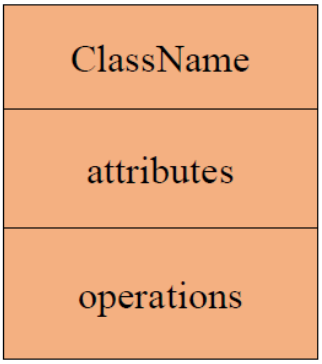
CLASS MODELLING: UML CLASS DIAGRAMS

Class Modelling

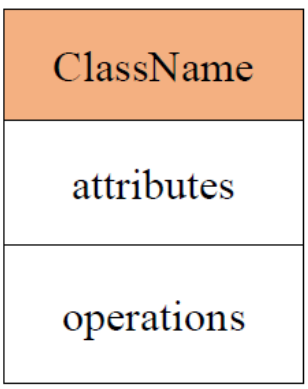
- A class model captures the static structure of a system by characterizing the objects in the system, the relationships between the objects and the attributes and operations for each class of objects.
- Class diagram is a graphical notation used to construct and visualize object-oriented systems.
- Class diagram can be mapped directly with object-oriented languages.
- Class diagrams capture the static structure of Object-Oriented systems as how they are structured rather than how they behave.
- It supports architectural design.
- They identify what classes are there, how they interrelate and how they interact.
- A UML class diagram is made up of:
    - A set of classes and
    - A set of relationships between classes

Class

- A class is a description of a set of objects that share the same attributes, operations, relationships and semantics.
- Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations in separate, designated compartments.

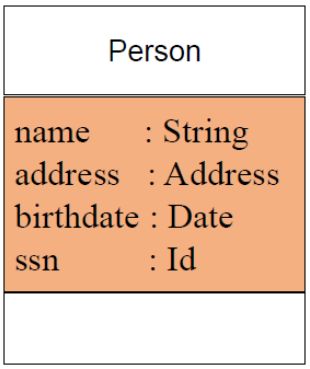


- The name of the class is the only required tag in the graphical representation of a class. It always appears in the top-most compartment.



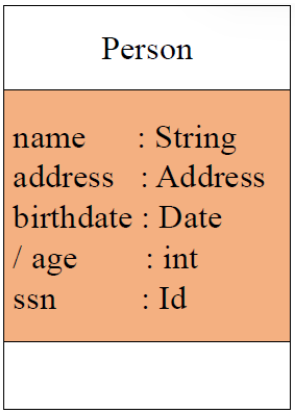
Attributes

- Each class can have attributes.
- An *attribute* is a named property of a class that describes the object being modelled.
- Each attribute has a type.
- In the class diagram, attributes appear in the second compartment just below the name-compartment.



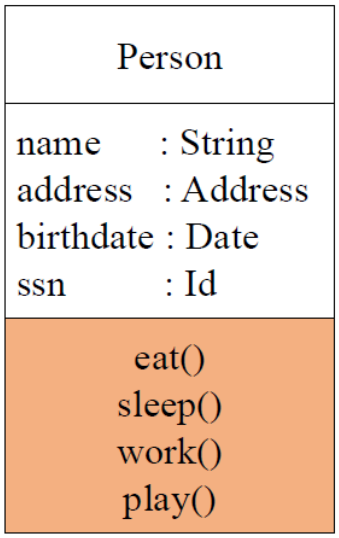
Derived Attributes

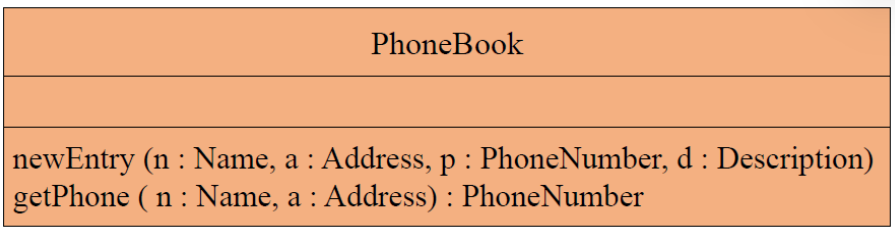
- Attributes are usually listed in the form:
  - attributeName : Type
- A *derived* attribute is one that can be computed from other attributes, but doesn't actually exist.
- For example, a Person's age can be computed from his birth date. A derived attribute is designated by a preceding '/' as in:
  - / age : int



Class Operations

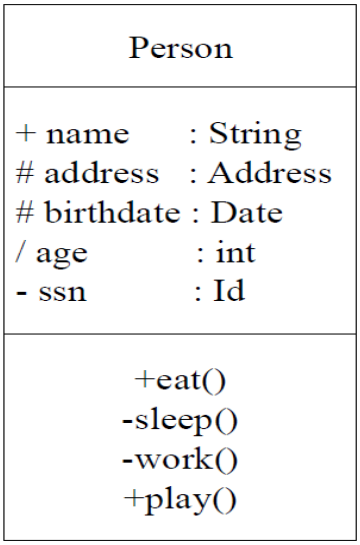
- *Operations* describe the class behaviour and appear in the third compartment.
- You can specify an operation by stating its signature: listing the name, type, and default value of all parameters, and, in the case of functions, a return type.





Visibility

- Attributes and operations can be declared with different visibility modes:
  - + public: any class can use the feature (attribute or operation);
  - # protected: any descendant of the class can use the feature;
  - private: only the class itself can use the feature



Finding Classes

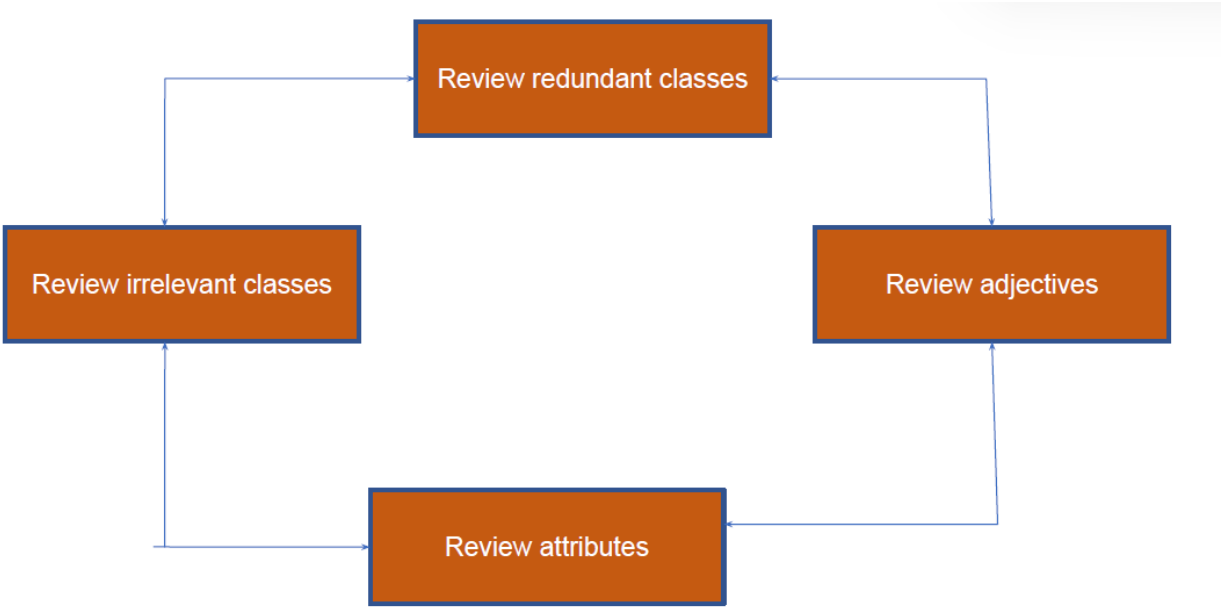
- **Finding classes in use case, or in text descriptions:**
  - Look for nouns and noun phrases in the description of a use case or a problem statement.
  - These are only included in the model if they explain the nature or structure of information in the application.
- **Don't create classes for concepts which:**
  - Are beyond the scope of the system
  - Refer to the system as a whole
  - Duplicate other classes
  - Are too vague or too specific (few instances)
- **Finding classes in other sources:**
  - Reviewing background information
  - Users and other stakeholders
  - Analysis patterns
  - CRC (Class Responsibility Collaboration) cards.

Approaches for identifying classes

1. Noun phrase approach
2. Common class patterns approach
3. Classes, responsibilities and collaborators (CRC) approach
4. Use case driven approach

Noun phrase approach

- This approach was proposed by Rebecca, Brian and Lauren.
- The requirement document needs to be read and noun phrase should be identified.
- Nouns to be considered as classes and verbs to be methods of the classes.
- All plurals changed to singular.
- Nouns are listed and divided into three categories: relevant, fuzzy and irrelevant classes
- Irrelevant classes need to be identified.
- Candidate classes can be identified from other two categories.
- Identifying tentative classes:
    - Look for noun and noun phrases in the use cases.
    - Some classes are implicit or taken from general knowledge.
    - Carefully choose and define class names.
- Finding classes is an incremental and iterative process.
- Guidelines to select candidate classes from the relevant and fuzzy categories:
    - **Redundant classes**: select one class if more than one class describes the same information. This is part of building a common vocabulary for the whole system. Choose the word that is used by the user.
    - **Attribute classes:** Tentative objects that are used only as values should be defined or restarted as attributes and not as a class. For example, client status can be the attribute not the class.
    - **Irrelevant classes:** class should be defined clearly with its purpose. If the purpose of statement is not possible for any class, class should be eliminated.
- The process of identifying classes and refining is the iterative process.
- This is not the sequential process. You can move back and forth.



Common class patterns approach

- Researchers like Shlaer and Mellor proposed some patterns for identifying the candidate class and objects
- **Concept class:**
  - Emphasis principles that are not tangible but used to organize or keep track of business activities or communications.
  - Ex: performance
- **Event class:**
  - These are points in time that must be recorded.
  - Things happen, usually to something else at given date and time or as a step in an ordered sequence.
  - Associated with things remembered are attributes such as who, what, when, where, how or why.
  - Ex: Landing, interrupt
- **Organization class:**
  - It's the collection of people, resources, facilities or groups to which the users belong.
  - Ex: An accounting department might be considered a potential class.
- **People class:**
  - It specifies different roles users play in interacting with the application.
  - Two types:
    - the users of the system (operator or clerk) or who interacts with the system.
    - Who do not use the system but whom information is kept by the system.
- **Tangible class and device class:**
  - This includes physical objects or groups of objects that are tangible.
  - The devices with which the application interacts.
  - Ex: cars, pressure sensors
- **Place class:**
  - Places are physical locations that the system must keep information.
  - Ex: buildings, stores and offices

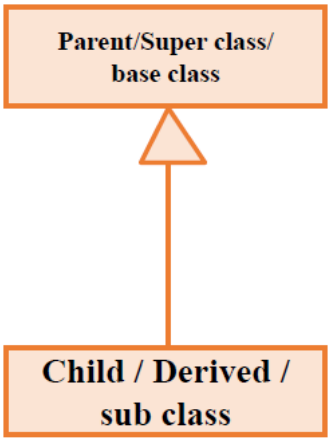
CLASS MODELLING: OO-RELATIONSHIPS

In UML, object interconnections (logical or physical), are modelled as relationships.
The type of relationships is listed below:

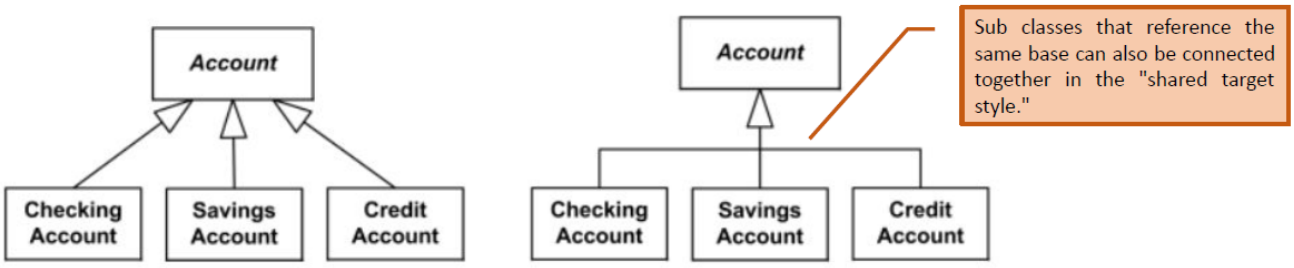
- Generalization- an inheritance relationship
- Abstraction
- Realization - interface implementation
- Dependency
- Association – using relationship
    - Aggregation
    - Composition

Generalization

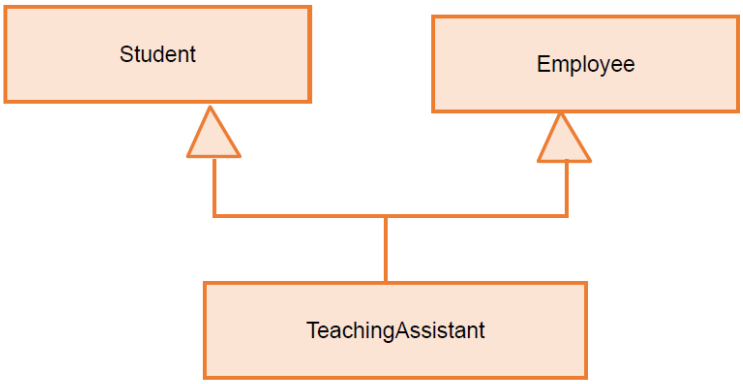
- Also known as Inheritance.
- It represents “is a” or “is a kind of” relationship since the child class is a type of the parent class.
- It is used to showcase reusable elements in the class diagram.
- The child classes “inherit” the common functionality (attributes and methods) defined in the parent class.
- A generalization is shown as a line with a hollow triangle as an arrowhead.
- The arrowhead points to the super class.



- UML permits a class to inherit from multiple super classes, although some programming languages (*e.g.,* Java) do not permit multiple inheritance.



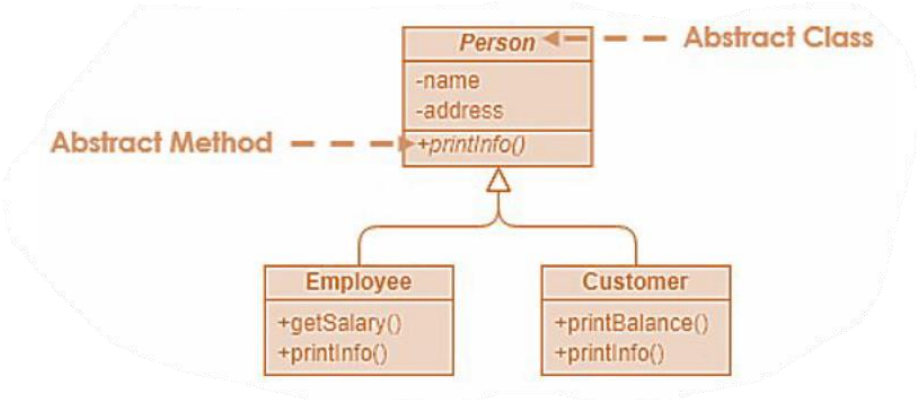
Example 1



Example 2

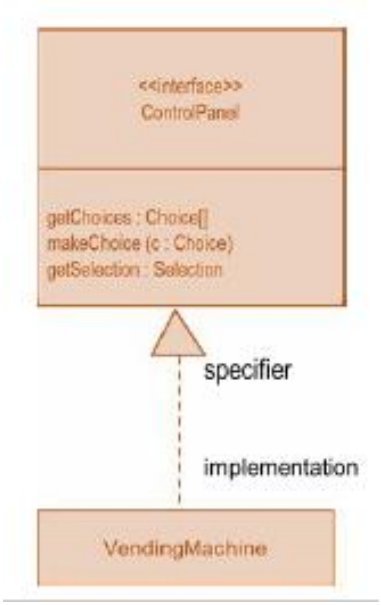
Abstraction

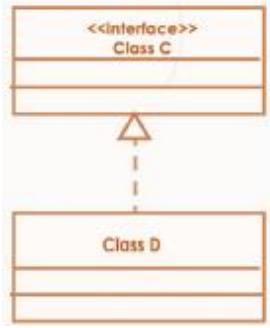
- In an inheritance hierarchy, subclasses implement specific details, whereas the parent class defines the framework its subclasses.
- The parent class also serves as a template for common methods that will be implemented by its subclasses.
- The name of an abstract Class is typically shown in italics; alternatively, an abstract Class may be shown using the textual annotation, also called stereotype {abstract} after or below its name.
- An abstract method is a method that do not have implementation. In order to create an abstract method, create an operation and make it italic.

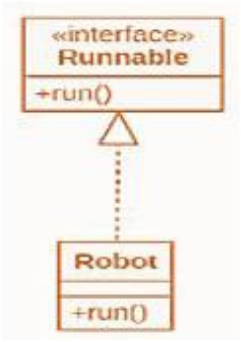


Realization/ Interfaces

- Realization is a specialized abstraction relationship between two things where one thing (an interface) specifies a contract that another thing (a class) guarantees to carry out by implementing the operations specified in that contract.
- Interface defines a set of functionalities as a contract and the other entity "realizes" the contract by implementing the functionality defined in the contract.
- Realization is shown as a dashed directed line with an open arrowhead pointing to the interface.

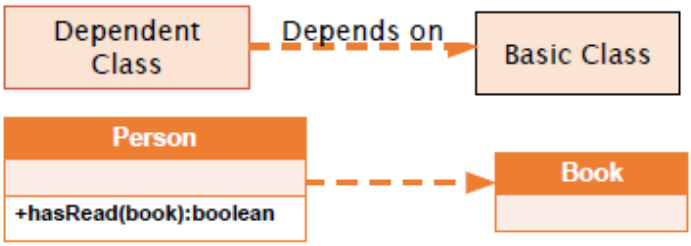






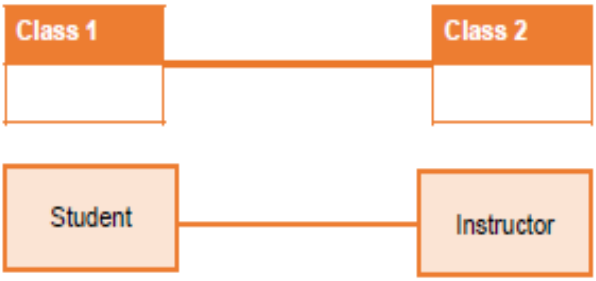
Dependency

- Dependency means "One class uses the other"
- A dependency relationship indicates that a change in one class may affect the dependent class, but not necessarily the reverse.
- A dependency relationship is often used to show that a method has object of a class as arguments.

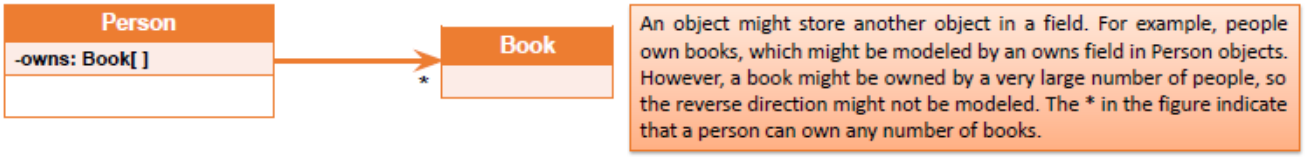


An object of one class might use an object of another class in the code of a method. If the object is not stored in any field, then this is modeled as a dependency relationship. For example, the Person class might have a hasRead method with Book as a parameter that returns true if the person has read the book.

Association Relationships



- If two classes in a model need to communicate with each other, there must be link between them. An *association* denotes that link.
- We can constrain the association relationship by defining the *navigability* of the association.
- Association is of two types. They are:
  - Unidirectional Association
  - Bidirectional Association
- In a unidirectional association, two classes are related, but only one class "knows" that the relationship exists.
- A unidirectional association is drawn as a solid line with an open arrowhead pointing to the known class.
- Uni-directional association includes a role name and a multiplicity value, but only for the known class.
- In below example Orderdetails class only knows that it has a relationship with Item class but Item class does not know about their relationship.



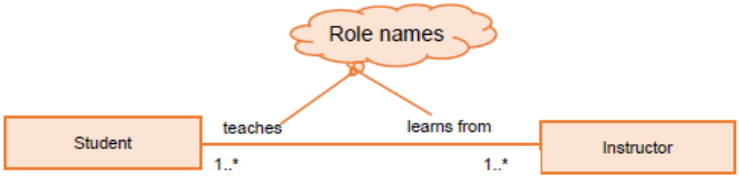
- We can indicate the multiplicity of an association by adding multiplicity adornments to the line denoting the association.
- The multiplicity of an association is the number of possible instances of the class associated with a single instance of the other end.
- Multiplicities are single number or ranges of numbers. The table below has the multiplicities used and their respective numbers.

Multiplicities	Meaning
0..1	zero or one instance.
0..* *or* *	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance
n..m	***n*** to ***m*** instances (n and m stand for numbers, e.g. 0..4, 3..15)
n	exactly n instance (where n stands for a number, e.g. 3)

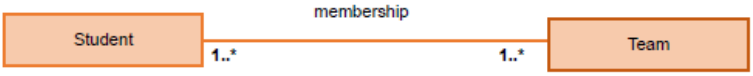
Student has one or more Instructors.

Every Instructor has one or more Students

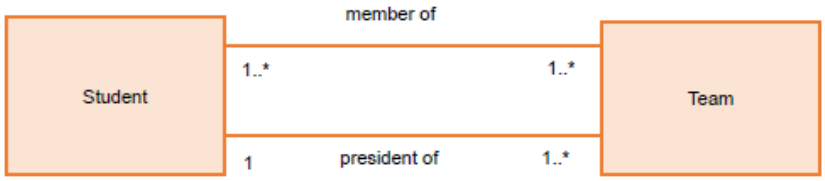
- We can also indicate the behaviour of an object in an association (i.e., the role of an object) using role names.



- We can also name the association.

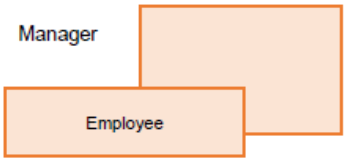


- We can specify dual associations using bidirectional association.

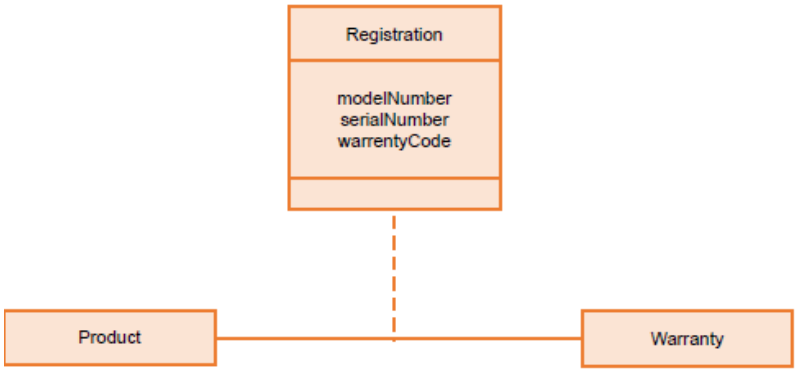


Two objects might store each other in fields. For example, in addition to a Person object listing all the books that the person owns, a Book object might list all the people that own it

- A class can have a self-association.

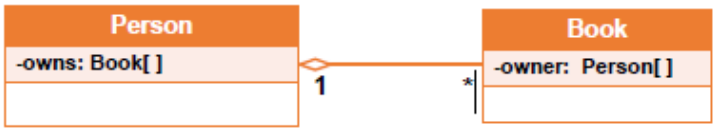


- Associations can also be objects themselves, called link classes or an association class.
- An association class is represented like a normal class, but it is linked to an association line with a dotted line.



Aggregation

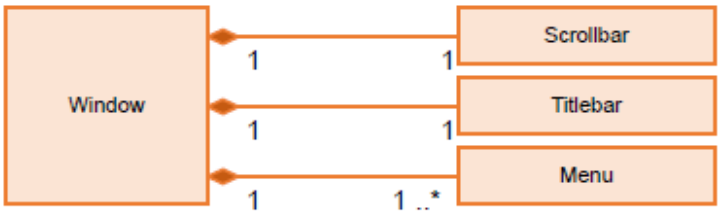
- We can model objects that contain other objects by way of special associations called aggregations and compositions. It is also known as “has a” relationship.
- An aggregation specifies a whole-part relationship between an aggregate (a whole) and a constituent part, where the part can exist independently from the aggregate. Aggregations are denoted by a hollow-diamond adornment on the association.

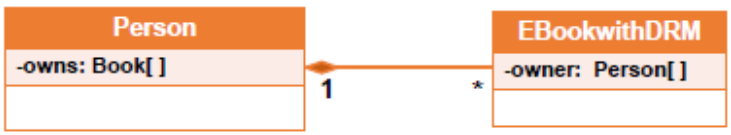


One object A has or owns another object B, and/or B is part of A. For example, suppose there are different Book objects for different physical copies. Then the Person object has/owns the Book object, and, while the book is not really part of the person, the book is part of the person's property. In this case, each book will (usually) have one owner. Of course, a person might own any number of books.

Composition

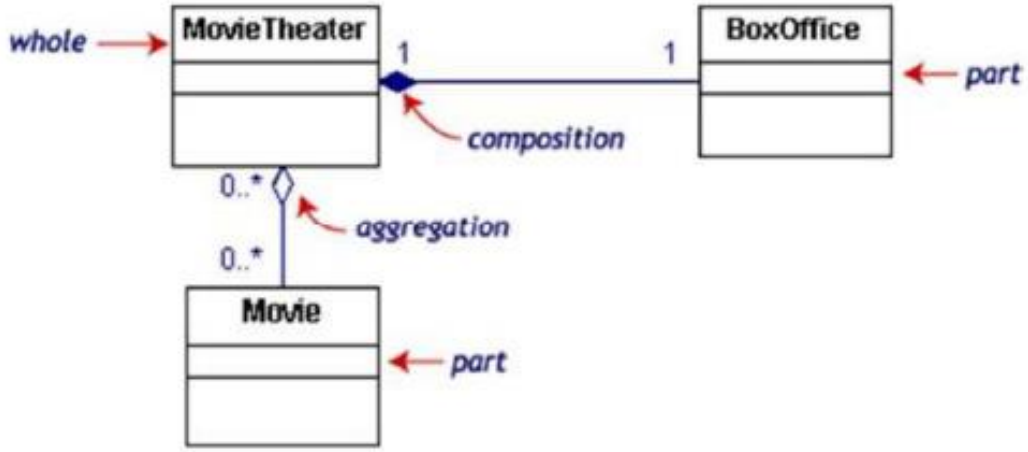
- A composition indicates a strong ownership and coincident lifetime of parts by the whole (i.e., they live and die as a whole).
- Compositions are denoted by a filled-diamond adornment on the association.





In addition to an aggregation relationship, the lifetimes of the objects might be identical, or near For example, in an idealized world of electronic books with DRM (Digital Rights Management), a person can own an ebook, but cannot sell it. After the person dies, no one else can access the ebook. [This is idealized, but might be considered less than ideal.]

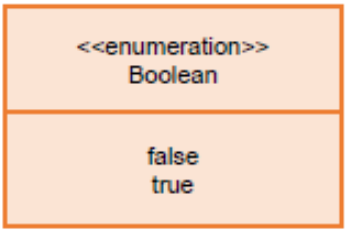
Composition/ Aggregation Example



If the movie theater goes away
so does the box office => composition
but movies may still exist => aggregation

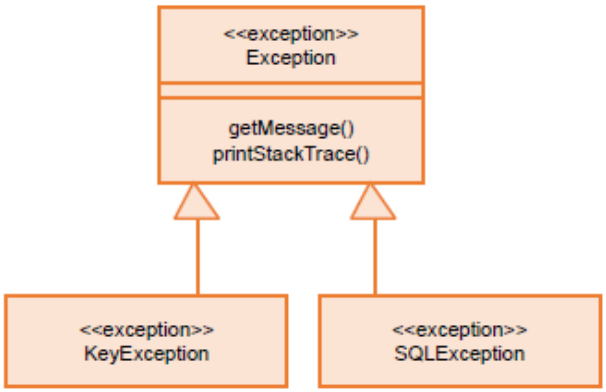
Enumeration

- An enumeration is a user-defined data type that consists of a name and an ordered list of enumeration literals.



Exceptions

- Exceptions can be modelled just like any other class.
- Notice the <<exception>> stereotype in the name compartment.

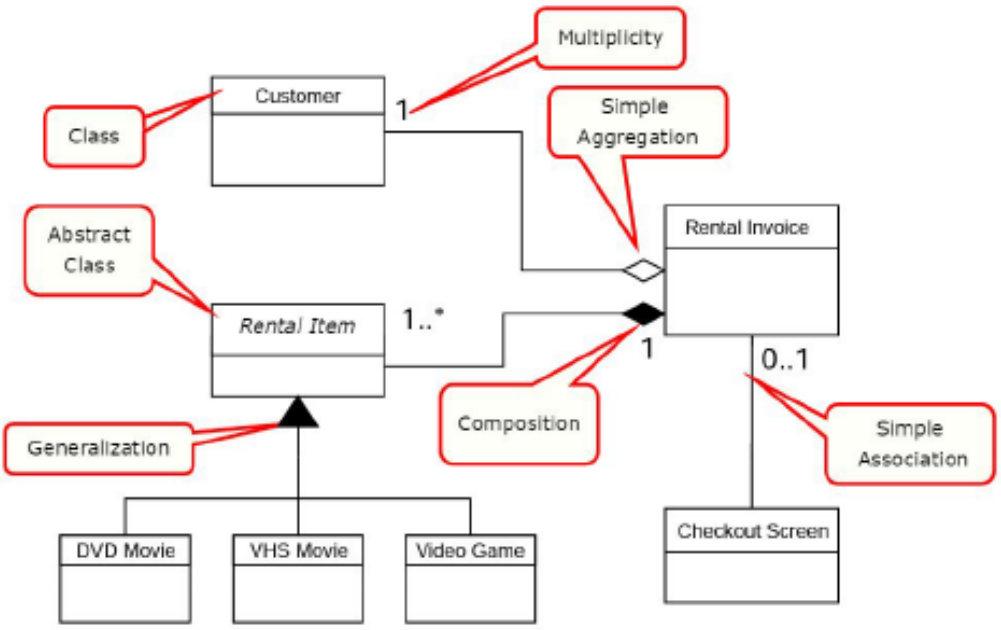


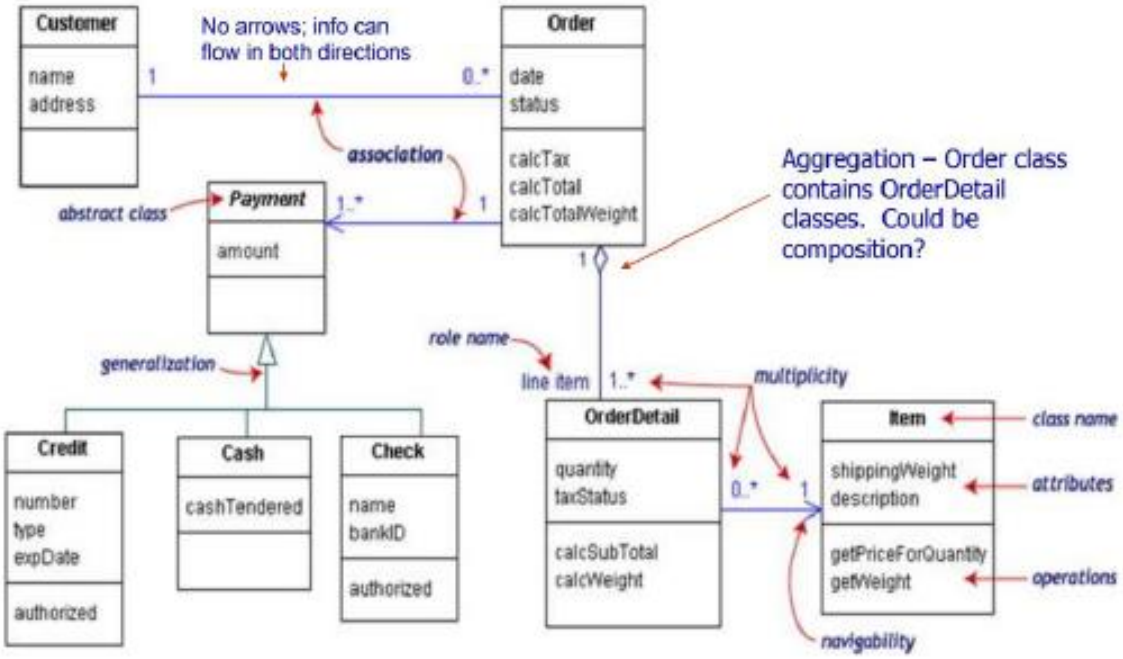
How to make a class diagram?

1. Identify all the classes participating in the software solution.
2. Draw them in a class diagram.
3. Identify the attributes.
4. Identify the methods.
5. Add associations, generalizations, aggregations and dependencies.
6. Add other stuff (roles, constraints, ...)

Example class diagram

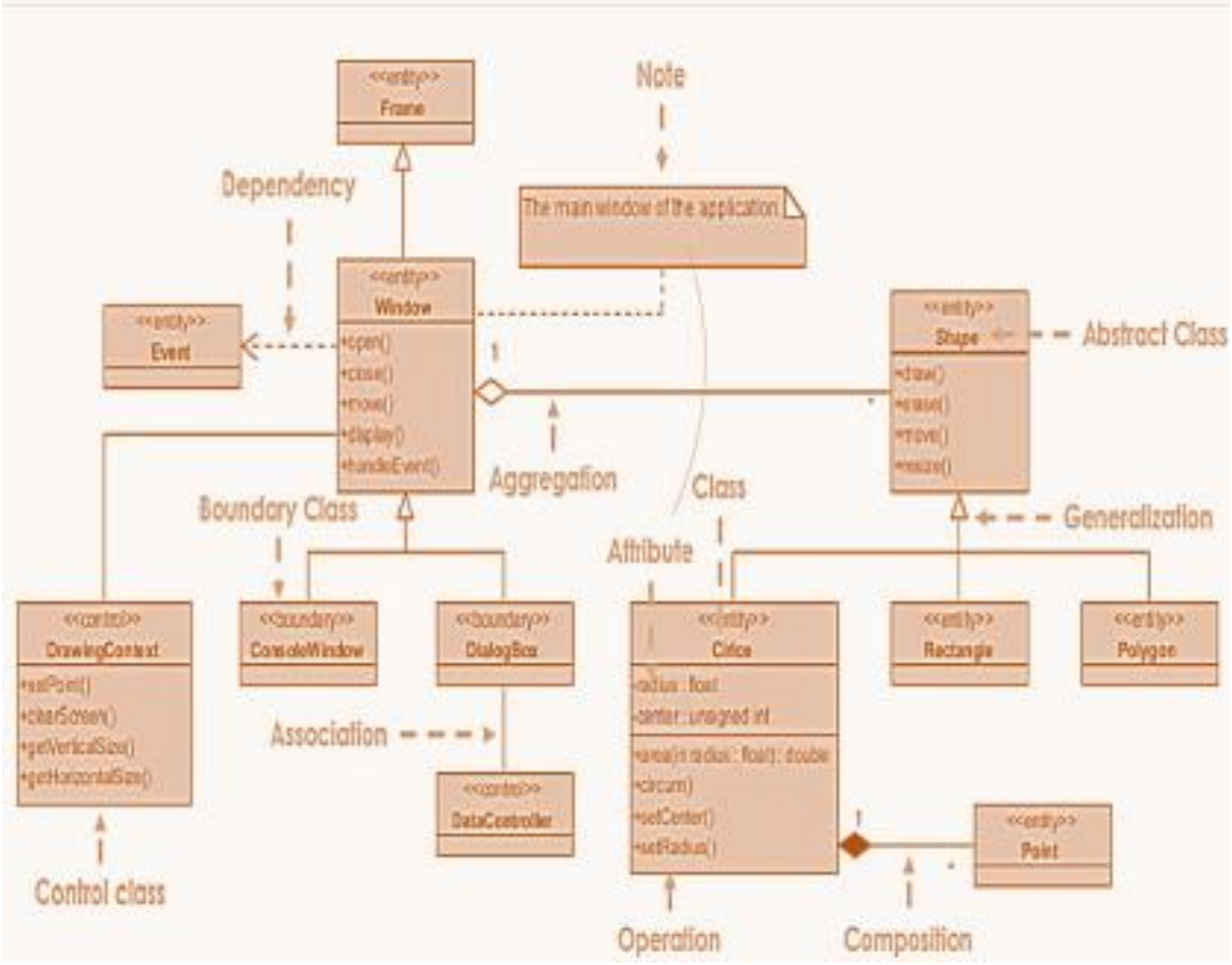
1. Video Store



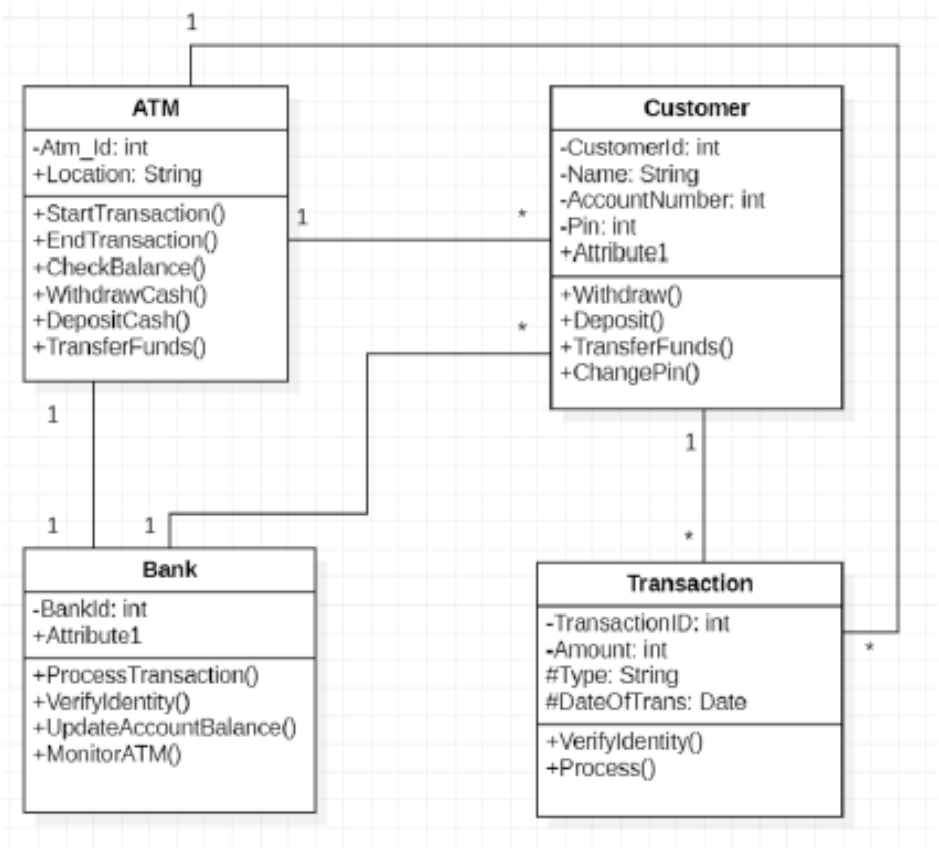


2. Shapes

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



3. Bank ATM Machine



CRC: CLASS-RESPONSIBILITY-COLLABORATORS

- CRC stands for Class, Responsibility and Collaboration.
  - **Class**
    - An object-oriented class name.
    - Include information about super- and sub-classes
  - **Responsibility**
    - What information this class stores
    - What this class does
    - The behaviour for which an object is accountable
  - **Collaboration**
    - Relationship to other classes
    - Which other classes this class uses

Class Name	
Responsibilities	**Collaborators**
Class: Account	
Responsibilities	**Collaborators**
Know balance	
Deposit Funds	Transaction
Withdraw Funds	Transaction, Policy
Standing Instructions	Transaction, StandingInstruction Policy, Account

Ex: CRC card

- CRC cards are tool used for brainstorming in OO design and agile methodologies.
- CRC cards are created from **Index cards**.
- Each member of the project team writes one CRC card for **each relevant class/object** of their design.
- Objects need to interact with other objects (Collaborators) in order to fulfil their responsibilities.
- Since the cards are small, prevents to get into details and give too many responsibilities to a class.
- They can be easily placed on the table and rearranged to describe and evolve the design.
- CRC cards are an aid to a group role-playing activity.

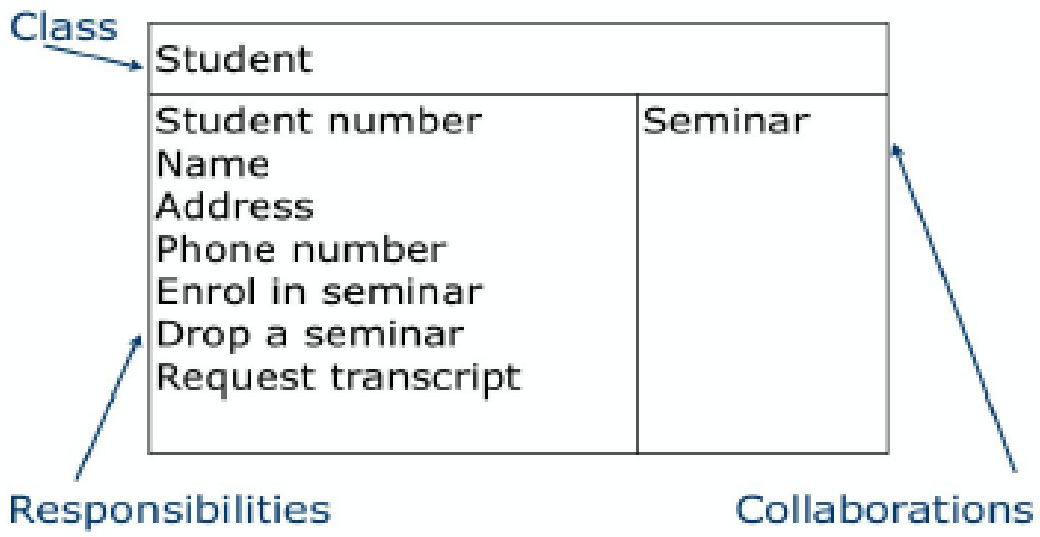
- Index cards are used in preference to pieces of paper due to their robustness and to the limitations that their size (approx. 15cm x 8cm) imposes on the number of responsibilities and collaborations that can be effectively allocated to each class.
- A **class name** is entered at the top of each card and responsibilities and collaborations are listed underneath as they become apparent.
- For the sake of clarity, each collaboration is normally listed next to the corresponding
- responsibility.
- From a UML perspective, use of CRC cards is **in analysing the object interaction** that is triggered by a particular use case scenario.

The process of using CRC cards is usually structured as follows.

1. Conduct a session to identify which objects are involved in the use case.
2. Allocate each object to a team member who will play the role of that object.
3. Act out the use case.
4. Identify and record any missing or redundant objects.

Example: Student CRC card.

- A responsibility is anything that a class knows or does. The things a class knows and does constitute its responsibilities.
- For example, students have names, addresses, and phone numbers. These are the things a student knows.
- Students also enroll in seminars, drop seminars, and request transcripts. These are the things a student does.
- Students only have information about themselves (their names and so forth), and not about seminars.
- What the student needs to do is collaborate/interact with the card labelled Seminar to sign up for a seminar.
- Therefore, Seminar is included in the list of collaborators of Student.



Example: Social Media Platform

User	
Responsibilities	**Collaborators**
Create, edit, delete account	
Manage profiles & personalization	Post
Authentication & authorization	Notification

Post	
Responsibilities	**Collaborators**
Create post	User
Share post	Friend
Manage likes/comments	Notification
Manage visibility	Friend

Friend	
Responsibilities	**Collaborators**
Manage friend requests	User, Notification
Add/Remove friend	User

Notification	
Responsibilities	**Collaborators**
Handle user alerts(likes, comments, requests)	User
Customization/changing settings	User, Post

Example: Online Shopping Platform

USER	
Responsibilities	**Collaborators**
Manages authentication	Customer
Manages personal info	Seller

Customer	
Responsibilities	**Collaborators**
Searches for products	User
Adds items to the cart	Product
Reviews product details	Shopping Cart
Reviews/updates cart	

Seller	
Responsibilities	**Collaborators**
Manages inventory	User
Reviews/updates inventory	Product
Adds new products	Inventory
Reads sales/profit	

Product	
Responsibilities	**Collaborators**
Manages product info	Customer
	Seller

Shopping Cart	
Responsibilities	**Collaborators**
Manages items in the cart	Customer

Inventory	
Responsibilities	**Collaborators**
Manages product stock	Seller