Table of Contents:

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.
4. As a sketch:
- Can be used to sketch a high level view of the system.
- **Forward engineering:** Describes the concepts we need to implement.
- **Reverse engineering:** Explains how parts of the code work.
5. As a blueprint:
- Should be complete and describes the system in detail.
- **Forward engineering:** Model as a detailed specification for the programmer.
- **Reverse engineering:** Model as a code browser.

- Tools provide both forward and reverse engineering to move back and forth between the program and the code.

6. As a programming language:

- UML diagrams can be automatically compiled into working code using sophisticated tools, such as Visual Paradigm.

- **Things to Model:**
- Structure of the code:
- Code dependencies.
- Components and couplings.
- Behaviour of the code:
- Execution traces.
- State machine models of complex objects.
- Function of the code:
- What function does it provide to the user?

Class Diagram:

- **Introduction to Class Diagrams:**
- 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. It displays the system's classes, attributes, and methods. It is helpful in recognizing the relationship between different objects as well as 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
- **Notation:**
- 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.

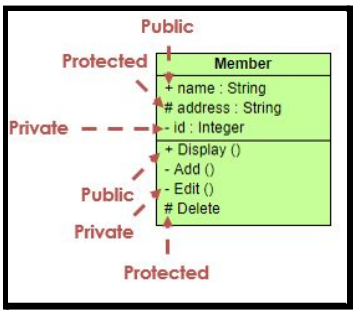
2. **Data members/Attributes**

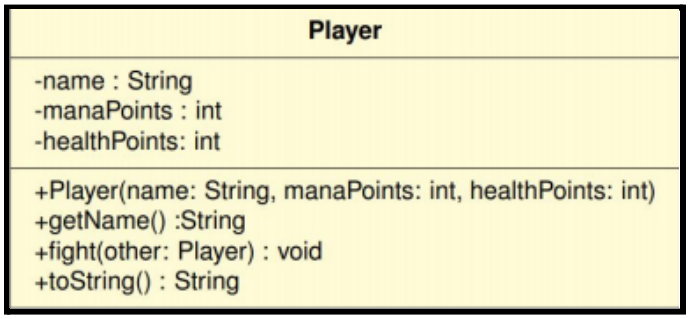
- The data members section of a class lists each of the class's data members on a separate line.
- Each line uses this format: **attributeName : type**
  E.g. **name : String**
- We must underline static attributes.

3. **Methods/Operations**

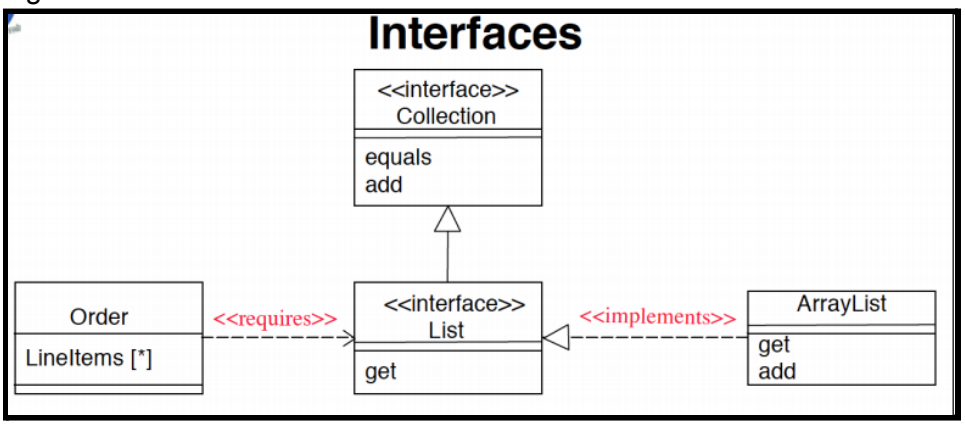
- The methods of a class are displayed in a list format, with each method on its own line.
- Each line uses this format:
  methodName(param1: type1, param2: type2, ...) : returnType
  E.g. **distance(p1: Point, p2: Point) : Double**
- We may omit setters and getters. However, don't omit any methods from 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.



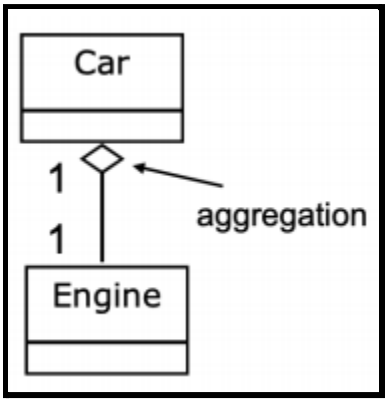


- Inheritance/Generalization and Realization Relationships:
- **Generalization/inheritance** is when a class extends another class while **realization** is when a class implements an interface.
- Generalization represents a “IS-A” relationship.
- Hierarchies are drawn top down with arrows pointing upward 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.



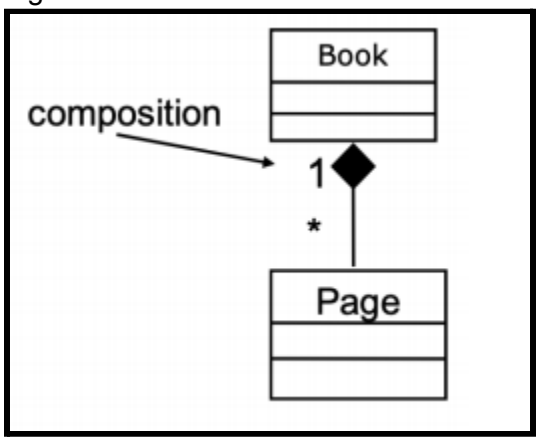
- Association:
- An **association** represents a relationship between two classes. It also defines the multiplicity between objects.
- Association can be represented by a line between the classes with an arrow indicating the navigation direction.
  Note: Sometimes, association can be represented just 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.

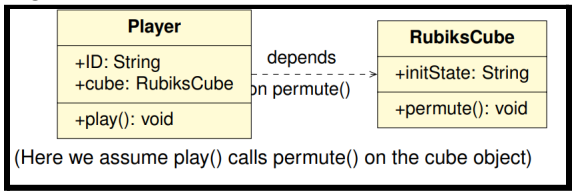


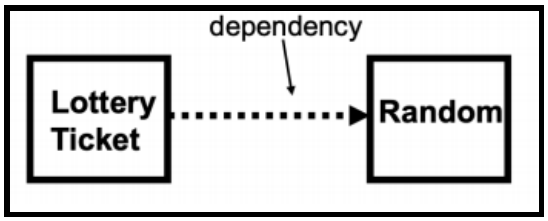
- Aggregation is considered as a weak type of association.
- Composition:
- A special type of association. Composition is considered as a strong type of association.
- It is a stronger version of aggregation where if you delete the parent class, then all the child classes are also deleted.
  I.e. Composition 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.

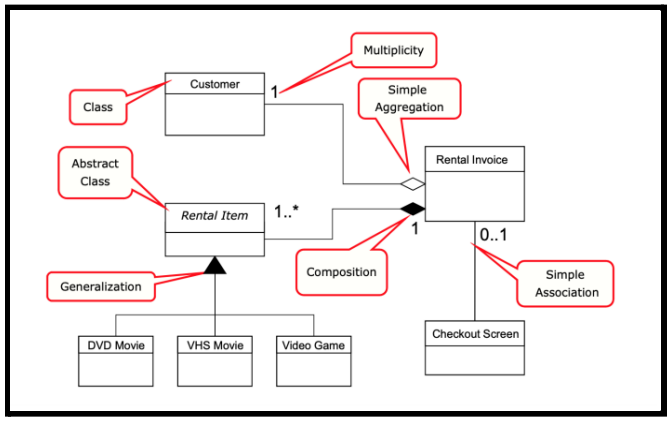


- Dependency:
- Is a special type of association.
- Dependency indicates a “uses” relationship between two classes. If a change in structure or behaviour of one class affects another class, then there is a dependency between those two classes.
- Dependency is represented by a dotted arrow where the arrowhead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

- **Summary:**
- 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.
- 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.

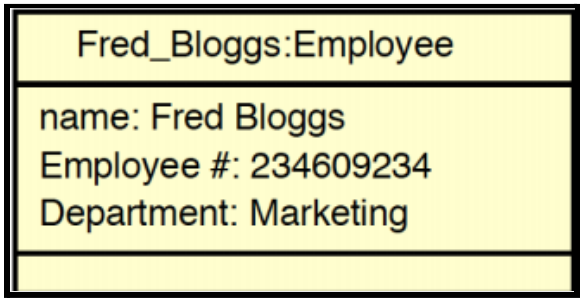
- Others:

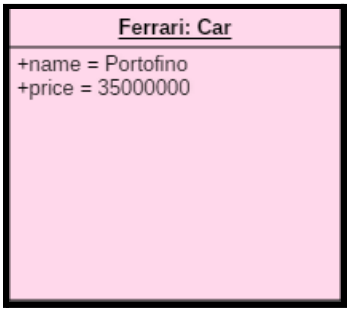
Item	Explanation	Depiction
Generalization/inheritance	When a class extends another class. Generalization represents a “IS-A” relationship.	For a class, draw a solid line with a black arrow pointing to the parent class. For an abstract class, draw a solid line with a white arrow pointing to the parent abstract class. For an interface, draw a dashed line with a white arrow pointing to the interface.
Realization	When a class implements an interface	
Association	Represents a relationship between two classes. It also defines the multiplicity between objects.	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.
Aggregation	A special type of association. It is a weak 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.	An arrow with a clear white diamond arrowhead pointing to the parent class.

Composition	A special type of association. Composition is considered as a strong 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.	An arrow with a black diamond arrowhead pointing to the parent class.
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.	A dotted arrow where the arrowhead points to the independent element.

Object Diagram:

- Object diagrams look very similar to class diagrams.
- **Naming Convention:**
  Object name: Type
  Attribute: Value (Sometimes, it's Attribute = Value)
  E.g.



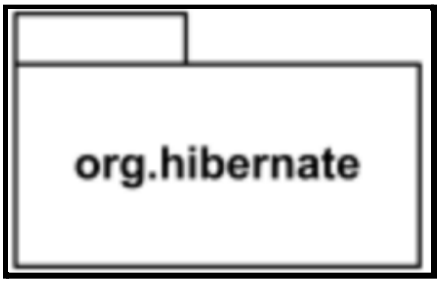


- **Note:** 2 different objects may have identical attribute values.
- **Purpose:**
- It is used to describe the static aspect of a system.
- It is used to represent an instance of a class.
- It can be used to perform forward and reverse engineering on systems.
- It is used to understand the behavior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

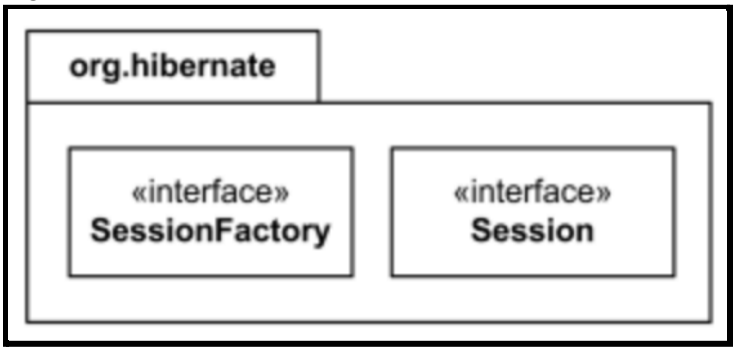
UML Packages:

- **Introduction:**
- A **package** is a namespace used to group together elements that are semantically related and might change together. It is a general purpose mechanism to organize elements into groups to provide a better structure for a system model.
- **UML package diagrams** are structural diagrams used to show the organization and arrangement of various model elements in the form of packages. A package is a grouping of related UML elements, such as diagrams, documents, classes, or even other packages. Each element is nested within the package, which is depicted as a file folder, and then is arranged hierarchically within the diagram. Package diagrams are most commonly used to provide a visual organization of the layered architecture within any UML classifier, such as a software system.
- Package diagrams are UML structure diagrams which show packages and dependencies between the packages.
  Note: Structure diagrams do not utilize time related concepts and do not show the details of dynamic behavior.
- If a package is removed from a model, so are all the elements owned by the package.
- A package could also be a member of other packages.
- A package in the UML helps:
- To group elements.
- To provide a namespace for the grouped elements.
- Provide a hierarchical organization of packages.
- Benefits of UML package diagrams:
- They provide a clear view of the hierarchical structure of the various UML elements within a given system.
- These diagrams can simplify complex class diagrams into well-ordered visuals.
- They offer valuable high-level visibility into large-scale projects and systems.
- Package diagrams can be used to visually clarify a wide variety of projects and systems.
- These visuals can be easily updated as systems and projects evolve.
- **Terminology:**
- **Package:** A namespace used to group together logically related elements within a system. Each element contained within the package should be a packageable element and have a unique name.
- **Packageable element:** A named element, possibly owned directly by a package. These can include events, components, use cases, and packages themselves. Packageable elements can also be rendered as a rectangle within a package, labeled with the appropriate name.
- **Dependencies:** A visual representation of how one element or set of elements depends on or influences another. Dependencies are divided into two groups: access and import dependencies.
- **Access dependency:** Indicates that one package requires assistance from the functions of another package.
  I.e. One package requires help from functions of another package. (Making an API call for example)
- **Import dependency:** Indicates that functionality has been imported from one package to another.
  I.e. One package imports the functionality of another package. (Importing a package)

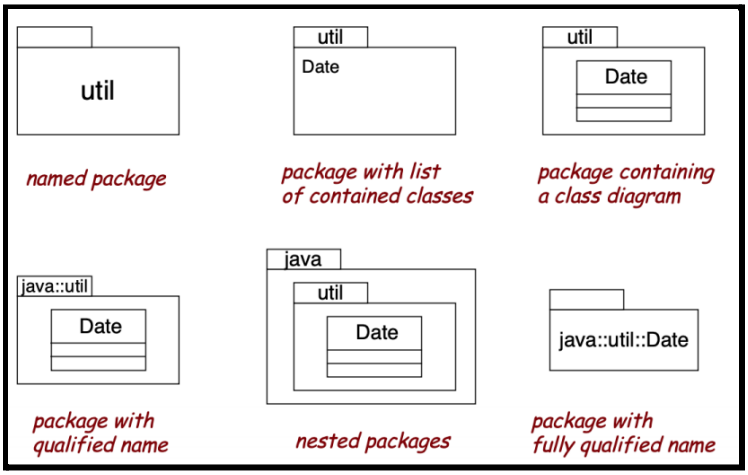
- **Notation:**
- A package is rendered as a rectangle with a small tab attached to the left side of the top of the rectangle. If the members of the package are not shown inside the package rectangle, then the name of the package should be placed inside.
  E.g.



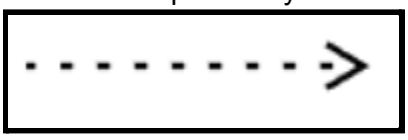
- The members/elements of the package may be shown within the boundaries of the package. If the names of the members of the package are shown, then the name of the package should be placed on the tab.
  E.g.



  Here, Package org.hibernate contains SessionFactory and Session.
- More examples:



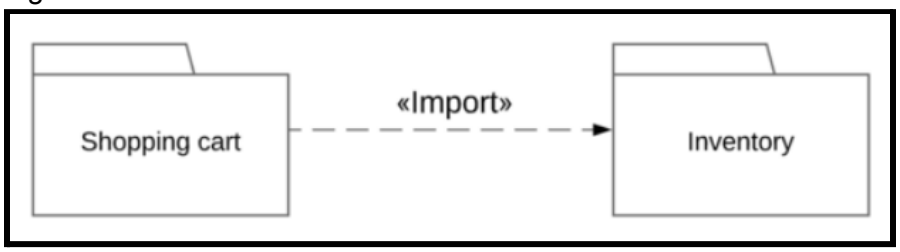
- To show a dependency between 2 packages, you draw a dotted arrow,



  , between the 2 packages such that the arrow is pointing to the independent package.

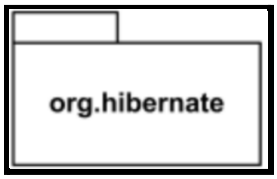
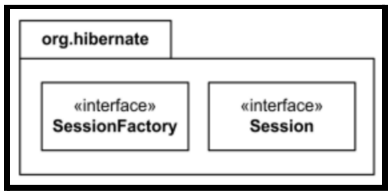
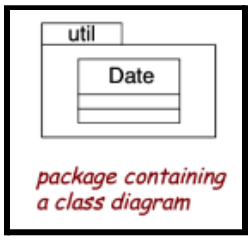
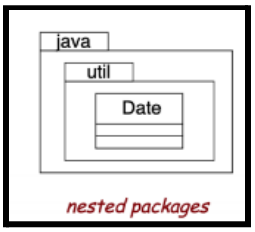
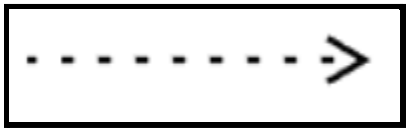
- To show an access dependency, write <<Access>> on the dotted arrow.
  E.g.

- To show an import dependency, write <<Import>> on the dotted arrow.
  E.g.



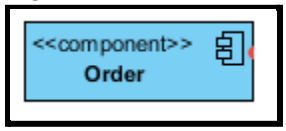
- **Criteria for Decomposing a System into Packages:**
- Different owners - who is responsible for working on which diagrams?
- Different applications - each problem has its own obvious partitions.
- Clusters of classes with strong cohesion - E.g. course, course description, instructor, student, etc.
- Or: Use an architectural pattern to help find a suitable decomposition such as the MVC Framework.
- **Other Guidelines for Packages:**
- Gather model elements with strong cohesion in one package.
- Keep model elements with low coupling in different packages.
- Minimize relationships, especially associations, between model elements in different packages.
- Namespace implication: An element imported into a package does not know how it is used in the imported package.
- We want to avoid dependency cycles.

- **Summary:**

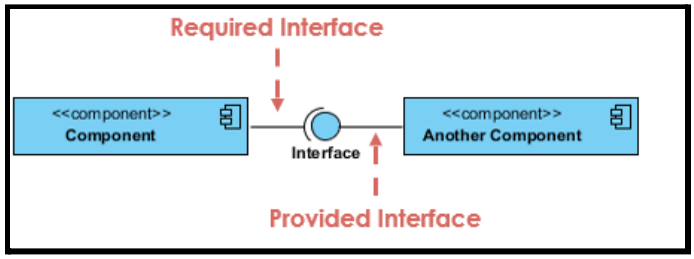
Item & Example	Description	Depiction
Package  	A namespace used to group together logically related elements within a system. Each element contained within the package should be a packageable element and have a unique name.	A rectangle with a small tab attached to the left side of the top of the rectangle. If the members of the package are not shown inside the package rectangle, then the name of the package should be placed inside.
Packageable element  	A named element, possibly owned directly by a package. These can include events, components, use cases, and packages themselves.	Packageable elements can also be rendered as a rectangle within a package, labeled with the appropriate name.
Dependency 	A visual representation of how one element or set of elements depends on or influences another. Dependencies are divided into two groups: access and import dependencies.	To show a dependency between 2 packages, you draw a dotted arrow, between the 2 packages such that the arrow is pointing to the independent package.
Access dependency 	Indicates that one package requires assistance from the functions of another package.	To show an access dependency, write <<Access>> on the dotted arrow.
Import dependency 	Indicates that functionality has been imported from one package to another.	To show an import dependency, write <<Import>> on the dotted arrow.

Component Diagrams:

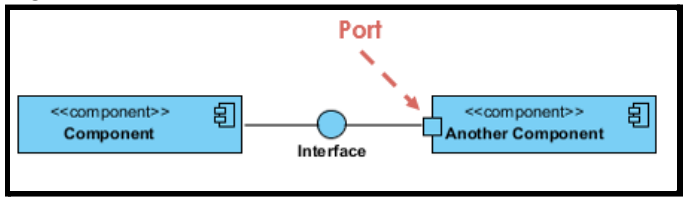
- **Introduction:**
- **Component diagrams** are used in modeling the physical aspects of object-oriented systems that are used for visualizing, specifying, and documenting component-based systems and also for constructing executable systems through forward and reverse engineering.
- A component diagram breaks down the actual system under development into various high levels of functionality.
- Each component is responsible for one clear aim within the entire system and only interacts with other essential elements on a need-to-know basis.
- Component diagrams can help your team:
    - Imagine the system's physical structure.
    - Pay attention to the system's components and how they relate.
    - Emphasize the service behavior as it relates to the interface.
- A component diagram gives a bird's-eye view of your software system. Understanding the exact service behavior that each piece of your software provides will make you a better developer. Component diagrams can describe software systems that are implemented in any programming language or style.
- **Notation:**
- **Component:** A rectangle with the component's name, stereotype text, and icon. A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.
  E.g.



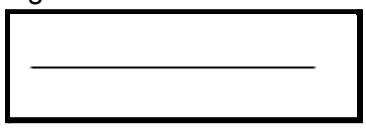
- **Interface:** There are 2 types of interfaces, **provided interface** and **required interface**.
  Provided interface: A complete circle with a line connecting to a component. Provided interfaces provide items to components.
  Required Interface: A half circle with a line connecting to a component. Required interfaces are used to provide required information to a provided interface.
  E.g.



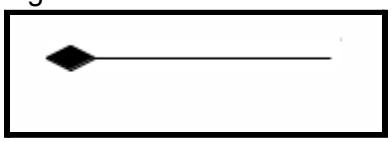
- **Port:** A square along the edge of the system or a component. A port is often used to help expose required and provided interfaces of a component. Ports are used to hook up other elements in a component diagram.
  E.g.



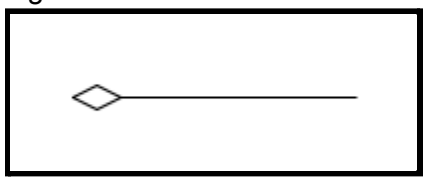
- **Association:** An association specifies a relationship that can occur between two instances. You represent an association using a straight line connecting 2 components.
  E.g.



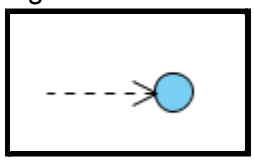
- **Composition:** Composition is a stronger form of aggregation that requires a part instance to be included in at most one composite at a time. If a composite is deleted, all of its parts are normally deleted with it. Composition is a type of association. You can represent a composition using an arrow where the arrowhead is filled in and points to the parent class.
  E.g.



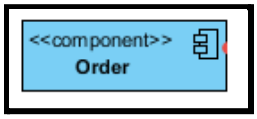
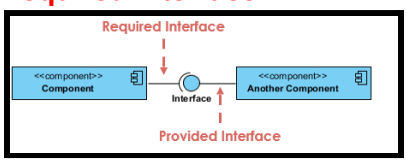
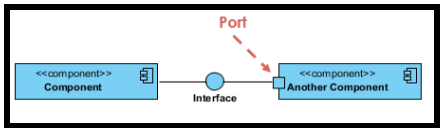
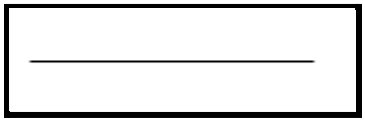
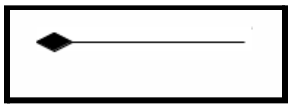
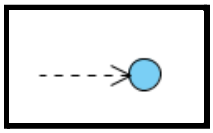
- **Aggregation:** Aggregation implies a relationship where the child class can exist independently of the parent class. This means that if you remove/delete the parent class, the child class still exists. It is a special type of association and a weak form of association. You can represent an aggregation using an arrow where the arrowhead is not filled in and points to the parent class.
  E.g.



- **Dependency:** A dependency is a relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation. It is denoted as a dotted arrow with a circle at the tip of the arrow.
  E.g.



- **Summary:**

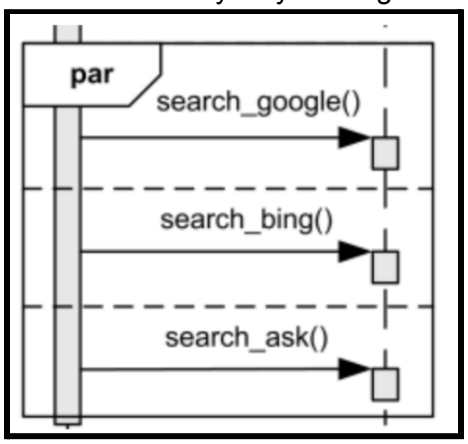
Item & Example	Description	Depiction
Component 	A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.	A rectangle with the component's name, stereotype text, and icon.
Provided interface **Required Interface** 	Provided interfaces provide items to components. Required interfaces are used to provide required information to a provided interface.	A complete circle with a line connecting to a component. A half circle with a line connecting to a component.
Port 	A port is often used to help expose required and provided interfaces of a component. Ports are used to hook up other elements in a component diagram.	A square along the edge of the system or a component.
Association 	An association specifies a relationship that can occur between two instances.	A straight line connecting 2 components.
Aggregation 	Aggregation implies a relationship where the child class can exist independently of the parent class. It is a weak form of association.	An arrow where the arrowhead is not filled in and points to the parent class.
Composition 	Composition is a stronger form of aggregation that requires a part instance to be included in at most one composite at a time. Composition is a type of association.	An arrow where the arrowhead is filled in and points to the parent class.
Dependency 	A dependency is a relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation.	It is denoted as a dotted arrow with a circle at the tip of the arrow.

Interaction Diagrams:

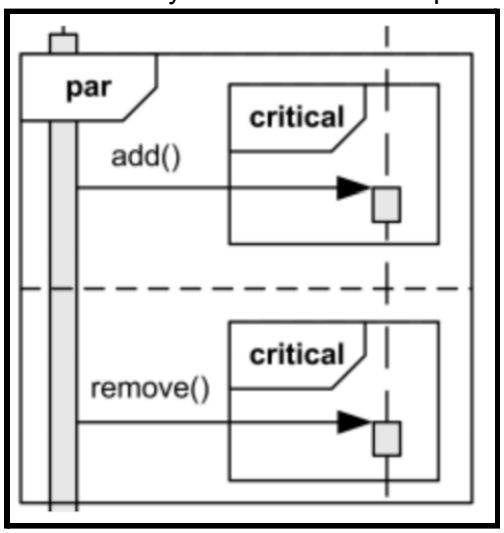
- **Interaction diagrams** describe how a group of objects collaborate in some behavior. They commonly contain objects, links and messages.
- Objects communicate with each other through function/method calls called **messages**.
- An interaction is a set of messages exchanged among a set of objects in order to accomplish a specific goal.
- Interaction diagrams:
  - Are used to model the dynamic aspects of a system.
  - Aid the developer in visualizing the system as it is running.
  - Are storyboards of selected sequences of message traffic between objects.
- After class diagrams, interaction diagrams are possibly the most widely used UML diagrams.
- A **lifeline** represents a single participant in an interaction. It describes how an instance of a specific classifier participates in the interaction. A lifeline represents a role that an instance of the classifier may play in the interaction.
- A **message** is the vehicle by which communication between objects is achieved. A function/method call is the most common type of message. The return of data as a result of a function call is also considered a message.
- A message may result in a change of state for the receiver of the message.
- The receipt of a message is considered an instance of an event.
- Interactions model the dynamic aspects of a system by showing the message traffic between a group of objects. Showing the time-ordering of the message traffic is a central ingredient of interactions.
- Graphically, a message is represented as a directed line that is labeled.
- The **sequence diagram** is the most commonly used UML interaction diagram. Typically a sequence diagram captures the behavior of a group of objects in a single scenario.
- Interaction Frame Operators:

Operator	**Name**	**Meaning**
Opt	Option	An operand is executed if the condition is true. (E.g. If-else)
Alt	Alternative	The operand, whose condition is true, is executed. (E.g. Switch)
Loop	Loop	It is used to loop an instruction for a specified period.
Break	Break	It breaks the loop if a condition is true or false, and the next instruction is executed.
Ref	Reference	It is used to refer to another interaction.
Par	Parallel	All operands are executed in parallel.
Region	Critical Region	Only 1 thread can execute this frame at a time.
Neg	Negative	Frame shows an invalid interaction.
Sd	Sequence Diagram	(Optional) Used to surround the whole diagram.

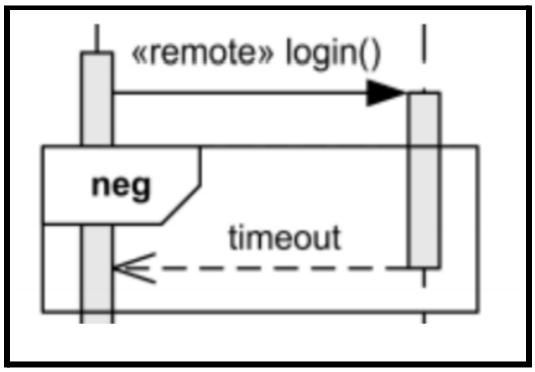
- **Parallel Example:** The interaction operator par defines potentially parallel execution of behaviors of the operands of the combined fragment. Different operands can be interleaved in any way as long as the ordering imposed by each operand is preserved.



- **Region Example:** The interaction operator region defines that the combined fragment represents a critical region. A critical region is a region with traces that cannot be interleaved by other occurrence specifications on the lifelines covered by the region.

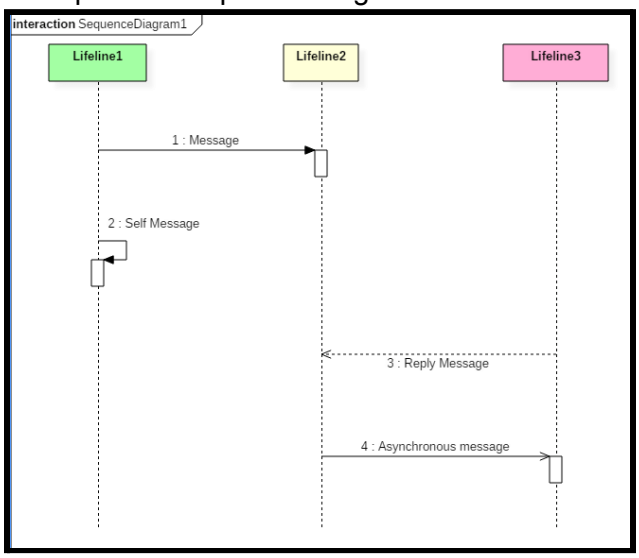


- **Negative Example:** The interaction operator neg describes a combined fragment of traces that are defined to be negative (invalid). Negative traces are the traces which occur when the system has failed. All interaction fragments that are different from the negative are considered positive, meaning that they describe traces that are valid and should be possible.

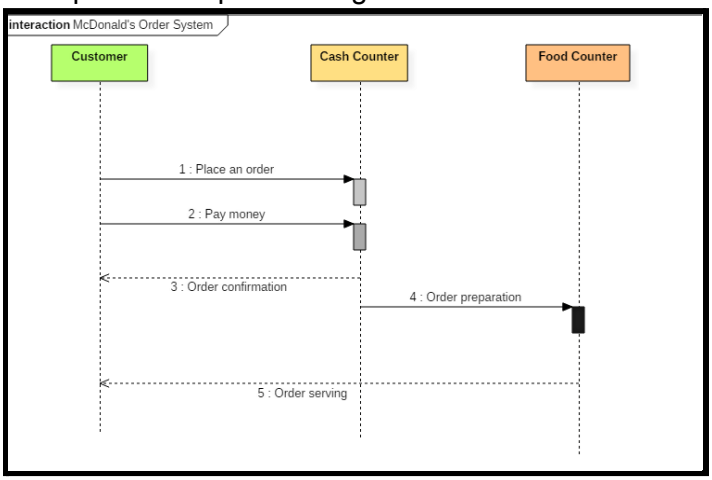


Sequence Diagrams:

- **Introduction:**
- A **sequence diagram** depicts interactions between objects in a sequential order. The purpose of a sequence diagram in UML is to visualize the sequence of a message flow in the system. The sequence diagram shows the interaction between two lifelines as a time-ordered sequence of events.
- A sequence diagram shows an implementation of a scenario in the system. Lifelines in the system take part during the execution of a system.
- In a sequence diagram, a lifeline is represented by a vertical bar.
- A message flow between two or more objects is represented using a vertical dotted line which extends across the bottom of the page.
- Sequence diagrams are built around an X-Y axis.
- Objects are aligned at the top of the diagram, parallel to the X axis.
- Messages travel parallel to the X axis.
- Time passes from top to bottom along the Y axis.
- Sequence diagrams most commonly show relative timings, not absolute timings.
- Links between objects are implied by the existence of a message.
- Example of a sequence diagram:

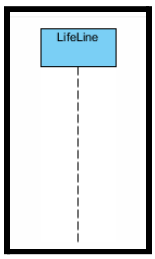
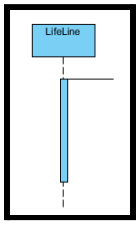
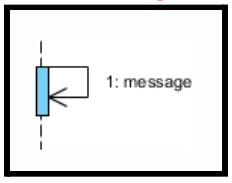


- Example of a sequence diagram:



- **Benefits of a sequence diagram:**
- Sequence diagrams are used to explore any real application of a system.
- Sequence diagrams are used to represent the message flow from one object to another.
- Sequence diagrams are easy to maintain and generate.
- Sequence diagrams can be easily updated according to the changes within a system.
- Sequence diagrams allow both reverse and forward engineering.
- **Drawbacks of a sequence diagram:**
- Sequence diagrams can become complex when too many lifelines are involved in the system.
- If the order of message sequence is changed, then incorrect results are produced.
- Each sequence needs to be represented using different message notation, which can be a little complex.
- The type of message decides the type of sequence inside the diagram.
- **When to use sequence diagrams:**
    1. Comparing Design Options:
        - Shows how objects collaborate to carry out a task.
        - Graphical form shows alternative behaviours.
    2. Assessing Bottlenecks
    3. Explaining Design Patterns:
        - Enhances structural models.
        - Good for documenting behaviour of design features.
    4. Elaborating Use Cases:
        - Shows how the user expects to interact with the system.
        - Shows how the user interface operates.
- **Modelling Control Flow By Time:**
- Determine what scenarios need to be modeled.
- Identify the objects that play a role in the scenario.
- Lay the objects out in a sequence diagram left to right, with the most important objects on the left.
  Most important in this context means objects that are the principle initiators of events.
- Draw in the message arrows, top to bottom.
  Adorn the message as needed with detailed timing information.
- **Style Guide for Sequence Diagrams:**
    1. Spatial Layout:
        - Strive for left-to-right ordering of messages.
        - Put proactive actors on the left.
        - Put reactive actors on the right.
    2. Readability:
        - Keep diagrams simple.
        - Don’t show obvious return values.
        - Don’t show object destruction.
    3. Usage:
        - Focus on critical interactions only.
    4. Consistency:
        - Class names must be consistent with class diagram.
        - Message routes must be consistent with navigable class associations.

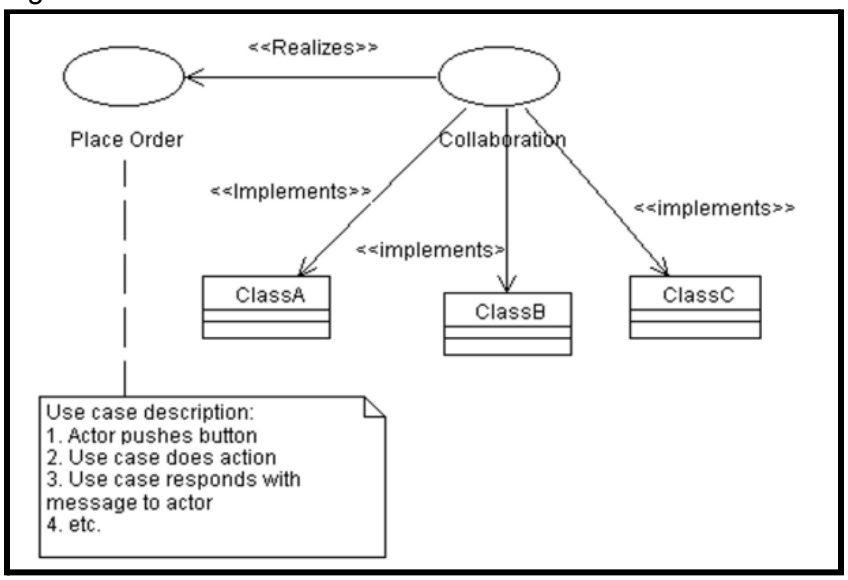
- **Summary:**

Item & Example	Description	Depiction
Lifeline 	A lifeline represents an individual participant in the interaction. Lifelines represent the passage of time as it extends downward. The dashed vertical line shows the sequential events that occur to an object during the charted process.	A labeled rectangle shape with a dotted line extending from its bottom.
Activation box 	Represents the time needed for an object to complete a task. The longer the task will take, the longer the activation box becomes.	A thin rectangle on a lifeline. The top and the bottom of the rectangle are aligned with the initiation and the completion time respectively.
Message 	A message defines a particular communication between lifelines.	A solid line with a solid arrowhead.
Asynchronous Message 	Asynchronous messages don't require a response before the sender continues. Only the call should be included in the diagram.	A solid line with a lined arrowhead.
Reply Message 	A return message is a kind of message that represents the pass of information back to the caller of a corresponded former message.	A dashed line with a lined arrowhead.
Self Message 	A self message is a kind of message that represents the invocation of a message of the same lifeline.	A solid line with a lined arrowhead pointing to the same lifeline that it originated from.

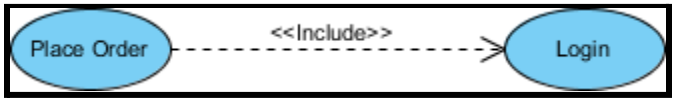
Use Case Diagrams:

- **Introduction:**
- A **use case diagram** is the primary form of system/software requirements for a new software program.
- Use cases specify the expected behavior (what), and not the exact method of making it happen (how).
- A key concept of use case modeling is that it helps us design a system from the end user's perspective. It is an effective technique for communicating a system's behavior in the user's terms.
- Use case diagrams are used to gather the requirements of a system including internal and external influences.
- A use case:
    - Specifies the behavior of a system or some subset of a system.
    - Is a set of scenarios tied together by a common user goal.
    - Does not indicate how the specified behavior is implemented, only what the behavior is.
    - Performs a service for some user of the system, called an **actor**.
- A use case represents a functional requirement of the system. A requirement:
    - Is a design feature, property, or behavior of a system.
    - States what needs to be done, but not how it is to be done.
    - Is a contract between the customer and the developer.
    - Can be expressed in various forms, including use cases.
- In brief, the purposes of use case diagrams are as follows:
    - Used to gather the requirements of a system.
    - Used to get an outside view of a system.
    - Identify the external and internal factors influencing the system.
    - Show the interaction among the requirements of the actors.
- An actor:
    - Is a role that the user plays with respect to the system. The user does not have to be human.
    - Is associated with one or more use cases.
    - Is most typically represented as a stick figure of a person labeled with its role name. Note that the role names should be nouns.
    - May exist in a generalization relationship with other actors in the same way as classes may maintain a generalization relationship with other classes.
- **Note:** Use cases diagrams do not show the order in which the steps are performed to achieve the goals of each use case. It only shows the relationship between actors, systems and use cases.
- Use cases are a technique for capturing the functional requirements of a system. Use cases work by describing the typical interactions between the users of a system and the system itself, providing a narrative of how the system is used.
- Use case development process:
    1. Develop multiple scenarios.
    2. Distill the scenarios into one or more use cases where each use case represents a functional requirement.
    3. Establish associations between the use cases and actors.
- A use case is graphically represented as an oval with the name of its functionality written inside. The functionality is always expressed as a verb or a verb phrase.
- A use case may exist in relationships with other use cases much in the same way as classes maintain relationships with other classes.

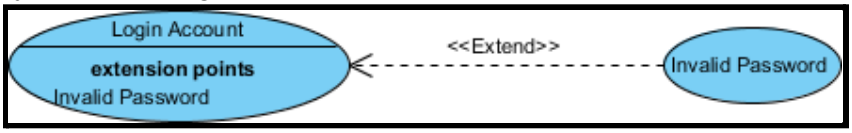
- As stated earlier, a use case by itself does not describe the flow of events needed to carry out the use case. The flow of events can be described using informal text, pseudocode, or activity diagrams.
  I.e. You can attach a note to a use case to show the flow of the event. Be sure to address exception handling when describing the flow of events.
  E.g.



- **Relationships Between Use Cases:**
- A use case may have a relationship with other use cases.
- Generalization between use cases is used to extend the behavior of a parent use case.
- An **<<include>> relationship** between use cases means that the base use case explicitly incorporates the behavior of another use case at a location specified in the base.
  Note: Sometimes **<<uses>>** is used instead of **<<include>>**.
  When a use case is depicted as using the functionality of another use case, the relationship between the use cases is named as an **<<include>>** or **<<uses>>** relationship.

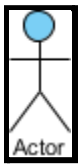


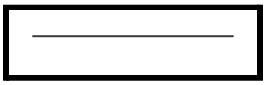
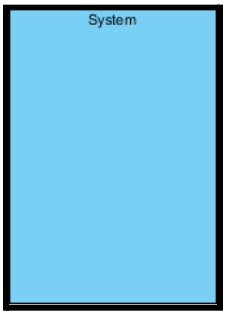
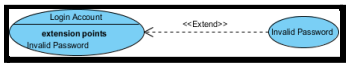
- An **<<extend>> relationship** between use cases means that the base use case implicitly incorporates the behavior of another use case at a location specified indirectly by the extending use case.



- Extended behavior is optional behavior, while included behavior is required behavior.
  I.e. Extended means “may use” while include/uses means “will use”.
- Extend occurs when one use case adds a behaviour to a base use case while include occurs when one use case invokes another.
- **Actor Classes:**
- Identify classes of actors.
- Actors inherit use cases from the class.

- **Describing Use Cases:**
- For each use case, a flow of events document, written from the actor's point of view, describes what the system must provide to the actor when the use case is executed.
- Typical contents:
  - How the use case starts and ends.
  - Normal flow of events.
  - Alternate flow of events.
  - Exceptional flow of events.
- Documentation style:
  - Activity Diagrams - Good for business process.
  - Collaboration Diagrams - Good for high level design.
  - Sequence Diagrams - Good for detailed design.
- **Finding Use Cases:**
- Noun phrases may be domain classes.
- Verb phrases may be operations and associations.
- Possessive phrases may indicate attributes.
- **For each actor, ask the following questions:**
  1. What functions does the actor require from the system?
  2. What does the actor need to do?
  3. Does the actor need to read, create, destroy, modify or store information in the system?
  4. Does the actor have to be notified about events in the system?
  5. Does the actor need to notify the system about something?
  6. What do these events require in terms of system functionality?
  7. Could the actor's daily work be simplified or made more efficient through new functions provided by the system?
- **Summary:**

Item & Example	Description	Depiction
Actors 	Is a role that the user plays with respect to the system. The user does not have to be human.	A stick figure.
Use Case 	Represents a functional requirement of the system. It specifies the behavior of a system or some subset of a system. It is a set of scenarios tied together by a common user goal. It does not indicate how the specified behavior is implemented, only what the behavior is.	An oval

Association 	Shows which actors use which use cases.	A line connecting an actor to a use case.
System Boundary Box 	The system boundary is potentially the entire system as defined in the requirements document. It is a box that sets a system scope to use cases. All use cases outside the box would be considered outside the scope of that system. For large and complex systems, each module may be the system boundary.	A blue box containing all the relevant use cases.
<<include>>/<<uses>> relationship 	An <<include>>/<<uses>> relationship between use cases means that the base use case explicitly incorporates the behavior of another use case at a location specified in the base.	A dotted line with a lined arrowhead originating from a use case pointing to the used use case. It has <<includes>> or <<uses>> written on the arrow.
<<extend>> relationship 	An <<extend>> relationship between use cases means that the base use case implicitly incorporates the behavior of another use case at a location specified indirectly by the extending use case.	A dotted line with a lined arrowhead originating from a use case pointing to the used use case. It has <<extends>> written on the arrow.