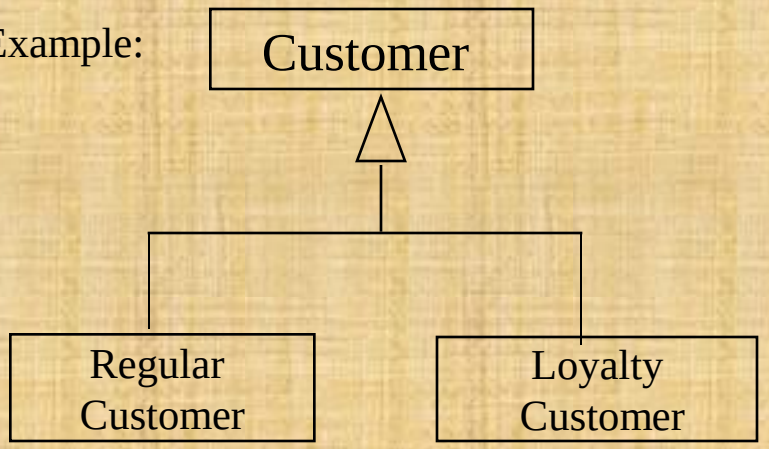
OO Relationships

- There are two kinds of Relationships
  - Generalization (parent-child relationship)
  - Association (student enrolls in course)
- Associations can be further classified as
  - Aggregation
  - Composition

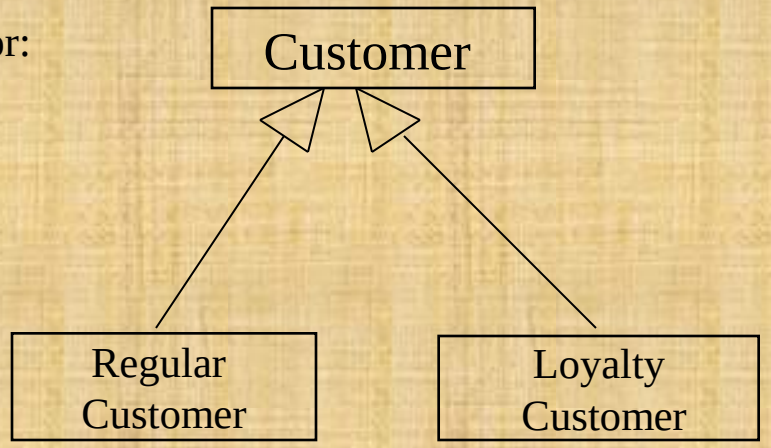
OO Relationships: **Generalization**



Example:



or:



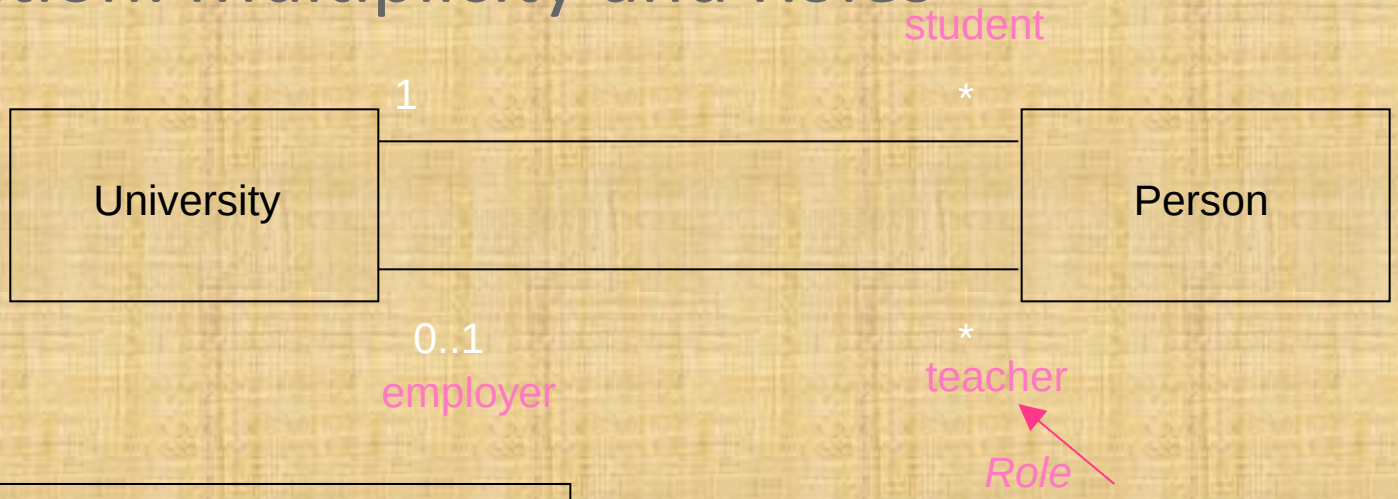
- Generalization expresses a parent/child relationship among related classes.

- Used for abstracting details in several layers

OO Relationships: **Association**

- Represent relationship between instances of classes
  - Student enrolls in a course
  - Courses have students
  - Courses have exams
  - Etc.
- Association has two ends
  - Role names (e.g. enrolls)
  - Multiplicity (e.g. One course can have many students)
  - Navigability (unidirectional, bidirectional)

Association: Multiplicity and Roles



Multiplicity

Symbol	Meaning
1	One and only one
0..1	Zero or one
M..N	From M to N (natural language)
*	From zero to any positive integer
0..*	From zero to any positive integer
1..*	From one to any positive integer

Role

"A given university groups many people; some act as students, others as teachers. A given student belongs to a single university; a given teacher may or may not be working for the university at a particular time."

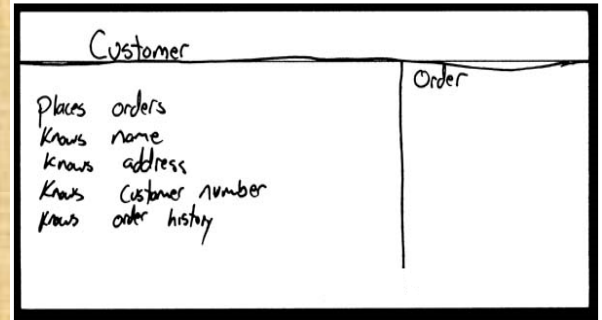
Design phase

- **design**: specifying the structure of how a software system will be written and function, without actually writing the complete implementation

- a transition from "what" the system must do, to "how" the system will do it
  - What classes will we need to implement a system that meets our requirements?
  - What fields and methods will each class have?
  - How will the classes interact with each other?

How do we design classes?

- class identification from project spec / requirements
  - nouns are potential classes, objects, fields
  - verbs are potential methods or responsibilities of a class
- CRC card exercises
  - write down classes' names on index cards
  - next to each class, list the following:
    - **responsibilities**: problems to be solved; short verb phrases
    - **collaborators**: other classes that are sent messages by this class (asymmetric)
- UML diagrams
  - class diagrams



Class design exercise

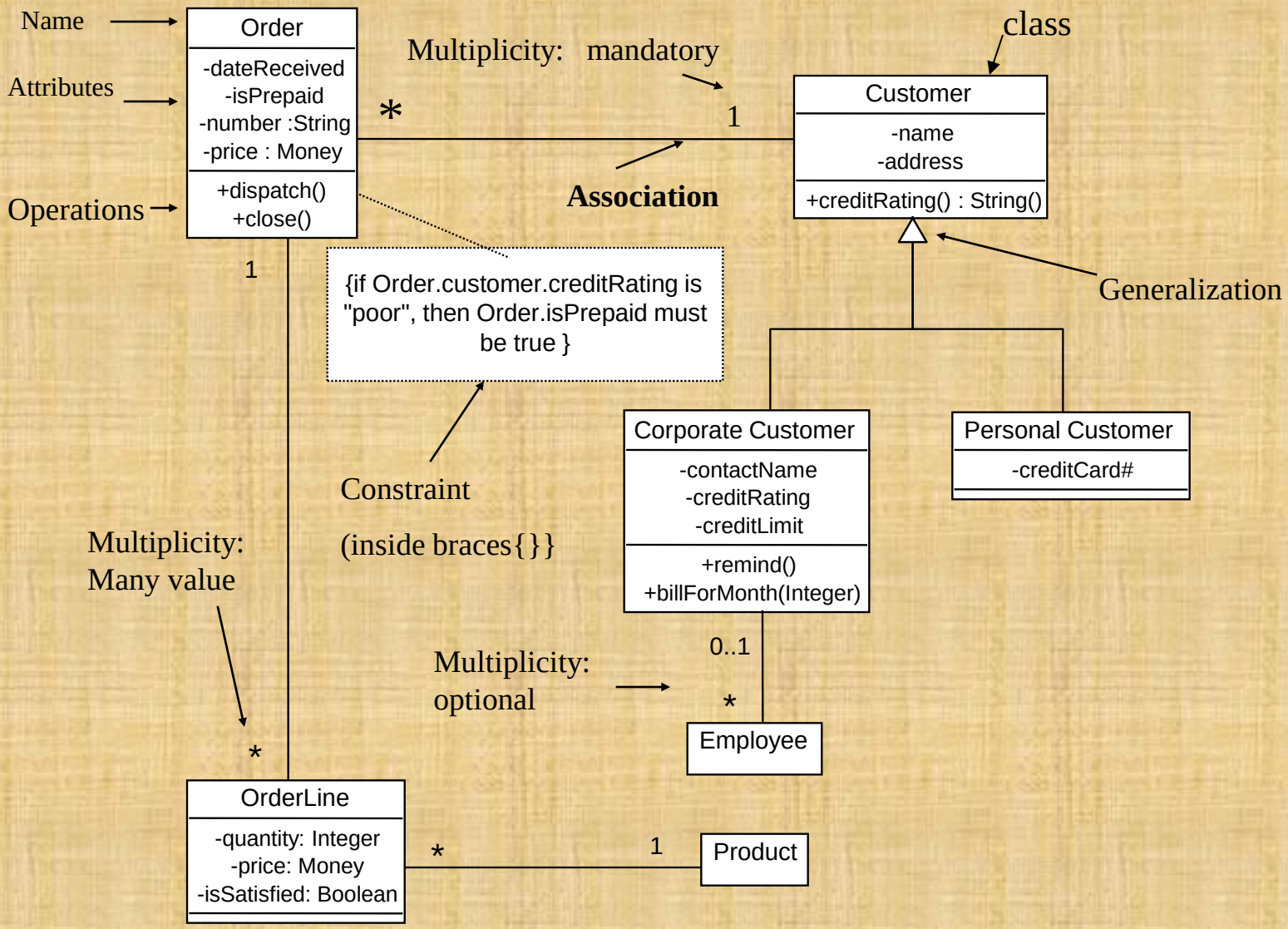
- Consider the following system: We want to write software for a Texas Hold 'em poker game in Java. The game will have a graphical user interface. The game allows 2 to 8 human or computer players. Each player has a name and a stack of chips representing their money. Computer players also each have a difficulty setting of easy, medium, or hard.

  At the start of each round of the game, the dealer collects ante from the appropriate players, shuffles the deck, and deals each player a hand of 2 cards from the deck. A round of betting takes place, followed by dealing several shared cards from the card deck. As shared cards are dealt, more betting rounds occur, where each player can fold, check, or raise. At the end of a round, if more than one player is remaining, players' hands are compared, and the best hand wins the pot of all chips bet so far.

UML Class Diagrams

- Class diagram describes
  - Types of objects in the system
  - Static relationships among them
- Two principal kinds of static relationships
  - Associations between classes
  - Subtype relationships between classes
- Class descriptions show
  - Attributes
  - Operations

Class Diagram



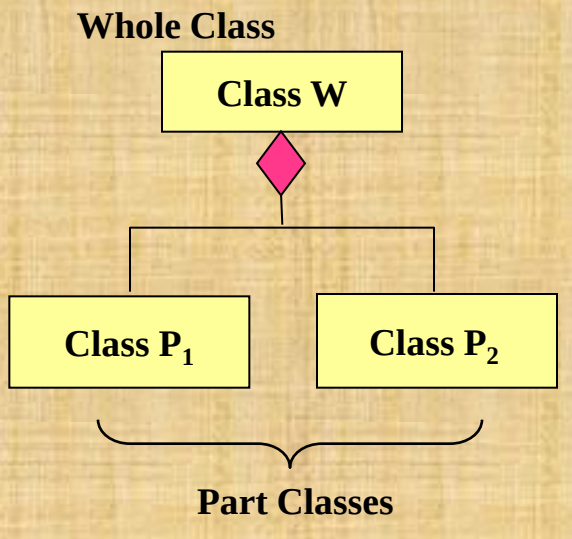
[from *UML Distilled Third Edition*]

Association: Model to Implementation

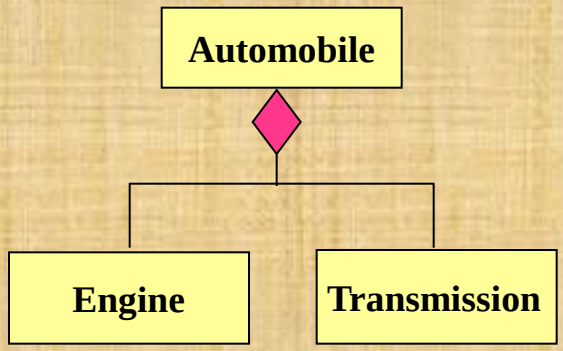

```
Class Student {
    Course enrolls[4];
}

Class Course {
    Student have[];
}
```

OO Relationships: **Composition**



Example



Composition: **expresses a relationship among instances of related classes. It is a specific kind of Whole-Part relationship.**

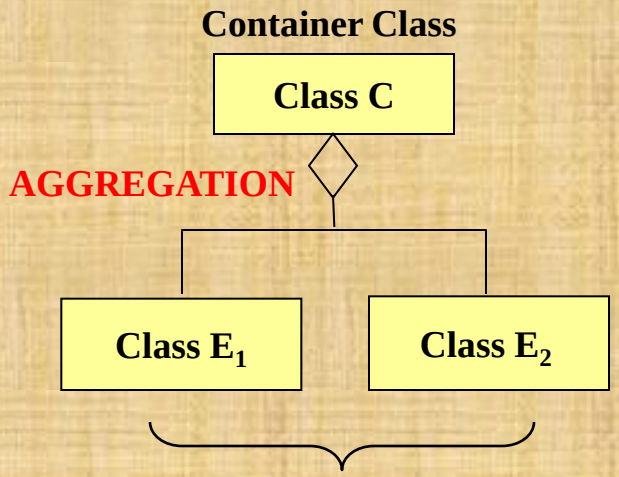
It expresses a relationship where an instance of the Whole-class has the responsibility to create and initialize instances of each Part-class.

It may also be used to express a relationship where instances of the Part-classes have privileged access or visibility to certain attributes and/or behaviors defined by the Whole-class.

Composition should also be used to express relationship where instances of the Whole-class have exclusive access to and control of instances of the Part-classes.

Composition should be used to express a relationship where the behavior of Part instances is undefined without being related to an instance of the Whole. And, conversely, the behavior of the Whole is ill-defined or incomplete if one or more of the Part instances are undefined.

OO Relationships: **Aggregation**



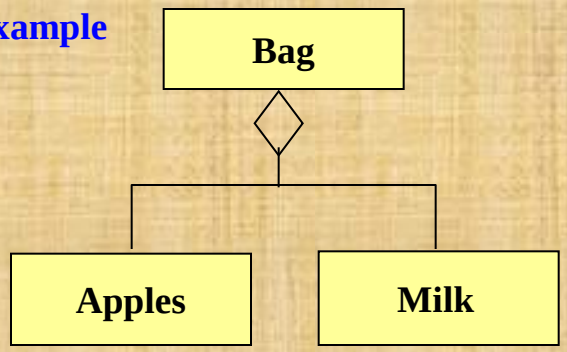
Aggregation: **expresses a relationship among instances of related classes. It is a specific kind of Container-Containee relationship**

It expresses a relationship where an instance of the Container-class has the responsibility to hold and maintain instances of each Containee-class that have been created outside the auspices of the Container-class.

Aggregation should be used to express a more informal relationship than composition expresses. That is, it is an appropriate relationship where the Container and its Containees can be manipulated independently.

Aggregation is appropriate when Container and Containees have no special access privileges to each other.

Example



Aggregation vs. Composition

- **Composition** is really a strong form of **aggregation**
  - components have only one owner
  - components cannot exist independent of their owner
  - components live or die with their owner

  e.g. Each car has an engine that can not be shared with other cars.

- **Aggregations** may form "part of" the aggregate, but may not be essential to it. They may also exist independent of the aggregate.

  e.g. Apples may exist independent of the bag.

Use Case Revisited

Use Cases and Business Rules

- Much written today about separating business rules from other dimensions of automated business systems.
- Without proper separation, they operate in enterprises without a great deal of thought given to them.
- Ironically, they may be the most important dimension because they represent important business thinking behind processes, use cases, for example.
- Various approaches for dealing with business rules and use cases. First, we look at three important questions

Question #1: Why Separate Business Rules?

- The primary benefit of separating them is to manage them separately. If we separate them properly, we can:
  - Make changes in one with minimal impact on the other. This is desirable because business rules often change more frequently than do use cases.
  - Share business rules across many use cases.
  - Define, analyze, and test business rules prior to, or in parallel with, development of use cases.
  - Iterate between use case development and business rule capture.
  - Involve different business people in use case development versus business rule capture, as appropriate.

Question #2: How to Separate Business Rules?

- To separate business rules from use cases, they must be different.
  - Note: business rules and use cases are fundamentally different considerations. Use cases are primarily about actor interactions. Business rules are about reasoning and logic.
- The second way is to pull them out of the use case altogether. We accomplish this by providing pointers to them in a business rule repository.
- Both approaches separate them from the major emphasis of the use case, actor interactions.

How to Promote Business Rules to a New Dimension?

- They must not only be different from other dimensions, they must have their own existence, independent of how and where executed, and whether automated or not.
- The combination of their own existence and independence means we should be able to cast them in their own model that is recognizably different (in structure and behavior) from all other kinds of models.

Identify Use Cases

- Capture the specific ways of using the system as dialogues between an actor and the system.
- Use cases are used to
  - Capture system requirements
  - Communicate with end users and Subject Matter Experts
  - Test the system

Specifying Use Cases

- Create a written document for each Use Case
  - Clearly define intent of the Use Case
  - Define Main Success Scenario (Happy Path)
  - Define any alternate action paths
  - Use format of Stimulus: Response
  - Each specification must be testable
  - Write from actor's perspective, in actor's vocabulary

General Recommended Template

- Name
- Primary Actor
- Scope
- Level
- Stakeholders and Interests
- Minimal Guarantee
- Success Guarantee
- Main Success Scenario
- Extensions

- This is the basic format used in the text
- I prefer to modify it slightly to use the actor actions and system response in tabular form. Larman calls this the Two-Column Variation.

Optional Items

- You can add some of the following items
  - Trigger (after Success Guarantee)
- (at end:)
  - Special requirements
  - Technology and Data Variations
  - Frequency of Occurrence
  - Open Issues

Use Case UC1: - Create Shopping List Scenario

Scope : Menu Planner Application
Level: User Goal
Primary Actor: Person who completes grocery store shopping operation
Stakeholders and Interests:
-Person who completes store shopping:
Desires accurate list of items to purchase for the next week's meal requirements.
-Cook or Chef:
Requires that all items needed to prepare food are in stock and readily available

Pre conditions:
Budget amount accepted as input
Menu application has access to the database of food inventory.
Inventory process has not been corrupted.
Network and printer availability.

Success Guarantee - Post conditions:
Shopping list generated to maximize budget and dietary preferences.

Main Success Scenario (or Basic Flow)

1. Shopper issues command to generate shopping list
2. Menu Planner application checks database of items that are required:
   a. All items used during recipe preparations are added to shopping list
   b. All additional items that were entered throughout the week by users as requested foods or "running low on" foods are added to list.
3. Depending on user preference and upon a database query a shopping list is generated via paper, email, or transmission to palm pilot.

Extensions (or Alternative Flows)

A. At any time, a network connection is lost
   a. Error message generated requesting user to reestablish connection
   b. Ability to work off-line if user is unable to reestablish connection
   c. Shopping list available via print command
B. At any time, inventory database connection is lost
   a. Error message generated
   b. Procedure to re-attempt database connection started
   c. If not able to reconnect, system prints shopping list as per last known state of database
C. At any time, User cancels operation
   a. Database rollback to indicate no items will be purchased to re-supply inventory

SAMPLE: Fully Dressed Use Case

Elements in the Preface

- Only put items that are important to understand before reading the Main Success Scenario. These might include:
- Name (*Always needed for identification*)
- Primary Actor
- Stakeholders and Interests List
- Preconditions
- Success Conditions (Post Conditions)

Naming Use Cases

- Must be a complete process from the viewpoint of the end user.
- Is in verb-object form, ex. Buy Pizza
- Use enough detail to make it specific
- Use active voice, not passive
- From viewpoint of the actor, not the system

Hint

- Appropriate use case names are very important. Because they are selected early, they tend to set the direction for the entire project.
- Most common errors in use case diagrams are poor names, showing procedures instead of complete user processes, and not including the boundary and system name.
- Rational Rose does not show the boundary and name, so you need to provide this in assignments turned in using that tool. (Rational Rose is preferred for assignments.)

Golden Rule of Use-Case Names

- Each use case should have a name that indicates what value (or goal) is achieved by the actor's interaction with the system
- Here are some good questions to help you adhere to this rule:
  - *Why would the actor initiate this interaction with the system?*
    What goal does the actor have in mind when undertaking these actions?
    What value is achieved and for which actor?

Use Case Name Examples

- Excellent - Purchase Concert Ticket
- Very Good - Purchase Concert Tickets
- Good - Purchase Ticket (insufficient detail)
- Fair - Ticket Purchase (passive)
- Poor - Ticket Order (system view, not user)
- Unacceptable - Pay for Ticket (procedure, not process)

CRUD

- Examples of bad use case names with the acronym CRUD. (All are procedural and reveal nothing about the actor's intentions.)
- C - actor Creates data
- R - actor Retrieves data
- U - actor Updates data
- D - actor Deletes data

Singular or Plural

- This is usually determined by context.
- There is a preference for the simplest form, but most common form can be better.
- In the example of concert tickets, most people buy more than one, but a significant number buy only one.
- At a supermarket, Buy Items would be best.
- At a vending machine, it would be Buy Item.

Identify Actors

- We cannot understand a system until we know who will use it
  - Direct users
  - Users responsible to operate and maintain it
  - External systems used by the system
  - External systems that interact with the system

Specifying Actors

- Actors are external to the system
- Actors are non-deterministic
- What interacts with the system?
- Actors may be different roles that one individual user interacts with the system
- Actors may be other systems
- Don't assume that Actor = Individual

Types of Actors

- Primary Actor
  - Has goals to be fulfilled by system
- Supporting Actor
  - Provides service to the system
- Offstage Actor
  - Interested in the behavior, but no contribution
- Generally: In diagrams, Primary actors go on the left and others on the right.

Define Actors

- Actors should not be analyzed or described in detail unless the application domain demands it.
- Template for definition:
  - Name
  - Definition
- Example for an ATM application:

Customer: Owner of an account who manages account by depositing and withdrawing funds

Identifying Actors

- Primary Actor
  - Emphasis is on the primary actor for the use case.
- Stakeholders and Interests
  - Other actors are listed as stakeholders.
  - The interests of each key actor should be described.

Working with Use Cases

- Determine the actors that will interact with the system
- Examine the actors and document their needs
- For each separate need, create a use case
- During Analysis, extend use cases with interaction diagrams

Preconditions

- Anything that must always be true before beginning a scenario is a precondition.
- Preconditions are assumed to be true, not tested within the Use Case itself.
- Ignore obvious preconditions such as the power being turned on. Only document items necessary to understand the Use Case.

Success Guarantees

- Success Guarantees (or Postconditions) state what must be true if the Use Case is completed successfully. This may include the main success scenario and some alternative paths. For example, if the happy path is a cash sale, a credit sale might also be regarded a success.
- Stakeholders should agree on the guarantee.

Scenarios

- The Main Success Scenario, or “happy path” is the expected primary use of the system, without problems or exceptions.
- Alternative Scenarios or Extensions are used to document other common paths through the system and error handling or exceptions.

Documenting the “Happy Path”

- The Success Scenario (or basic course) gives the best understanding of the use case
- Each step contains the activities and inputs of the actor and the system response
- If there are three or more items, create a list
- Label steps for configuration management and requirements traceability
- Use present tense and active voice
- Remember that User Interface designers will use this specification

Note: Do not use the term “happy path” in formal documents.

Extensions (Alternative Flows)

- Extensions or Alternative Flow Use Cases allow the specification of
  - Different ways of handling transactions
  - Error processes
- Sections are convenient way to handle alternative courses of action
  - Sections are a segment of a use case executed out of sequence

Two Parts for Extensions

- Condition
  - Describe the reason for the alternative flow as a condition that the user can detect
- Handling
  - Describe the flow of processing in the same manner as the happy path, using a numbering system consistent with the original section.

Documenting Extensions

- Use same format as Happy Path
- Document actions that vary from ideal path
- Include error conditions
- Number each alternate, and start with the condition:

3A. Condition: If [*actor*] performs [*action*] the system …

- If subsequent steps are the same as the happy path, identify and label as (same)
- Steps not included in alternate course are assumed not to be performed.

Requirements and Use Cases

- Requirements capture
- Requirements modelling
- Use-cases
- UML Use-Case Diagrams
  - Basic notation
  - Use-Case Descriptions
  - Other notation: generalizes, includes, extends

Requirements Capture

- Aim to develop system to meet user needs
- Capture user requirements capture through:
  - Background reading/research
  - Interviews with users/clients
  - Observation of current practices
  - Sampling of documents
  - Questionnaires
- This is hard!!!!!

Functional and Non-Functional Requirements

- Functional requirements:
  - What processing system must perform
  - Details of inputs and outputs
  - Details of data held by system
- Non-functional requirements:
  - Performance criteria (time/space)
  - Security
  - Usability requirements
  - HCI issues

ATM Case Study: Statement of Purpose

The design task is to implement an Automated Teller Machine (ATM):

"An ATM is an electronic device designed for automated dispensing of money. A user can withdraw money quickly and easily after authorization. The user interacts with the system through a card reader and a numerical keypad. A small display screen allows messages and information to be displayed to the user. Bank members can access special functions such as ordering a statement"

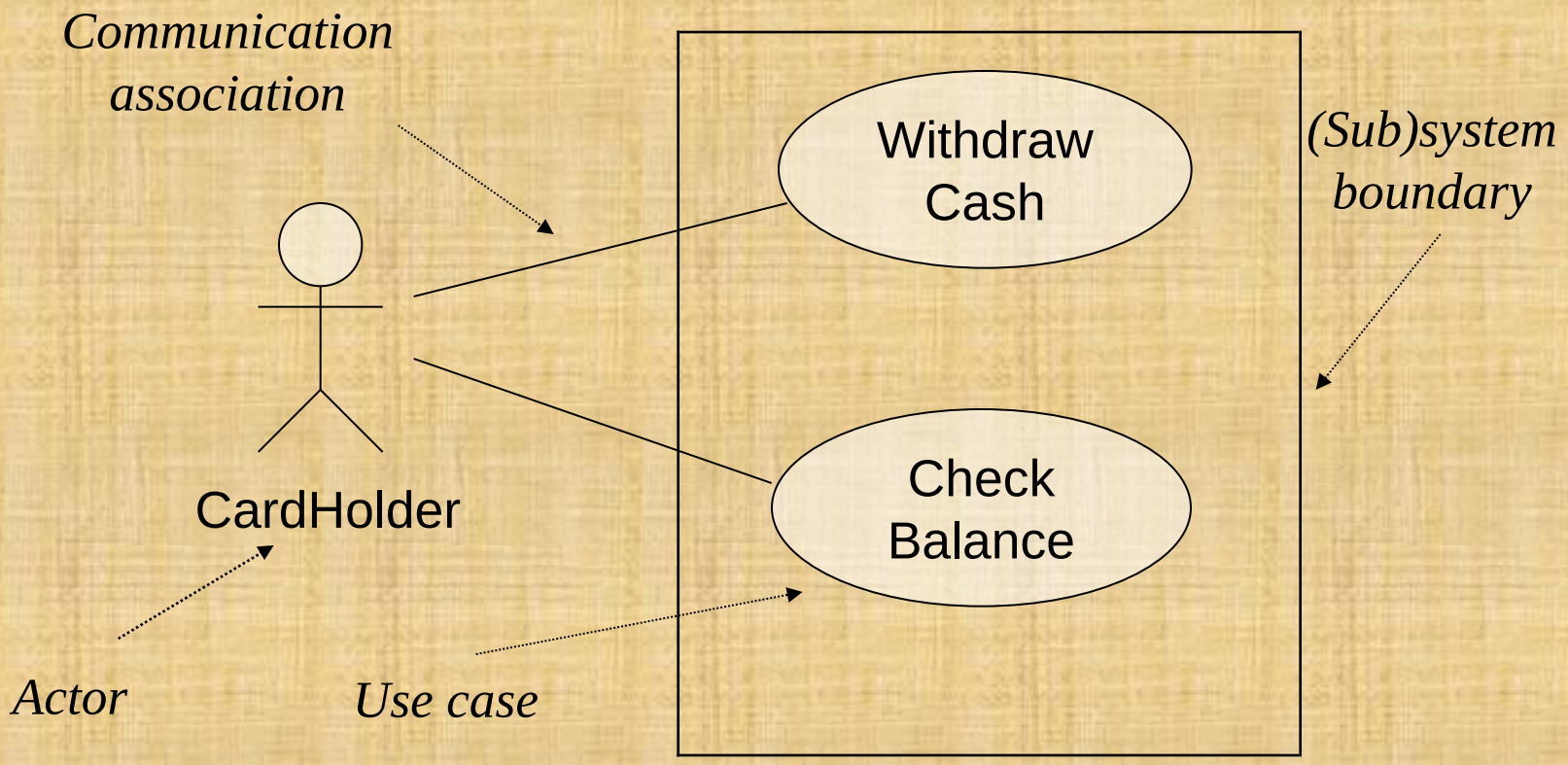
ATM Case Study: Requirements Summary

1. **To allow card holders to make transactions**
    1. To view and/or print account balances
    2. To make cash withdrawals
2. **To allow bank members to access special services**
    1. To order a statement
    2. To change security details
3. **To allow access to authorized bank staff**
    1. To re-stock the machine
4. **To keep track of how much money it contains**

Use Cases

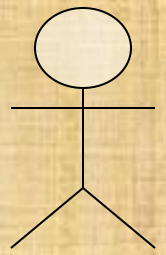
- Use cases used to model functionality:
  - What system does, not how
  - Focus on functionality from users' perspective
  - not appropriate for non-functional requirements
- UML use case diagrams:
  - Document system functionality
  - Useful for communicating with clients/users
  - Supported by more detailed descriptions of system behaviour (e.g. text documents, other UML diagrams)

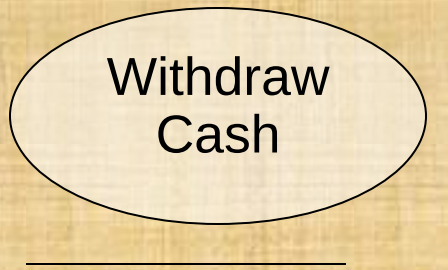
UML Use Case Diagrams



Basic Notation

- Use case diagrams show:
  - **Actors**: people or other systems interacting with system being modelled
  - **Use cases**: represent sequences of actions carried out by the system
  - **Communication:** between actors and use cases





Behaviour Specifications

- Document sequence of actions carried out by system to realize a use case
  - Other UML diagrams: e.g.
    - sequence diagrams
    - activity diagrams
  - Textual description:
    - use-case descriptions
    - use-case scenarios