**UML Theory**

**Reference URLs:**

<https://sea.ucar.edu/best-practices/design>

<https://www.javatpoint.com/uml>

<https://www.javatpoint.com/uml-building-blocks>

<https://www.lucidchart.com/pages/uml-class-diagram>

<https://www.javatpoint.com/uml-object-diagram>

<https://www.javatpoint.com/uml-activity-diagram>

<https://www.smartdraw.com/use-case-diagram/>

**What is Software Modelling?**

Software models are ways of expressing a software design. Usually some sort of abstract language or pictures are used to express the software design. For object-oriented software, an object modeling language such as UML is used to develop and express the software design. There are several tools that you can use to develop your UML design.

In almost all cases a modeling language is used to develop the design not just to capture the design after it is complete. This allows the designer to try different designs and decide which will be best for the final solution. Think of designing your software as you would a house. You start by drawing a rough sketch of the floor plan and layout of the rooms and floors. The drawing is your modeling language and the resulting blueprint will be a model of your final design. You will continue to modify your drawings until you arrive at a design that meets all your requirements. Only then should you start cutting boards or writing code.

Again the benefit of designing your software using a modeling language is that you discover problems early and and fix them without refactoring your code.

**What is and Why UML?**

UML (Unified Modeling Language) is a general-purpose, graphical modeling language in the field of Software Engineering. UML is used to specify, visualize, construct, and document the artifacts (major elements) of the software system. It was initially developed by Grady Booch, Ivar Jacobson, and James Rumbaugh in 1994-95 at Rational software, and its further development was carried out through 1996. In 1997, it got adopted as a standard by the Object Management Group.

**What is UML?**

The UML stands for Unified modeling language, is a standardized general-purpose visual modeling language in the field of Software Engineering. It is used for specifying, visualizing, constructing, and documenting the primary artifacts of the software system. It helps in designing and characterizing, especially those software systems that incorporate the concept of Object orientation. It describes the working of both the software and hardware systems.

The UML was developed in 1994-95 by Grady Booch, Ivar Jacobson, and James Rumbaugh at the Rational Software. In 1997, it got adopted as a standard by the Object Management Group (OMG).

The Object Management Group (OMG) is an association of several companies that controls the open standard UML. The OMG was established to build an open standard that mainly supports the interoperability of object-oriented systems. It is not restricted within the boundaries, but it can also be utilized for modeling the non-software systems. The OMG is best recognized for the Common Object Request Broker Architecture (CORBA) standards.

**Goals of UML**

Since it is a general-purpose modeling language, it can be utilized by all the modelers.

UML came into existence after the introduction of object-oriented concepts to systemize and consolidate the object-oriented development, due to the absence of standard methods at that time.

The UML diagrams are made for business users, developers, ordinary people, or anyone who is looking forward to understand the system, such that the system can be software or non-software.

Thus it can be concluded that the UML is a simple modeling approach that is used to model all the practical systems.

**Characteristics of UML**

The UML has the following features:

It is a generalized modeling language.

It is distinct from other programming languages like C++, Python, etc.

It is interrelated to object-oriented analysis and design.

It is used to visualize the workflow of the system.

It is a pictorial language, used to generate powerful modeling artifacts.

**Conceptual Modeling**

Before moving ahead with the concept of UML, we should first understand the basics of the conceptual model.

A conceptual model is composed of several interrelated concepts. It makes it easy to understand the objects and how they interact with each other. This is the first step before drawing UML diagrams.

Following are some object-oriented concepts that are needed to begin with UML:

**Object:** An object is a real world entity. There are many objects present within a single system. It is a fundamental building block of UML.

**Class:** A class is a software blueprint for objects, which means that it defines the variables and methods common to all the objects of a particular type.

**Abstraction:** Abstraction is the process of portraying the essential characteristics of an object to the users while hiding the irrelevant information. Basically, it is used to envision the functioning of an object.

**Inheritance:** Inheritance is the process of deriving a new class from the existing ones.

**Polymorphism:** It is a mechanism of representing objects having multiple forms used for different purposes.

**Encapsulation:** It binds the data and the object together as a single unit, enabling tight coupling between them.

**UML-Building Blocks**

UML is composed of three main building blocks, i.e., things, relationships, and diagrams. Building blocks generate one complete UML model diagram by rotating around several different blocks. It plays an essential role in developing UML diagrams. The basic UML building blocks are enlisted below:

Things

Relationships

Diagrams

**Things**

Anything that is a real world entity or object is termed as things. It can be divided into several different categories:

Structural things

Behavioral things

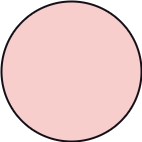
**Structural things**

Nouns that depicts the static behavior of a model is termed as structural things. They display the physical and conceptual components. They include class, object, interface, node, collaboration, component, and a use case.

**Class:** A Class is a set of identical things that outlines the functionality and properties of an object. It also represents the abstract class whose functionalities are not defined. Its notation is as follows;

**Object:** An individual that describes the behavior and the functions of a system. The notation of the object is similar to that of the class; the only difference is that the object name is always underlined and its notation is given below;

**Interface:** A set of operations that describes the functionality of a class, which is implemented whenever an interface is implemented.



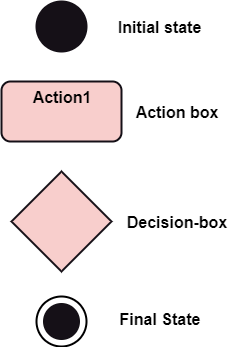
**Use case:** Use case is the core concept of object-oriented modeling. It portrays a set of actions executed by a system to achieve the goal.

**Actor:** It comes under the use case diagrams. It is an object that interacts with the system, for example, a user.

**Behavioral Things**

They are the verbs that encompass the dynamic parts of a model. It depicts the behavior of a system. They involve state machine, activity diagram, interaction diagram, grouping things, annotation things.

**Activity Diagram:** It portrays all the activities accomplished by different entities of a system. It is represented the same as that of a state machine diagram. It consists of an initial state, final state, a decision box, and an action notation.



**Relationships**

It illustrates the meaningful connections between things. It shows the association between the entities and defines the functionality of an application. There are four types of relationships given below:

**Dependency:** Dependency is a kind of relationship in which a change in target element affects the source element, or simply we can say the source element is dependent on the target element. It is one of the most important notations in UML. It depicts the dependency from one entity to another.

It is denoted by a dotted line followed by an arrow at one side as shown below,

UML-Building Blocks

**Association:** A set of links that associates the entities to the UML model. It tells how many elements are actually taking part in forming that relationship.

It is denoted by a dotted line with arrowheads on both sides to describe the relationship with the element on both sides.

UML-Building Blocks

**Generalization:** It portrays the relationship between a general thing (a parent class or superclass) and a specific kind of that thing (a child class or subclass). It is used to describe the concept of inheritance.

It is denoted by a straight line followed by an empty arrowhead at one side.

UML-Building Blocks

**Realization:** It is a semantic kind of relationship between two things, where one defines the behavior to be carried out, and the other one implements the mentioned behavior. It exists in interfaces.

It is denoted by a dotted line with an empty arrowhead at one side.

UML-Building Blocks

**Diagrams**

The diagrams are the graphical implementation of the models that incorporate symbols and text. Each symbol has a different meaning in the context of the UML diagram. There are thirteen different types of UML diagrams that are available in UML 2.0, such that each diagram has its own set of a symbol. And each diagram manifests a different dimension, perspective, and view of the system.

UML diagrams are classified into two categories that are given below:

Structural Diagram

Behavioral Diagram

**Structural Diagram:** It represents the static view of a system by portraying the structure of a system. It shows several objects residing in the system. Following are the structural diagrams given below:

Class diagram

Object diagram

**Behavioral Diagram:** It depicts the behavioral features of a system. It deals with dynamic parts of the system. It encompasses the following diagrams:

Activity diagram

Use case diagram

Types of Modeling and diagrams(overview level):

Structural Modelling Diagrams:

Class Diagram:

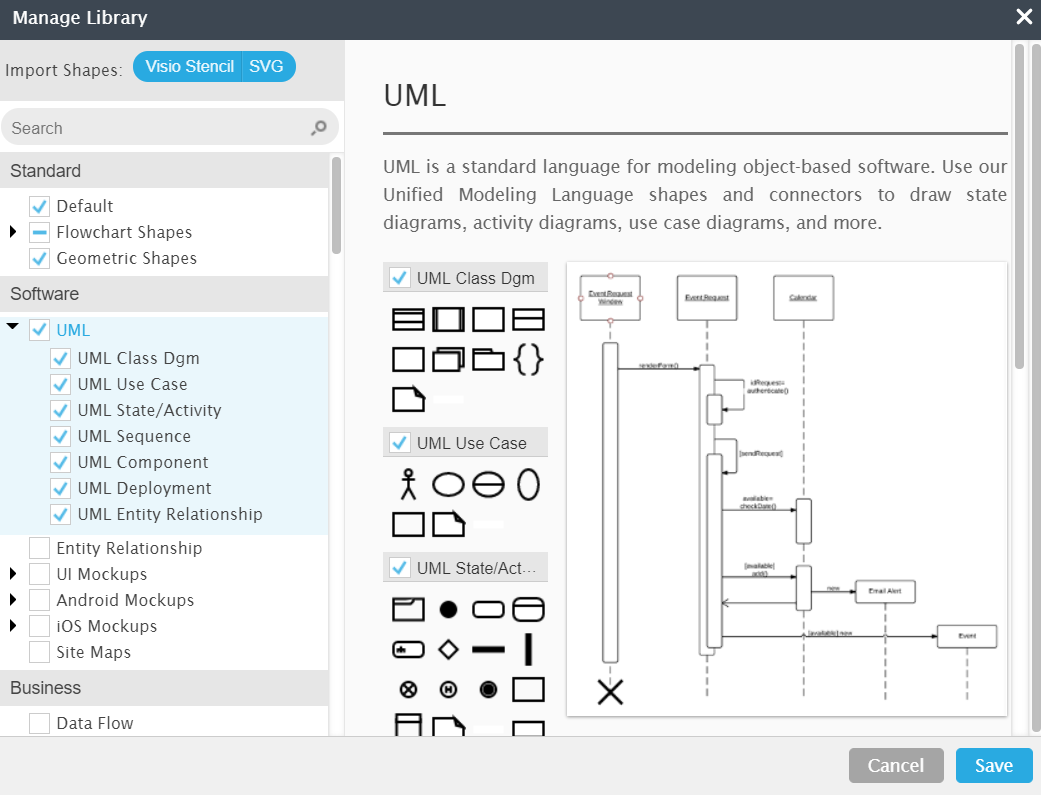
## What is a class diagram in UML?

The [Unified Modeling Language](https://www.lucidchart.com/pages/what-is-UML-unified-modeling-language) (UML) can help you model systems in various ways. One of the more popular types in UML is the class diagram. Popular among software engineers to document software architecture, class diagrams are a type of structure diagram because they describe what must be present in the system being modeled. No matter your level of familiarity with UML or class diagrams, our [UML software](https://www.lucidchart.com/pages/examples/uml_diagram_tool) is designed to be simple and easy to use.

UML was set up as a standardized model to describe an object-oriented programming approach. Since classes are the building block of objects, class diagrams are the building blocks of UML. The various components in a class diagram can represent the classes that will actually be programmed, the main objects, or the interactions between classes and objects.

The class shape itself consists of a rectangle with three rows. The top row contains the name of the class, the middle row contains the attributes of the class, and the bottom section expresses the methods or operations that the class may use. Classes and subclasses are grouped together to show the static relationship between each object.

The UML shape library in Lucidchart can help you create nearly any custom class diagram using our UML diagram tool.



## Benefits of class diagrams

Class diagrams offer a number of benefits for any organization. Use UML class diagrams to:

* Illustrate data models for information systems, no matter how simple or complex.
* Better understand the general overview of the schematics of an application.
* Visually express any specific needs of a system and disseminate that information throughout the business.
* Create detailed charts that highlight any specific code needed to be programmed and implemented to the described structure.
* Provide an implementation-independent description of types used in a system that are later passed between its components.

Diagramming is quick and easy with Lucidchart. Start a free trial today to start creating and collaborating.

[Create a UML Diagram](https://lucid.app/pricing/lucidchart?anonId=0.44e5c8e117debb5a6ab&sessionDate=2021-12-24T09%3A11%3A30.220Z&sessionId=0.47123ea317debb5a6ab&type=discovery)

## Basic components of a class diagram

The standard class diagram is composed of three sections:

* **Upper section:**Contains the name of the class. This section is always required, whether you are talking about the classifier or an object.
* **Middle section:**Contains the attributes of the class. Use this section to describe the qualities of the class. This is only required when describing a specific instance of a class.
* **Bottom section:**Includes class operations (methods). Displayed in list format, each operation takes up its own line. The operations describe how a class interacts with data.

### Member access modifiers

All classes have different access levels depending on the access modifier (visibility). Here are the access levels with their corresponding symbols:

* Public (+)
* Private (-)
* Protected (#)
* Package (~)
* Derived (/)
* Static (underlined)

### Member scopes

There are two scopes for members: classifiers and instances.

Classifiers are static members while instances are the specific instances of the class. If you are familiar with basic OO theory, this isn't anything groundbreaking.

### Additional class diagram components

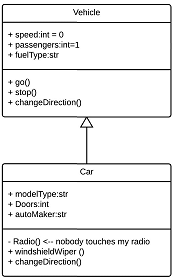
Depending on the context, classes in a class diagram can represent the main objects, interactions in the application, or classes to be programmed. To answer the question "What is a class diagram in UML?" you should first understand its basic makeup.

* **Classes:** A template for creating objects and implementing behavior in a system. In UML, a class represents an object or a set of objects that share a common structure and behavior. They're represented by a rectangle that includes rows of the class name, its attributes, and its operations. When you draw a class in a class diagram, you're only required to fill out the top row—the others are optional if you'd like to provide more detail.
  + **Name:** The first row in a class shape.
  + **Attributes:** The second row in a class shape. Each attribute of the class is displayed on a separate line.
  + **Methods:** The third row in a class shape. Also known as operations, methods are displayed in list format with each operation on its own line.
* **Signals**: Symbols that represent one-way, asynchronous communications between active objects.
* **Data types:** Classifiers that define data values. Data types can model both primitive types and enumerations.
* **Packages:** Shapes designed to organize related classifiers in a diagram. They are symbolized with a large tabbed rectangle shape.
* **Interfaces:**A collection of operation signatures and/or attribute definitions that define a cohesive set of behaviors. Interfaces are similar to classes, except that a class can have an instance of its type, and an interface must have at least one class to implement it.
* **Enumerations:** Representations of user-defined data types. An enumeration includes groups of identifiers that represent values of the enumeration.
* **Objects:** Instances of a class or classes. Objects can be added to a class diagram to represent either concrete or prototypical instances.
* **Artifacts:** Model elements that represent the concrete entities in a software system, such as documents, databases, executable files, software components, etc.

### Interactions

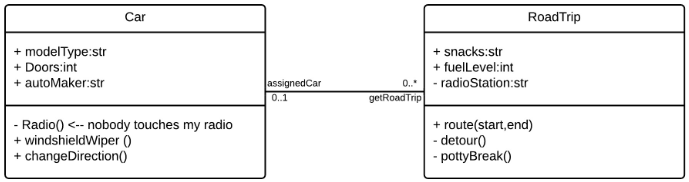
The term "interactions" refers to the various relationships and links that can exist in class and object diagrams. Some of the most common interactions include:

* **Inheritance:** The process of a child or sub-class taking on the functionality of a parent or superclass, also known as generalization. It's symbolized with a straight connected line with a closed arrowhead pointing towards the superclass.



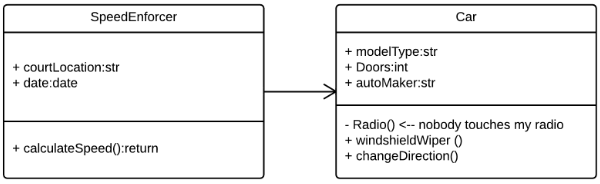
In this example, the object "Car" would inherit all of the attributes (speed, numbers of passengers, fuel) and methods (go(), stop(), changeDirection()) of the parent class ("Vehicle") in addition to the specific attributes (model type, number of doors, auto maker) and methods of its own class (Radio(), windshieldWiper(), ac/heat()). Inheritance is shown in a class diagram by using a solid line with a closed, hollow arrow.

* **Bidirectional association:** The default relationship between two classes. Both classes are aware of each other and their relationship with the other. This association is represented by a straight line between two classes.



In the example above, the Car class and RoadTrip class are interrelated. At one end of the line, the Car takes on the association of "assignedCar" with the multiplicity value of 0..1, so when the instance of RoadTrip exists, it can either have one instance of Car associated with it or no Cars associated with it. In this case, a separate Caravan class with a multiplicity value of 0..\* is needed to demonstrate that a RoadTrip could have multiple instances of Cars associated with it. Since one Car instance could have multiple "getRoadTrip" associations—in other words, one car could go on multiple road trips—the multiplicity value is set to 0..\*

* **Unidirectional association:** A slightly less common relationship between two classes. One class is aware of the other and interacts with it. Unidirectional association is modeled with a straight connecting line that points an open arrowhead from the knowing class to the known class.



As an example, on your road trip through Arizona, you might run across a speed trap where a speed cam records your driving activity, but you won't know about it until you get a notification in the mail. It isn't drawn in the image, but in this case, the multiplicity value would be 0..\* depending on how many times you drive by the speed cam.

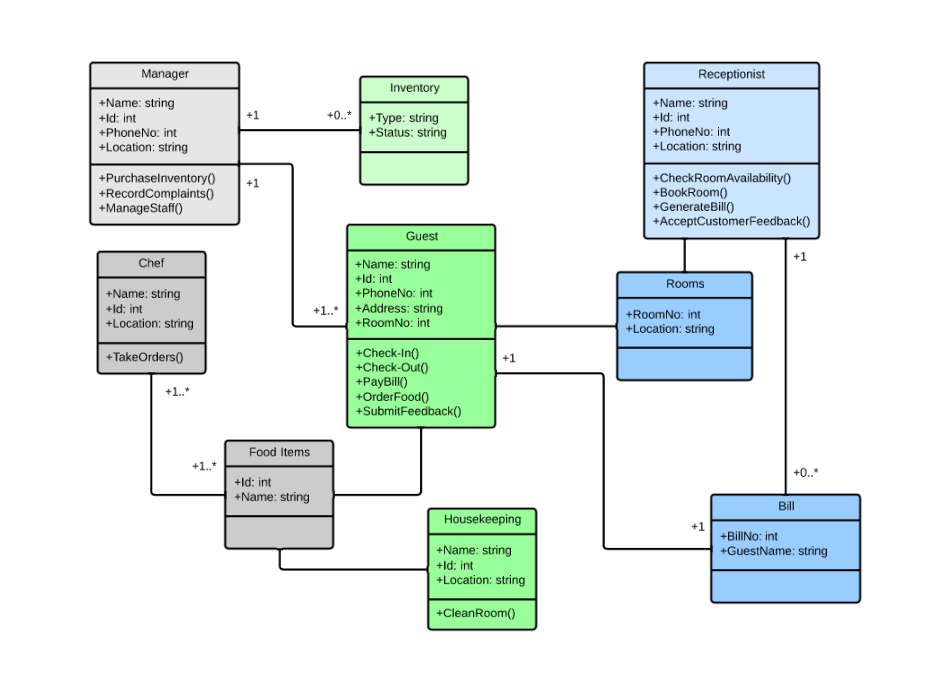
## Class diagram examples

Creating a class diagram to map out process flows is easy. Consider the three examples below as you build your own class diagrams in UML.

### UML Class Diagram

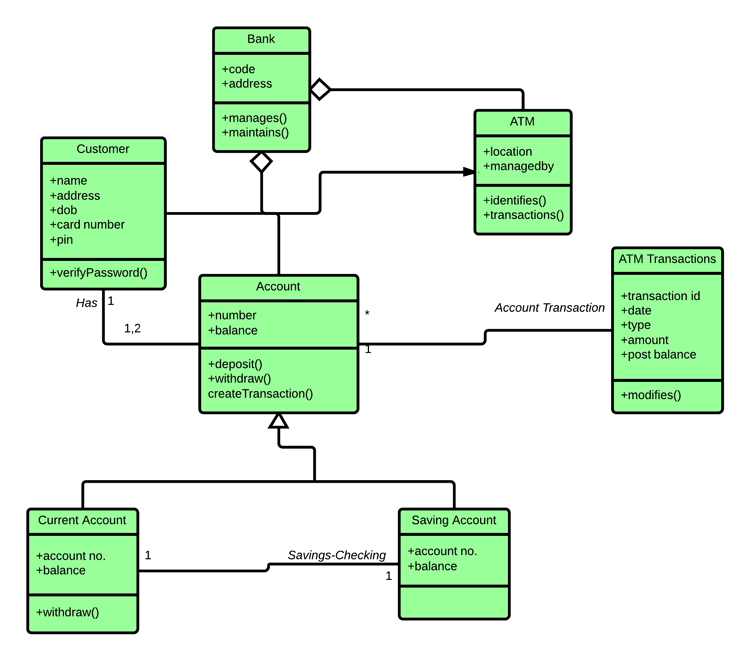
### Class diagram for a hotel management system

A class diagram can show the relationships between each object in a hotel management system, including guest information, staff responsibilities, and room occupancy. The example below provides a useful overview of the hotel management system. Get started on a class diagram by clicking the template below.

[Click here to use this template](https://www.lucidchart.com/documents/editNewOrRegister/d3ab2c31-8f2f-4ada-8f23-fb1654c0f0c9)

### Class diagram for an ATM system

ATMs are deceptively simple: although customers only need to press a few buttons to receive cash, there are many layers of security that a safe and effective ATM must pass through to prevent fraud and provide value for banking customers. The various human and inanimate parts of an ATM system are illustrated by this easy-to-read diagram—every class has its title, and the attributes are listed beneath. You can edit, save, and share this chart by opening the document and signing up for a free Lucidchart account.



[*Click here to use this template*](https://www.lucidchart.com/documents/editNewOrRegister/f3617503-abc4-4fa3-9b16-7dccea412d79)

## How to make a class diagram

In Lucidchart, creating a class diagram from scratch is surprisingly simple. Just follow these steps:

1. Open a blank document or start with a template.
2. Enable the UML shape library. On the left side of the Lucidchart editor, click "Shapes." Once you're in the Shape Library Manager, check "UML" and click "Save."
3. From the libraries you just added, select the shape you want and drag it from the toolbox to the canvas.
4. Model the process flow by drawing lines between shapes while adding text.

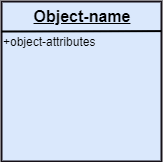
Dive into this guide on [how to draw a class diagram in UML](https://www.lucidchart.com/pages/how-to-draw-a-class-diagram-in-UML) for additional insight. In Lucidchart, it's easy to resize and style any element. You can even import SVG shapes and Visio files for a custom solution. If you'd like to learn more about UML, check out our tutorial, "[What Is UML](https://www.lucidchart.com/pages/what-is-UML-unified-modeling-language)?"

# **UML Object Diagram**

Object diagrams are dependent on the class diagram as they are derived from the class diagram. It represents an instance of a class diagram. The objects help in portraying a static view of an object-oriented system at a specific instant.

Both the object and class diagram are similar to some extent; the only difference is that the class diagram provides an abstract view of a system. It helps in visualizing a particular functionality of a system.

## Notation of an Object Diagram



## Purpose of Object Diagram

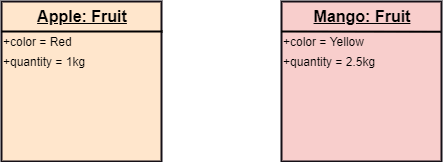
The object diagram holds the same purpose as that of a class diagram. The class diagram provides an abstract view which comprises of classes and their relationships, whereas the object diagram represents an instance at a particular point of time.

The object diagram is actually similar to the concrete (actual) system behavior. The main purpose is to depict a static view of a system.

Following are the purposes enlisted below:

* It is used to perform forward and reverse engineering.
* It is used to understand object behavior and their relationships practically.
* It is used to get a static view of a system.
* It is used to represent an instance of a system.

## Example of Object Diagram



## How to draw an Object Diagram?

1. All the objects present in the system should be examined before start drawing the object diagram.
2. Before creating the object diagram, the relation between the objects must be acknowledged.
3. The association relationship among the entities must be cleared already.
4. To represent the functionality of an object, a proper meaningful name should be assigned.
5. The objects are to be examined to understand its functionality.

## Applications of Object diagrams

The following are the application areas where the object diagrams can be used.

1. To build a prototype of a system.
2. To model complex data structures.
3. To perceive the system from a practical perspective.
4. Reverse engineering.

## Class vs. Object diagram

|  |  |  |
| --- | --- | --- |
| **Serial No.** | **Class Diagram** | **Object Diagram** |
| 1. | It depicts the static view of a system. | It portrays the real-time behavior of a system. |
| 2. | Dynamic changes are not included in the class diagram. | Dynamic changes are captured in the object diagram. |
| 3. | The data values and attributes of an instance are not involved here. | It incorporates data values and attributes of an entity. |
| 4. | The object behavior is manipulated in the class diagram. | Objects are the instances of a class. |

Behavioral Diagrams:

# **UML Activity Diagram**

In UML, the activity diagram is used to demonstrate the flow of control within the system rather than the implementation. It models the concurrent and sequential activities.

The activity diagram helps in envisioning the workflow from one activity to another. It put emphasis on the condition of flow and the order in which it occurs. The flow can be sequential, branched, or concurrent, and to deal with such kinds of flows, the activity diagram has come up with a fork, join, etc.

It is also termed as an object-oriented flowchart. It encompasses activities composed of a set of actions or operations that are applied to model the behavioral diagram.

## Components of an Activity Diagram

Following are the component of an activity diagram:

**Activities**

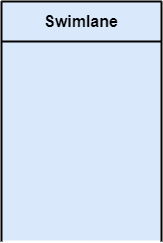
The categorization of behavior into one or more actions is termed as an activity. In other words, it can be said that an activity is a network of nodes that are connected by edges. The edges depict the flow of execution. It may contain action nodes, control nodes, or object nodes.

The control flow of activity is represented by control nodes and object nodes that illustrates the objects used within an activity. The activities are initiated at the initial node and are terminated at the final node.



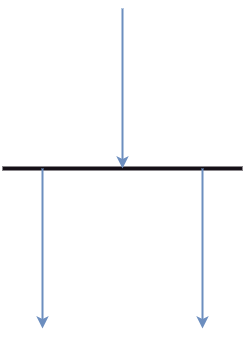
**Activity partition /swimlane**

The swimlane is used to cluster all the related activities in one column or one row. It can be either vertical or horizontal. It used to add modularity to the activity diagram. It is not necessary to incorporate swimlane in the activity diagram. But it is used to add more transparency to the activity diagram.



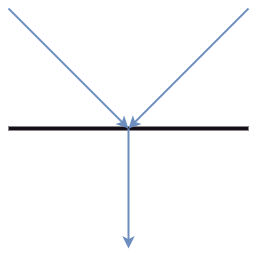
**Forks**

Forks and join nodes generate the concurrent flow inside the activity. A fork node consists of one inward edge and several outward edges. It is the same as that of various decision parameters. Whenever a data is received at an inward edge, it gets copied and split crossways various outward edges. It split a single inward flow into multiple parallel flows.



**Join Nodes**

Join nodes are the opposite of fork nodes. A Logical AND operation is performed on all of the inward edges as it synchronizes the flow of input across one single output (outward) edge.



**Pins**

It is a small rectangle, which is attached to the action rectangle. It clears out all the messy and complicated thing to manage the execution flow of activities. It is an object node that precisely represents one input to or output from the action.

## Notation of an Activity diagram

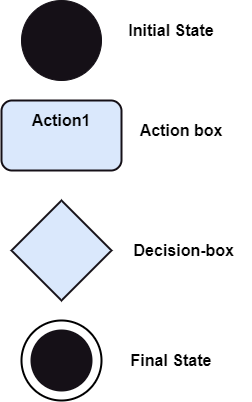
Activity diagram constitutes following notations:

**Initial State:** It depicts the initial stage or beginning of the set of actions.

**Final State:** It is the stage where all the control flows and object flows end.

**Decision Box:** It makes sure that the control flow or object flow will follow only one path.

**Action Box:** It represents the set of actions that are to be performed.



## Why use Activity Diagram?

An event is created as an activity diagram encompassing a group of nodes associated with edges. To model the behavior of activities, they can be attached to any modeling element. It can model use cases, classes, interfaces, components, and collaborations.

It mainly models processes and workflows. It envisions the dynamic behavior of the system as well as constructs a runnable system that incorporates forward and reverse engineering. It does not include the message part, which means message flow is not represented in an activity diagram.

It is the same as that of a flowchart but not exactly a flowchart itself. It is used to depict the flow between several activities.

## How to draw an Activity Diagram?

An activity diagram is a flowchart of activities, as it represents the workflow among various activities. They are identical to the flowcharts, but they themself are not exactly the flowchart. In other words, it can be said that an activity diagram is an enhancement of the flowchart, which encompasses several unique skills.

Since it incorporates swimlanes, branching, parallel flows, join nodes, control nodes, and forks, it supports exception handling. A system must be explored as a whole before drawing an activity diagram to provide a clearer view of the user. All of the activities are explored after they are properly analyzed for finding out the constraints applied to the activities. Each and every activity, condition, and association must be recognized.

After gathering all the essential information, an abstract or a prototype is built, which is then transformed into the actual diagram.

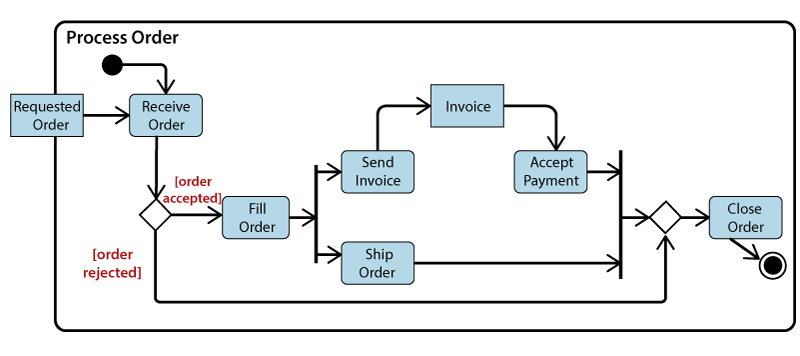
Following are the rules that are to be followed for drawing an activity diagram:

1. A meaningful name should be given to each and every activity.
2. Identify all of the constraints.
3. Acknowledge the activity associations.

## Example of an Activity Diagram

An example of an activity diagram showing the business flow activity of order processing is given below.

Here the input parameter is the Requested order, and once the order is accepted, all of the required information is then filled, payment is also accepted, and then the order is shipped. It permits order shipment before an invoice is sent or payment is completed.

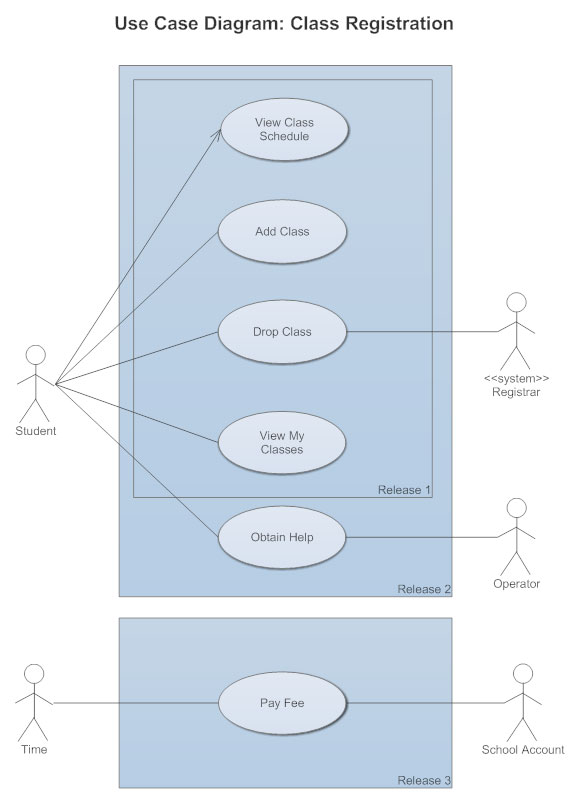


## When to use an Activity Diagram?

An activity diagram can be used to portray business processes and workflows. Also, it used for modeling business as well as the software. An activity diagram is utilized for the followings:

1. To graphically model the workflow in an easier and understandable way.
2. To model the execution flow among several activities.
3. To model comprehensive information of a function or an algorithm employed within the system.
4. To model the business process and its workflow.
5. To envision the dynamic aspect of a system.
6. To generate the top-level flowcharts for representing the workflow of an application.
7. To represent a high-level view of a distributed or an object-oriented system.

# **Use Case Diagram**



## What is a Use Case Diagram?

A use case diagram is a dynamic or behavior diagram in [UML](https://www.smartdraw.com/uml-diagram/). Use case diagrams model the functionality of a system using actors and use cases. Use cases are a set of actions, services, and functions that the system needs to perform. In this context, a "system" is something being developed or operated, such as a web site. The "actors" are people or entities operating under defined roles within the system.

## Why Make Use Case Diagrams?

Use case diagrams are valuable for visualizing the functional requirements of a system that will translate into design choices and development priorities.

They also help identify any internal or external factors that may influence the system and should be taken into consideration.

They provide a good high level analysis from outside the system. Use case diagrams specify how the system interacts with actors without worrying about the details of how that functionality is implemented.

## Basic Use Case Diagram Symbols and Notations

**System**  
Draw your system's boundaries using a rectangle that contains use cases. Place actors outside the system's boundaries.



**Use Case**  
Draw use cases using ovals. Label the ovals with verbs that represent the system's functions.



**Actors**  
Actors are the users of a system. When one system is the actor of another system, label the actor system with the actor stereotype.



**Relationships**  
Illustrate relationships between an actor and a use case with a simple line. For relationships among use cases, use arrows labeled either "uses" or "extends." A "uses" relationship indicates that one use case is needed by another in order to perform a task. An "extends" relationship indicates alternative options under a certain use case.

