# What Is UML Diagram?

UML (Unified Modeling Language) diagram is a way of **visualizing a software program using a collection of diagrams**.

## Why Do We Need UML Diagrams?

Suppose you have to build a system. Before implementing a bunch of classes and functions, you’ll want to **have a conceptual understanding of the system** – that is, what classes/functions do you need? What functionality and information will these classes/functions have? How do they interact with one another? Who can see these classes/functions? And so on.

That’s where UML diagrams come in. They are a neat way of visualizing your system **before you actually start coding them up**.

Moreover, as your system **scales and grows**, it becomes increasingly difficult to keep track of the code. Having precise, organized, and straight-forward diagrams to do that for you is therefore very necessary.

## Types of UML Diagrams?

There are several types of UML diagrams and each one of them serves a different purpose. The two most broad categories that encompass all other types are Structural UML diagram and Behavioral UML diagram. The different types are broken down as follows:

### Structural UML Diagram

Structure diagrams show the **static structure** of the system, its parts on different **abstraction and implementation levels** and how they are related to each other.

* Class Diagram
* Object Diagram
* Component Diagram
* Composite Structure Diagram
* Deployment Diagram
* Package Diagram
* Profile Diagram

### Behavioral UML Diagram

Behavior diagrams show the **dynamic behavior** of the objects in a system, which can be described as a series of changes to the system over time.

* Use Case Diagram
* Sequence Diagram
* Activity Diagram
* Interaction Overview Diagram
* Timing Diagram
* State Machine Diagram
* Communication Diagram

Not all of these 14 different types of UML diagrams are used on a regular basis when documenting systems and/or architectures. The Pareto principle seems to apply in terms of UML diagram usage as well – 20% of the diagrams are being used 80% of the time by developers. The most frequently used ones in software development are: **Use Case diagrams**, **Class diagrams**, and **Sequence diagrams**. ﻿

# Class Diagram

Should watch first: https://www.youtube.com/watch?v=UI6lqHOVHic

## What Is A Class Diagram?

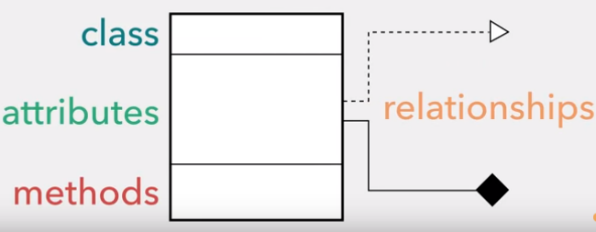
A class UML diagram shows overall structure of the system and subsystem as related **classes and interfaces**, with their features, constraints and relationships.

Because most software being created nowadays is based on the **OOP** paradigm (based on classes and the relations between them), using class diagrams to design and implement software turns out to be the most common-sense solution.

A class diagram contains **different classes and their relationships** (represented by a connecting line).

More specifically, each class has 3 fields:

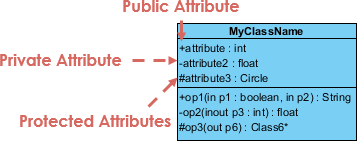
* **Class name** at the top
* **Class attributes** (data fields) right below the name
* **Class methods** (member functions or operations) at the bottom



## Main Components

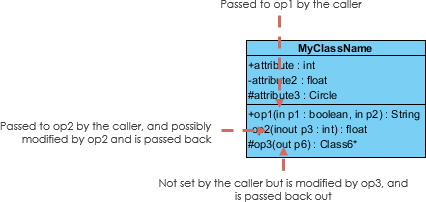
### Class Visibility

The **+**, **-** and **#** symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.



### Parameter Directionality

Each parameter in an operation (method) may be denoted as **in**, **out** or **inout** which specifies its direction with respect to the caller. This directionality is shown before the parameter name.

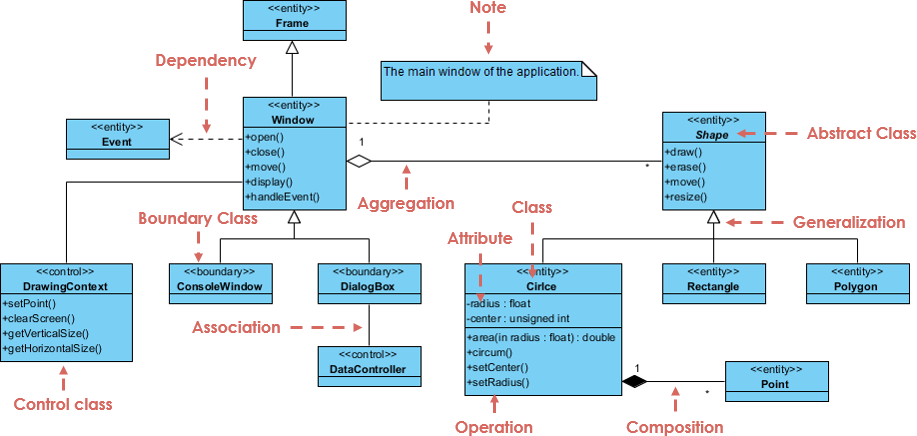


### Class Relationships

A class may be involved in one or more relationships with other classes. A relationship can be one of the following types:

|  |  |
| --- | --- |
| **Association**  (Bidirectional) | Both classes know about each other.  For example,  class Parent {  Child\* children;  // or void setChild(Child c)  }  class Child {  Parent\* parent;  }  You can go from the *Parent* to its *Child*, and vice-versa. In other words, the *Parent* knows about its *Child*, and the *Child* knows about its *Parent*.  So, Parent---------Child  Multiplicity: |
| **Association**  (Unidirectional) | Only one of the classes know about the other one.  Take the example above, but this time, there is no pointer to *Parent* in *Child*: class Child { }  In other word, you can go from *Parent* to *Child*, but you cannot go back from *Child* to *Parent*.  So, Parent---------->Child |
| **Inheritance**  or **Generalization** | A sub-class takes on the functionality of a super-class. This is a "is a" relationship.    For example, we have an *Animal* parent class with all public member fields. Because *Duck* “is a” *Animal*, itschild class can inherit all members from *Animal*. The same goes for *Fish* and *Zebra.*  They also implement their own unique member fields, including *swim(), run()* and *quack()* method. |
| **Realization**  or **Implementation** | One element implements/executes the behavior that another specifies. In other words, you can understand this as the relationship **between the interface and the implementing class**.  For example, the *Movable* interface specify *moveUp()*, *moveDown()*, *moveLeft()*, and *moveRight()* methods. The *MovablePoint* and *MovableCircle* classes implement these methods, possibly in very different ways. |
| **Dependency** | An object of one class might **use** an object of another class in the code of a method.  For example, *Person* class might have a *hasRead* method with a *Book* parameter that returns true if the person has read the book (perhaps by checking some database).  Also, *Person* class depends on *Book* parameter. In other word, a change to the definition of *Book* cause changes to *Person* (but not the other way around).  Dependency |
| **Aggregation** | This is a "has a" or "part of" relationship.  For example,   * *Class1* "has a" *Class2.* In other word, *Class2* is "part of" *Class1*. * When *Class1* is destroyed, *Class2* is NOT destroyed. |
| **Composition** | A special type of aggregation, where parts are destroyed when the whole is destroyed.  For example,   * When *Class1* is destroyed, *Class2* is destroyed too. |

Should read more: [Association vs Aggregation vs Composition](http://aviadezra.blogspot.com/2009/05/uml-association-aggregation-composition.html)



In the example above, we can interpret the meaning of the above class diagram by reading through the points as following.

* *Shape* is an abstract class. It is shown in ***Italics***.
* *Shape* is a superclass. *Circle*, *Rectangle* and *Polygon* are derived from *Shape*. In other words, a *Circle* is-a *Shape*.
* There is an association between *DialogBox* and *DataController*.
* *Shape* is part-of *Window*. *Shape* can exist without Window.
* *Point* is part-of *Circle*. *Point* cannot exist without a *Circle*.
* *Window* is dependent on *Event*. However, *Event* is not dependent on Window.
* The attributes and method names of *Rectangle* are hidden. Some other classes in the diagram also have their attributes and method names hidden.

## How to Draw

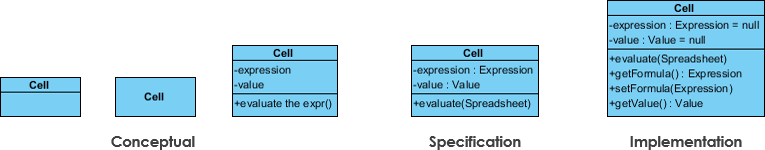
### Perspectives of Class Diagram

The perspective **affects the amount of detail** to be supplied and the kinds of relationships worth presenting.

The choice of perspective depends on how far along you are in the development process. During the formulation of a domain model, for example, you would seldom move past the **conceptual perspective**. Analysis models will typically feature a mix of conceptual and **specification perspective**. Design model development will typically start with heavy emphasis on the specification perspective and evolve into the **implementation perspective**.

A diagram can be interpreted from various perspectives:

* Conceptual: represents the concepts in the domain
* Specification: focus is on the interfaces of Abstract Data Type (ADTs) in the software
* Implementation: describes how classes will implement their interfaces



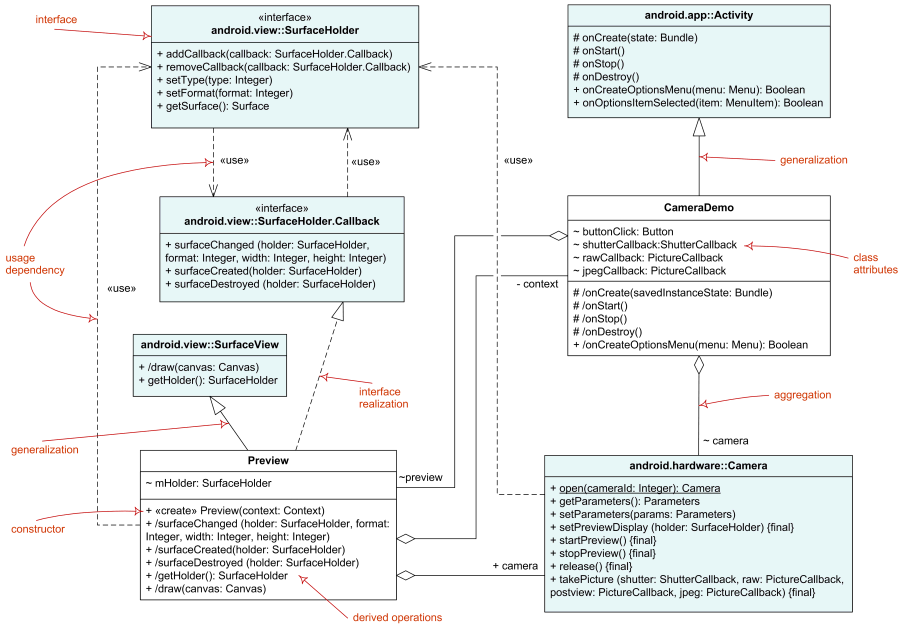
### Multiple or Single Class Diagram?

Inevitably, if you are modeling a large system or a large business area, there will be numerous entities you must consider. Should we use multiple or a single class diagram for modeling the problem? The answer is:

* Instead of modeling every entity and its relationships on a single class diagram, it is better to use multiple class diagrams.
* Dividing a system into multiple class diagrams makes the system easier to understand, especially if each diagram is a graphical representation of a specific part of the system.

### Tips

* For each class, **minimum number of properties should be specified**, as unnecessary properties will make the diagram complicated.
* **Use notes whenever required** to describe some aspect of the diagram. At the end of the drawing it should be understandable to the developer. ﻿



# Use Case Diagram

Should watch first: <https://www.youtube.com/watch?v=zid-MVo7M-E>

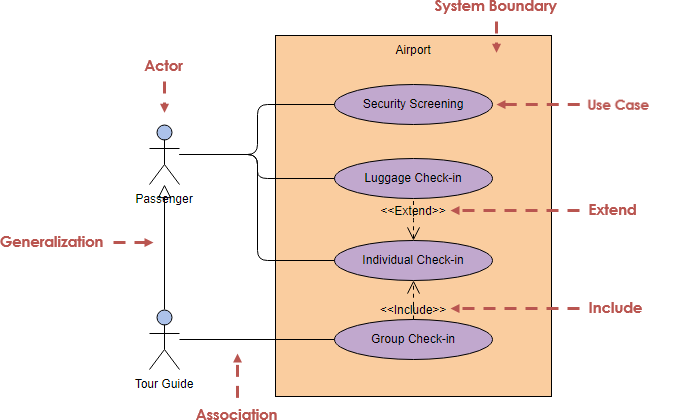
## What Is Use Case Diagram?

A use case diagram describes **how a user uses a system** to accomplish a particular goal. It consists of the **system** (what is being described), **actors** (who is using the system), **use cases** (what do the actors want to achieve), **relationships** (how the actors and the use cases, or different use cases, relate to each other). Thus, it provides a higher-level view of the system, ensuring that the correct system is developed by capturing the requirements from the user's point of view.

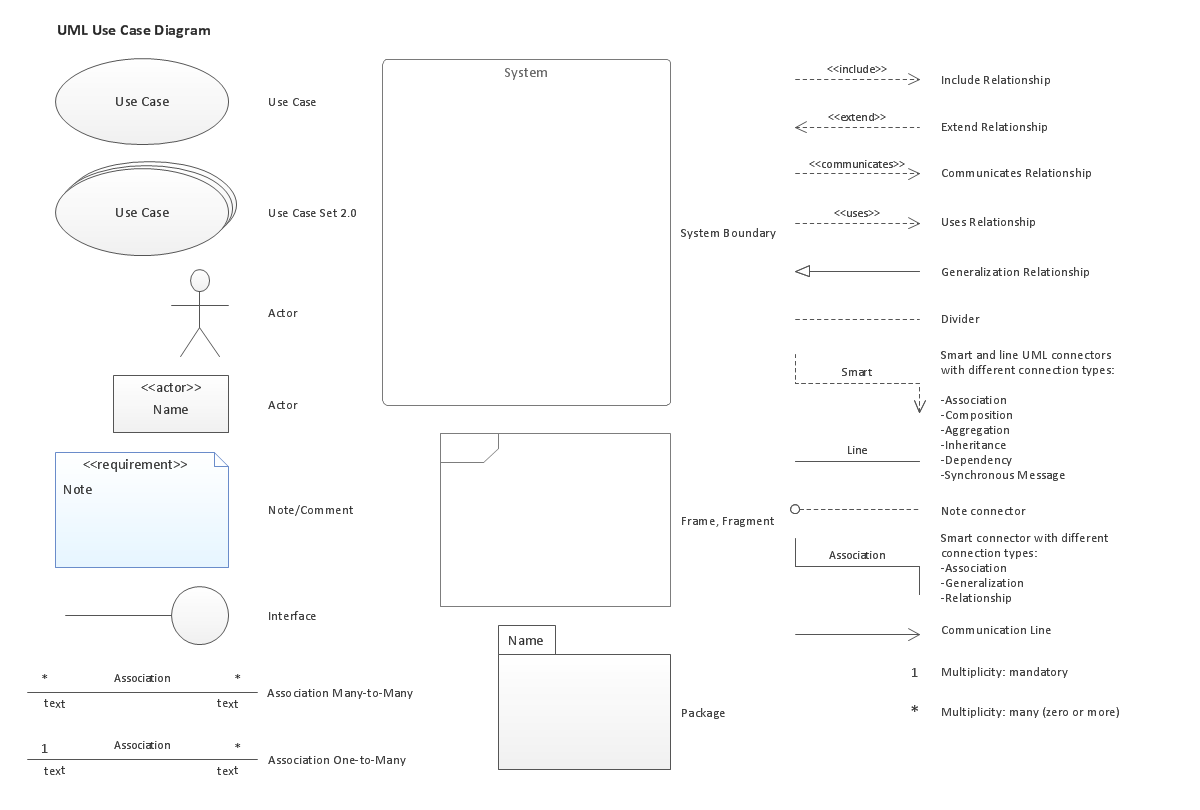
Use case vs test case

Hai khái niệm này rất dễ bị nhầm lẫn vì cả hai đều là những actions/events đứng trên góc độ người dùng. Use case mang tính tổng quát hơn nhiều so với test case; một use case có thể chứa nhiều test case vì lý do đó. Nói cách khác, nếu use case mô tả một general action, thì mỗi test case mô tả một specific action cho general action đó.

## Example



## Main Components

**System Boundary**

The system boundary defines the system in relation to the world around it.

**Actor**

An actor can be a **human** or **another** **external system** that is involved with the system (but outside the system) and defined according to their roles.

There are two types of actors:

* Primary actor: Initializes the use of the system
* Secondary actor: Acts once the primary actor asks for something

**Note**: There can be many primary actors and secondary actors

**Use Case**

A use case describes how the actors (both primary and secondary) use a system to accomplish a particular goal. It is an **action** which is initiated bythe actors.

**Relationship**

The relationships among the actors and the use cases. There are four main relationships in a use case diagram:

* **Association**
* **Include**: Whenever the base use case is executed, the included use case is executed as well. In other words, the base use case requires the included use case to be able to work.
* **Extend**: When the base use case is executed, the extended use case is executed sometimes, but not every time.
* **Generalization (or inheritance)**: parent-child relationship between use cases.

**Package**

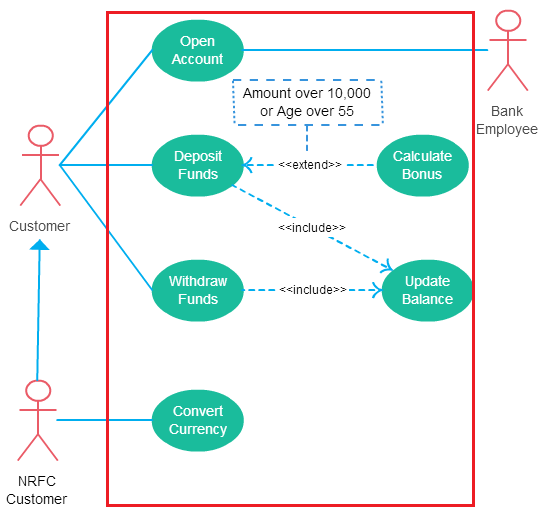
The package is an optional element. It is used to **group together use cases**, making it very useful in complex diagrams.

## How to Draw

**Steps**

A use case diagram SHOULD be developed by following the steps below.

* Define all actors and their roles
* Define all use cases
* Make relationships among actors and use cases
* Make relationships among use cases



**Tips**

* Always structure and organize the use case diagram **from the perspective of actors**.
* Use cases should start off **simple and at the highest view possible**. Only then can they be refined and detailed further.
* Use case diagrams are based upon functionality and thus should focus on the "what" and not the "how".

Use Case Diagram Guidelines: <https://creately.com/blog/diagrams/use-case-diagram-guidelines/>

# Sequence Diagram

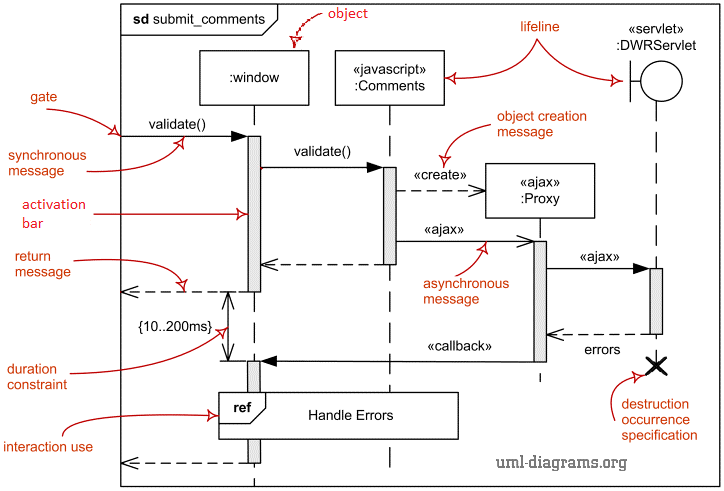
Should watch first: <https://www.youtube.com/watch?v=pCK6prSq8aw&t=>

## What Is A Sequence Diagram?

A sequence diagram describes **interactions among objects in a single use case** in terms of an exchange of **events and messages** over time. It helps predict how a system will behave, how different parts of a system interact with each other to carry out a function, in which order the interactions occur.

Compared to use case diagrams or class diagrams, sequence diagrams give us a **much more detailed** view of the system.

## Example



## Main Components

**Lifeline**

|  |  |
| --- | --- |
| Sequence diagram - Lifeline | A sequence diagram is made up of several lifeline notations that are arranged horizontally across the top of the diagram. They represent different objects or parts that interact with each other in the system during the sequence. |
| lifeline with an actor element symbol | A lifeline notation with an actor element symbol is used when the particular sequence diagram is **owned by a use case**. |
| Entity Lifeline | A lifeline with an entity element represents **system data**. For an example, in a customer service application, the ‘Customer’ entity would manage all data related to a customer. |
| Boundary Lifeline | A lifeline with a boundary element indicates a system boundary/ software element in a system. For example, **GUI elements** such as user interface screens, database gateways or menus which users interact with, are boundaries. |
| Control Lifeline | A lifeline with a control element indicates a **controlling entity or manager**. It organizes and schedules the interactions between the boundaries and entities and serves as the mediator between them. |

**Activation Bars**

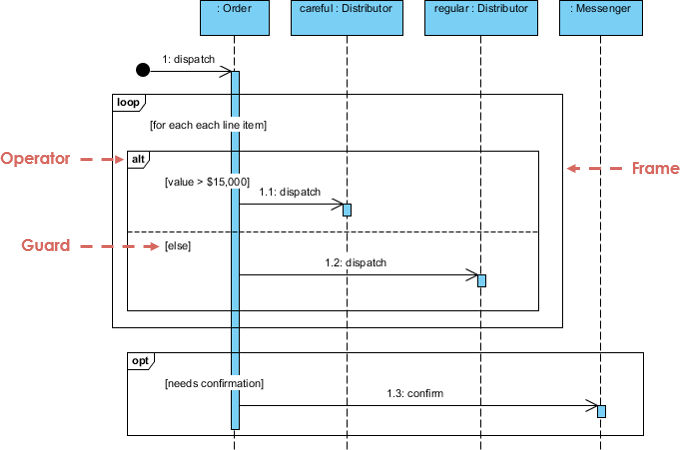
Activation bar (also called execution occurrence) is placed on the lifeline to indicate that an object is active (or instantiated) during an interaction (sending/receiving messages) between two objects. Its length indicates the duration of the objects staying active.

**Messages**

Messages are arrows that represent communication between objects. Some types of messages in sequence diagrams are:

|  |  |
| --- | --- |
| **Synchronous message**  Synchronous Message Arrow | It’s sent from a sender that will WAIT for a receiver to process the message and return before continuing with another message. |
| **Asynchronous message** | It’s sent from a sender that will NOT WAIT for a receiver to process the message and return before continuing with another message. |
| **Reply or return message** | It’s used to indicate that the receiver is done processing the message and is returning control over to the sender.  Tip: You can avoid cluttering up your diagrams by minimizing the use of return messages since the return value can be specified in the initial message arrow itself. |
| **Object creation message**  *Image below* | Objects do not necessarily live for the entire duration of the sequence of events, but some are created according to the message being sent.  A create message is used to create a new object (notated by a dropped object box).  If the created object does something immediately after its creation, you should add an activation bar right below the object box. |
| **Object destruction message**  Participation Destruction Message | Likewise, object when no longer needed can be deleted from a sequence diagram. This is done by adding an ‘X’ at the end of its lifeline. |
| **Reflexive or self message**  Reflexive message | When an object sends a message to itself, it is called a reflexive or self message. |

**Combined Fragments**



*Alternatives*

The ‘alt’ fragment is used when a choice needs to be made between two or more message sequences. It models the “*if … else*” logic.

To show two or more alternatives, the rectangle is divided into interaction **operands** using a dashed line. Each operand has a **guard** to test against and is placed at the top left corner of the operand.

*Options*

The ‘opt’ fragment is used to indicate a sequence that will only occur under a certain condition; otherwise, the sequence won’t occur. It models the “*if*” logic.  

Unlike the alternative fragment, the option fragment is not divided into two or more operands. Option’s guard is placed at the top left corner.

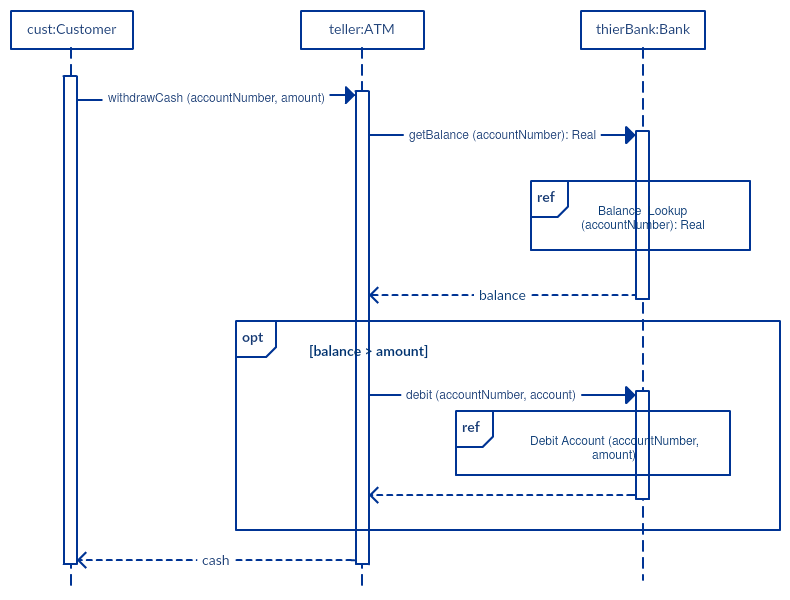
*Loops*

They are used to represent a repetitive sequence. It models the “*while*” or “*for*” logic. In addition to the Boolean test, the guard in a loop fragment can have two other special conditions tested against.

More combined fragments: <https://www.uml-diagrams.org/sequence-diagrams-reference.html>

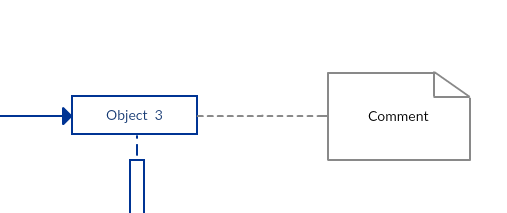
**Interaction Fragment**

The ‘ref’ fragment is used to manage the size of large sequence diagrams. It allows you to reuse part of one sequence diagram in another, or in other words, you can **reference part of a diagram in another diagram** using the ref fragment.



**Comment (Note)**

The comment is a rectangle with a folded-over corner. It can be linked to the related object with a dashed line.



## How To Draw

**Steps**

Determine:

1. A specific use case and its flow
2. How many objects and what they are
3. The interactions between objects

# Activity Diagram

Should watch first: <https://www.youtube.com/watch?v=Wf_xlagfHmg>

## What Is An Activity Diagram?

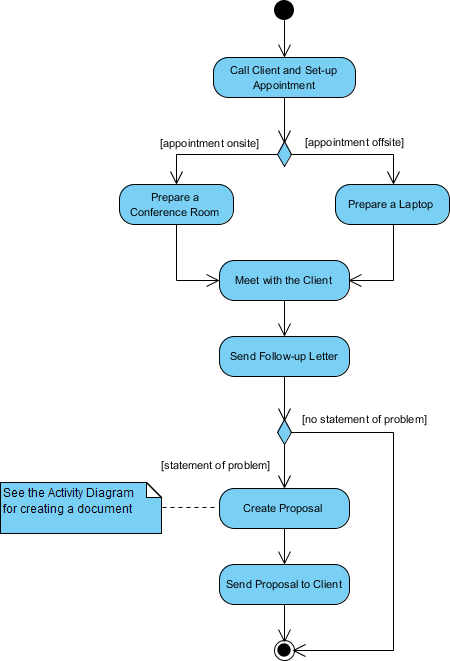
An activity diagram presents a series of actions or **flow of control** in a system, similar to a [data flow diagram](#_1fob9te). But it focuses on condition of flow and the sequence in which it happens. This flow can be sequential, branched, or concurrent.

Activity diagrams are often used in business process modeling. They can also describe the steps in a [use case diagram](#_gjdgxs).

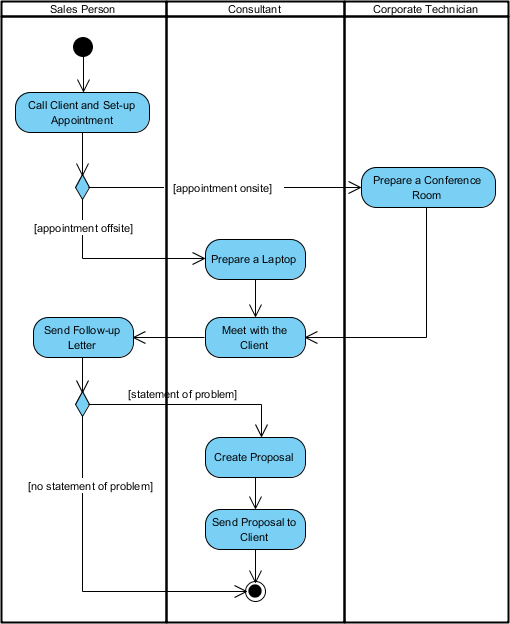
Other than [sequence diagrams](#_30j0zll) which are used to show the message flow from one object to another, activity diagrams are used to show message flow from one activity to another.

## Examples

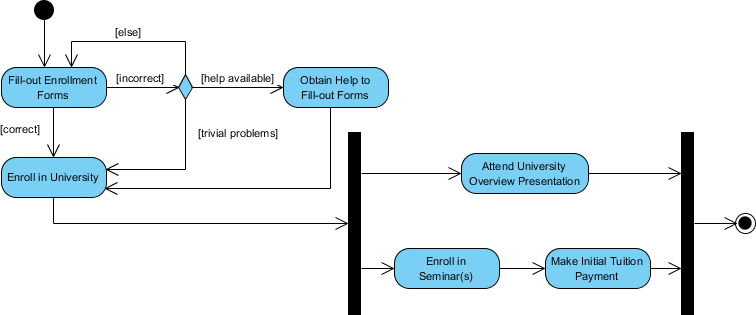
**Example 1**

****

**Example 2:** with swinlane

****

**Example 3**

****

## Main Components

|  |  |  |
| --- | --- | --- |
| **Components** | **Symbols** | **Descriptions** |
| Start Point |  | The initial state (or entry point) for any activity diagram. It’s usually placed in the top left corner of the first column.  A process can have only one initial state unless we are depicting nested activities. |
| End point |  | The state which the system reaches when a particular process or activity end. |
| Action  or Activity State |  | An activity represents execution of an action on objects or by objects. |
| Action flow  or Control flows |  | Also referred to as paths and edges. Used to show the transition from one activity state to another.  An activity state can have multiple incoming and outgoing action flows. |
| Decision and Branching |  | When we need to make a decision before deciding the flow of control, we use the decision node.  The outgoing arrows from the decision node can be labelled with conditions (or guard expressions). |
| Jork |  | Used to support concurrent activities. |
| Join |  | Used to support concurrent activities converging into one. |
| Merge Event |  | Used when activities which are not being executed concurrently have to be merged. |
| Time Event |  | Used when an event takes some time to complete. |
| Interrupting Edge |  | An event, such as a cancellation, that interrupts the flow denoted with a lightning bolt. |
| Swimlane |  | Used for grouping related activities in one column, performed by the same actor or in the same thread.  Swimlanes can be vertical and horizontal. It is not mandatory to use swimlanes. |

# Data Flow Diagram

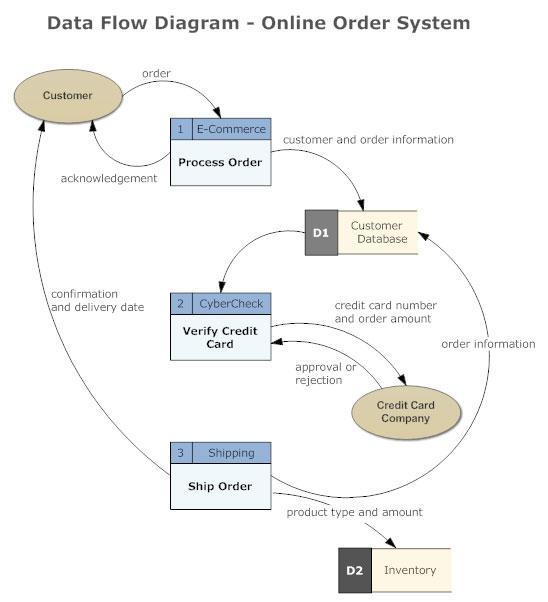
Should watch first: <https://www.youtube.com/watch?v=6VGTvgaJllM>

**Note**: Data flow diagram (DFD) is not UML-standarded. For which UML diagram lookings like a DFD, check [here](https://stackoverflow.com/questions/7716196/what-is-the-uml-analogue-to-the-data-flow-diagram-from-structured-analysis).

## What Is A Data Flow Diagram?

A data flow diagram (DFD) illustrates how data is processed by a system in terms of inputs and outputs. It focuses on the **flow of information (where data comes from, where it goes and how it gets stored)**.

## Examples



## Main Components

There are essentially two different types of symbols for a DFD (Yourdon & Coad or Gane & Sarson). Yourdon & Coad type is usually used for system analysis and design, while Gane & Sarson type is more common for visualizing information systems. The biggest difference between these two types is how processes look.

|  |  |  |  |
| --- | --- | --- | --- |
| **Component** | **Notation** | | **Description** |
| **Yourdon & Coad** | **Gane & Sarson** |
| Dataflow |  | | Arrow through packets of information flow, labeled with the name of the data moving through it. |
| Process |  |  | Transforms incoming dataflow into outgoing dataflow. Perform computations, or sort data, or direct the data flow based on business rules. |
| Datastore |  |  | Repositories of data in the system, sometimes referred to as files. |
| External Entity |  |  | Objects outside the system, with which the system communicates. In other words, sources and destinations of the system's inputs and outputs.  Might be an outside organization or person, a computer system or a business system. They are typically drawn on the edges of the diagram. |

## DFD Rules and Tips

* Each process should have at least one input and one output.
* Each data store should have at least one data flow in and one data flow out.
* Data stored in a system must go through a process.
* All processes in a DFD go to another process or a data store.

# State Machine Diagram

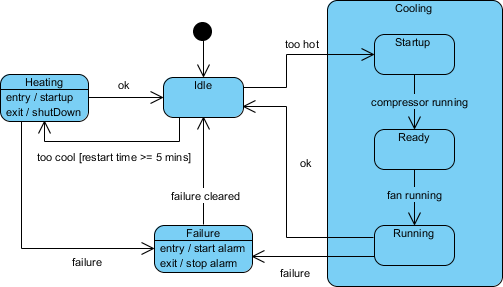
## What Is A State Machine Diagram?

Firstly, what is a state machine? It is any device which stores the status of an object at a given time and can change to a different status based on the input it receives. States refer to the different combinations of information that an object can hold, not how the object behaves.

So, a state machine diagram can help **visualize all of the possible states and how an object gets to each state**.

## Examples

Example 1: Heater



Example 2: Student enrollment



## Main Components

|  |  |  |
| --- | --- | --- |
| **Components** | **Symbols** | **Descriptions** |
| Initial state |  | A marker for the first state in the process. |
| Final state |  | The point at which the process is terminated. |
| Exit point |  | The point at which the process is not completed but has to be terminated due to an error or other issues. |
| State  &  Transition  &  Trigger |  | A state indicates the current status of an object in the life cycle.  A transition indicates running from one state to another.  A trigger (or event) is the cause of a transition. It’s denoted by a message written above the transition arrow. |
| Composite state |  | A state that has substates nested into it. |
| State with activities |  | A state that has activities. It has:  - Entry activity: action performed on entry to state  - Do activity: action performed while in state  - Exit activity: action performed on leaving state |
| Decision and Branching |  | When we need to make a decision before deciding whether to go to a certain state, we use the decision node.  The outgoing arrows from the decision node can be labelled with conditions (or guard expressions). |
| Fork |  | Used to support concurrent states. |
| Join |  | Used to support concurrent states converging into one. |

# Timing Diagram

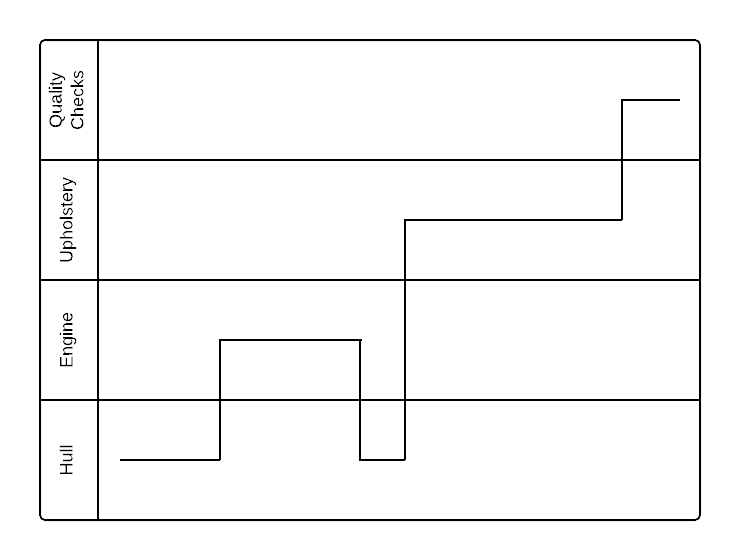
## What Is A State Machine Diagram?

In a timing diagram, time passes on the x-axis from left to right, with different components of the system interacting with each other on the y-axis. It shows **how long each step of a process takes** so that we can identify and improve which steps of a process which require too much time.

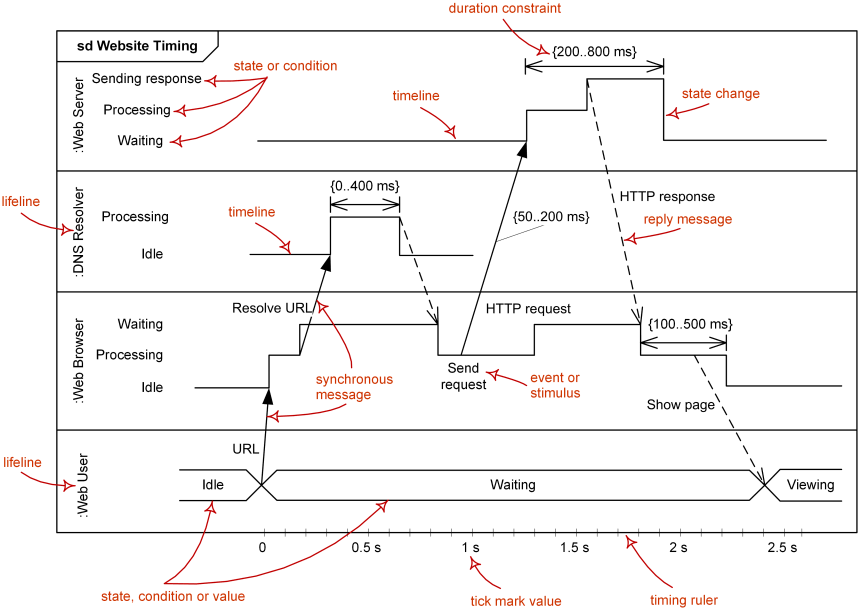
## Examples

**Example 1: Boat manufacturing plant**

In this example, the timing diagram shows that too much time is spent on the upholstery stages of production. As a result, factory administrators may assign more employees to the upholstery stations or seek out ways to increase efficiency.



## Main Components



[*Website latency*](https://www.uml-diagrams.org/website-latency-uml-timing-diagram-example.html)

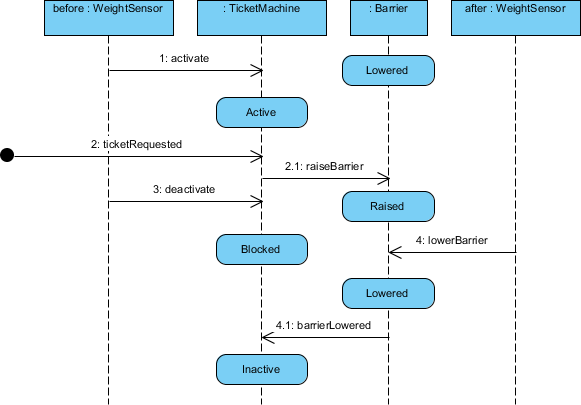
## How To Draw

**1. Model consistency among interaction diagrams**

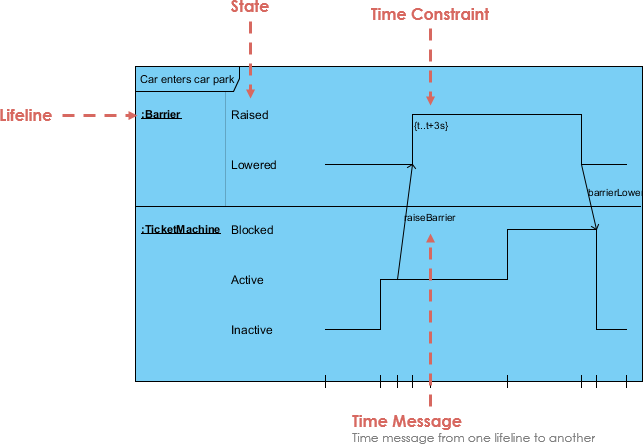
**A timing diagram should be consistent with a relevant sequence diagram and state machine diagram**. To do so, we can attach states in the lifeline for each of the objects in the sequence diagram. We can then derive the corresponding timing diagram much easier by inspecting the message passing between the objects against the states attached in the lifeline.

The Carpark example below shows the model consistency between two interaction diagrams.

First, the sequence diagram:



Second, the corresponding timing diagram:

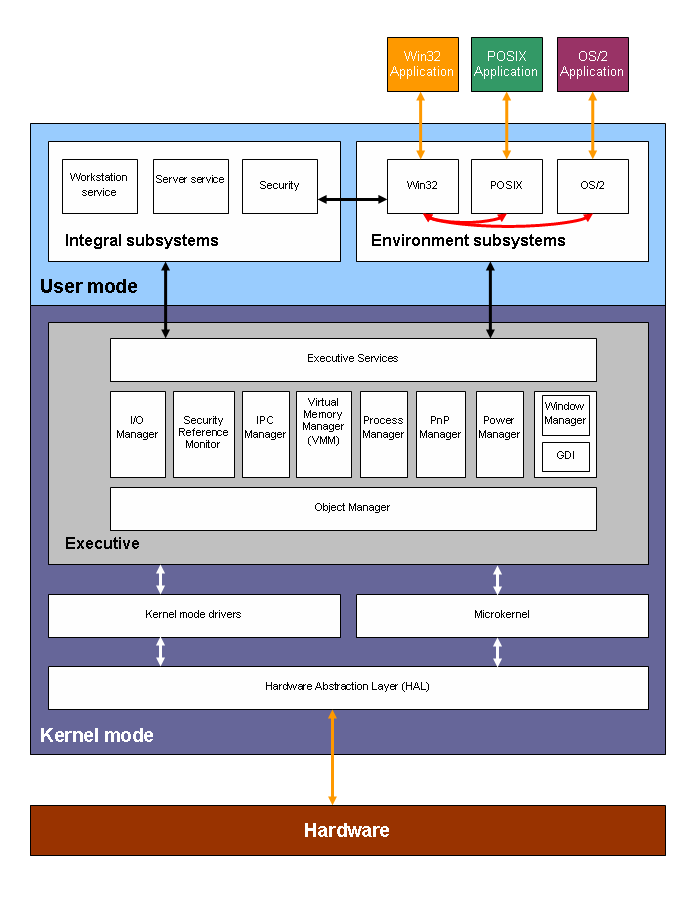


# Block Diagram

## What Is A Block Diagram?

A block diagram is a high-level system modularization that **separates the overall system into sub-systems or modules**. It especially focused on the input and output of a system, and it cares less about what happens getting from input to output. This principle is referred to as **black box** in engineering.

## Examples



## Main Components

**Block**

Block diagrams use very basic geometric shapes: boxes and circles. Principal parts of the system are represented by blocks connected by straight and segmented lines illustrating relationships.

**Arrow**

When block diagrams are used in electrical engineering, the arrows connecting components represent the direction of signal flow through the system.

## How To Draw

**Steps**

1. Identify the system: Determine the system to be illustrated. Define components, inputs, and outputs.
2. Create and label the diagram: Add a symbol for each component of the system, connecting them with arrows to indicate flow. Also, label each block so that it is easily identified.
3. Indicate input and output: Label the input that activates a block, and label that output that ends the block.

**Tips**