Building Models:

- **Introduction to Models:**
- **Forward engineering** is where we take the requirements and produce a model from it. This model can either be very high level or very low level.
  If we want a high level point of view, we usually sketch the components of the model.
  If we want a low level point of view, we need to add a lot of details.
- **Reverse engineering** is where we're given a code base and we extract the requirements from it. Creating higher level views can help us better understand the project.
- For reverse engineering, we would like to know the following information:
  1. **Structure of the code:**
     - This involves knowing the dependencies of the code and all the different components and how they couple together.
  2. **Behaviour of the code:**
     - This involves knowing how the code executes.
     - For more complex code, we may need to make state machine models.
  3. **Function of the code:**
     - This involves knowing what functions the code provides to the user.
- Modelling can guide your exploration.
  - It can help you figure out what questions to ask.
  - It can help to reveal key design decisions.
  - It can help you to uncover problems.
- Modelling can help us check our understanding.
  - We can use the model to understand its consequences and know if it has the properties we expect.
  - We can animate the model to help us visualize/validate software behaviour.
- Modelling can help us communicate.
  - Modelling provides useful abstractions that focus on the point you want to make without overwhelming people with detail.
  - Furthermore, both technical and non-technical people can understand.
- However, the exercise of modelling is more important than the model itself.
  Time spent perfecting the models might be time wasted.
- **Dealing with the Problem Complexity:**
- There are 4 ways to dealing with problem complexity:
  1. **Abstraction:**
     - Allows us to ignore details and look at the big picture.
     - We can treat objects as the same by ignoring certain differences.
     - **Note:** Every abstraction involves choice over what is important.
  2. **Decomposition:**
     - Allows us to break a problem into many independent pieces so that we can study each piece separately.
     - **Note:** The pieces are rarely independent.
  3. **Projection:**
     - Allows us to separate different points of view and describe them separately. I.e. Divide and conquer.
     - This is different from decomposition as it does not partition the problem space.
     - **Note:** Different views will be inconsistent most of the time.
  4. **Modularization:**
     - Allows us to choose structures that are stable over time to localize change.
     - **Note:** Some structures will make localizing changes easier and others will make it harder.

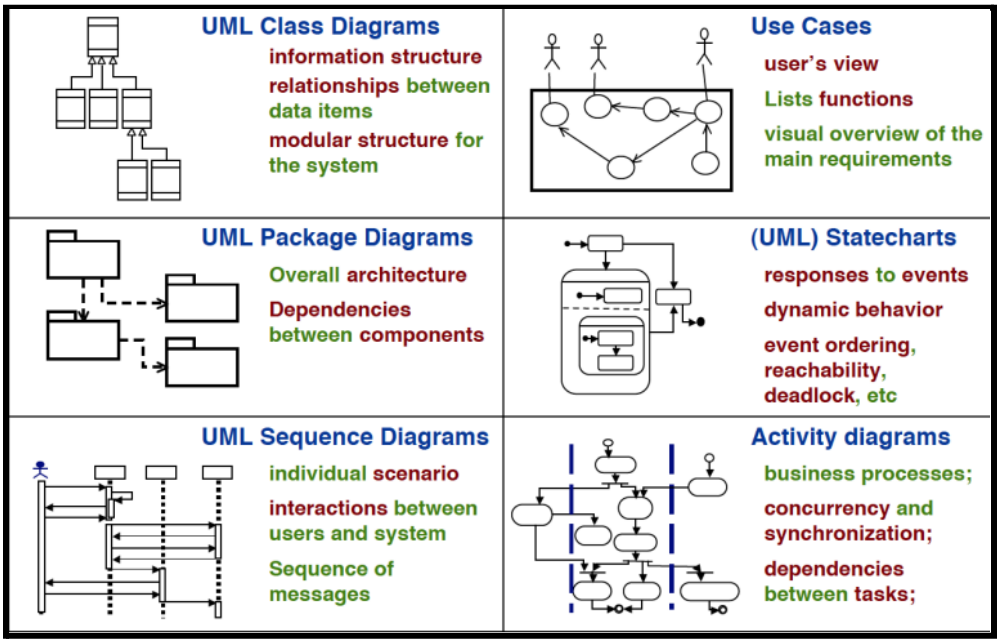
- **Design Phase:**
- **Design** is specifying the structure of how a software system will be written and function, without actually writing the complete implementation.
- During the design phase we transition from "what" the system must do to "how" the system will do it.
- The design phase involves answering the following questions:
    - 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?

Unified Modeling Language (UML):

- **Introduction to UML:**
- **Unified Modeling Language (UML)** allows us to express the design of a program before writing any code.
- It is language-independent.
- It is an extremely expressive language.
- UML is a graphical language for visualizing, specifying, constructing, and documenting information about software-intensive systems.
- UML can be used to develop diagrams and provide programmers with ready-to-use, expressive modeling examples. Some UML tools can generate program language code from UML. UML can be used for modeling a system independent of a platform language.
- UML is a picture of an object oriented system. Programming languages are not abstract enough for object oriented design. UML is an open standard and lots of companies use it.
- Legal UML is both a descriptive language and a prescriptive language. It is a descriptive language because it has a rigid formal syntax, like programming languages, and it is a prescriptive language because it is shaped by usage and convention.
- It's okay to omit things from UML diagrams if they aren't needed by the team/supervisor/instructor.
- **History of UML:**
- In an effort to promote object oriented designs, three leading object oriented programming researchers joined forces to combine their languages. They were:
    1. Grady Booch (BOOCH)
    2. Jim Rumbaugh (OML: object modeling technique)
    3. Ivar Jacobsen (OOSE: object oriented software eng)
- They came up with an industry standard in the mid 1990's.
- UML was originally intended as a design notation and had no modelling associated with it.
- **UML diagrams can help engineering teams:**
- Bring new team members or developers switching teams up to speed quickly.
- Navigate source code.
- Plan out new features before any programming takes place.
- Communicate with technical and non-technical audiences more easily.
- **Uses of UML:**

1. It can be used as a sketch to communicate aspects of the system.
- **Forward design:** Doing UML before coding.
- **Backward design:** Doing UML after coding as documentation.
2. It can be used as a blueprint to show a complete design that needs to be implemented. This is sometimes done with CASE (Computer-Aided Software Engineering) tools. One of these tools is visual paradigm.
3. It can be used as a programming language.
- Some UML tools can generate program language code from UML.

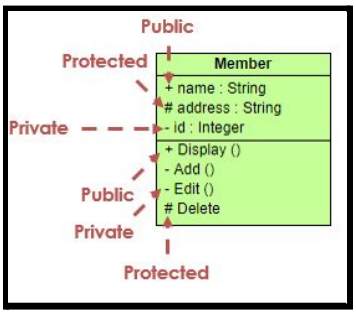
- **UML Notations:**

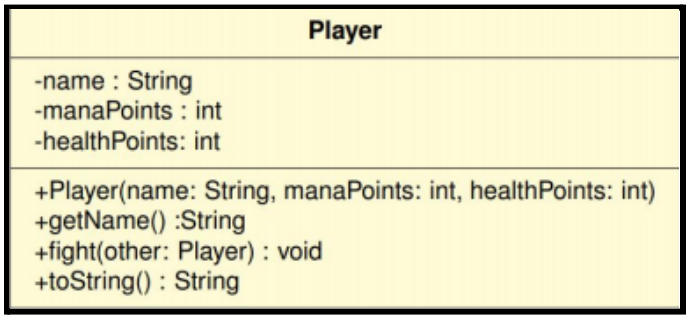


- **Class Diagram:** It displays the system's classes, attributes, and methods. It is helpful in recognizing the relationship between different objects as well as classes.
- **Object Diagram:** It describes the static structure of a system at a particular point in time. It can be used to test the accuracy of class diagrams. It represents distinct instances of classes and the relationship between them at a time.
- **Use Case Diagram:** It represents the functionality of a system by utilizing actors and use cases. It encapsulates the functional requirement of a system and its association with actors. It portrays the use case view of a system.
- **Package Diagram:** It is used to illustrate how the packages and their elements are organized. It shows the dependencies between distinct packages. It manages UML diagrams by making it easily understandable. It is used for organizing the class and use case diagrams.
- **Statechart:** It is a behavioral diagram. It portrays the system's behavior by utilizing finite state transitions. It models the dynamic behavior of a class in response to external stimuli.
- **Sequence Diagram:** It shows the interactions between objects in terms of messages exchanged over time. It delineates in what order and how the object functions are in a system. Time does not play a role.
- **Activity Diagram:** It models the flow of control from one activity to the other. With the help of an activity diagram, we can model sequential and concurrent activities. It visually depicts the workflow as well as what causes an event to occur. Time plays a role.
- **Note:** Statechart depicts the state transition. E.g. Finite state machine. Activity diagrams depict the various activities that occur. In activity diagrams, the time plays a role while in sequence diagrams, time does not play a role. An activity diagram is a sequence diagram but with timing.

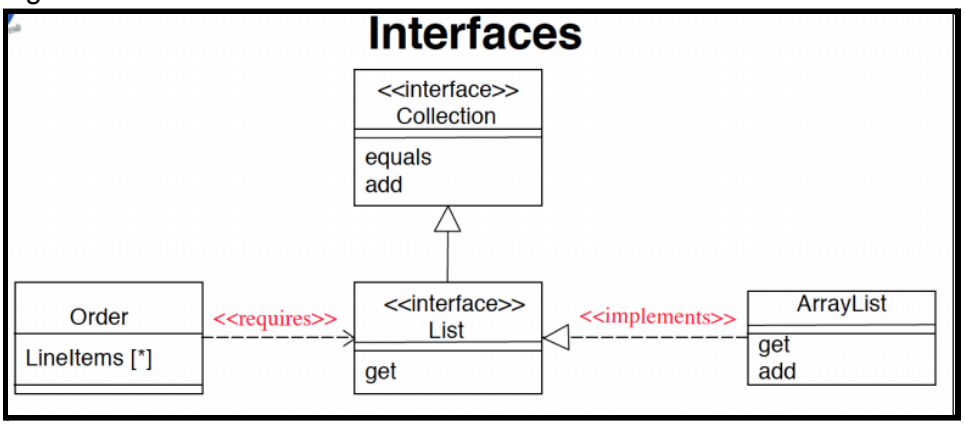
- **UML Class Diagram:**
- A class describes a group of objects with:
  - Similar attributes
  - Common operations
  - Common relationships with other objects
  - Common meaning
- A **class diagram** describes the structure of an object oriented system by showing the classes in that system and the relationships between the classes. A class diagram also shows the constraints, and attributes of classes.
  I.e.
  A UML class diagram is a picture of:
  - The classes in an object oriented system.
  - Their fields and methods.
  - Connections between the classes that interact or inherit from each other.
- Some things that are not represented in a UML class diagram are:
  - Details of how the classes interact with each other.
  - Algorithmic details, like how a particular behavior is implemented.
- **Note:** Coupling between classes must be kept low, while cohesion within a class must be kept high. Furthermore, we should respect the SOLID principles.
- UML class diagrams cans show:
  1. Division of responsibility
  2. Subclassing/Inheritance
  3. Visibility of objects and methods
  4. Aggregation/Composition
  5. Interfaces
  6. Dependencies
- Naming Convention:
  1. **Class name**
  - Use **<<interface>>** on top of interface names.
  - To show that a class is abstract, either italicize the class name or put **<<abstract>>** on top of the abstract class name.
  2. **Data members/Attributes**
  - The data members section of a class lists each of the class's data members on a separate line.
  - Each line uses this format: **attributeName : type**
    E.g. **name : String**
  - We must underline static attributes.
  3. **Methods/Operations**
  - The methods of a class are displayed in a list format, with each method on its own line.
  - Each line uses this format:
    methodName(param1: type1, param2: type2, ...) : returnType
    E.g. **distance(p1: Point, p2: Point) : Double**
  - We may omit setters and getters. However, don't omit any methods from an interface.
  - Furthermore, do not include inherited methods.
  - We must underline static methods.

- Visibility:
- − means that it is private.
- + means that it is public.
- # means that it is protected.
- ~ means that it is a package.
- / means that it is a **derived attribute**. A derived attribute is an attribute whose value is produced or computed from other information.
- **Note:** Everything except / is common for both methods and attributes.
- E.g.

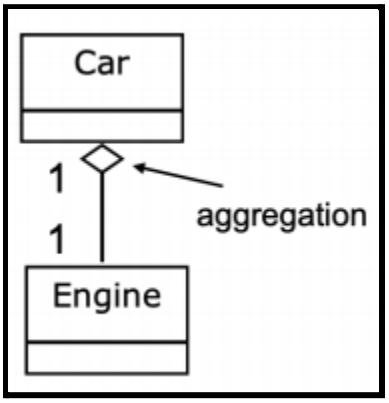




- Inheritance/Generalization and Realization Relationships:
- **Generalization/inheritance** is when a class extends another class while **realization** is when a class implements an interface.
- Generalization represents a “IS-A” relationship.
- Hierarchies are drawn top down with arrows pointing upward to the parent class. I.e. The parent class is above the child class and the arrow goes from the child class to the parent class.
- For a class, draw a solid line with a black arrow pointing to the parent class.
- For an abstract class, draw a solid line with a white arrow pointing to the parent abstract class.
- For an interface, draw a dashed line with a white arrow pointing to the interface.
- E.g.

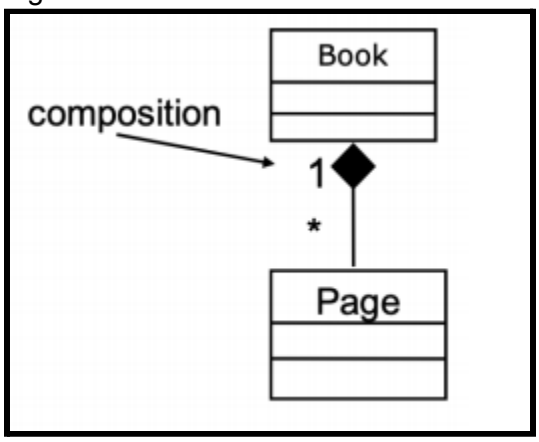


- Association:
- An association represents a relationship between two classes. It also defines the multiplicity between objects.
- Association can be represented by a line between the classes with an arrow indicating the navigation direction.
  Note: Sometimes, association can be represented just by a line between the classes. This means that information can flow in both directions.
- We need the following items to represent association between 2 classes:
  1. The multiplicity
  2. The name of the relationship
  3. The direction of the relationship
- Aggregation, composition and dependency are all types of association.
- Multiplicity:
- * means 0 or more.
- 1 means 1 exactly.
- 2..4 means 2 to 4, inclusive.
- 3..* means 3 or more.
- There are other relationships such as 1-to-1, 1-to-many, many-to-1 and many-to-many.
- Aggregation:
- A special type of association.
- Aggregation implies a relationship where the child class can exist independently of the parent class. This means that if you remove/delete the parent class, the child class still exists.
  I.e. Aggregation represents a “HAS-A” or “PART-OF” relationship.
  E.g. Say we have 2 classes, Teacher (the parent class) and Student (the child class). If we delete the Teacher class, the Student class still exists.
- Aggregation is symbolized by an arrow with a clear white diamond arrowhead pointing to the parent class.
- E.g.

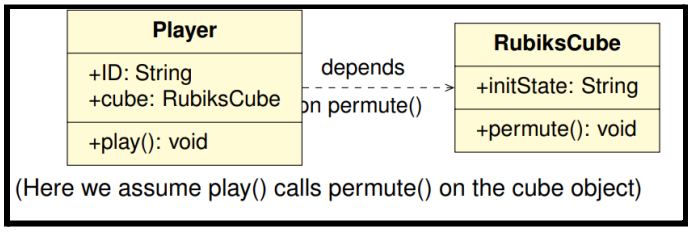


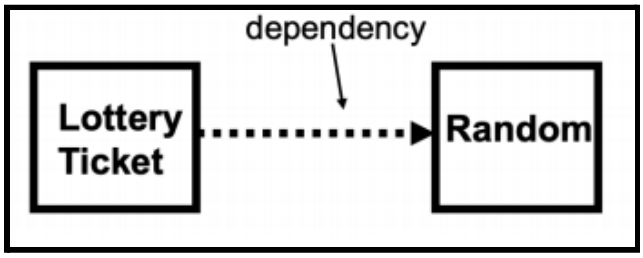
- Aggregation is considered as a weak type of association.
- Composition:
- A special type of association.
- It is a stronger version of aggregation where if you delete the parent class, then all the child classes are also deleted.
  I.e. Composition represents a “ENTIRELY MADE OF” relationship.
  E.g. Say we have 2 classes, House (the parent class) and Room (the child class). If we delete the House class, the Room class is also deleted.
- Composition is symbolized by an arrow with a black diamond arrowhead pointing to the parent class.

- E.g.

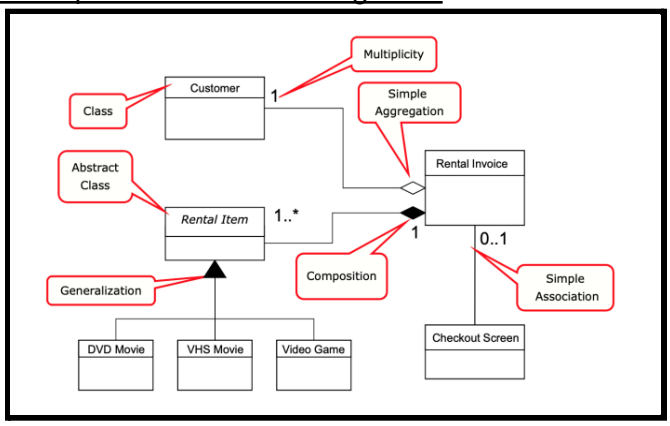


- Composition is considered as a strong type of association.
- Dependency:
- Is a special type of association.
- Dependency indicates a “uses” relationship between two classes. If a change in structure or behaviour of one class affects another class, then there is a dependency between those two classes.
- Dependency is represented by a dotted arrow where the arrowhead points to the independent element.
- E.g.

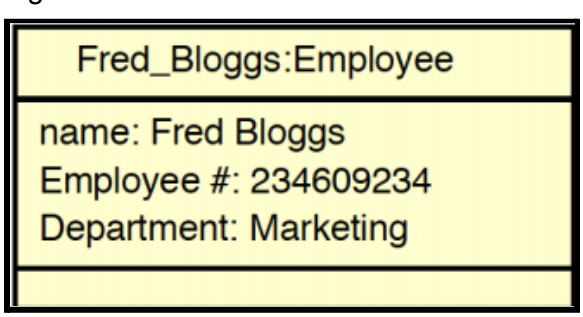


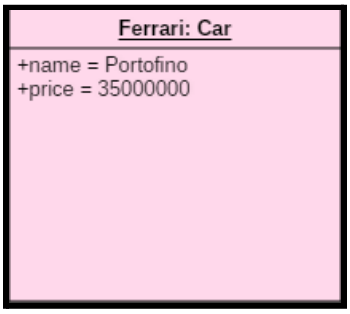


- Examples of UML class diagrams:



- How to draw class diagrams:
    1. Identify the objects in the problem and create classes for each of them
    2. Add attributes
    3. Add operations
    4. Connect classes with relationships
    5. Specify the multiplicities for association connections.
- **UML Object Diagram:**
- Object diagrams look very similar to class diagrams.
- Naming Convention:
  Object name: Type
  Attribute: Value (Sometimes, it's Attribute = Value)
  E.g.





- **Note:** 2 different objects may have identical attribute values.
- Purpose:
- It is used to describe the static aspect of a system.
- It is used to represent an instance of a class.
- It can be used to perform forward and reverse engineering on systems.
- It is used to understand the behavior of an object.
- It can be used to explore the relations of an object and can be used to analyze other connecting objects.