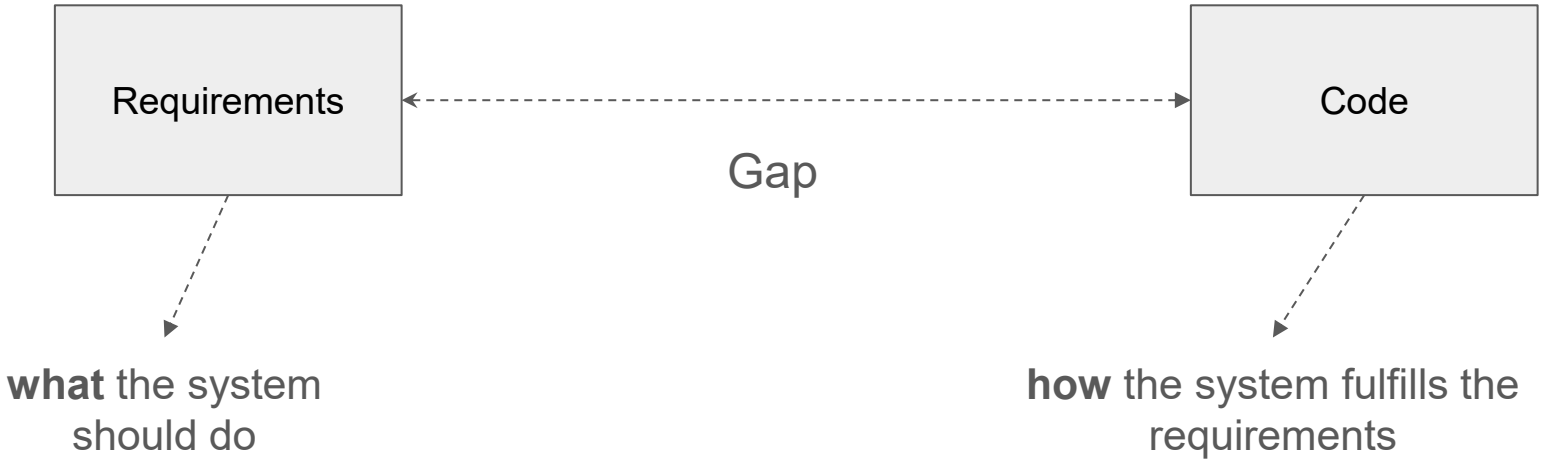
Software Engineering

UML part 1:
Use cases, Class and Sequence Diagrams

Motivation

- There is a gap between these two domains: requirements and code



Software design

- It is a **creative process** to transform users requirements into visualized form to describe **what** and **how** the system will be developed.
  - *a transition from "what" the system must do, to "how" the system will do it*
- The aim is to produce an implementable solution to a problem.
- An iterative process through which a customer’s requirements are translated into a blueprint for constructing the software code.
- All design work products must be traceable to the software requirements document (SRS)

Software design

- **Software Design :**
  - During the design process, a number of **artifacts** are produced including:
    - Conceptual Design (What) : meant for customers
      - *User Interface Designs.*
      - *Use-Case Diagrams*
    - Technical Design (How) : meant for developers.
      - *Structural Models*
      - *Behavioural Models*
      - *Architectural Designs*

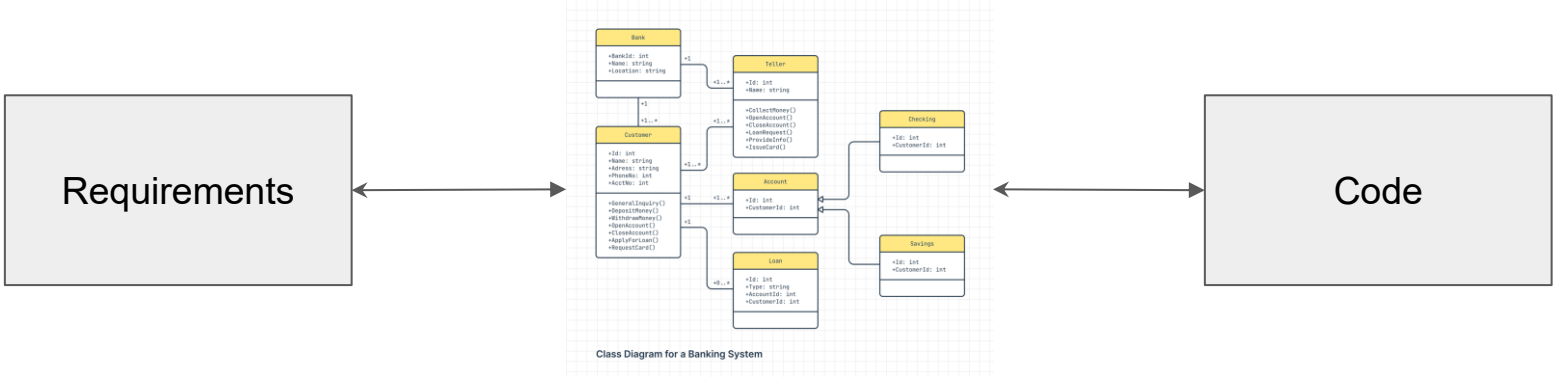
Software design

- **Why Software Modeling:**
  - Even provided the requirements, we cannot fully comprehend software products when it comes to their functionalities, structures and interactions.

  - Models can help to :
    - Clarify and visualize the software to be produced.
    - To specify the behaviour and structure of the software
    - Document decisions made for the development phase
    - Facilitate the communication between differents users.

Software Models

- Goal: bridge the gap between requirements and code
- Models document solutions to problems defined by requirements



Types of Software Models

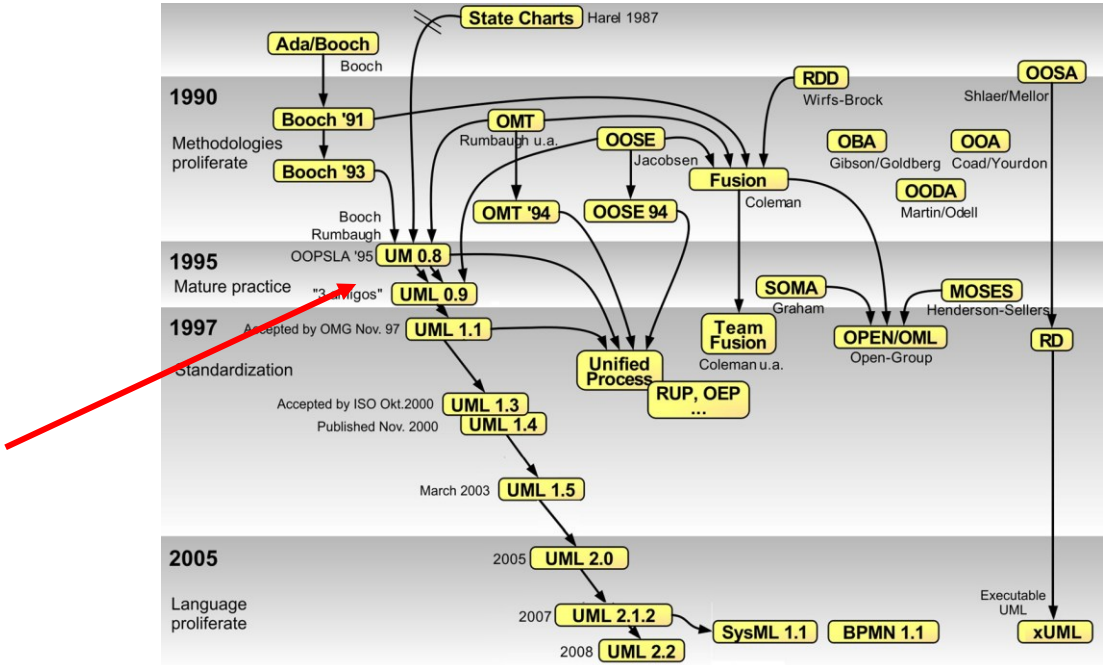
- Formal: less common and not covered in this course
- Graphical: UML is the most common notation

Types of Software Models

- **Software Modeling Languages:**
  - Many notations are developed over time
    - State machines
    - Entity-relationship diagrams
    - Data flow diagrams
    - ..
  - Most of them are graphical representations with some shapes, lines, arrows…

UML: Unified Modeling Language

- Proposed in 1995 to unify different modeling notations



Source: Wikipedia

UML

- **UML - Unified Modeling Language** :
  - is a general purpose notation that is used to
    - Visualize
    - Specify
    - construct and
    - Document the artifacts of :
      - Software-based systems
      - Business and similar processes,

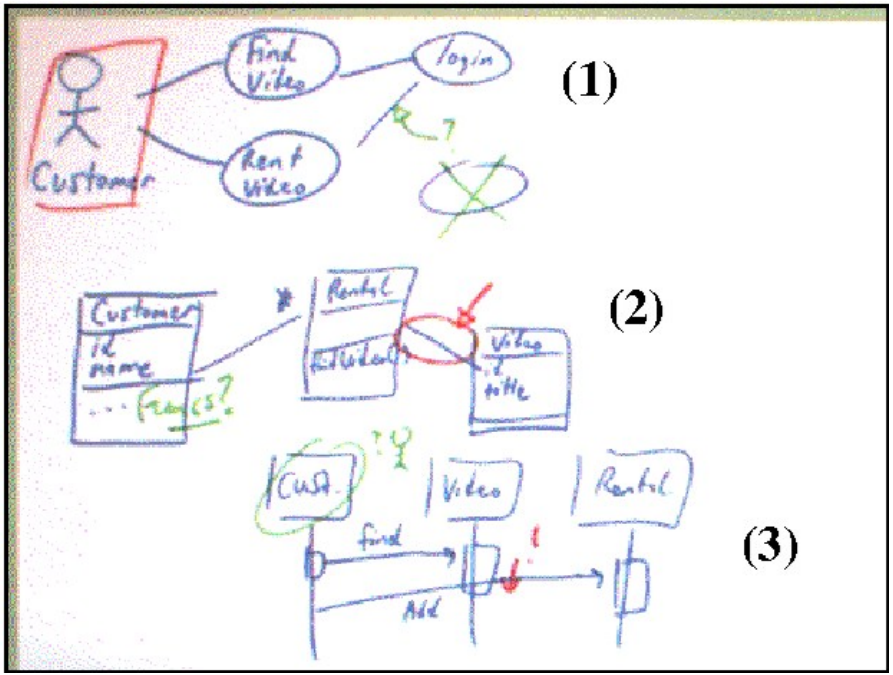
Main uses of UML

- As a blueprint (detailed plan)
- As a sketch (draft or outline)

UML as Sketch

- Common in agile methods
- Used to discuss or document parts of the design and code
- Lightweight and informal use of the notation
- The goal is not to create a complete, blueprint-style model

UML as Sketch



Q. Chen, J. Grundy, J. Hosking: SUMLOW: early design-stage sketching of UML diagrams on an E-whiteboard. Software Practice and Experience, 2008

UML sketches are useful in

both forward and reverse engineering

Forward Engineering

- Sketches are used to discuss alternative designs
- Before any code is written

Reverse Engineering

- Sketches are used to explain existing code
- Context: Software maintenance and evolution

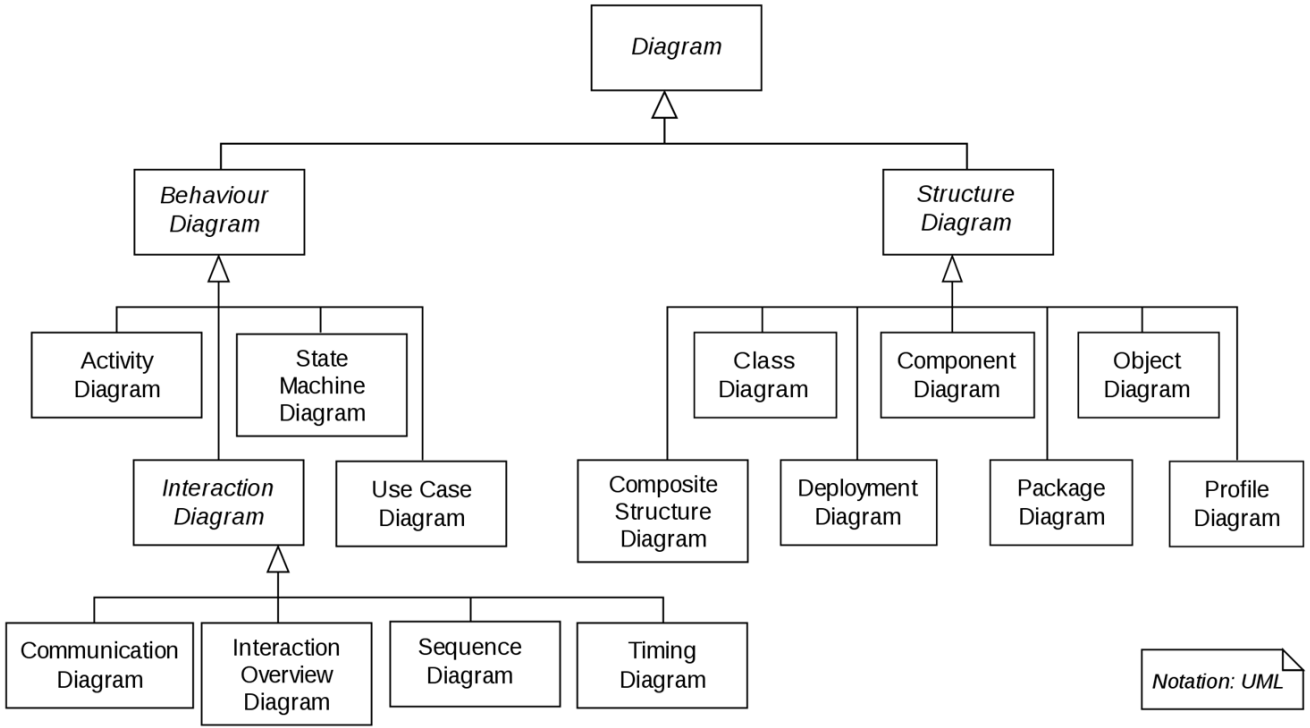
UML Diagrams

UML Diagrams

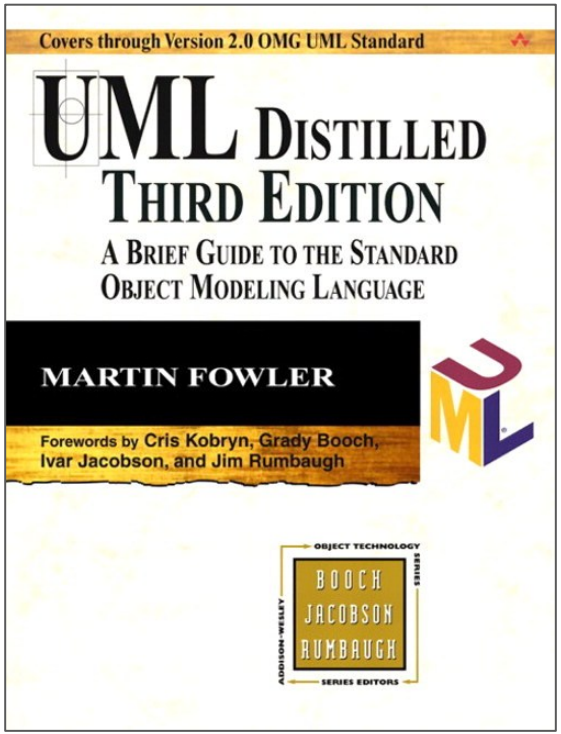
- Static Diagrams: model the code's structure
- Dynamic Diagrams: model the system's behavior at runtime

UML Diagrams

The diagrams shown in red are the ones we will study



We will use the UML version presented in this book



Use Cases Diagram

Use Case

- “A use case specifies the behavior of a system or a part of a system, and is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor.”- The UML User Guide, [Booch,99]

- “An actor is an idealization of an external person, process, or thing interacting with a system, subsystem, or class. An actor characterizes the interactions that outside users may have with the system.” - The UML Reference Manual, [Rumbaugh,99]

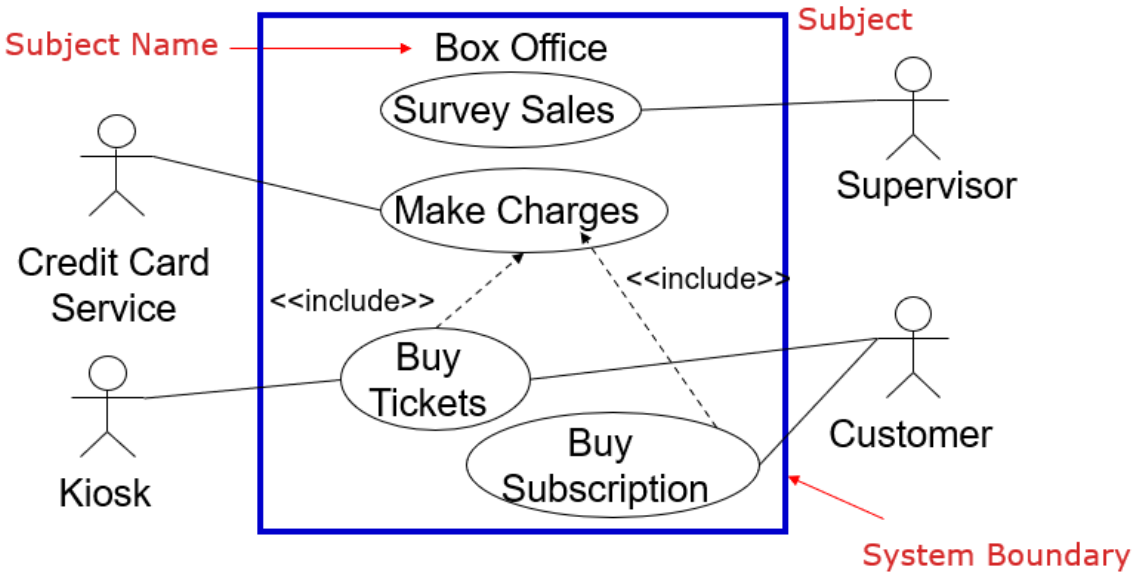
Use Cases Diagram

- Specifies the behavior of a system
- Sequence of actions to yield an observable result of value to an actor
- Capture the intended behavior (the what) of the system omitting the implementation of the behavior (the how)
- Customer requirements/ early analysis

Use Cases Diagram

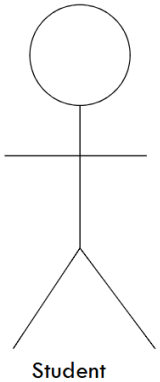
- **Elements of the Use Case Diagram** :
  1. Systems
  2. Actors
  3. Use Cases
  4. Relationships

Use Cases Diagram – System



- **Subject Symbol**
  - Indicate system boundary
  - System is represented as a rectangle with its name at the top

Use Cases Diagram - Actor



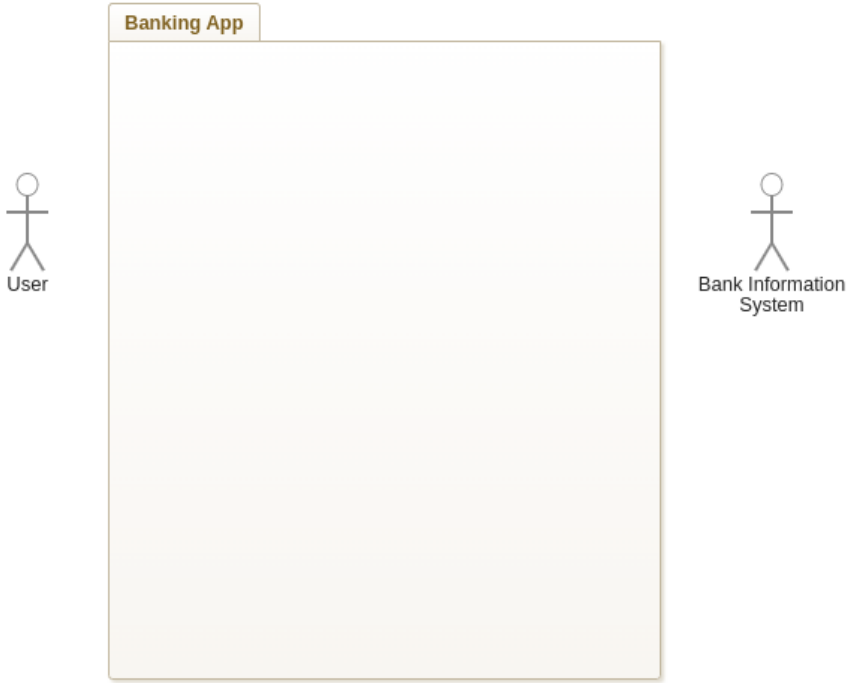
An actor is rendered as a stick figure in a use case diagram. Each actor participates in one or more use cases.

Use Cases Diagram - Actor

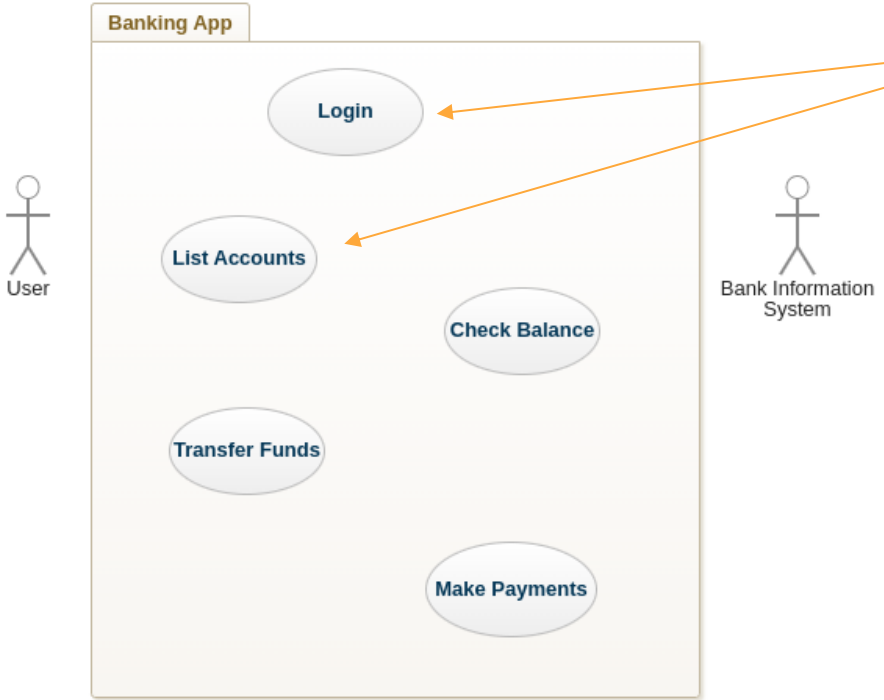
- **Actors :**
  - Two types of Actors:
    - **Primary Actors** : initiates the use of the system.

      Placed on the left side
    - **Secondary Actors** : are reactionary Actors where the system initiates interaction with them.

      Placed on the right side.



Use Cases Diagram – use case

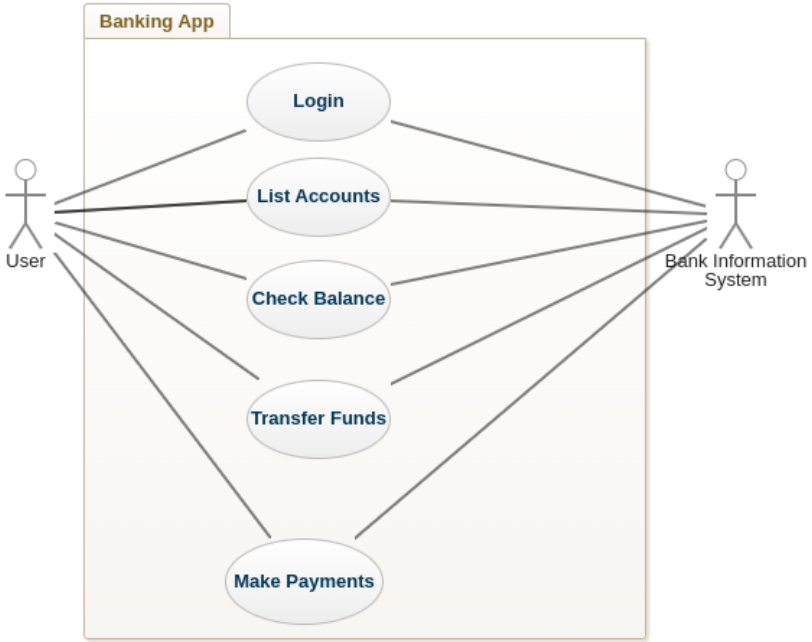


A use case :

- Represents a high-level action/feature to be provided by the system.
- Depicted by an Oval shape
- Each Use Case has a unique label

  (usually starting with a verb)

Use Cases Diagram – Relationships



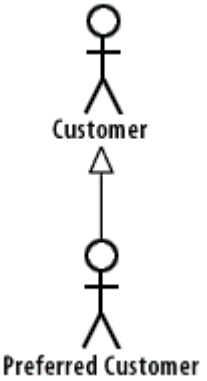
Association

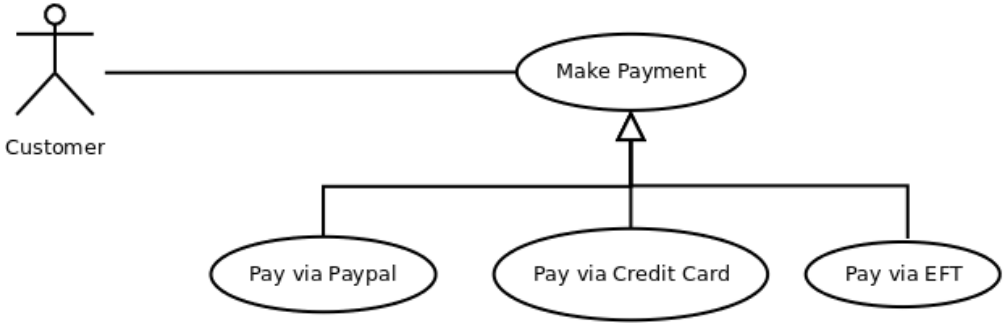
- Represent bi-directional communication between the actor and the system
- Represented as a solid line
- Drawn between an actor and a use case

Use Cases Diagram – Relationships

Generalization

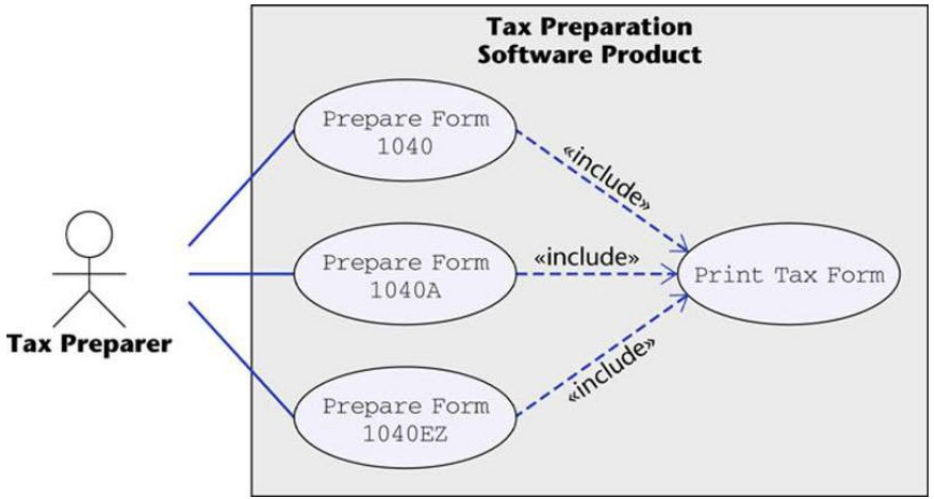
- Called also Inheritance, to show specialized use cases from the general base case.
- Depicted by an arrowed line.
  - Use Cases
  - Actors





Use Cases Diagram - Relationships

Include: a dotted line labeled <<include>> beginning at base use case and ending with an arrows 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.

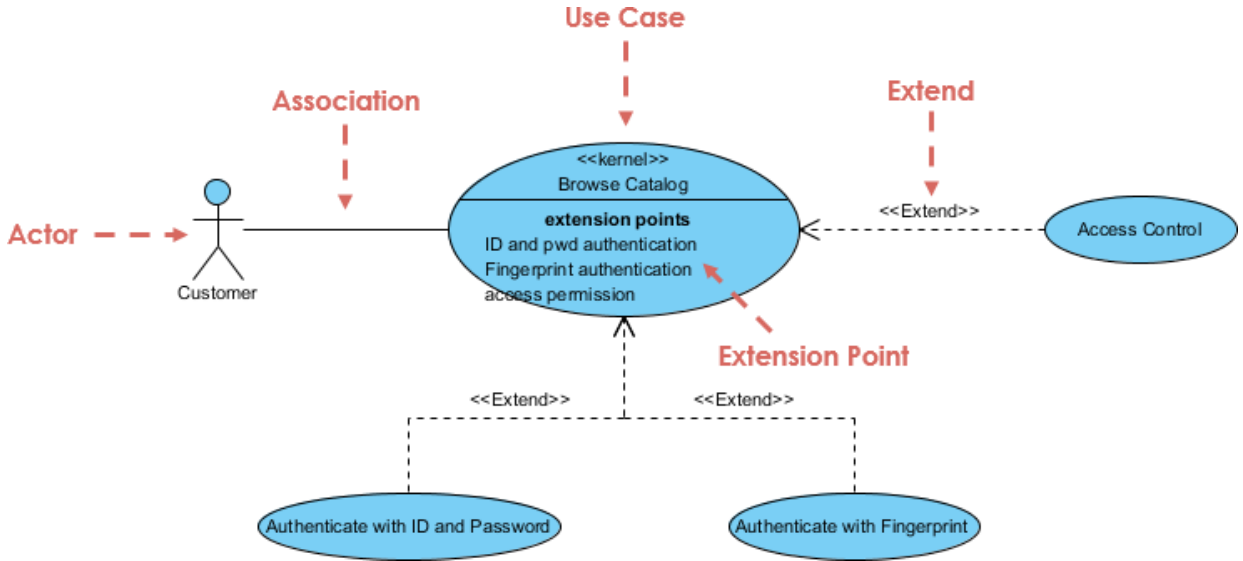


<<include>>

Use Cases Diagram - Relationships

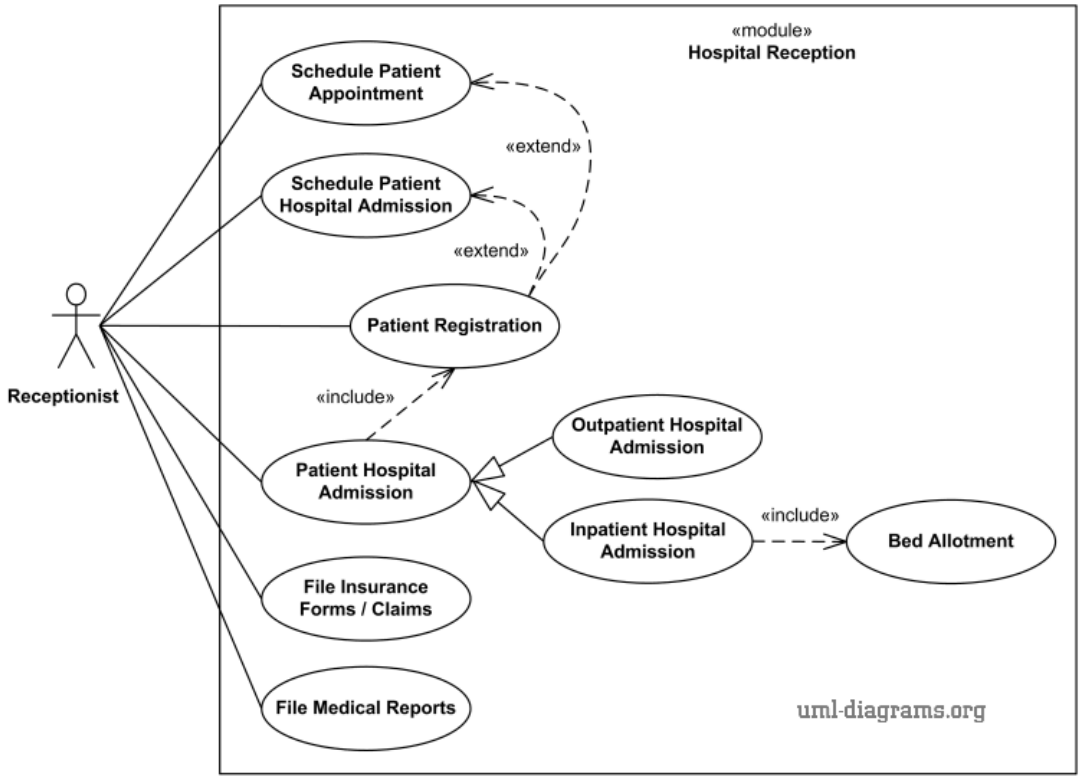
Extend: a dotted line labeled <<extend>> with an arrow toward the base case.

Extend **means the base use case can optionally be extended by another use case** — i.e., the “extended” use case may add extra behavior under certain conditions. The base class declares “extension points”.

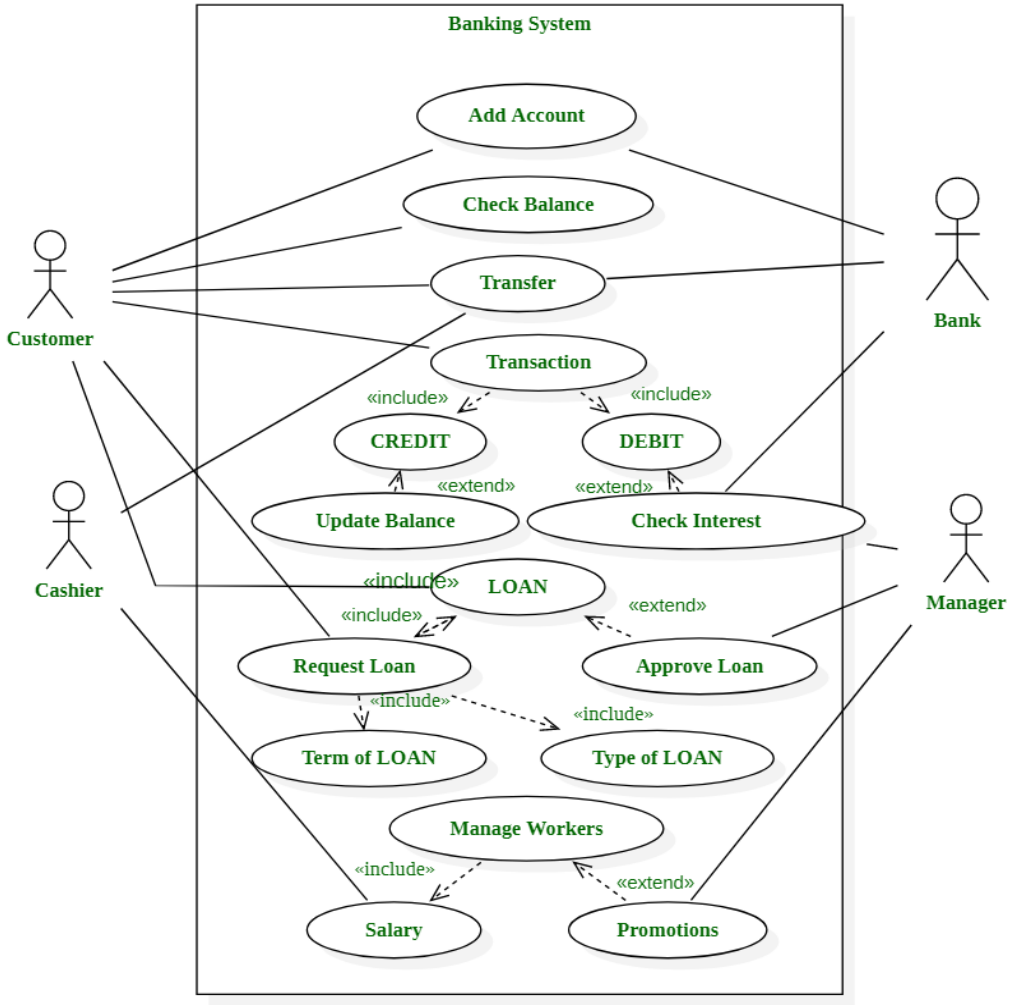


- Make sure that each use case describes a significant chunk of system usage that is understandable by both domain experts and programmers
- When defining use cases in text, use nouns and verbs accurately and consistently to help derive objects and messages for interaction diagrams
- Factor out common usages that are required by multiple use cases
    - If the usage is required use <<include>>
    - If the base use case is complete and the usage may be optional, consider use <<extend>>
- A use case diagram should
    - contain only use cases at the same level of abstraction
    - include only actors required
- Large numbers of use cases should be organized into packages

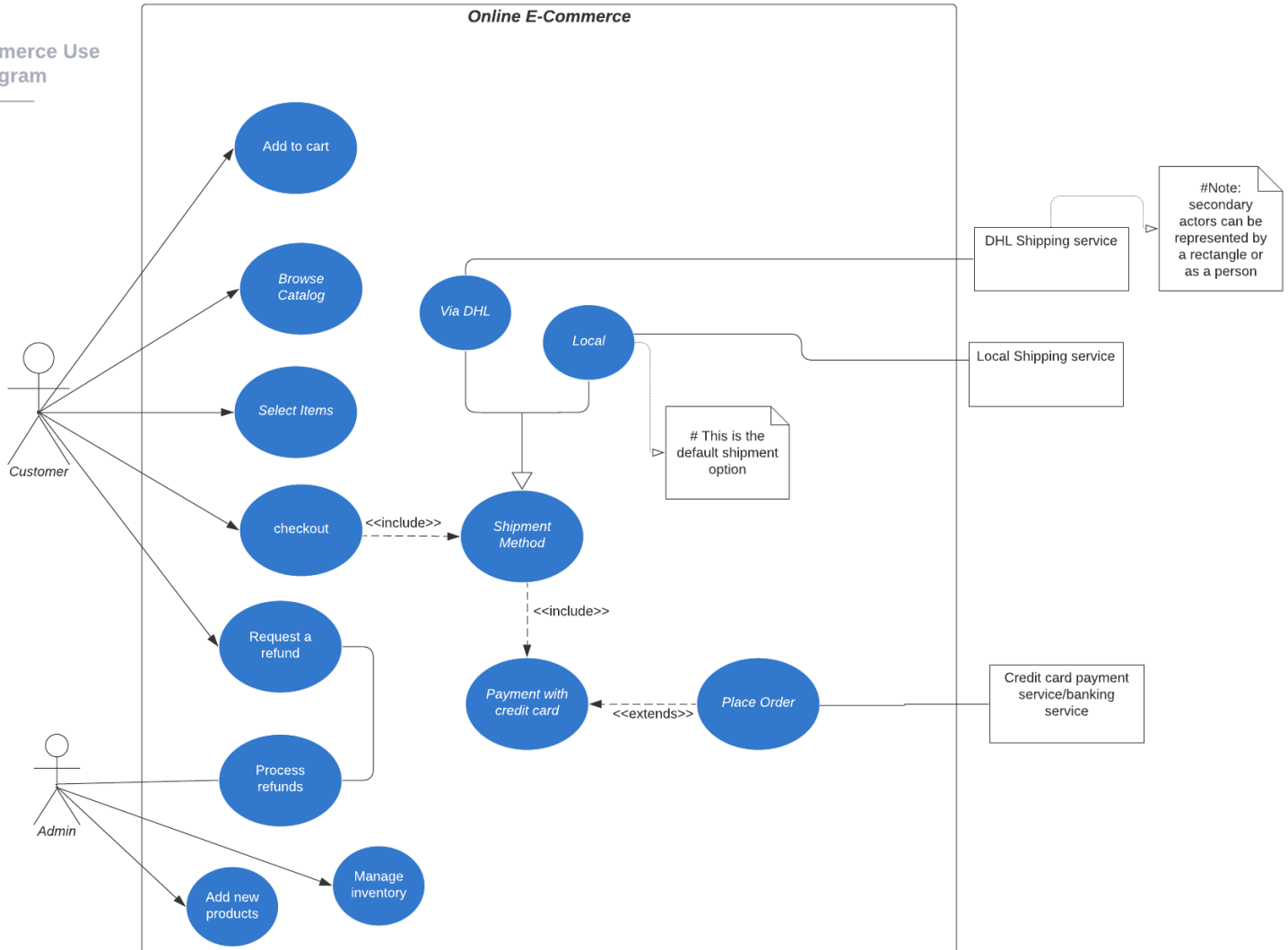
Use Cases Diagram - Example



Use Cases diagram



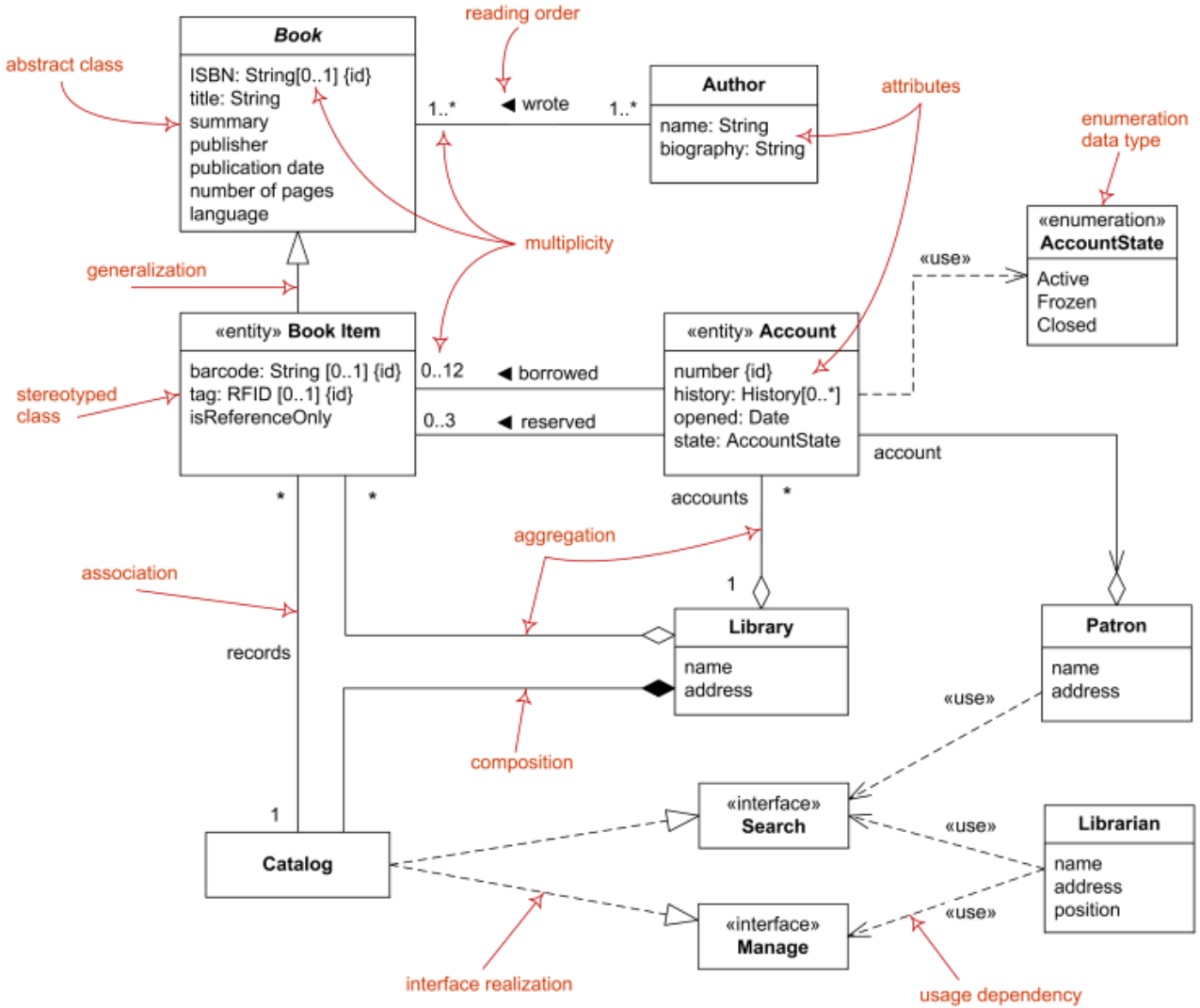
Use Cases diagram



Class Diagrams

Class Diagram

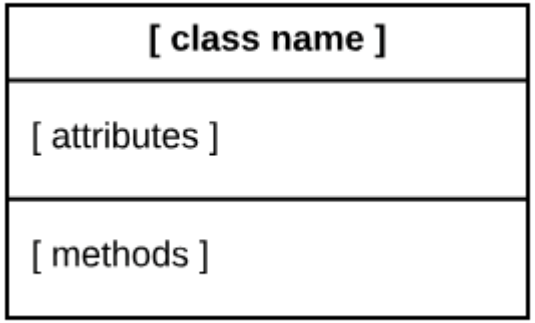
- Description of static structure
  - Showing the types of objects in a system and the relationships between them


Book
ISBN: String[0..1] {id}
title: String
summary
publisher
publication date
number of pages
language
abstract class
reading order
1..*
wrote
1..*
Author
name: String
biography: String
attributes
enumeration
data type
«enumeration»
AccountState
Active
Frozen
Closed
«use»
multiplicity
generalization
«entity» Book Item
barcode: String [0..1] {id}
tag: RFID [0..1] {id}
isReferenceOnly
stereotyped
class
0..12
borrowed
0..3
reserved
«entity» Account
number {id}
history: History[0..*]
opened: Date
state: AccountState
account
accounts
*
*
*
1
aggregation
association
Library
name
address
Patron
name
address
records
composition
«use»
1
«interface»
Search
«use»
Librarian
name
address
position
Catalog
«interface»
Manage
«use»
interface realization
usage dependency

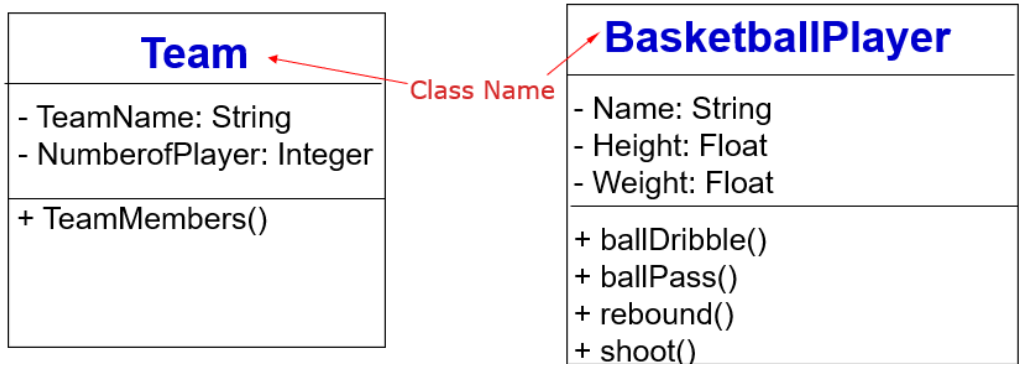
Classe Diagram

- **Elements of the Class Diagram**
  - Boxes:
    - Classes with Attributes and Operations
  - Relationships:
    - Association
    - Generalization/Inheritance
    - Composition/Aggregation
    - Dependence

Class Diagram - Classes

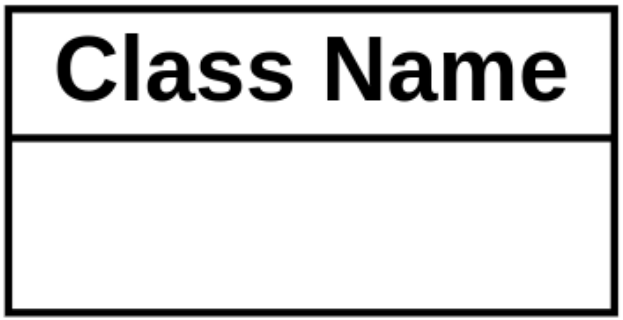


- Most important building block of any object-oriented system
- Description of a set of objects
- Abstraction of the entities
- Existing in the problem/solution domain
- Depicted as class-box



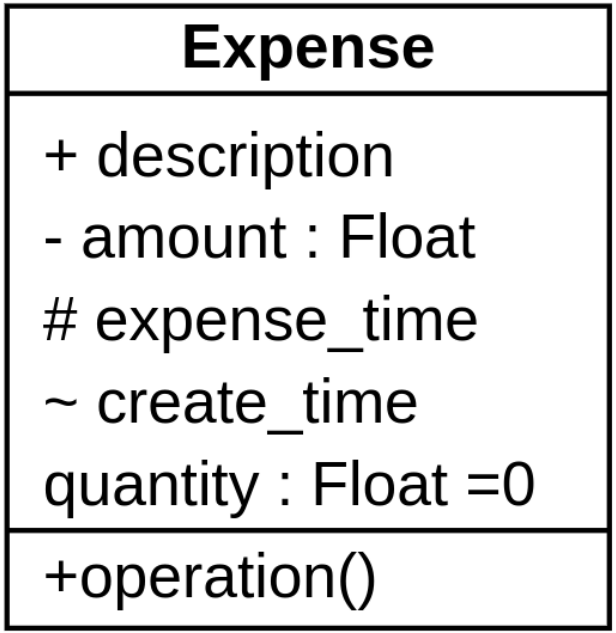
Class Diagram - Classes

- **Drawing the Class Diagram : Classes**
  - For conceptual design without details, the class box can be drawn with only two sections.



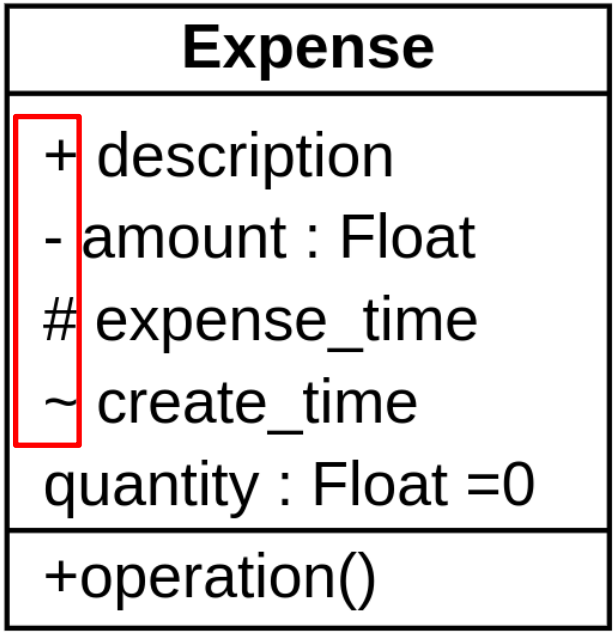
Class Diagram – Attributes

- **Attributes**
  - The attributes are the adjectives and properties of a given entity.
  - In programming language : **Instance Variables**
  - Within the attributes section, we can specify :
    - Visibility Scope.
    - Type of the attribute :
    - Default value



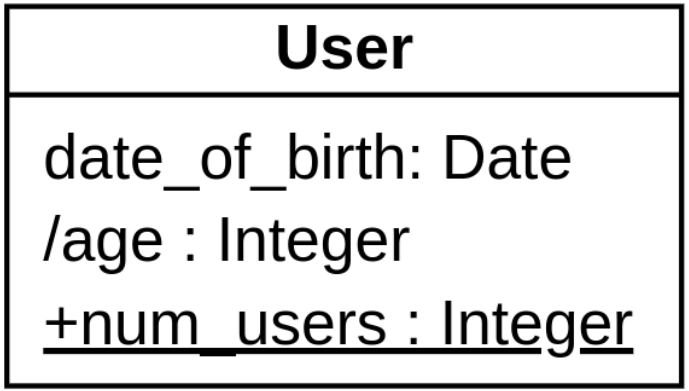
Class Diagram – Attributes

- Visibility Scope : relates to the level of information hiding to be enforced when accessing the attribute
- ***+ : Public*** *: it can be access from anywhere.*
- ***# : Protected*** *: Accessible to subclasses only.*
- ***- : Private*** *: Not accessible except from the class only*
- ***~ : Package*** *: Accessible from classes of the same package.*
- *Default (no symbol) usually :* ***private***



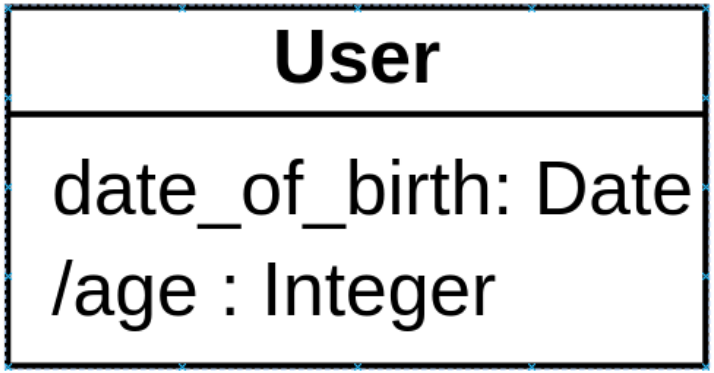
Class Diagram – Attributes

- **Static attributes** or global variables that belong to the class and shared between all objects of the same class, are represented by an **underline**.
  - num_users
- For constant variables, conventionally, the variable name is written in capital letters



Class Diagram – Attributes

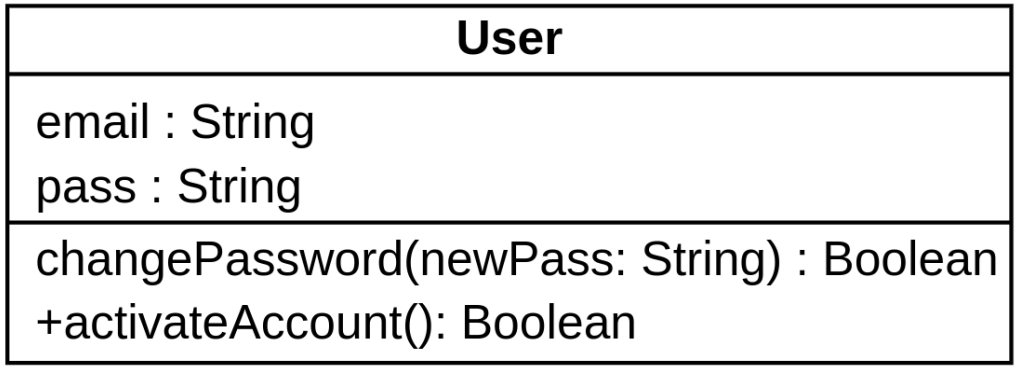
- Attributes can be non-stored and computed during their time of retrieval.
  - Example : Age of the person.
  - To indicate that an attribute is computed and non-stored, **the symbol /** is used just before the attribute name.



Class Diagram – Operations and Methods

Operations and Methods

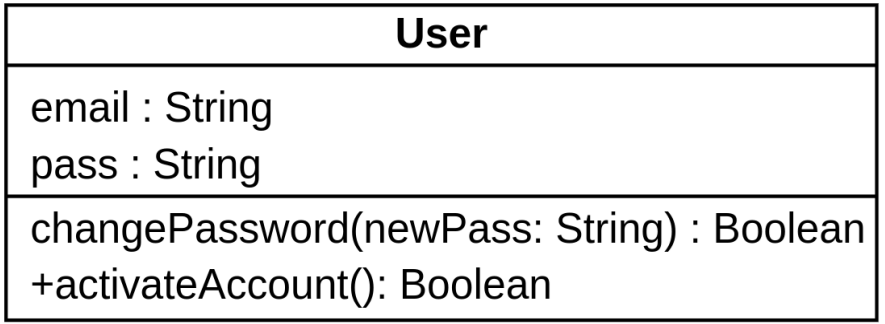
- Are action, functions or operations that can be performed by a given class.
- Class Operation can be specified by :
  - Visibility modifier
  - Operation name
  - Passed Parameters
  - Return Type



Class Diagram – Operations and Methods

Operations and Methods

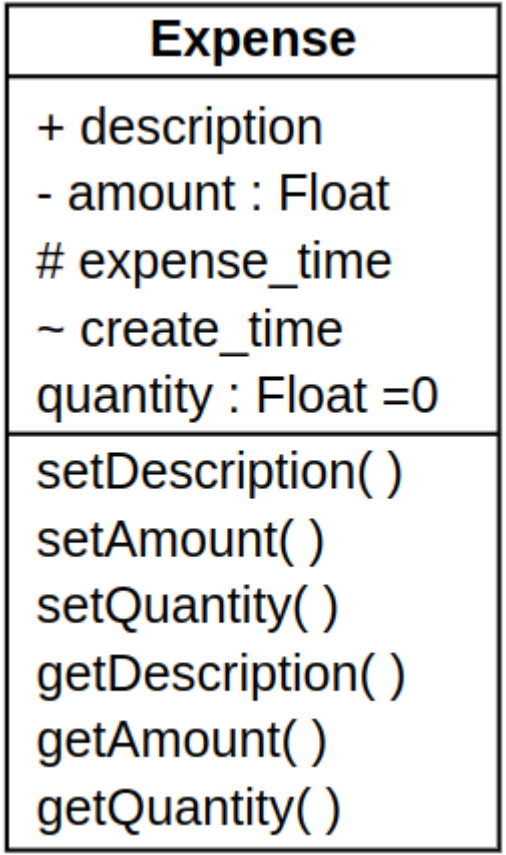
- Are action, functions or operations that can be performed by a given class.
- Class Operation can be specified by :
  - Visibility modifier
  - Operation name
  - Passed Parameters
  - Return Type



Class Diagram – Operations and Methods

Operations and Methods

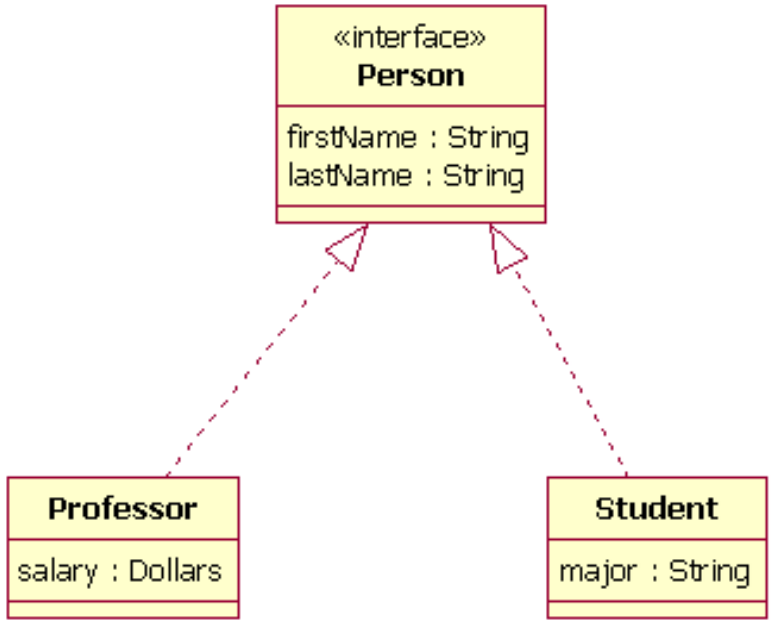
- Can we add the setters and getters ?
- Better not : The aim of UML diagrams is to simplify and convey our understanding to how we would design the system not to care about non-architectural details



Class Diagram – Interface

Interface

- Is composed of :
  - Attributes
  - Operations (Signatures) that need to be implemented inside a class
- The Interface is drawn the same way as the class but with annotation word <<interface>> above the interface name
- The interface **must have** at least one class to implement its operation



Class Diagram – Finding the Classes

- Read the **problem description**, **use cases**, and **data requirements** carefully.
  → Look for **nouns** and **noun phrases** — they often represent **candidate classes**.

- Example:
  “A *student* borrows a *book* from the *library*.”
  Candidate classes: Student, Book, Library

Class Diagram – Finding the Classes

Heuristic	Description	Example
Noun Extraction	Each noun in the requirements text is a potential class or attribute.	“User submits a form” → `User`, `Form`
Domain Concepts	Identify real-world entities the system must represent.	`Customer`, `Order`, `Invoice`
Responsibilities	Look for entities that have behavior or store information.	`Payment` (handles processing)
Avoid Functional Concepts	Ignore verbs, operations, or UI elements.	`Print`, `Click`, `Save` (avoid)
Refine Attributes	Some nouns are just attributes, not classes.	“Student has a *name*” → `name` is an attribute
Check Redundancy	Merge synonyms or overlapping concepts.	`Customer` vs. `Client`

Class Diagram – Finding the Classes

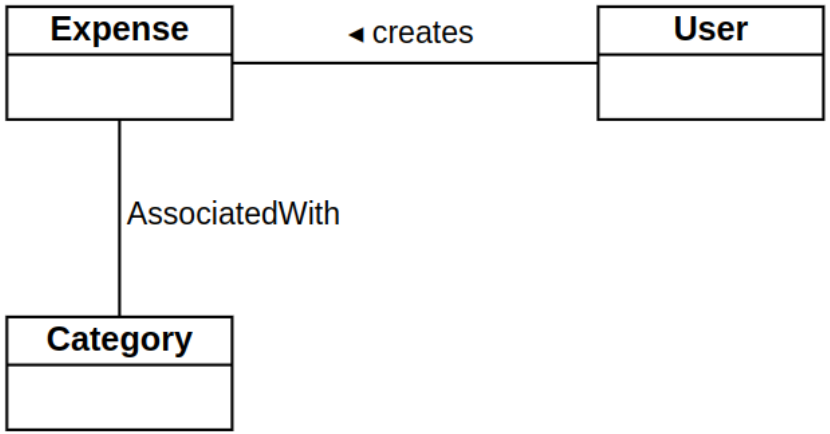
- **Example - Requirement fragment:**

“A customer places an order that contains several items. The system calculates the total and generates an invoice.”

- **Step 1 – Extract nouns:**
  Customer, Order, Item, System, Total, Invoice
- **Step 2 – Filter:**
  Keep: Customer, Order, Item, Invoice
  Drop: System (too generic), Total (attribute)
- **Resulting classes:**
  Customer, Order, Item, Invoice

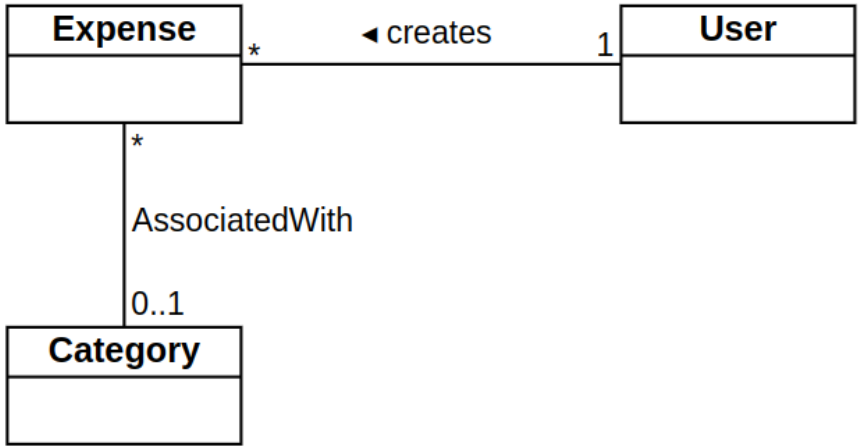
Class Diagram – Relationships

- **Simple Association**
  - Represents a structural link between two classes.
  - Drawn as a simple line
  - Optionally labeled with either the name of the relationship or the roles that the classes play in the relationship
  - An arrow/Triangle can be added to improve the readability of the association



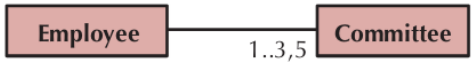
Class Diagram – Relationships

- **Simple Association**
- Multiplicity symbols : represent the minimum and maximum times a class instance can be associated with the related class instance



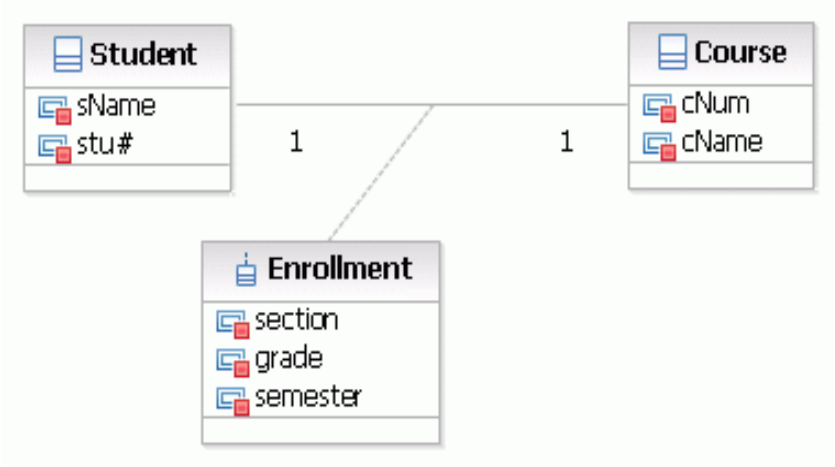
Class Diagram – Relationships

- **Simple Association**
- Multiplicity symbols : represent the minimum and maximum times a class instance can be associated with the related class instance

Exactly one	1		A department has one and only one boss.
Zero or more	0..*		An employee has zero to many children.
One or more	1..*		A boss is responsible for one or more employees.
Zero or one	0..1		An employee can be married to zero or one spouse.
Specified range	2..4		An employee can take from two to four vacations each year.
Multiple, disjoint ranges	1..3,5		An employee is a member of one to three or five committees.

Class Diagram – Relationships

- **Association Class**
  - It is a class which associates to other classes for the case when the relationship needs to have further attributes or information.
  - Linked to the association solid line by a dashed line
  - Association class is drawn as a normal class with attributes and operations.

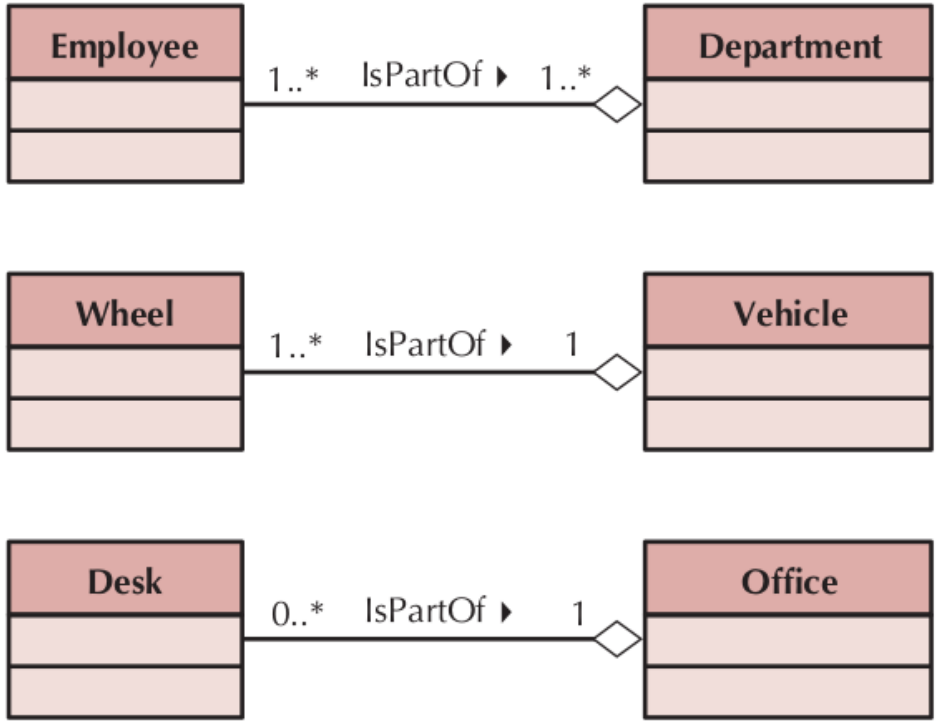


- You as a student, you associated directly with the School with no extra data needed? No
- The association needs further information like registration year, academic year, status…

Class Diagram – Relationships

Aggregation

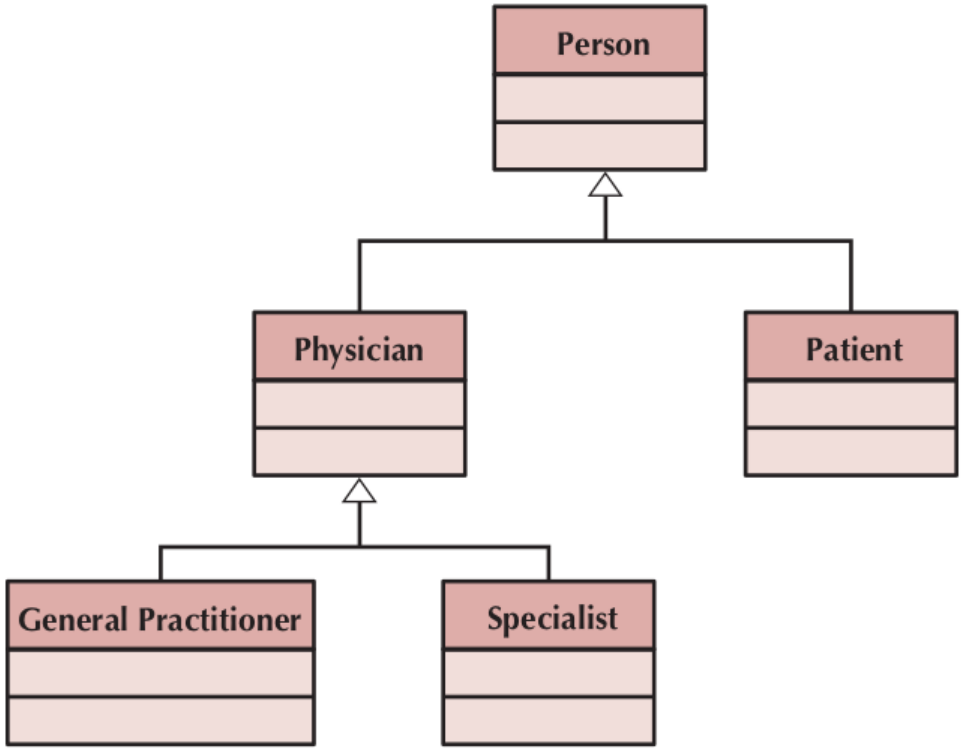
- Represents a logical a-part-of relationship between multiple classes or a class and itself.
- Logical implies:
  - *Whole Class can be removed without impacting the composing -Classes*
  - *Composing-classes can compose different whole classes at the same time (Multiplicity can be *)*



Class Diagram – Relationships

Inheritance or Generalization

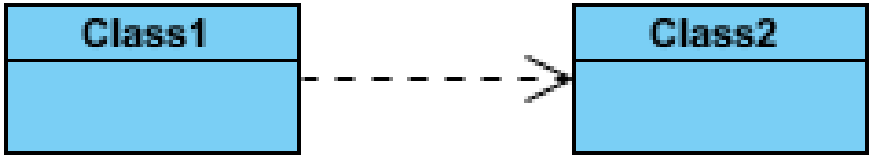
- A generalization association shows that one class (subclass) inherits from another class (superclass):
  - meaning that the properties and operations of the superclass are also valid for objects of the subclass.
- The generalization path is shown with a solid line from the subclass to the superclass and a hollow arrow pointing at the superclass



Class Diagram – Relationships

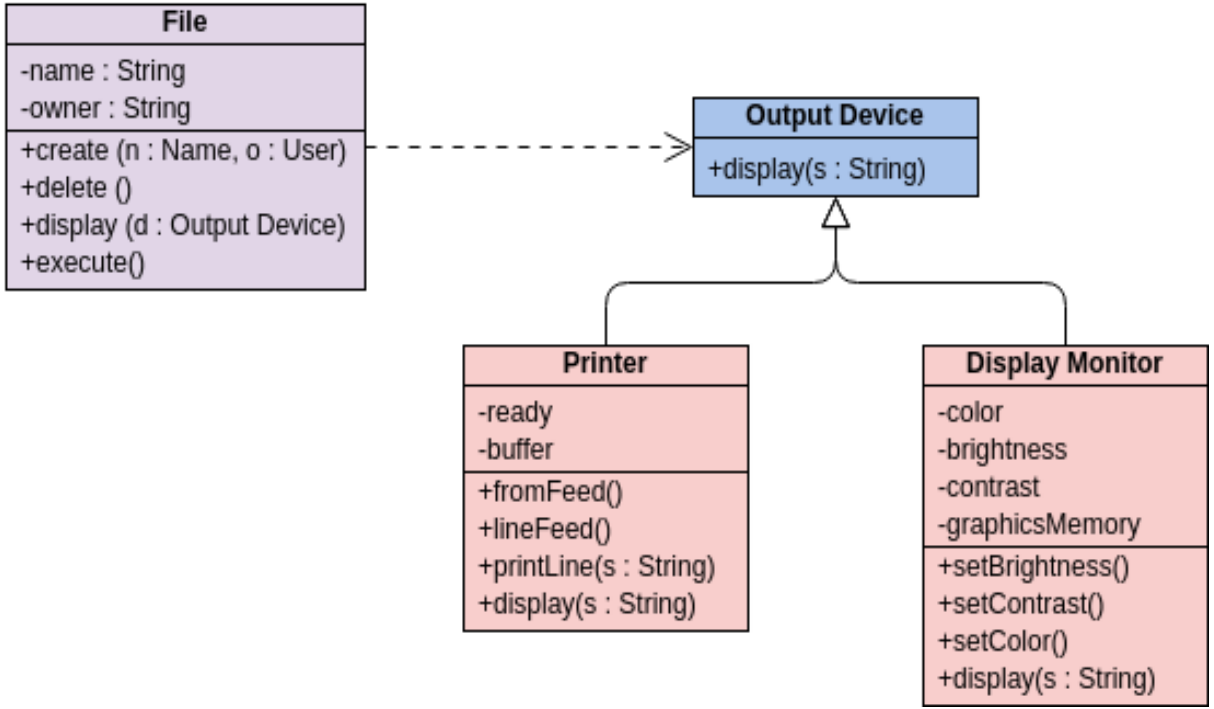
Dependencies

- is a directed relationship which is used to show that some UML element or a set of elements **requires**, **needs** or **depends** on other model elements for specification or implementation.
- If the changes to the definition of one may cause changes to the other (but not the other way around).
- Drawn as dashed-line arrow
  - Class 1 depends on Class 2

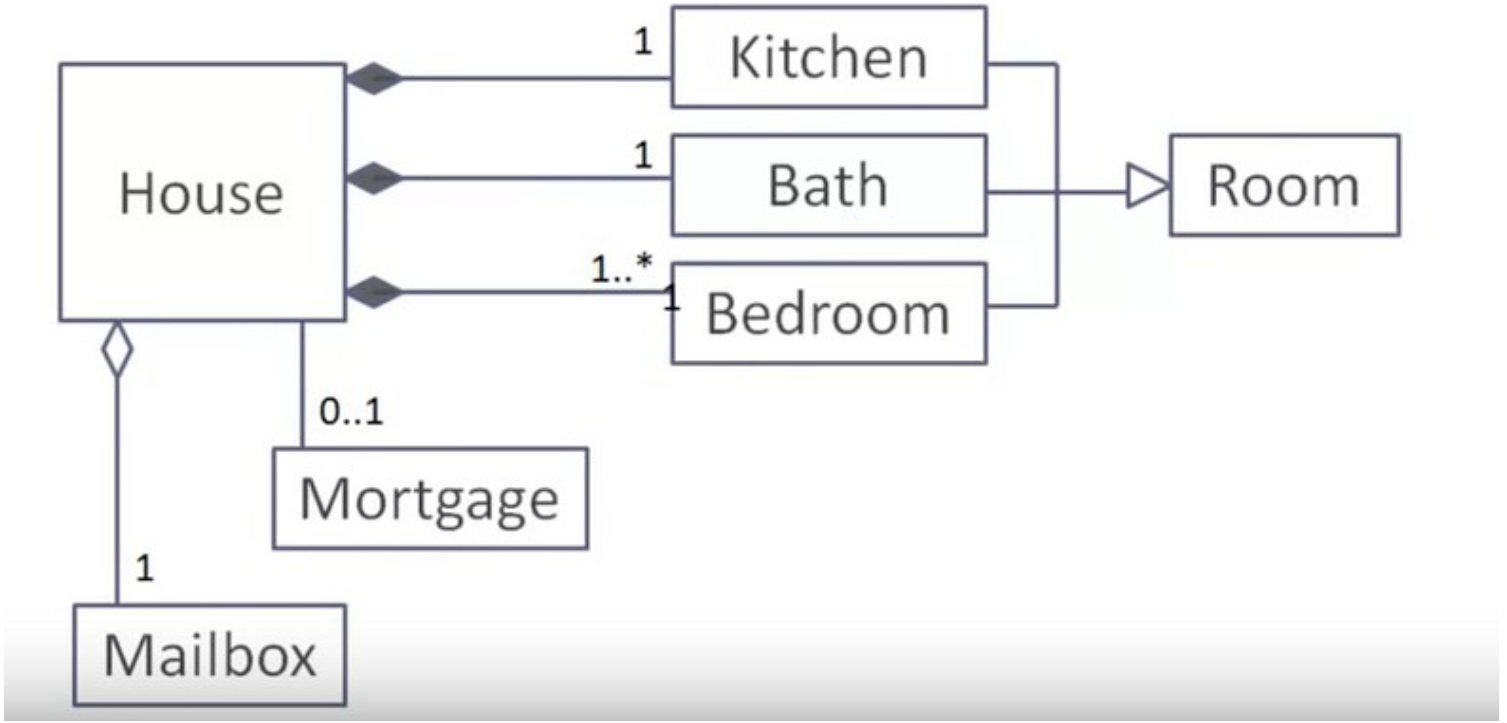


Class Diagram – Relationships

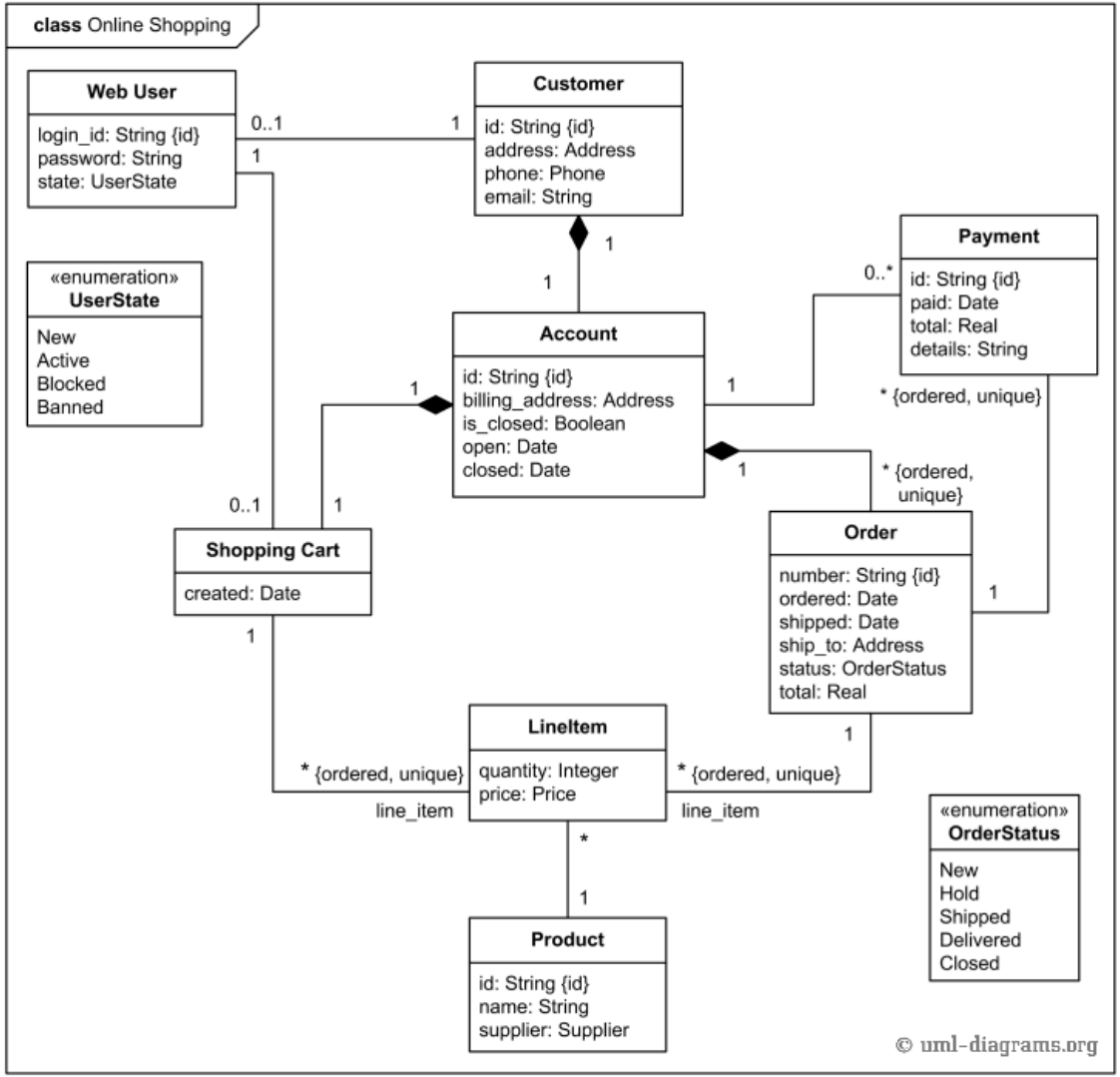
Dependencies



Class Diagram – Examples

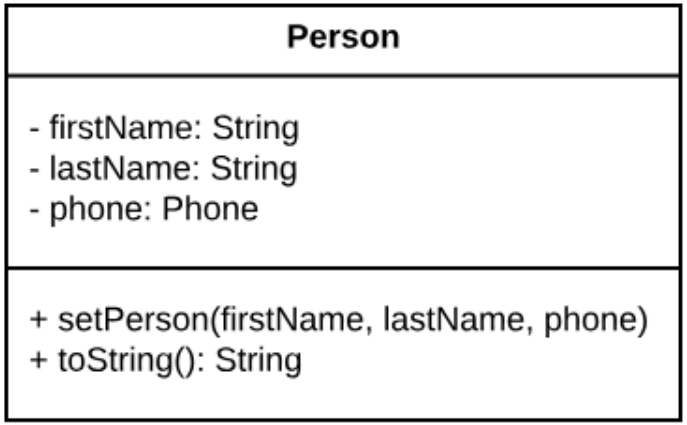


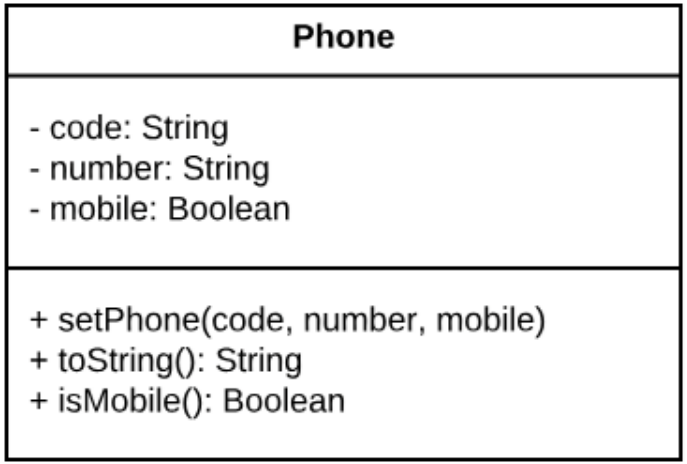
Class Diagram Examples



Interpreting UML Class Diagrams in Code

Example



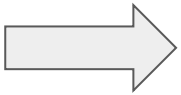


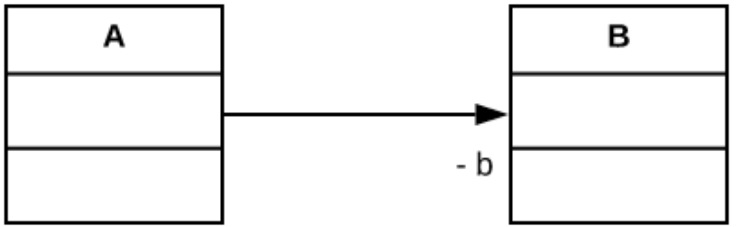
- : private
+: public

Association

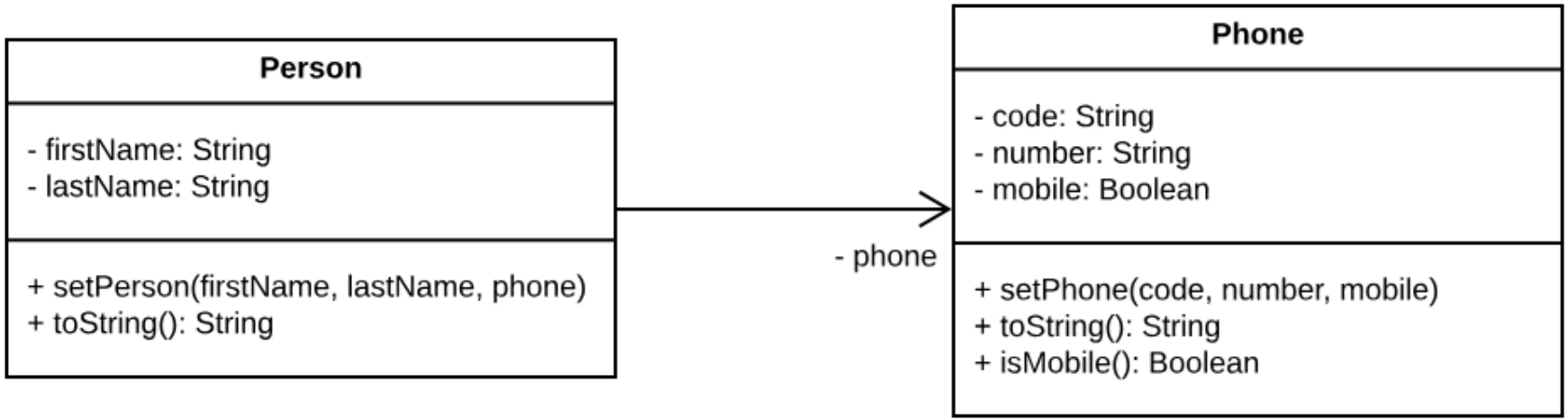
```
class A {
    ...
    private B b;
    ...
}


class B {
    ...
}
```

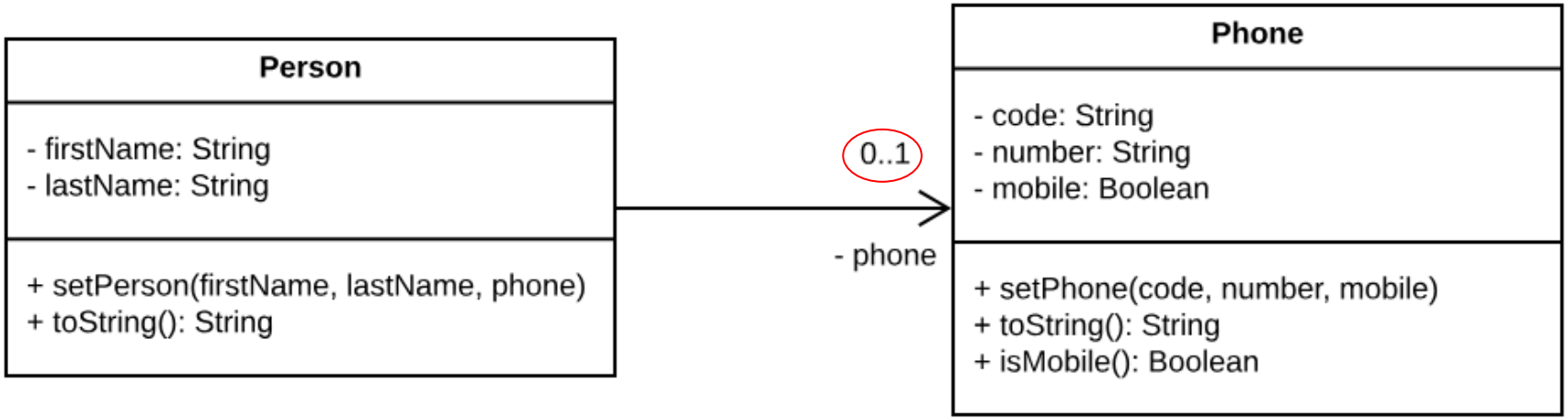




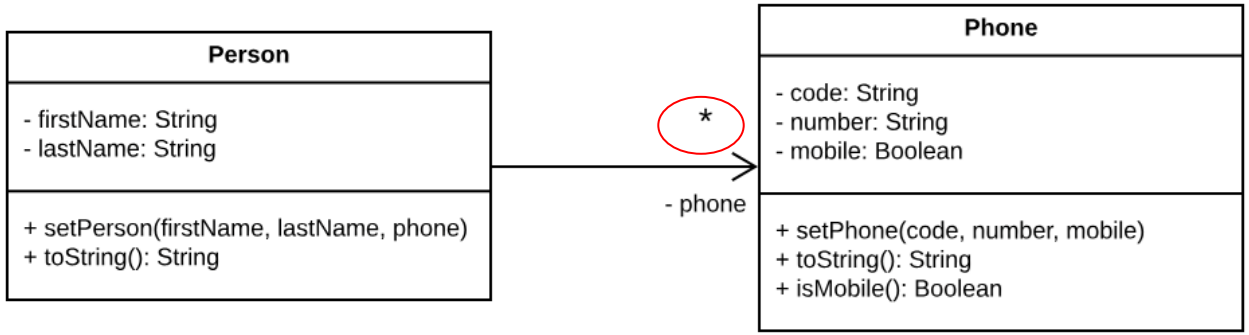
Association



Multiplicity



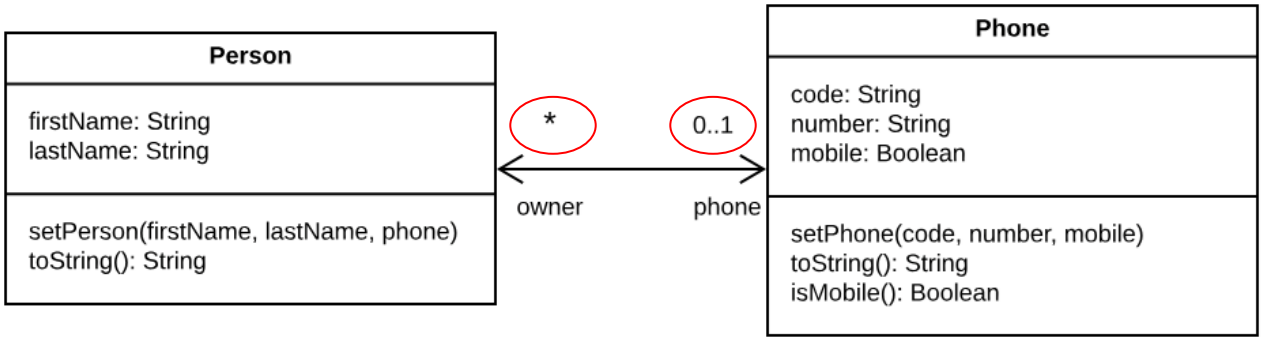
Multiplicity




```
class Person {
   private Phone[] phone;
   ...
}

class Phone {
   ...
}
```

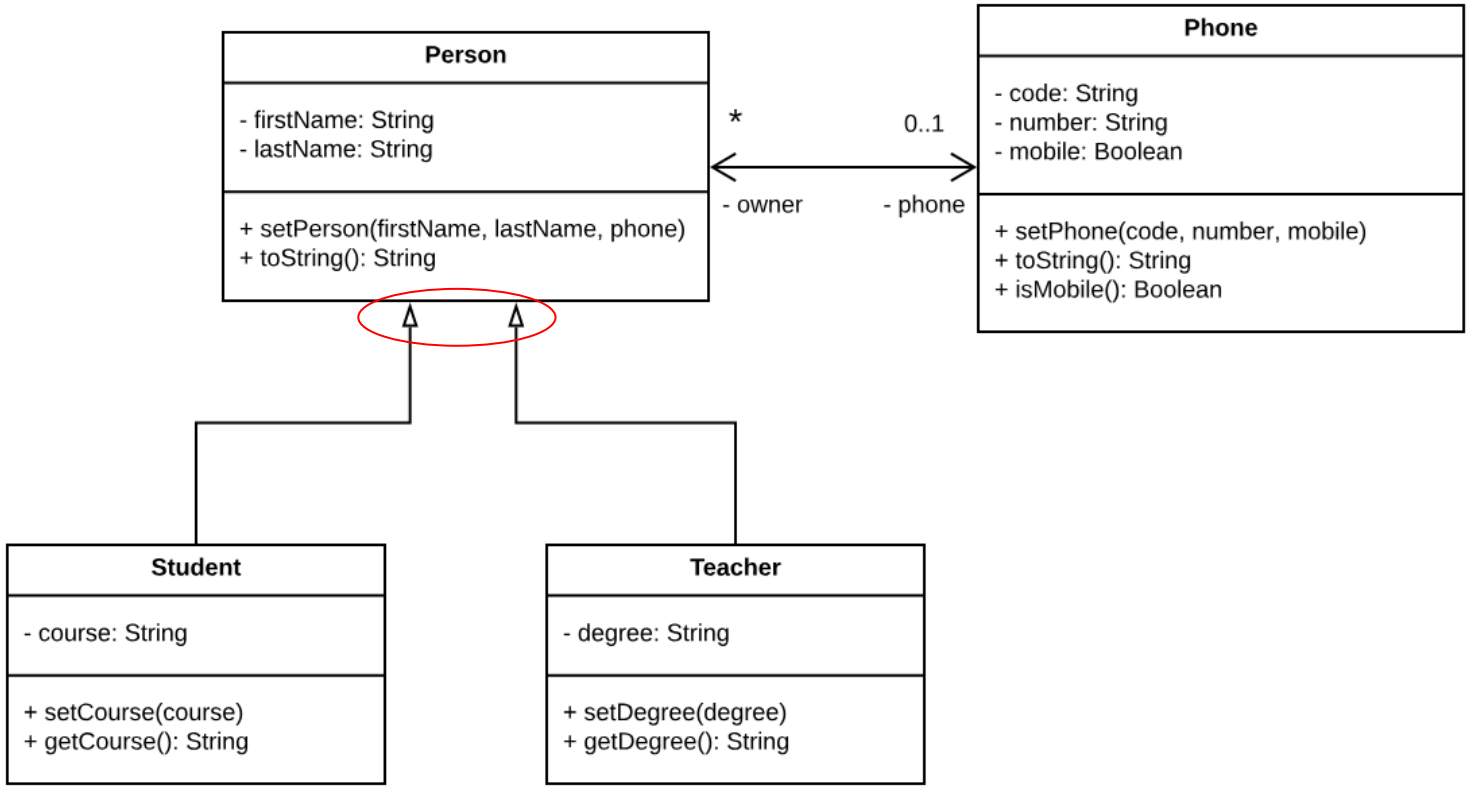
Bidirectional Associations




```
class Person {
   private Phone phone;
   ...
}

class Phone {
   private Person[] owner;
   ...
}
```

Inheritance

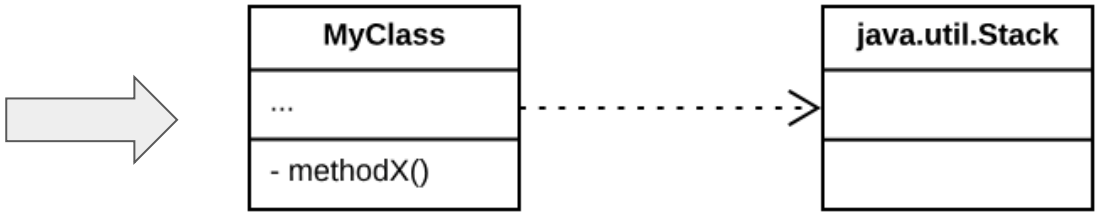


Dependencies (dashed arrows)

Represent relationships between classes that are not modeled as associations or inheritance

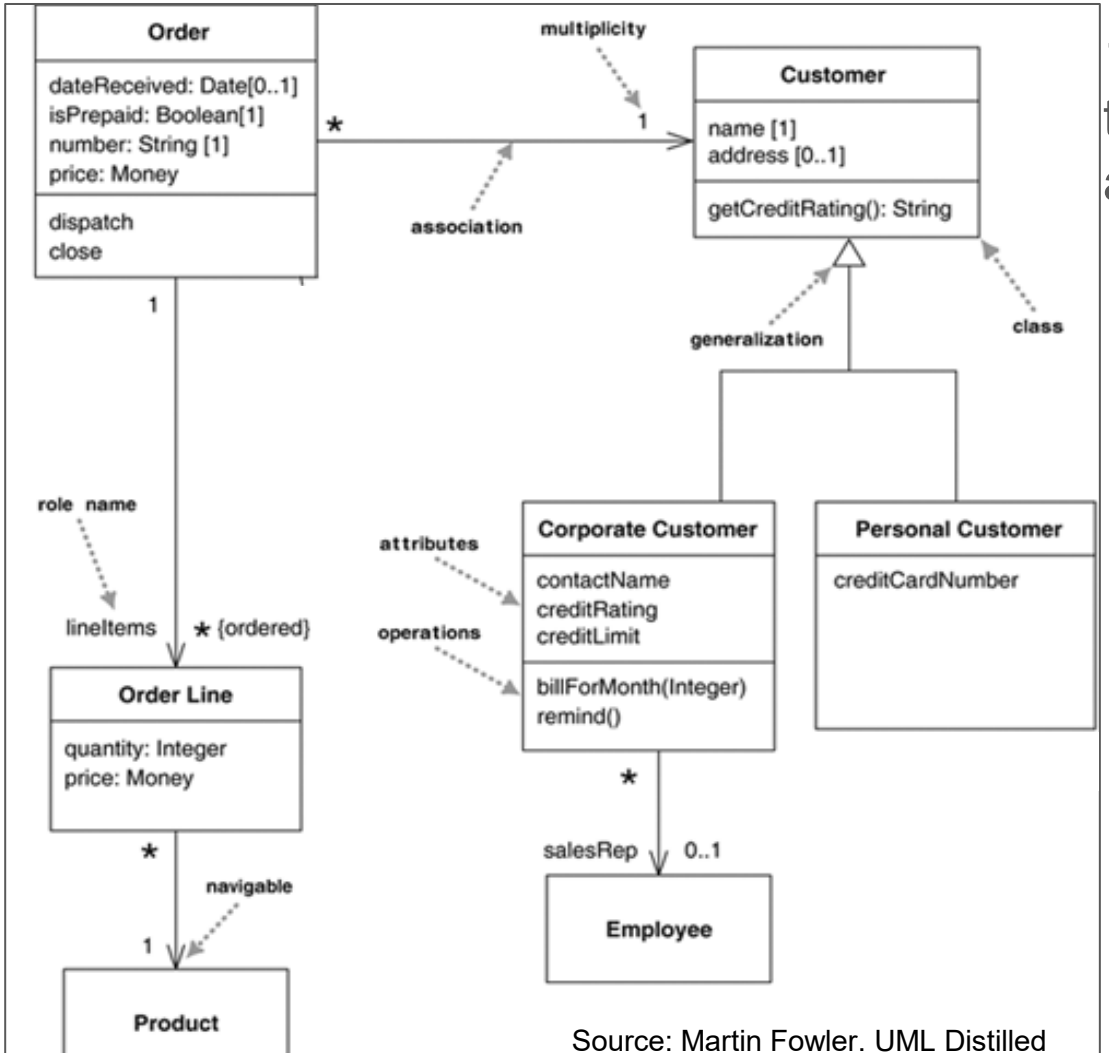
```
import java.util.Stack;

class MyClass {
   ...
   private void methodX() {
     Stack stack = new Stack();
     ...
   }
}
```



Dependencies do not indicate any multiplicity

Exercises



Source: Martin Fowler, UML Distilled

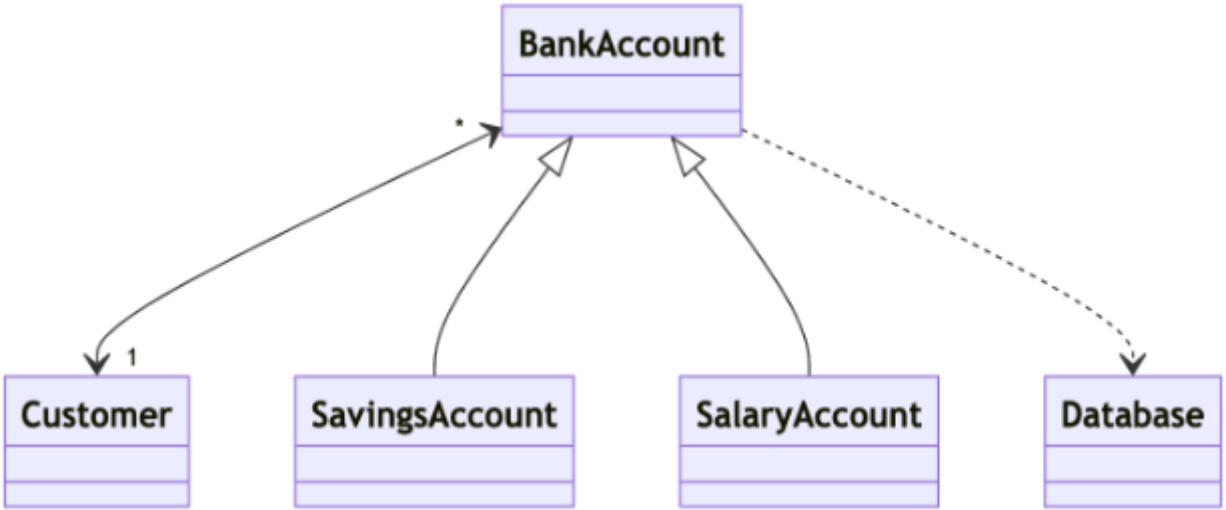
1. Examine this class diagram to understand its structure and relationships

2. Model using class diagrams. The classes are in a different font.

- A `BankAccount` has exactly one `Customer`, but a `Customer` can have several `BankAccount`, with bidirectional navigation.
- `SavingsAccount` and `SalaryAccount` are subclasses of `BankAccount`.
- The `BankAccount` code declares a local variable of type `Database`.
- An `OrderItem` refers to a single `Order` (without navigation). An `Order` can have several `OrderItem` (with navigation).
- The `Student` class has three private attributes: `name`, `ID`, `course`; and two public methods: `getCourse()` and `cancelEnrollment()`.

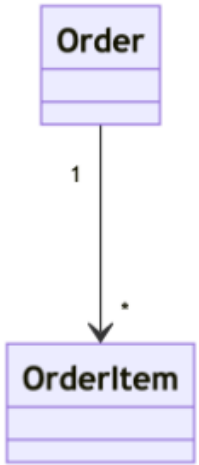
2. Model using class diagrams. The classes are in a different font.

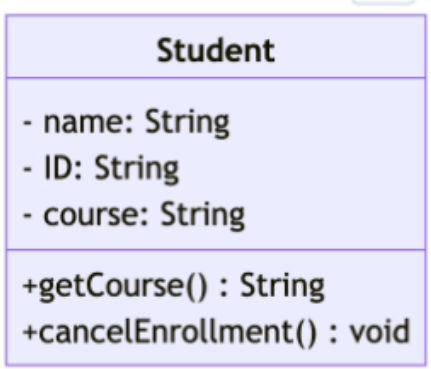
- A `BankAccount` has exactly one `Customer`, but a `Customer` can have several `BankAccount`, with bidirectional navigation.
- `SavingsAccount` and `SalaryAccount` are subclasses of `BankAccount`.
- The `BankAccount` code declares a local variable of type `Database`.



An `OrderItem` refers to a single `Order` (without navigation). An `Order` can have several `OrderItem` (with navigation).

- The `Student` class has three private attributes: `name`, `ID`, `course`; and two public methods: `getCourse()` and `cancelEnrollment()`.





3. Create class diagrams for the following Java programs:

(a)

```
class HelloFrame {
   public static void main(String[] args) {
     JFrame frame = new JFrame("Hello!");
     frame.setVisible(true);
   }
}
```

(b)

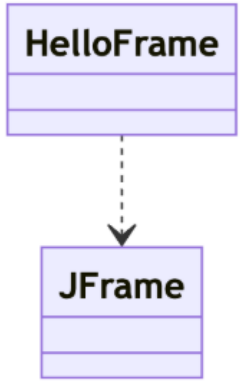
```
class HelloFrame extends JFrame {
   public HelloFrame() {
     super("Hello!");
   }
   public static void main(String[] args) {
     HelloFrame frame = new HelloFrame();
     frame.setVisible(true);
   }
}
```

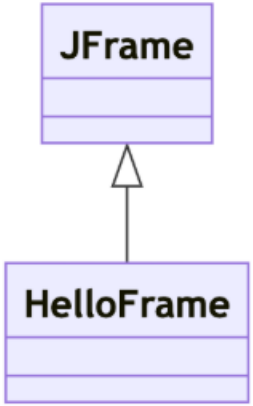
3. Create class diagrams for the following Java programs:

(a)

```
class HelloFrame {

   public static void main(String[] args) {
     JFrame frame = new JFrame("Hello!");
     frame.setVisible(true);
   }
}
```





(b)

```
class HelloFrame extends JFrame {
   public HelloFrame() {
     super("Hello!");
   }
   public static void main(String[] args) {
     HelloFrame frame = new HelloFrame();
     frame.setVisible(true);
   }
}
```