Unified Modeling Language

Dao Anh Vu

I. Introduction to UML

II. Basic Notations

III. Modeling types

IV Diagrams

V Exercises

VI Final project

Introduction

- Unified Modeling Language (UML) is a standard language for creating blueprints that depicts structure and design of the software system.
- A general-purpose, developmental, modeling language in the field of software engineering, that is intended to provide a standard way to visualize the design of a system – *Wikipedia*.

Introduction

- Rational Software Corporation defines UML as follows: “The Unified Modeling Language (UML) is a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system.”
- UML can be defined as a language for:
  - Specifying artifacts
  - Visualizing artifacts
  - Constructing artifacts
  - Documenting artifacts

History

- In late 1980s, object-oriented modeling languages were developed for analysis and design of the software.
- 1994 – 1995: Booch’s Booch’93 , Jacobson’s Object Oriented Software Engineering (OOSE) and Rumbaugh’s Object Modeling Technique-2 (OMT) were developed at Rational Software.
- In October 1994, the unification of Booch′93, OMT, and OOSE led to the release of version 0.9 and 0.91 of UML.
- In 1997, UML 1.0 was adopted as a standard by OMG.
- In 2005, UML 2.0 was introduced as the next standard by OMG.

Building Blocks

- UML building blocks include:
  - **Basic UML constituents**: Include the static, dynamic, grouping, and annotational constituents of UML.
  - **Relationships**: Depict the relations between various constituents of a UML model.
  - **Diagrams**: Represent the various artifacts of a system graphically.

Standard diagrams

- **Use case diagrams**
- **Class diagrams**
- Object diagrams
- Communication diagrams
- **Sequence diagrams**
- **Statechart diagrams**
- **Activity diagrams**
- Package Diagrams
- Component diagrams
- Deployment diagrams
- Timing Diagrams
- Composite Structure Diagrams
- Interaction Overview Diagrams

Modeling types

- **Structural modeling**
  - Classes diagrams
  - Objects diagrams
  - Deployment diagrams
  - Package diagrams
  - Composite structure diagram
  - Component diagram

- **Behavioral Modeling**
  - Activity diagrams
  - Interaction diagrams
  - Use case diagrams

- **Architectural Modeling**
  - ✓ **Structural modeling**
  - ✓ **Behavioral modeling**

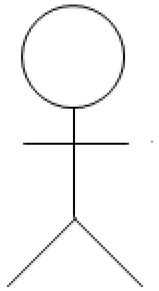
UML Diagrams

Use case diagram

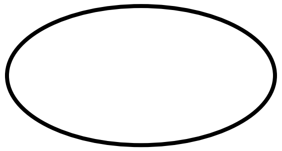
- A use case diagram:
  - Present the usage requirements of the system.
  - It is the "means" of communicating with user and other stakeholders what the system is intended to do.

- Use case diagram consists of:
  - **Actors**: entities those are external to the system
  - **Use cases**: A use case is a single unit of meaningful work. It provides a high-level view of behavior observable to someone or something outside the system. The notation for a use case is an ellipse
  - **Relationships**: Associations between actors and use cases

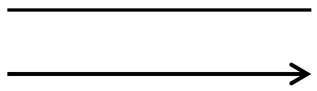
Use case Diagrams - Notations



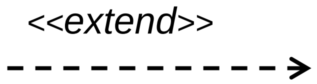
Actors: entities those are external to the system



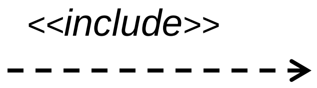
Use cases: A use case is a single unit of meaningful work. It provides a high-level view of behavior observable to someone or something outside the system.



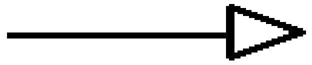
Associations between actors and use cases



Extend



Include



Generalization

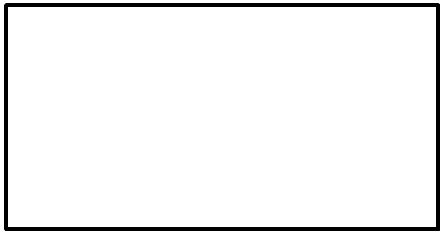
Finding Actors: Useful Questions

- Who is interested in a certain requirement?
- Where in the organization is the system used?
- Who will supply the system with this information, use this information, remove this information?
- Who will use this function?
- Who will support and maintain the system?
- Does the system use an external resource?
- What actors do the use cases need?
- Does one actor play several different roles? Do several actors play the same role?

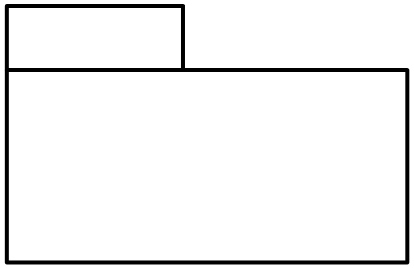
Finding Actors: Useful Questions

- Who is interested in a certain requirement?
- Where in the organization is the system used?
- Who will supply the system with this information, use this information, remove this information?
- Who will use this function?
- Who will support and maintain the system?
- Does the system use an external resource?
- What actors do the use cases need?
- Does one actor play several different roles? Do several actors play the same role?

Use case diagram - Notations



System boundary boxes: **define the scope of the use case (***optional***)**.

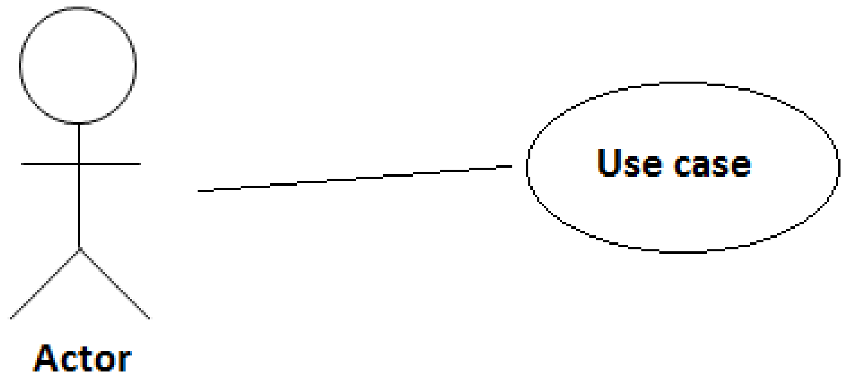


Packages are **used to group together use cases** *(Optional)*

Use case diagram

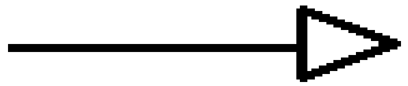
Actor - use case associations:

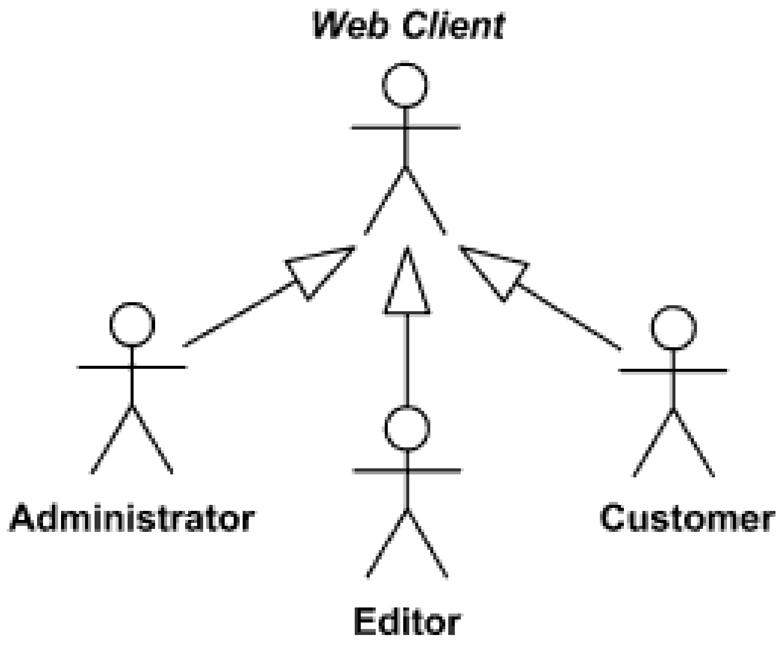
- An association between an actor and a use case indicates that the actor and the use case somehow interact or communicate with each other.
- Only binary associations are allowed between actors and use cases.
- An actor could be associated to one or several use cases.



Use case diagram – Relationship between actors

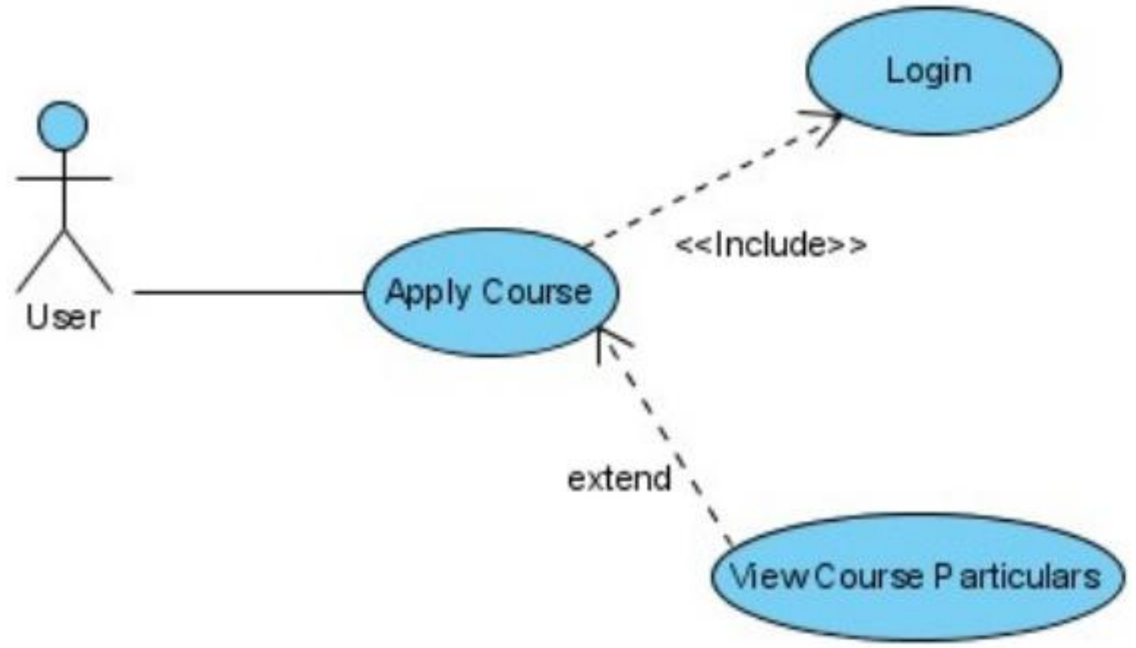
- Actor generalization:





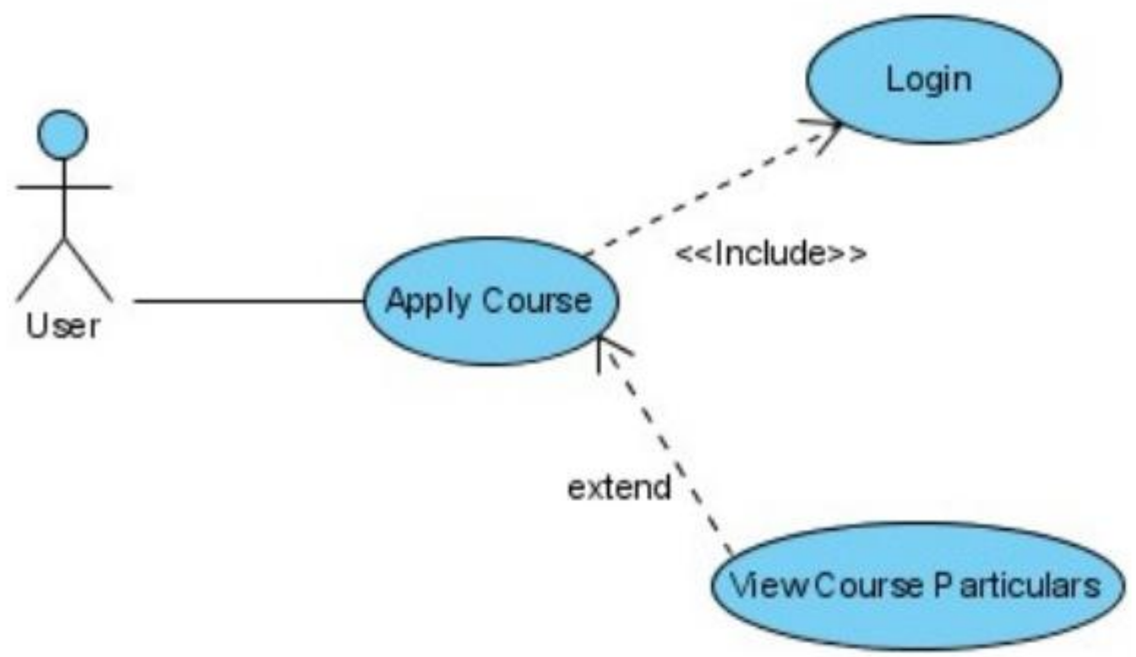
Use case diagram – Include

- Include dependency
  - A base use case is dependent on the included use case(s).
  - Without them the base use case is incomplete as the included use case(s) represent sub-sequences of the interaction.

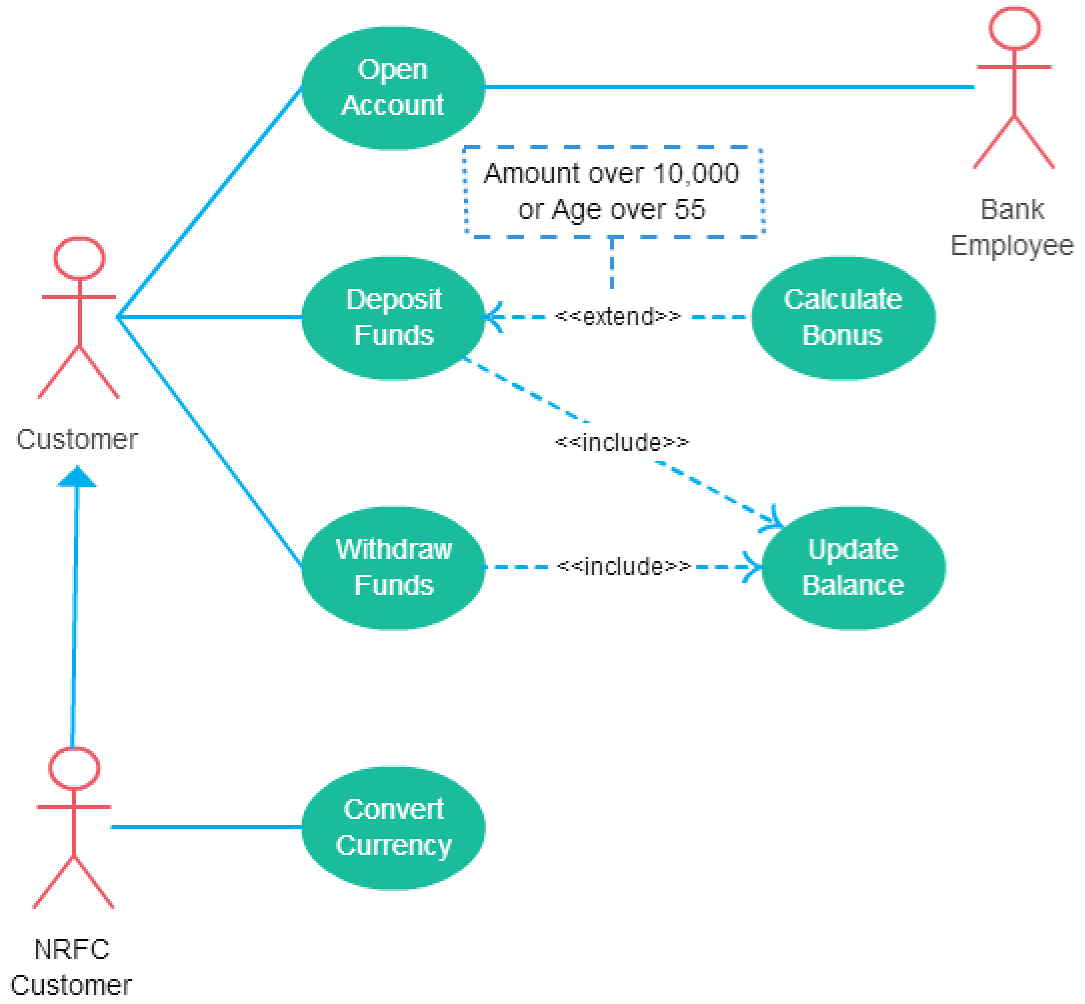


Use case diagram – Extend

- Extend dependency
  - Extends the base use case by inserting additional action sequence the base use case.
  - The base use case should be a fully functional use case in its own right without the extending use case's



Use case diagram - Example



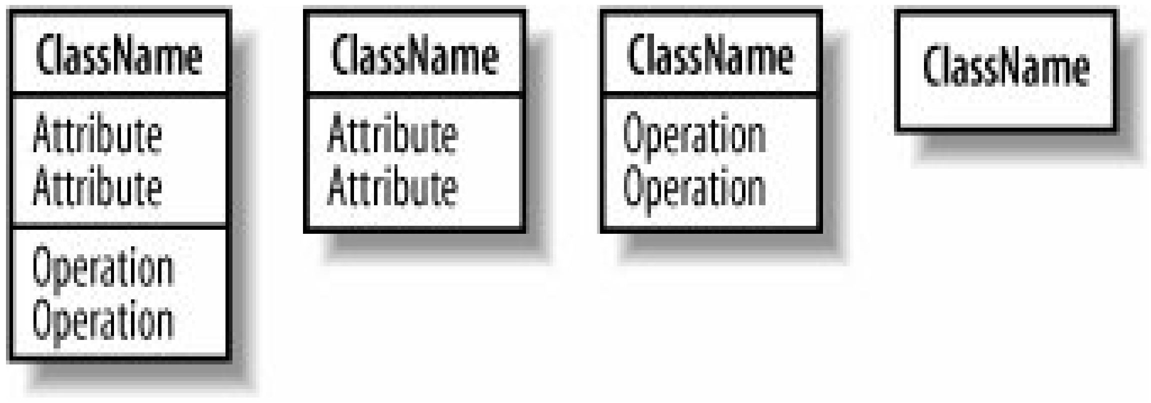
Use case diagram - Specification

- Writing a specification is the final step in defining a use case.
- This document outlines the actors, preconditions, flow of events of a use case.

Class diagram

- A class is a type of something. You can think of a class as a blueprint out of which objects can be constructed

- Class in UML is represented as a rectangle split into up to three sections. The top section contains the name of the class, the middle section contains the attributes of the class and the final section contains the operations (behaviors) of the class.



Class diagram

- There are four types of visibility characteristics to apply to the elements of a UML model.
  - Public visibility: accessible by any class
  - Protected visibility: more restricted than public, accessible by inherited classes
  - Package visibility: only accessible by classes reside in the same package
  - Private visibility: only accessible by the classes contain the element

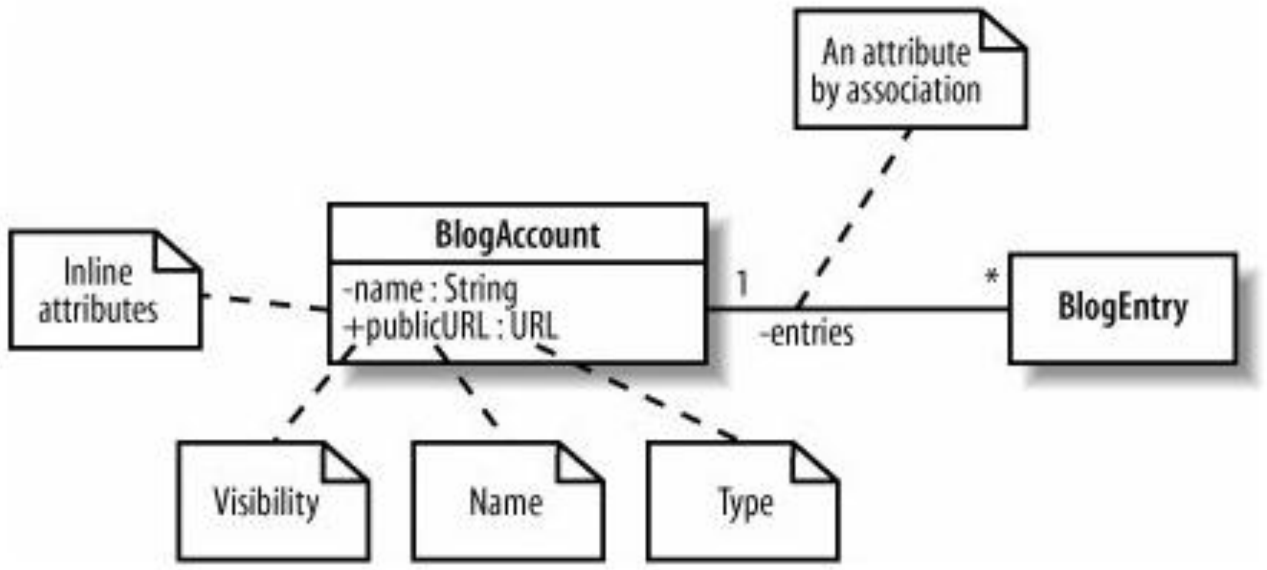
Name	Public	Protected	Package	Private
(Notation)	(+)	(#)	(~)	(-)

← **More accessible to the other parts of the system**

Less accessible to the other parts of the system →

Class diagram

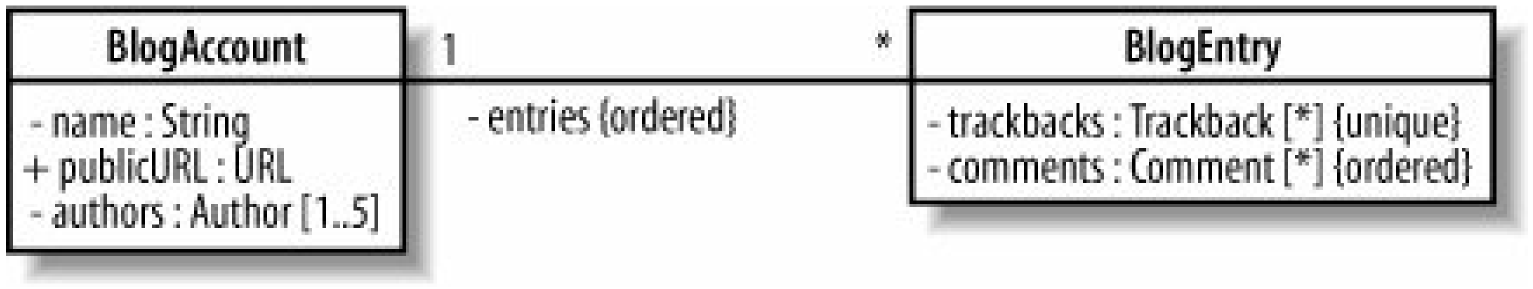
- Attributes (class states) are pieces of information that represent the state of an object. They can be represented by placing them inside the 2nd section of the class UML model or by association with another class.
- No two attributes in the same class can have the same name. The attribute type depends on how the class will be implemented.



Syntax: <visibility > <attribute_name> : <attribute_type>

Class diagram

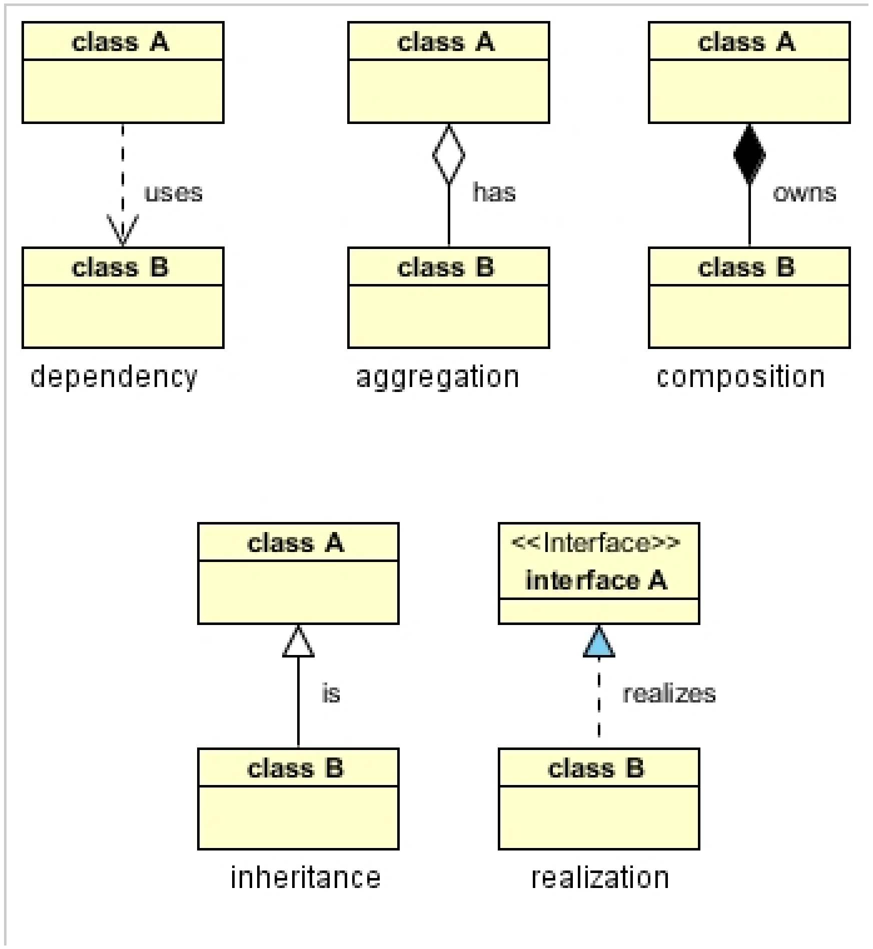
- Attribute can represents more than one object. Multiplicity used to specify that an attribute represents a collection of objects.


BlogAccount
- name : String
+ publicURL : URL
- authors : Author [1..5]
1
*
- entries {ordered}
BlogEntry
- trackbacks : Trackback [*] {unique}
- comments : Comment [*] {ordered}

- Operation describes what a class can do but not necessarily how.
  - Syntax: <visibility> <operation_name> () : <return_type>
  - Constructor doesn’t need to specify return type

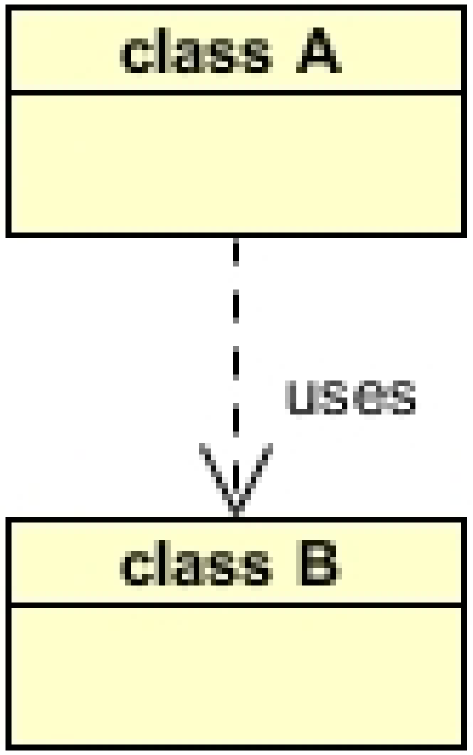
Class diagram

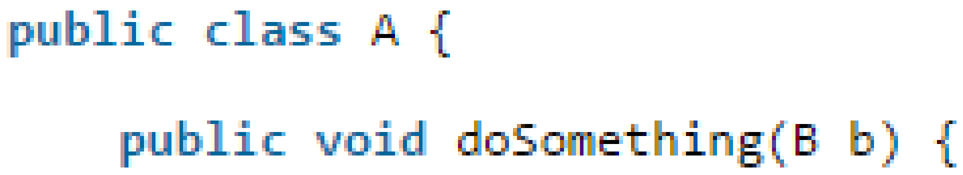
- There are five types of class relationship



Class diagram

- Dependency: is represented when a reference to one class is passed in as a method parameter to another class

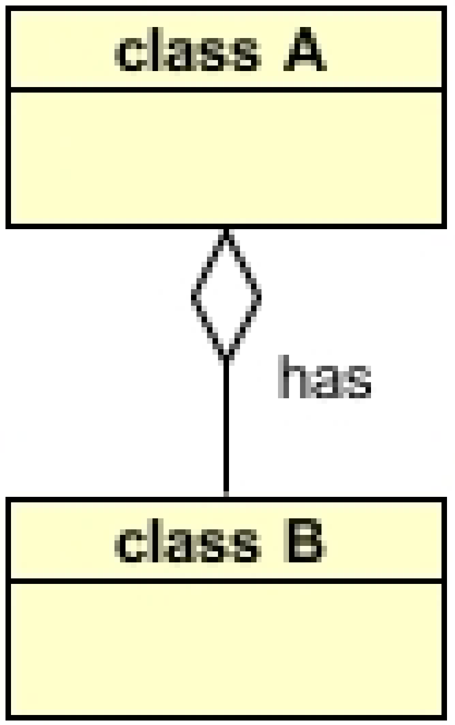




```
public class A {

    public void doSomething(B b) {
```

Class diagram

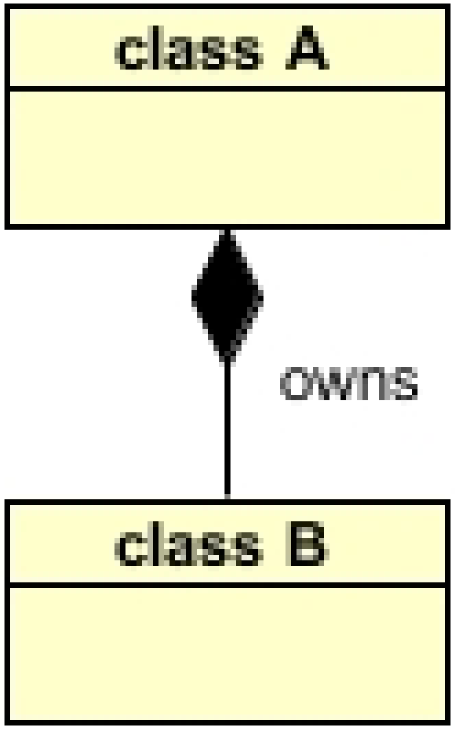
- **Aggregation**: if class A stored the reference to class B for later use we would have a different relationship



```
public class A {
    private B _b;
    public void setB(B b) { _b = b; }
```

Class diagram

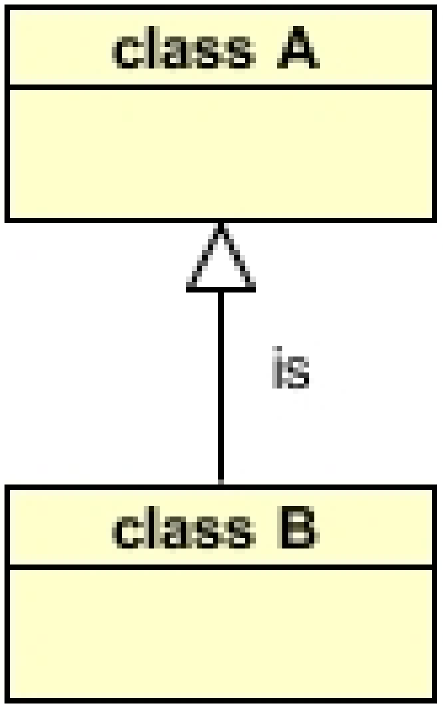
- Composition: Stronger than aggregation, the containing object is responsible for the creation and life cycle of the contained object



```
public class A {

    private B _b = new B();


public class A {

    private B _b;

    public A() {
        _b = new B();
    } // default constructor
```

Class diagram

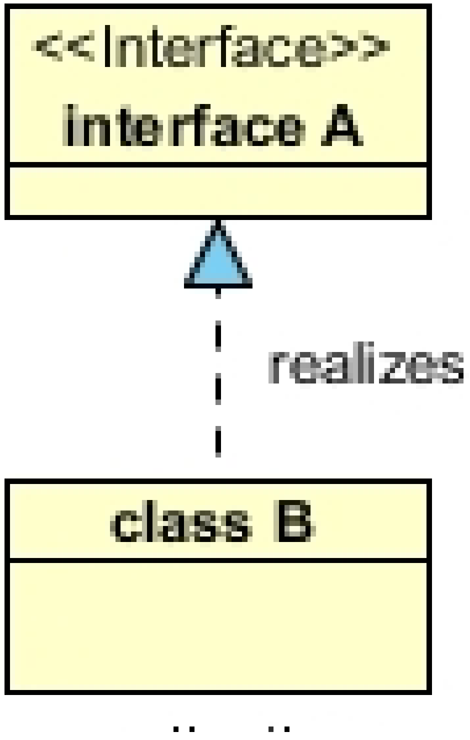
- Inheritance: used to describe a class that is a type of another class.
  - Has-a: association/aggregation/composition
  - Is-a: inheritance



```
public class A {

    ...

} // class A

public class B extends A {

    ....

} // class B
```

Class diagram

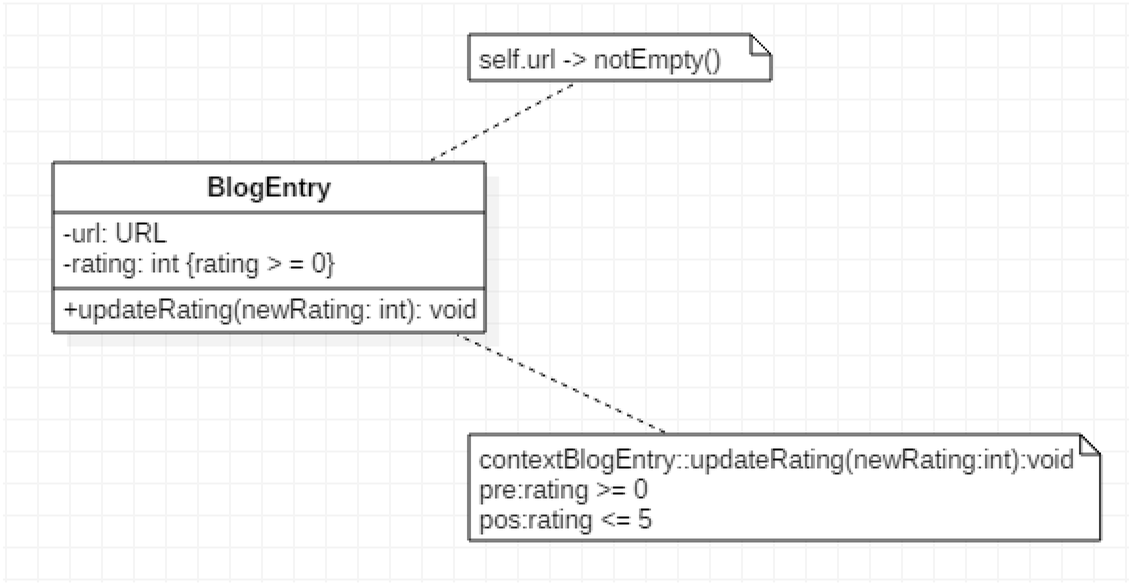
- **Realization** is also straightforward in Java and deals with implementing an interface



```
public interface A {

    ...

} // interface A

public class B implements A {

    ...

} // class B
```

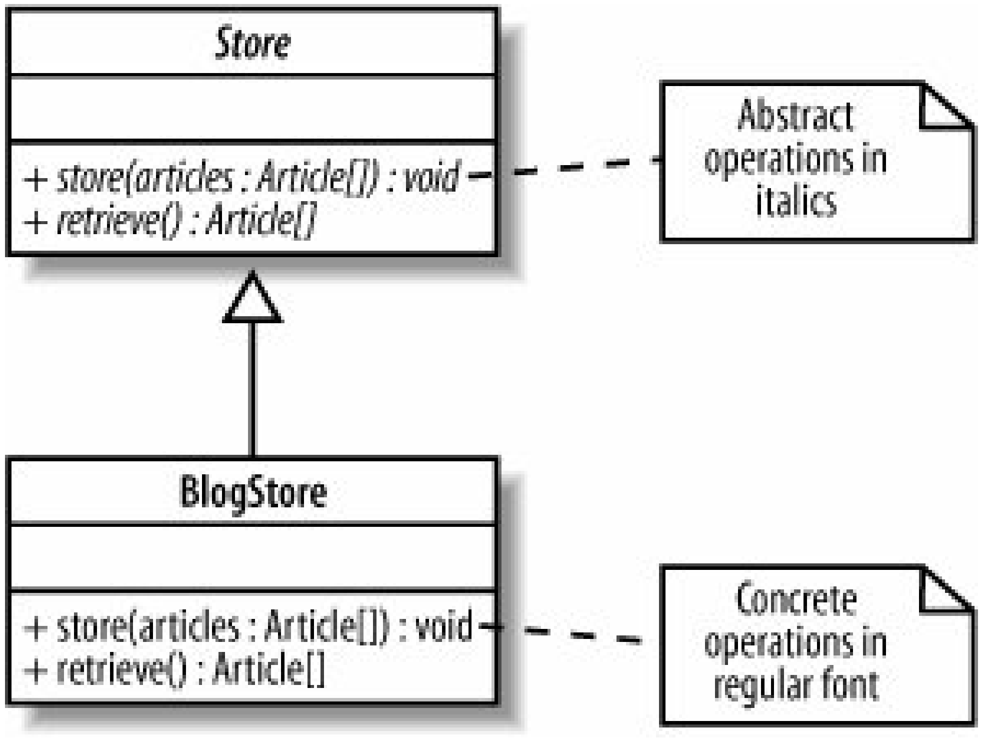
Class diagram

- There are 3 types of constraints to restrict the ways in which a class can operate
  - Invariants: is a constraint that must always be true, else the system is in an invalid state. Used on class attribute
  - Preconditions: is a defined constraint on a method and checked before method executes. Used to validate input parameters
  - Postconditions: is a defined constraint on a method and checked after method executes. Used to describe how values were changed by a method


self.url -> notEmpty()
BlogEntry
-url: URL
-rating: int {rating > = 0}
+updateRating(newRating: int): void
contextBlogEntry::updateRating(newRating:int):void
pre:rating >= 0
pos:rating <= 5

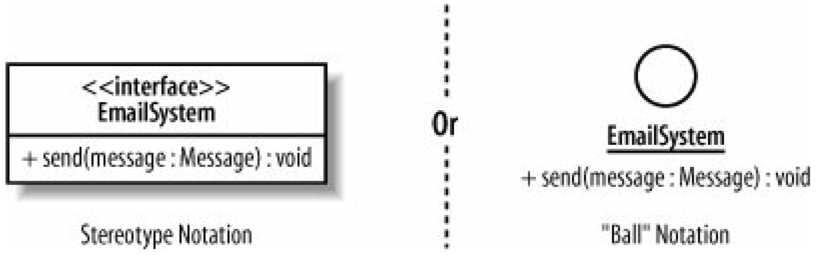
Class diagram

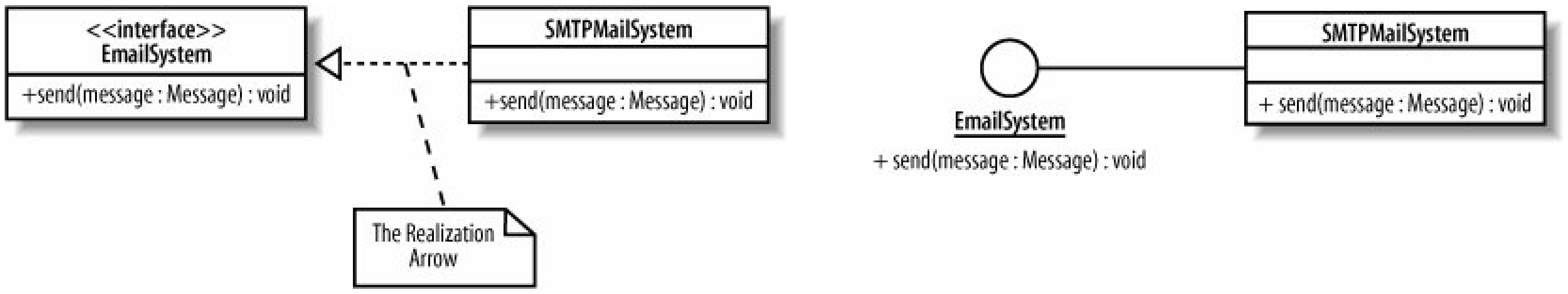
- Abstract class: use for inheritance and you won't be able to implement the behavior of the general class.


Store
+ store(articles : Article[]) : void
+ retrieve() : Article[]
Abstract operations in italics
BlogStore
+ store(articles : Article[]) : void
+ retrieve() : Article[]
Concrete operations in regular font

Class diagram

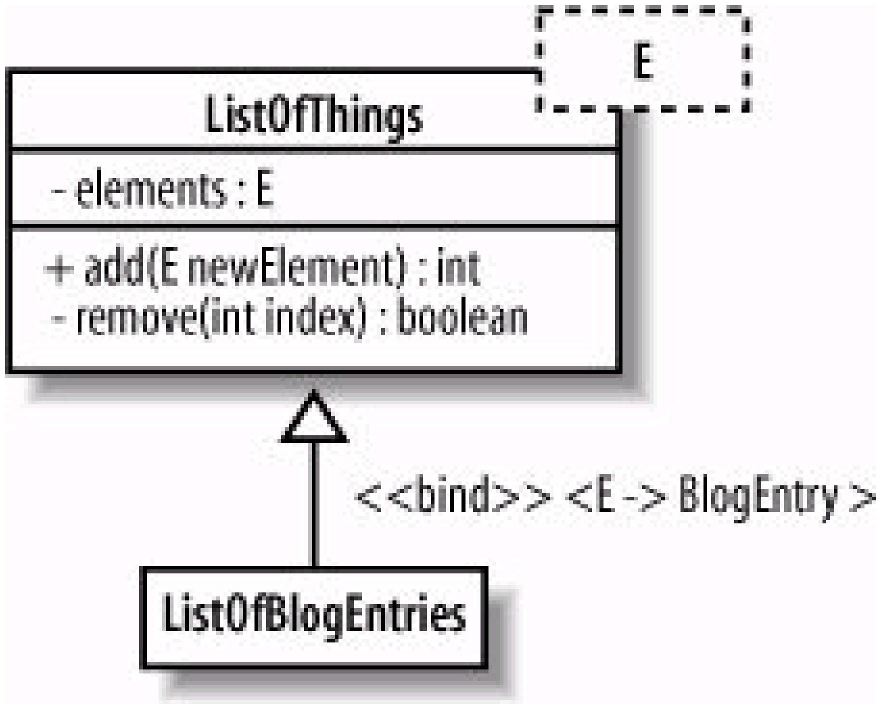
- An interface is a collection of operations but doesn't have an implementation of its. It's like an abstract class but with only abstract methods.




<<interface>>
EmailSystem
+send(message : Message) : void
SMTPMailSystem
+send(message : Message) : void
The Realization Arrow
EmailSystem
+ send(message : Message) : void
SMTPMailSystem
+ send(message : Message) : void

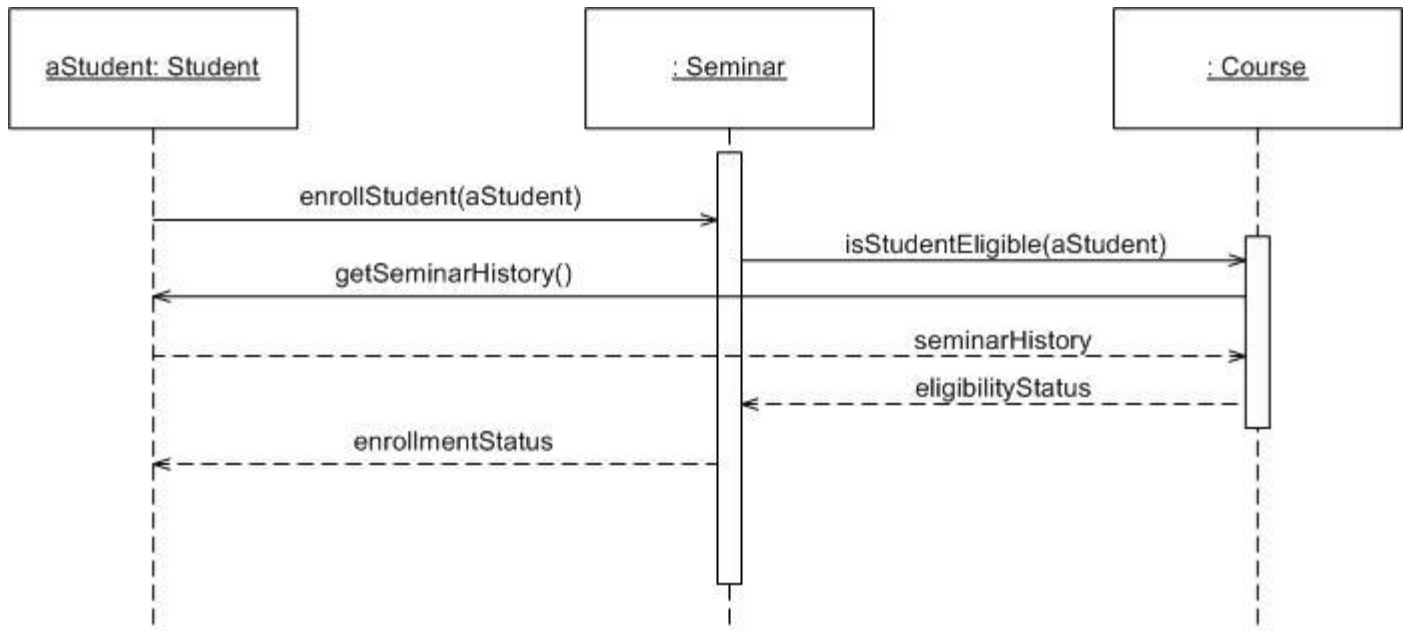
Class diagram

- Templates are parameterized class, as you want to postpone the decision as to which classes a class will work it. The binding process is done by subclassing

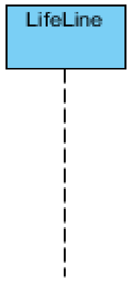

E
ListOfThings
- elements : E
+ add(E newElement) : int
- remove(int index) : boolean
<<bind>> <E -> BlogEntry >
ListOfBlogEntries

Sequence diagram

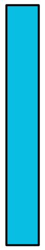
- A **Sequence diagram** is an interaction diagram that shows how processes operate with one another and in what order.
- Focus on identifying the behavior within the system being built.
- Associated with a use case.



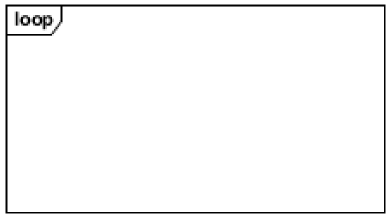
Sequence diagram



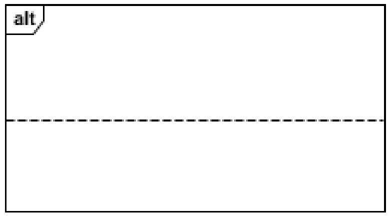
A **lifeline** represents an individual participant in a sequence diagram



Activation or **Execution Occurrence**

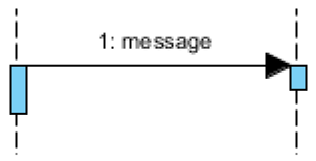


A **loop combined fragment** represents a loop. The loop operand will be repeated a number of times.

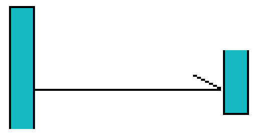


An **alternative combined fragment** represents a choice of behavior. At most one of the operands will be chosen.

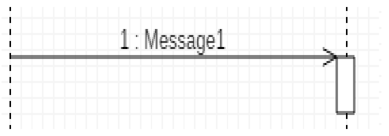
Sequence diagram



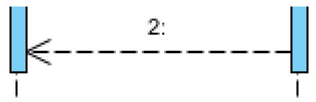
A **message** defines a particular communication between Lifelines of an Interaction. In most case, it denotes a method call.



An **asynchronous message** is one where the sender doesn't wait for the result of the message, instead it processes the result when and if it ever comes back.

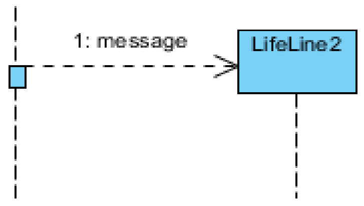


A **simple message** and also used for asynchronous.

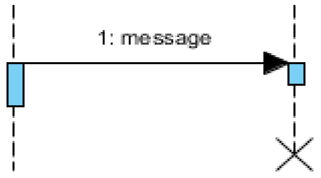


Return message is a kind of message that represents the pass of information back to the caller of a corresponded former message.

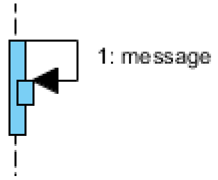
Sequence diagram



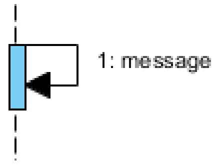
Create message is a kind of message that represents the instantiation of (target) lifeline.



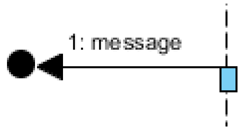
Destroy message is a kind of message that represents the request of destroying the lifecycle of target lifeline.



Recursive message is a kind of message that represents the invocation of message of the same lifeline.

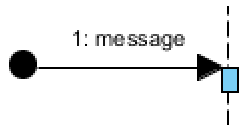


Self message is a kind of message that represents the invocation of message of the same lifeline.

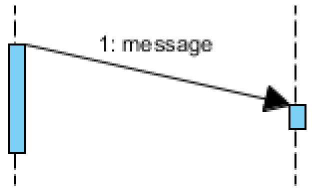


A **lost message** is a message where the sending event occurrence is known, but there is no receiving event occurrence.

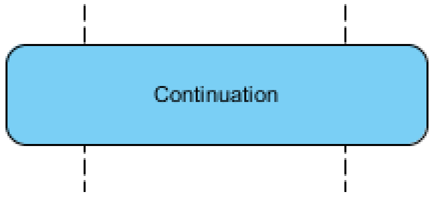
Sequence diagram



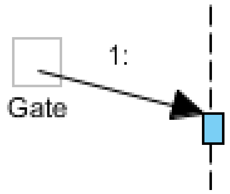
A **found message** is a message where the receiving event occurrence is known, but there is no (known) sending event occurrence.



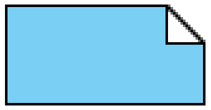
Duration message shows the distance between two time instants for a message invocation.



A **Continuation** is a syntactic way to define continuations of different branches of an Alternative CombinedFragment.

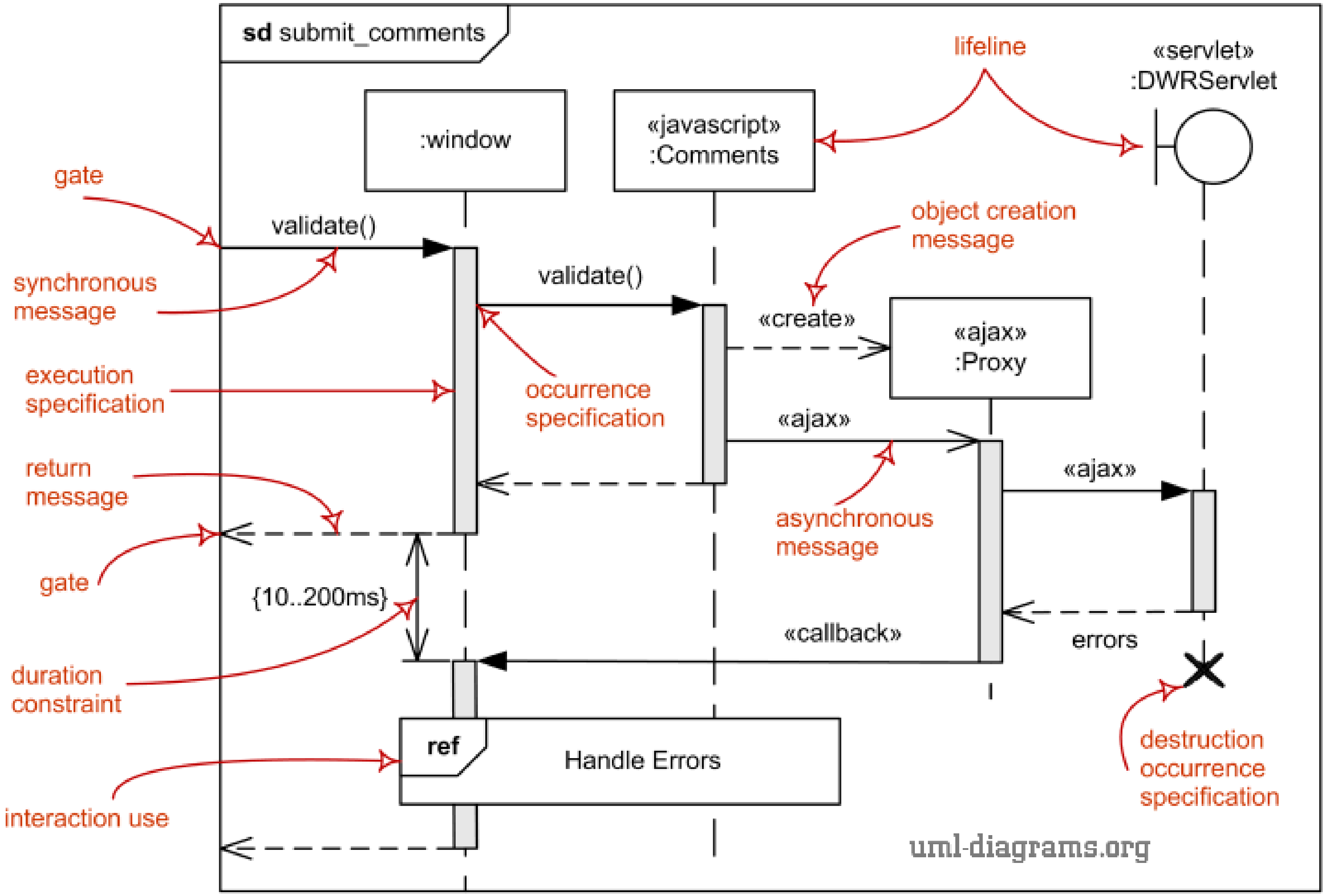


A **gate** is a connection point for connecting a message inside a fragment with a message outside a fragment.



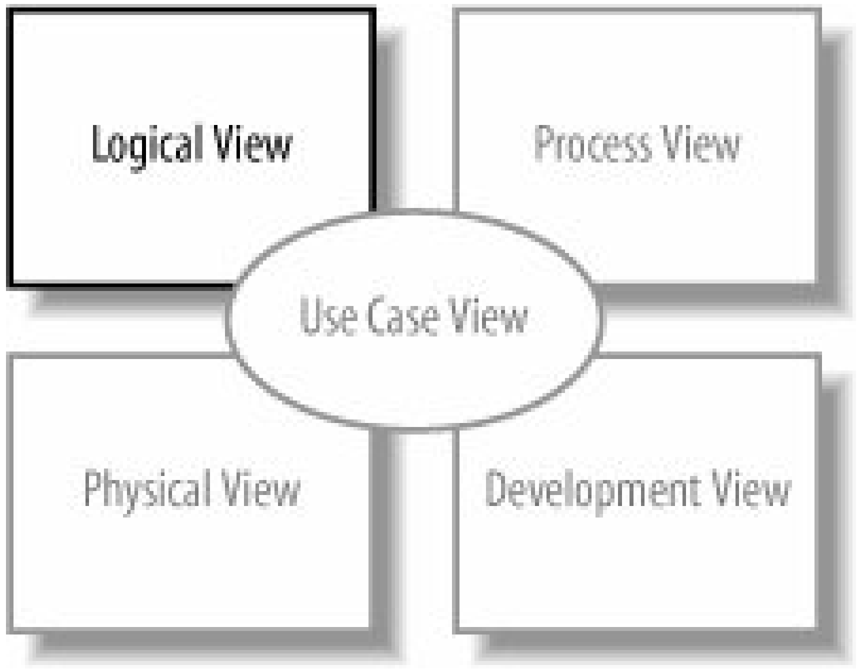
A **note** (comment) gives the ability to attach various remarks to elements

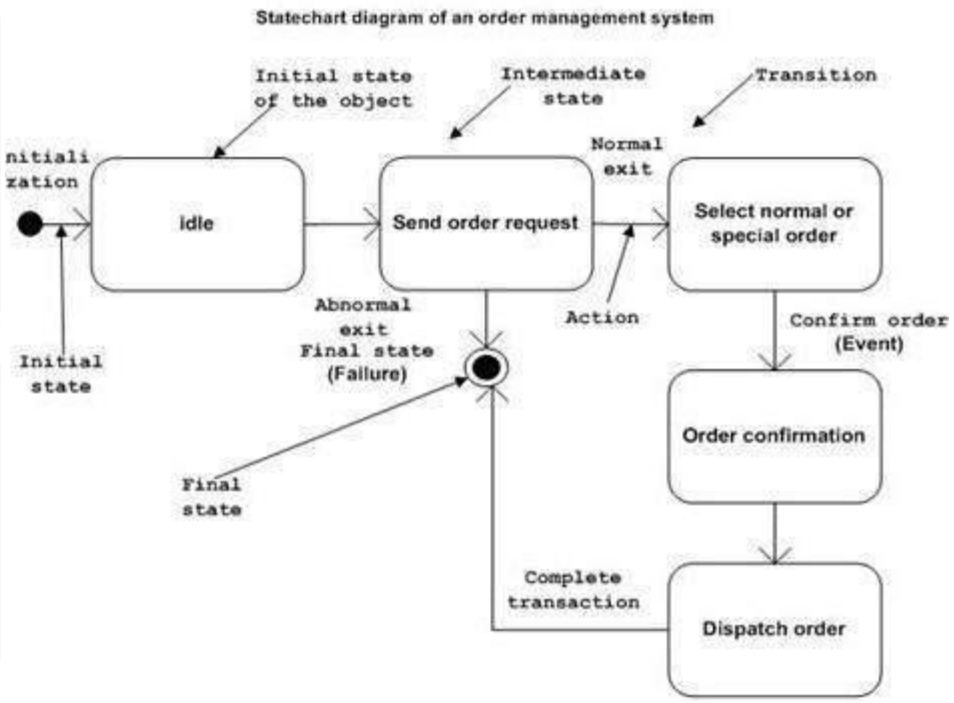
Sequence diagram



Statechart diagram

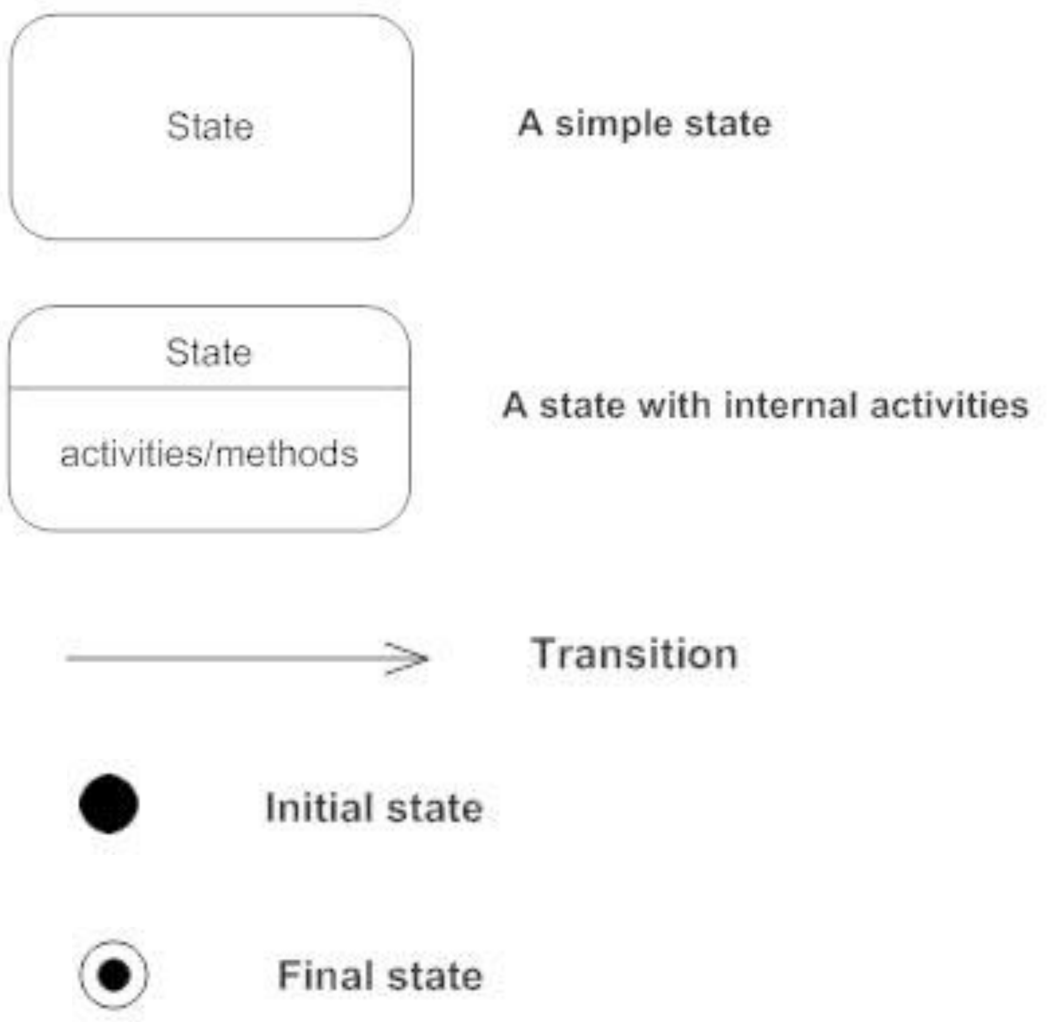
- Describes different states of a component in a system.
- The states are specific to a component/object of a system.
- Statechart diagram is one of the five UML diagrams used to model dynamic nature of a system.
- They define different states of an object during its lifetime.



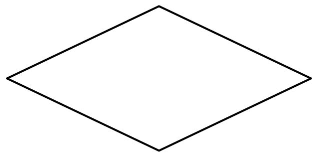


CSC Proprietary and Confidential

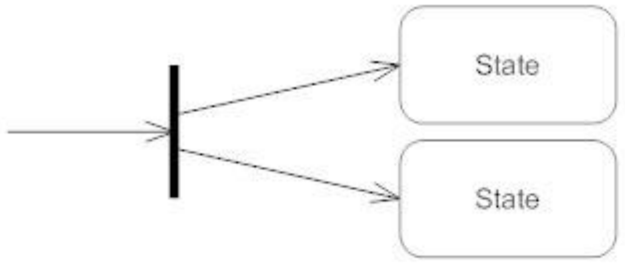
Statechart diagram - Notations



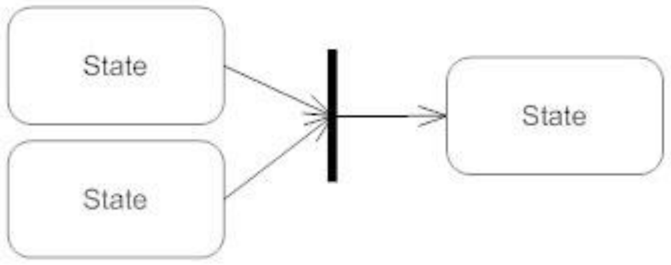
State machine diagram



Choice / Condition



Fork

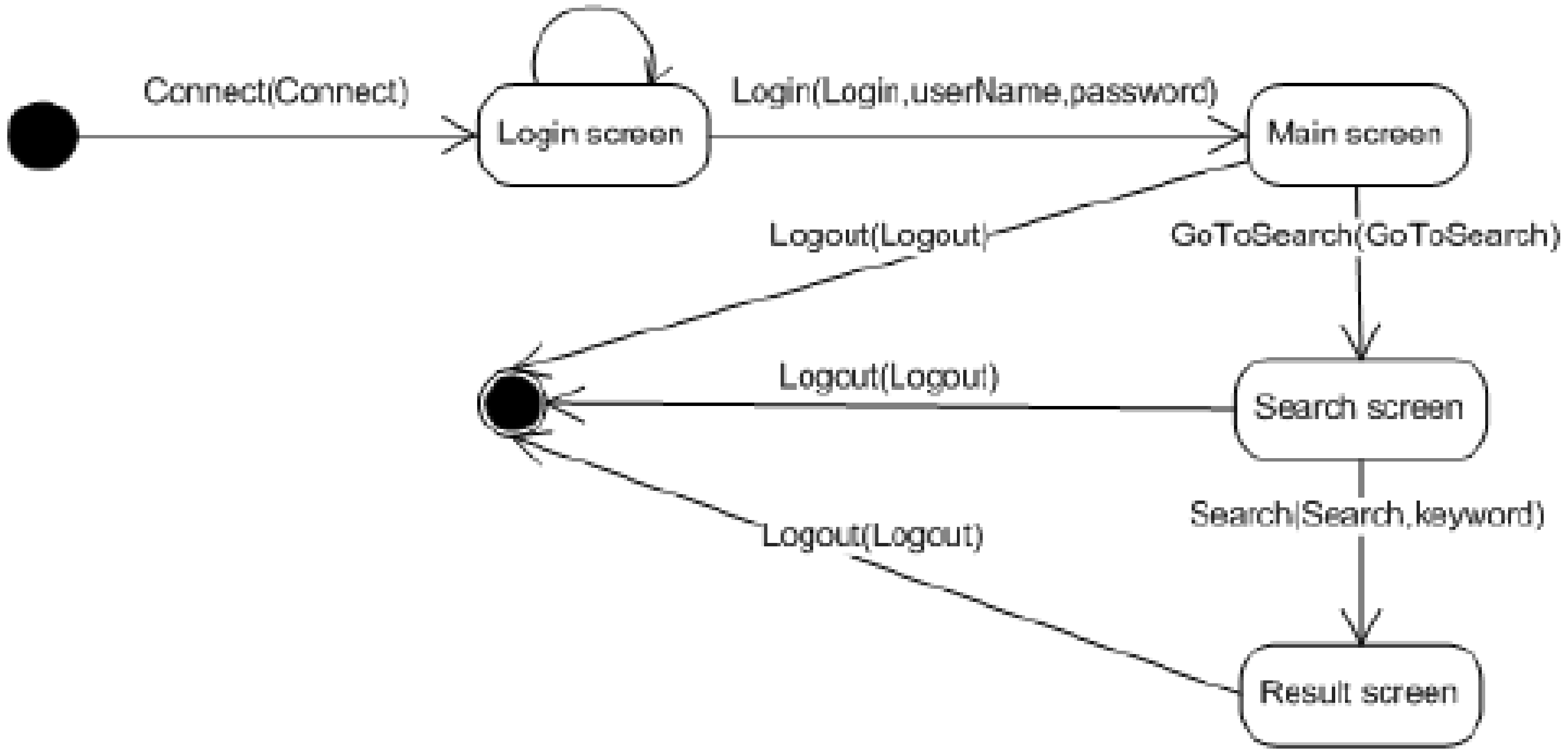


Join

[Guard Condition]

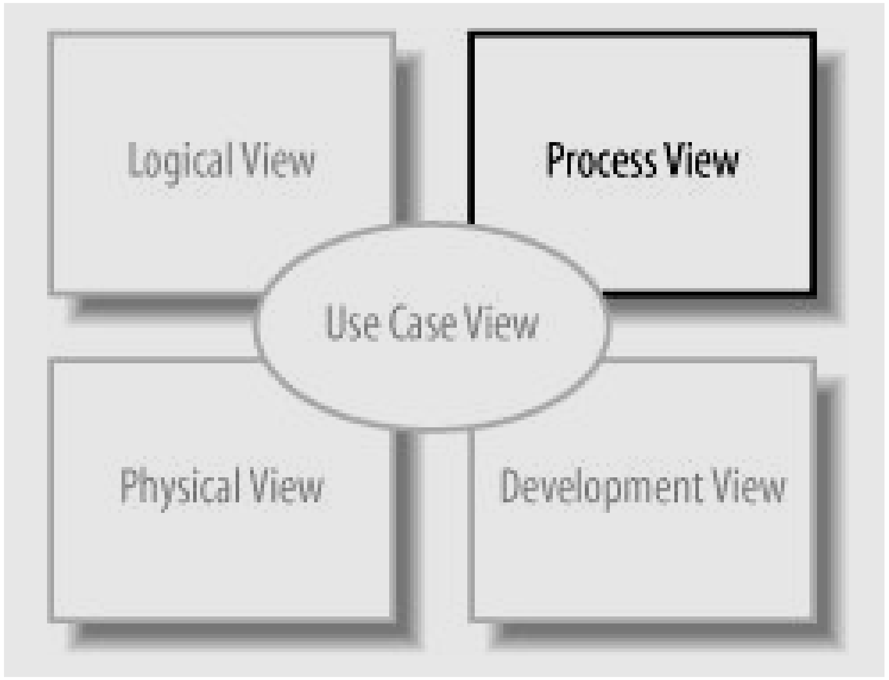
Guard conditions can be used to document that a certain event, depending on the condition, can lead to different transitions

Statechart diagram



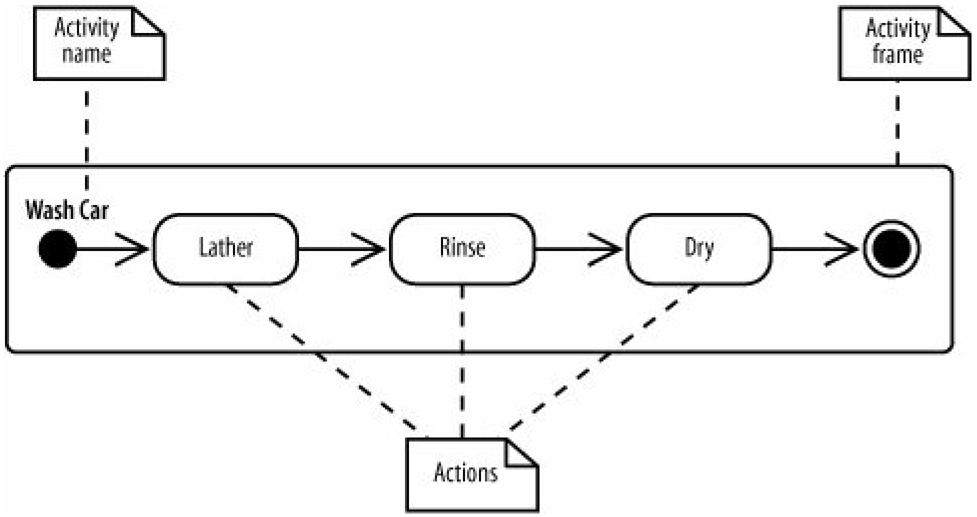
Activity diagram

- Activity diagram is basically a flow chart to represent the flow form one activity to another activity.
- The activity can be described as an operation of the system.
- It captures the dynamic behavior of the system.
- Activity diagram consists of:
  - ✓ Activities
  - ✓ Association
  - ✓ Conditions
  - ✓ Constraints



Activity diagram

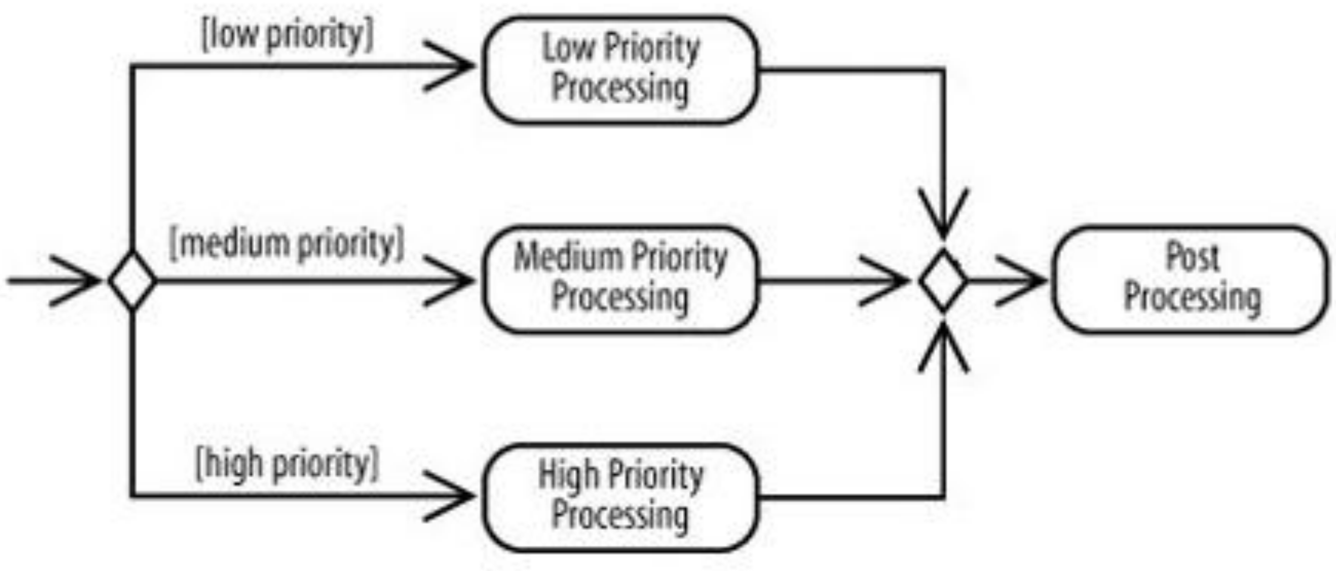
- Activities and actions
  - Actions are steps in the completion of a process.



  - The Activity Wash Car is comprise of 3 actions: lather, rinse, dry

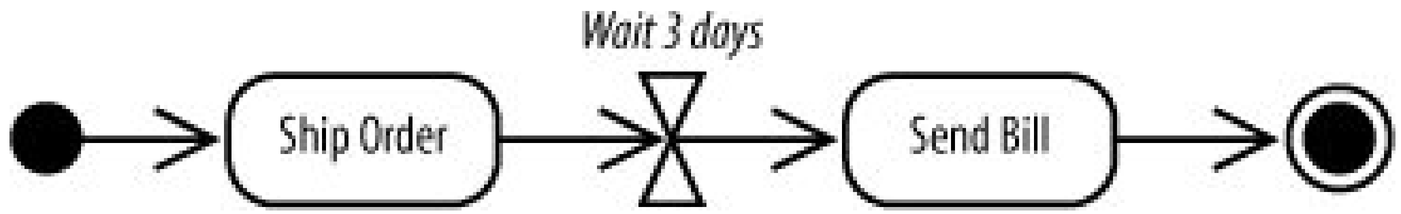
Activity diagram

- Decisions and Merges
  - Decisions are used when you want to execute a different sequence of actions depending on a condition. Decisions are drawn as diamond-shaped nodes with one incoming edge and multiple outgoing edges



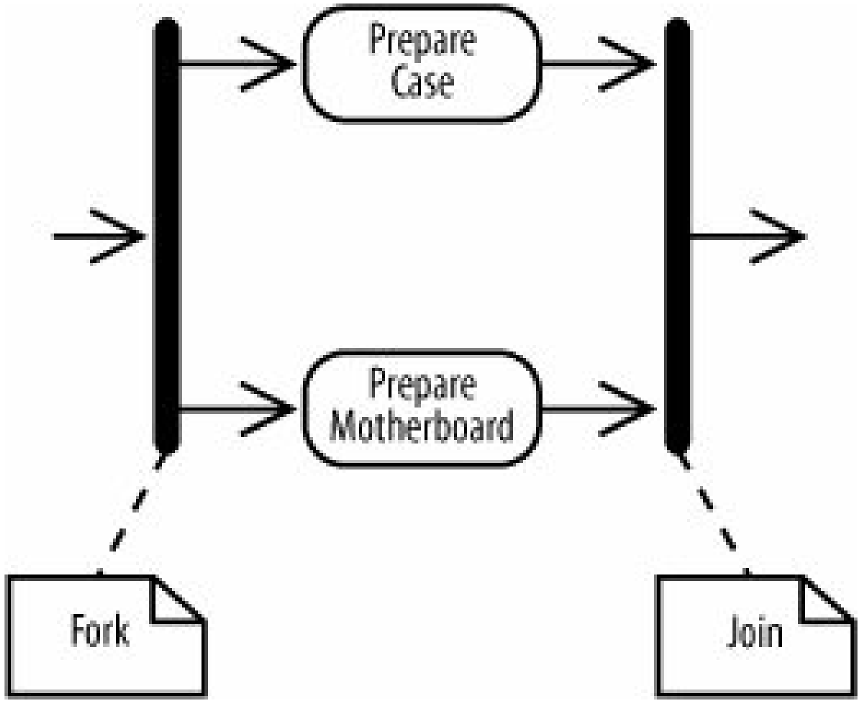
Activity diagram

- Time Events
  - Time events are drawn with an hourglass symbol.



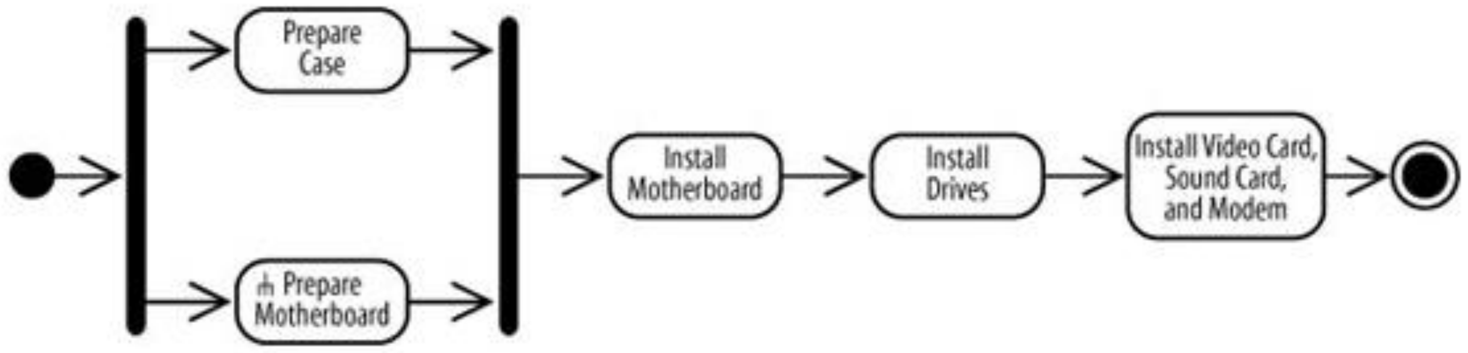
Folks and joins

- Used to split and join actions that can be processed in parallel at the same time

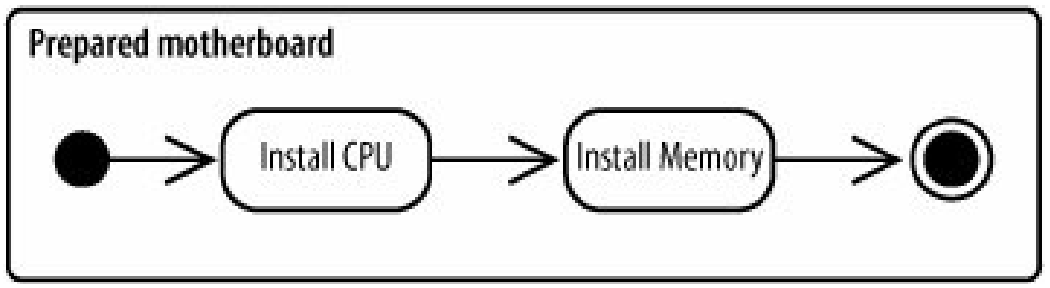


Activity diagram

- Calling other activities:
  - You can simplify your diagram by making your activity diagram a call activity node which has an upside down pitch fork symbol

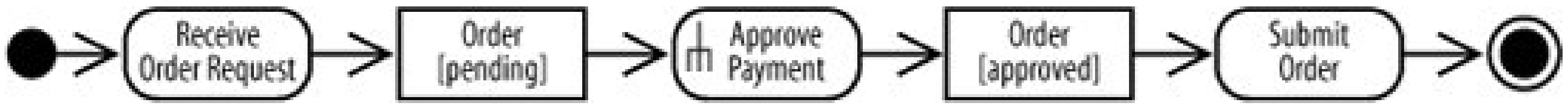


  - And the gives the detail of the call activity node as below

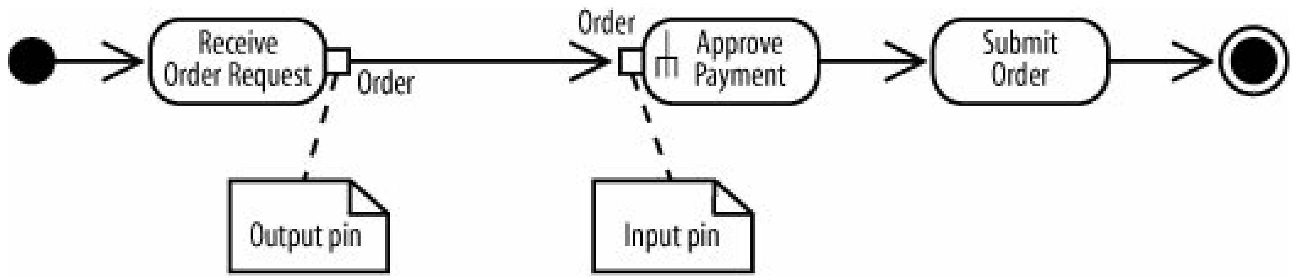


Activity diagram

- Passing object:
  - An object is drawn as a rectangle, can include its status, to present that it's available at a particular point in the activity

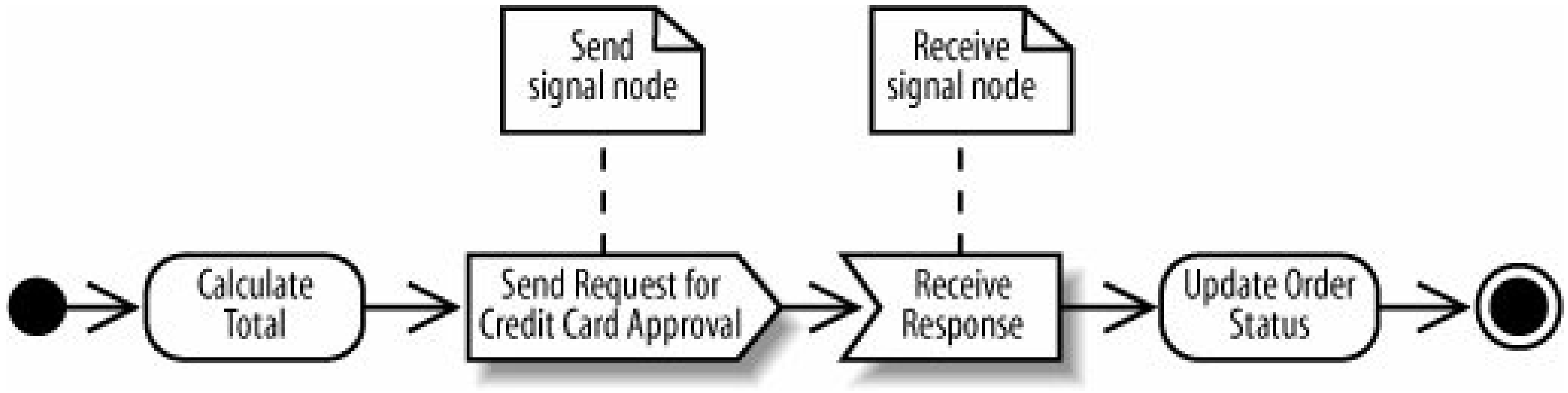


  - An object is drawn as a rectangle, can include its status, to present that it's available at a particular point in the activity
  - You can show input and output as below:



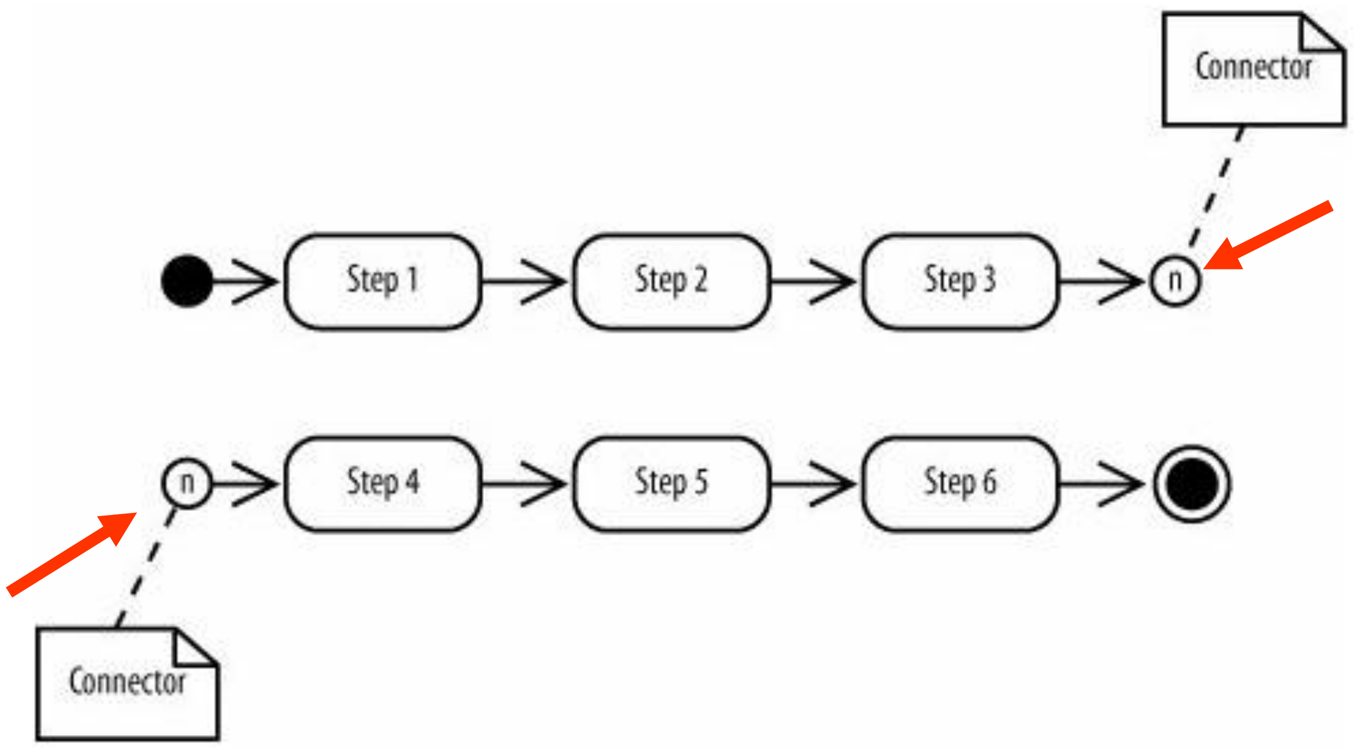
Activity diagram

- Sending and Receiving Signals
  - Activities may interact with external people, systems or processes. Can be demonstrated as below



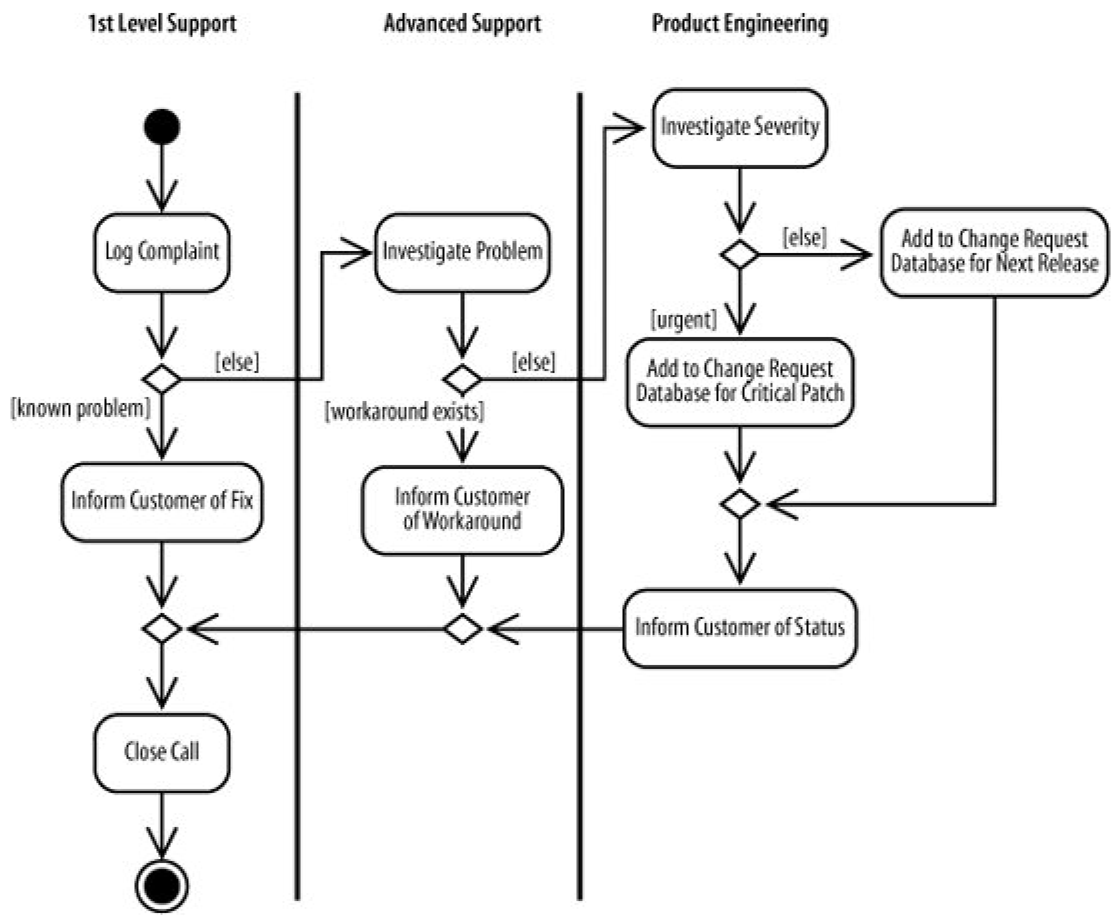
Connectors

- Used to manage the complexity of the activity diagram (by fragmentation)



Activity diagram

- Partitions (Swimlanes)
  - Activities may involve with different participants, thus the diagram requires multiple participants to be completed.



References