<u>Difference Between Link and Association</u>**

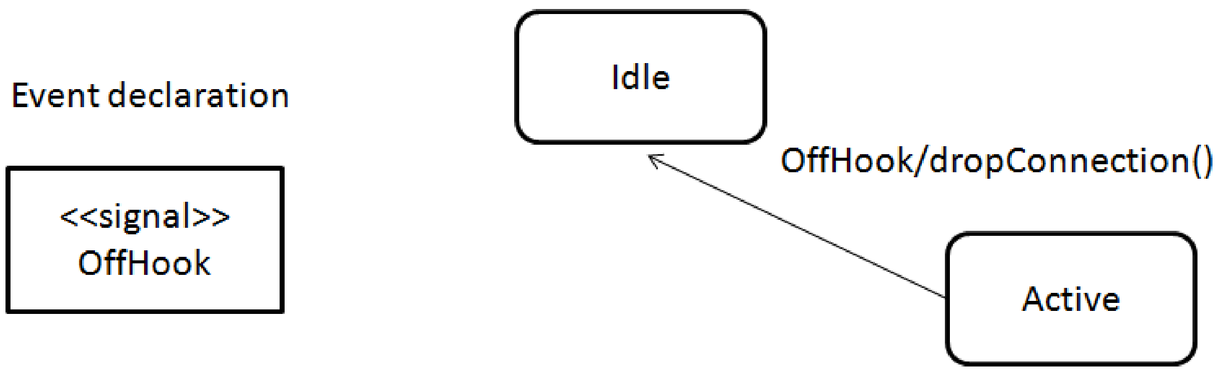
The major difference between link and association is that link is a physical or theoretical connection between the objects whereas association is a group of links with same structure and semantics. Associations are implemented in programming languages as a reference model in which one object is referenced from the another. While links cannot be referenced as these are not objects by itself, but rely on the objects.

BASIS FOR COMPARISON	**LINK**	**ASSOCIATION**
Basic	A link can be defined as a theoretical and physical connection between objects.	An association is a specification of a collection of links.
Function	Relationship among objects.	Connects related classes.
UML design symbol	Line segment between objects.	Also uses line segment but it shows the connection between classes.

In state machines (sequence of states), we use events to model the occurrence of a stimulus that can trigger an object to move from one state to another state. Events may include signals, calls, the passage of time or a change in state.

In UML, each thing that happens is modeled as an event. An event is the specification of a significant occurrence that has a location in time and space. A signal, passing of time and change in state are asynchronous events. Calls are generally synchronous events, representing invocation of an operation.

UML allows us to represent events graphically as shown below. Signals may be represented as stereotyped classes and other events are represented as messages associated with transitions which cause an object to move from one state to another.



Events & signals

Types of Events

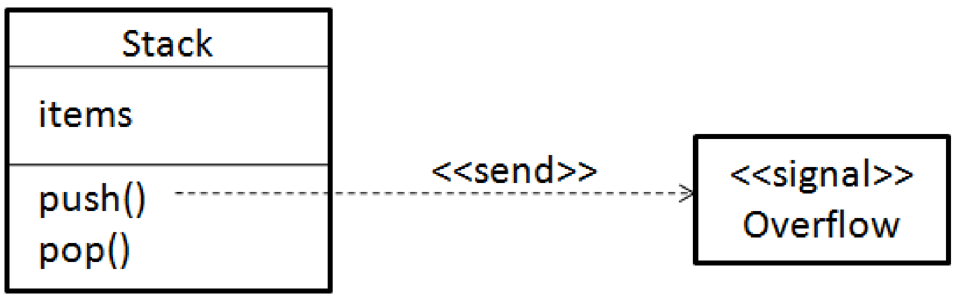
Events may be external or internal. Events passed between the system and its actors are external events. For example, in an ATM system, pushing a button or inserting a card are external events. Internal events are those that are passed among objects living inside the system. For example, a overflow exception generated by an object is an internal event.

In UML, we can model four kinds of events namely: signals, calls, passing of time and change in state.

Signals

A signal is a named object that is sent asynchronously by one object and then received by another. Exceptions are the famous examples for signals. A signal may be sent as the action of a state in a state machine or as a message in an interaction. The execution of an operation can also send signals.

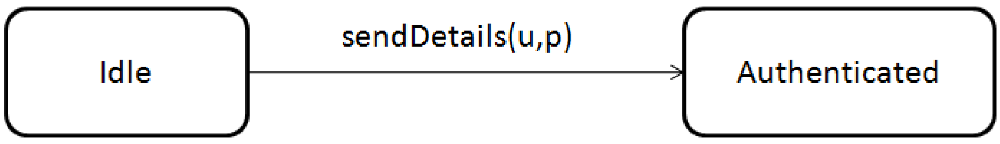
In UML, we model the relationship between an operation and the events using a dependency stereotyped with “send”, which indicates that an operation sends a particular signal.



Call Events

A call event represents the dispatch of an operation from one object to another. A call event may trigger a state change in a state machine. A call event, in general, is synchronous.

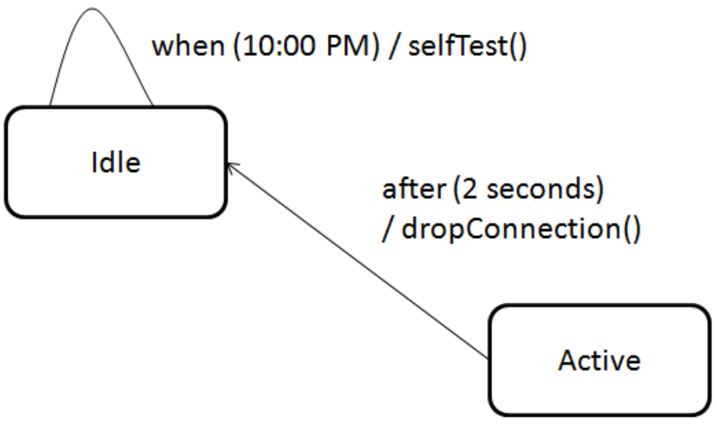
This means that the sender object must wait until it gets an acknowledgment from the receiver object which receives the call event. For example, consider the states of a customer in an ATM application:



Time and Change Events

A time event represents the passage of time. In UML, we model the time event using the “after” keyword followed by an expression that evaluates a period of time.

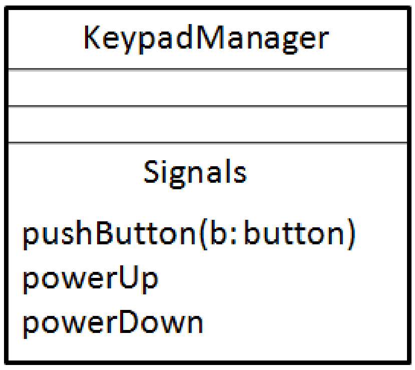
A change event represents an event that represents a change in state or the satisfaction of some condition. In UML, change event is modeled using the keyword “when” followed by some Boolean expression.



Sending and Receiving Events

Any instance of a class can receive a call event or signal. If this is a synchronous call event, the sender is in locked state with receiver. If this is a signal, then the sender is free to carry its operations without any concern on the receiver.

In UML, call events are modeled as operations on the class of an object and signals that an object can receive are stored in an extra component in the class as shown below:



-

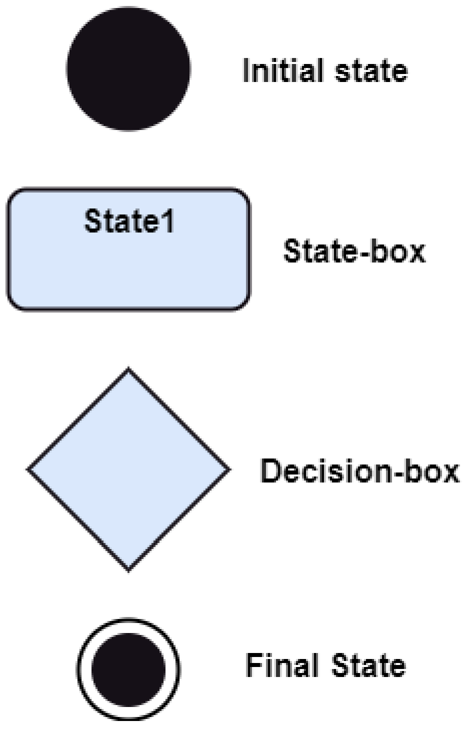
<u>**State Machine Diagram**</u>

The state machine diagram is also called the Statechart or State Transition diagram, which shows the order of states underwent by an object within the system. It captures the software system's behavior. It models the behavior of a class, a subsystem, a package, and a complete system.

It tends out to be an efficient way of modeling the interactions and collaborations in the external entities and the system. It models event-based systems to handle the state of an object. It also defines several distinct states of a component within the system. Each object/component has a specific state.

N<u>otation of a State Machine Diagram</u>

Following are the notations of a state machine diagram enlisted below:


Initial state
State1
State-box
Decision-box
Final State

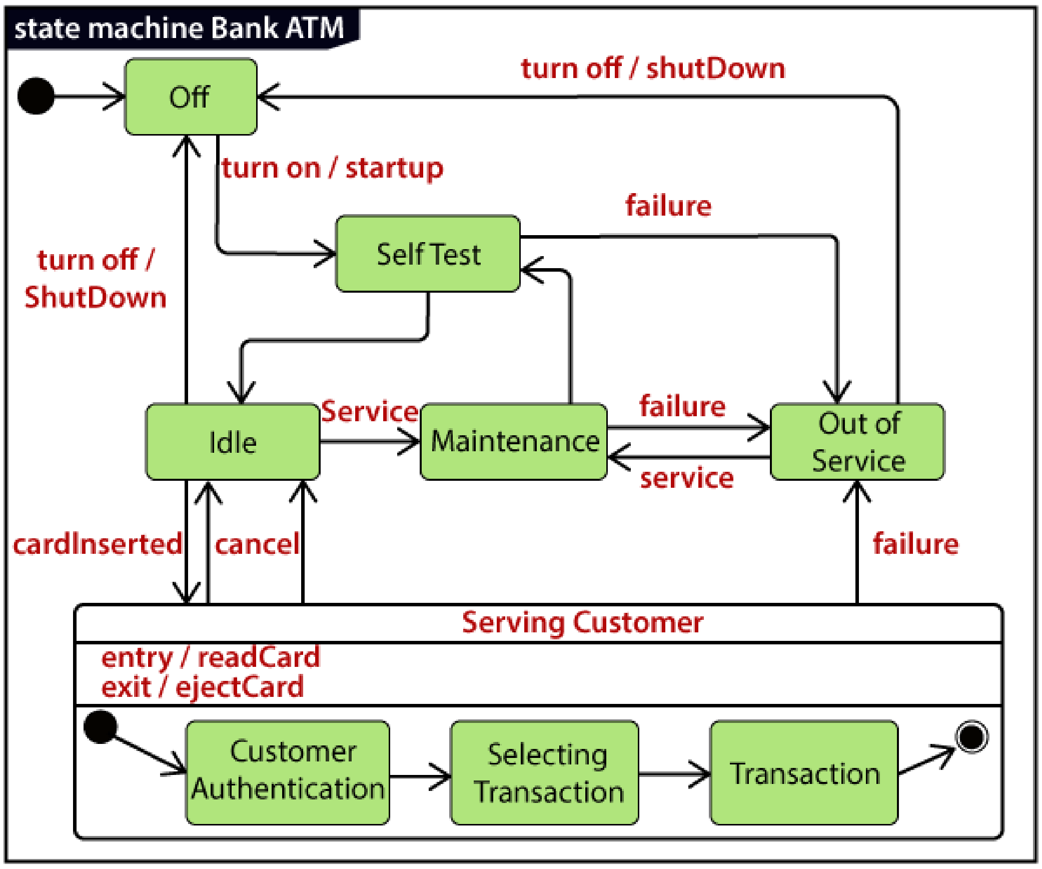
a. **Initial state:** It defines the initial state (beginning) of a system, and it is represented by a black filled circle.
b. **Final state:** It represents the final state (end) of a system. It is denoted by a filled circle present within a circle.
c. **Decision box:** It is of diamond shape that represents the decisions to be made on the basis of an evaluated guard.
d. **Transition:** A change of control from one state to another due to the occurrence of some event is termed as a transition. It is represented by an arrow labeled with an event due to which the change has ensued.
e. **State box:** It depicts the conditions or circumstances of a particular object of a class at a specific point of time. A rectangle with round corners is used to represent the state box.

Example of a State Machine Diagram

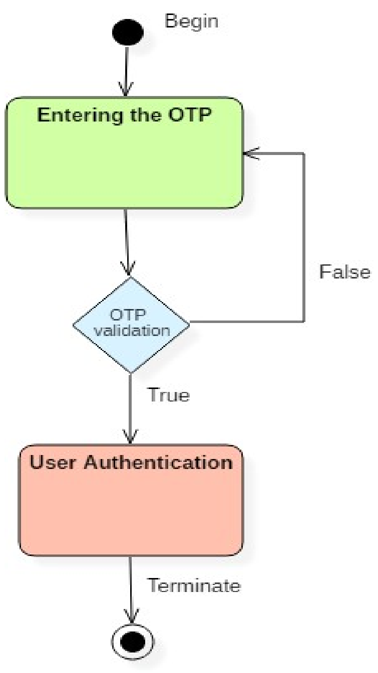
An example of a top-level state machine diagram showing Bank Automated Teller Machine (ATM) is given below.

Initially, the ATM is turned off. After the power supply is turned on, the ATM starts performing the startup action and enters into the **Self Test** state. If the test fails, the ATM will enter into the **Out Of Service** state, or it will undergo **a triggerless transition** to the **Idle** state. This is the state where the customer waits for the interaction.

Whenever the customer inserts the bank or credit card in the ATM's card reader, the ATM state changes from **Idle** to **Serving Customer**, the entry action **readCard** is performed after entering into **Serving Customer** state. Since the customer can cancel the transaction at any instant, so the transition from **Serving Customer** state back to the **Idle** state could be triggered by **cancel** event.



Following state chart diagram represents the user authentication process.:



UML Timing Diagram

In UML, the timing diagrams are a part of Interaction diagrams that do not incorporate similar notations as that of sequence and collaboration diagram. It consists of a graph or waveform that depicts the state of a lifeline at a specific point of time. It illustrates how conditions are altered both inside and between lifelines alongside linear time axis.

The timing diagram describes how an object underwent a change from one form to another. A waveform portrays the flow among the software programs at several instances of time.

Following are some important key points of a timing diagram:

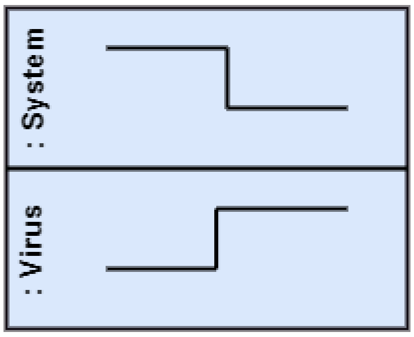
1. It emphasizes at that particular time when the message has been sent among objects.
2. It explains the time processing of an object in detail.
3. It is employed with distributed and embedded systems.
4. It also explains how an object undergoes changes in its form throughout its lifeline.
5. As the lifelines are named on the left side of an edge, the timing diagrams are read from left to right.
6. It depicts a graphical representation of states of a lifeline per unit time.
7. In UML, the timing diagram has come up with several notations to simplify the transition state among two lifelines per unit time.

Basic concepts of a Timing Diagram

In UML, the timing diagram constitutes several major elements, which are as follows:

Lifeline

As the name suggests, the lifeline portrays an individual element in the interaction. It represents a single entity, which is a part of the interaction. It is represented by the classifier's name that it depicts. A lifeline can be placed within a "swimlane" or a diagram frame.

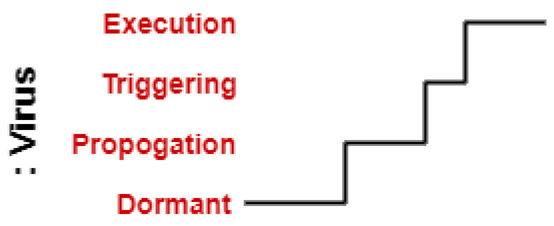


Lifelines representing instances of a System and Virus

State or Condition Timeline

The timing diagram represents the state of a classifier or attributes that are participating, or some testable conditions, which is a discrete value of the classifier.

In UML, the state or condition is continuous. It is mainly used to show the temperature and density where the entities endure a continuous state change.

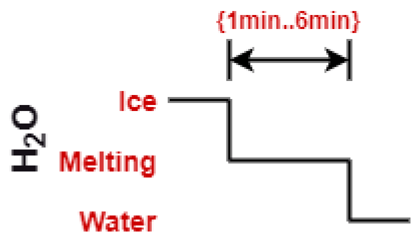


Timeline showing the change in the state of virus between dormant, Propagation, Triggering, Execution

Duration Constraint

The duration constraint is a constraint of an interval, which refers to duration interval. It is used to determine if the constraint is satisfied for a duration or not. The duration constraint semantics inherits from the constraints.

The negative trace defines the violated constraints, which means the system is failed. A graphical association between duration interval and the construct, which it constrains, may represent a duration constraint.

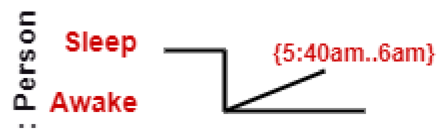


Ice should melt into the water in 1 to 6 mins.

Time Constraint

Time constraint is an interval constraint that refers to a time interval. The time interval is time expression used to determine whether the constraint is satisfied.

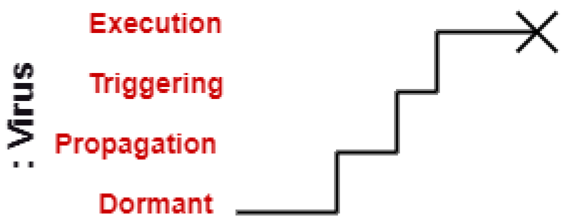
The graphical association is mainly represented by a small line in between a time interval and an occurrence specification.



A person should wakeup in between 5:40 am, and 6 am

Destruction Occurrence

The destruction occurrence refers to the occurrence of a message that represents the destruction of an instance is defined by a lifeline. It may subsequently destruct other objects owned by the composition of this object, such that nothing occurs after the destruction event on a given lifeline. It is represented by a cross at the end of a timeline.



Virus lifeline is terminated

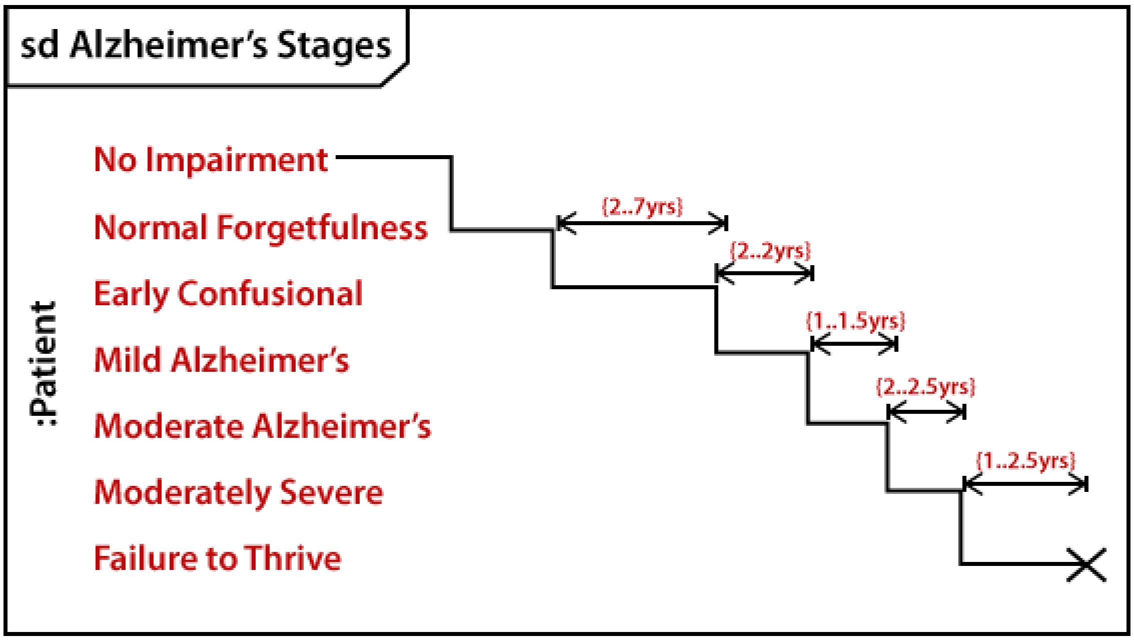
Example of a Timing Diagram

A timing diagram example of a medical domain that depicts different stages of Alzheimer's disease (AD) is explained below.

Since Alzheimer's is a very progressive fatal brain disease, it leads to memory loss and intellectual abilities. The reason behind this disease is yet to be discovered. It cannot be cured as well as one of the main reasons for rising death rates in the United States.

The doctor may require a diagnostic framework with three to seven-stage, such that its evolution may last for about 8 to 10 years. Also, in some cases, it lasts up to 20years from the time neuron starts changing.

The example given below constitutes timing for a seven-stage framework. The given example is just a UML diagram and should not be considered as a reference to medical research. The medical details are provided for you to better understand the UML diagram.



Following are the seven-stage Alzheimer disease framework explained below:

- **No Impairment, Normal State**
  It is the stage where the memory and cognitive abilities look normal.
- **Normal Aged Forgetfulness**
  It is mostly seen in people with an age group of 65 who experience subjective complaints of cognitive and/or functional difficulties, which means they face problems in recalling the name and past 5 to 10 years of history.
- **Early Confusional, Mild Cognitive Impairment**
  It causes a problem in retrieving words, planning, organizing, objects misplacing as well as forgetting fresh learning, which in turn affects the surrounding.
- **Late Confusional, Mild Alzheimer's**
  In this, a person forgets the most recent events and conversations. The person remembers himself and his family, but face problems while carrying out sequential tasks such as cooking, driving, etc. Its duration is about two years,
- **Early Dementia, Moderate Alzheimer's**
  In this, the person cannot manage independently. He faces difficulty in recalling the past details and contact information. It lasts for about 1.5 years.
- **Middle Dementia, Moderately Severe Alzheimer's**
  It leads to insufficient awareness about current events, and the person is unable to recall the past. It causes an inability in people to take a bath and dress up independently. It lasts for about 2.5 years approximately.
- **Late or Severe Dementia, Failure to Thrive**
  It is severely limited intellectual ability. In this, a person either communicates through short words or cries, which leads health to decline as it shut down the body system. Its duration is 1 to 2.5 years.

Benefits of Timing Diagram

1. It depicts the state of an object at a particular point in time.
2. It implements forward and reverses engineering.
3. It keeps an eye on every single change that happens within the system.

UML Component Diagram

A component diagram is used to break down a large object-oriented system into the smaller components, so as to make them more manageable. It models the physical view of a system such as executables, files, libraries, etc. that resides within the node.

It visualizes the relationships as well as the organization between the components present in the system. It helps in forming an executable system. A component is a single unit of the system, which is replaceable and executable. The implementation details of a component are hidden, and it necessitates an interface to execute a function. It is like a black box whose behavior is explained by the provided and required interfaces.

Purpose of a Component Diagram

Since it is a special kind of a UML diagram, it holds distinct purposes. It describes all the individual components that are used to make the functionalities, but not the functionalities of the system. It visualizes the physical components inside the system. The components can be a library, packages, files, etc.

The component diagram also describes the static view of a system, which includes the organization of components at a particular instant. The collection of component diagrams represents a whole system.

The main purpose of the component diagram are enlisted below:

1. It envisions each component of a system.
2. It constructs the executable by incorporating forward and reverse engineering.
3. It depicts the relationships and organization of components.

When to use a Component Diagram?

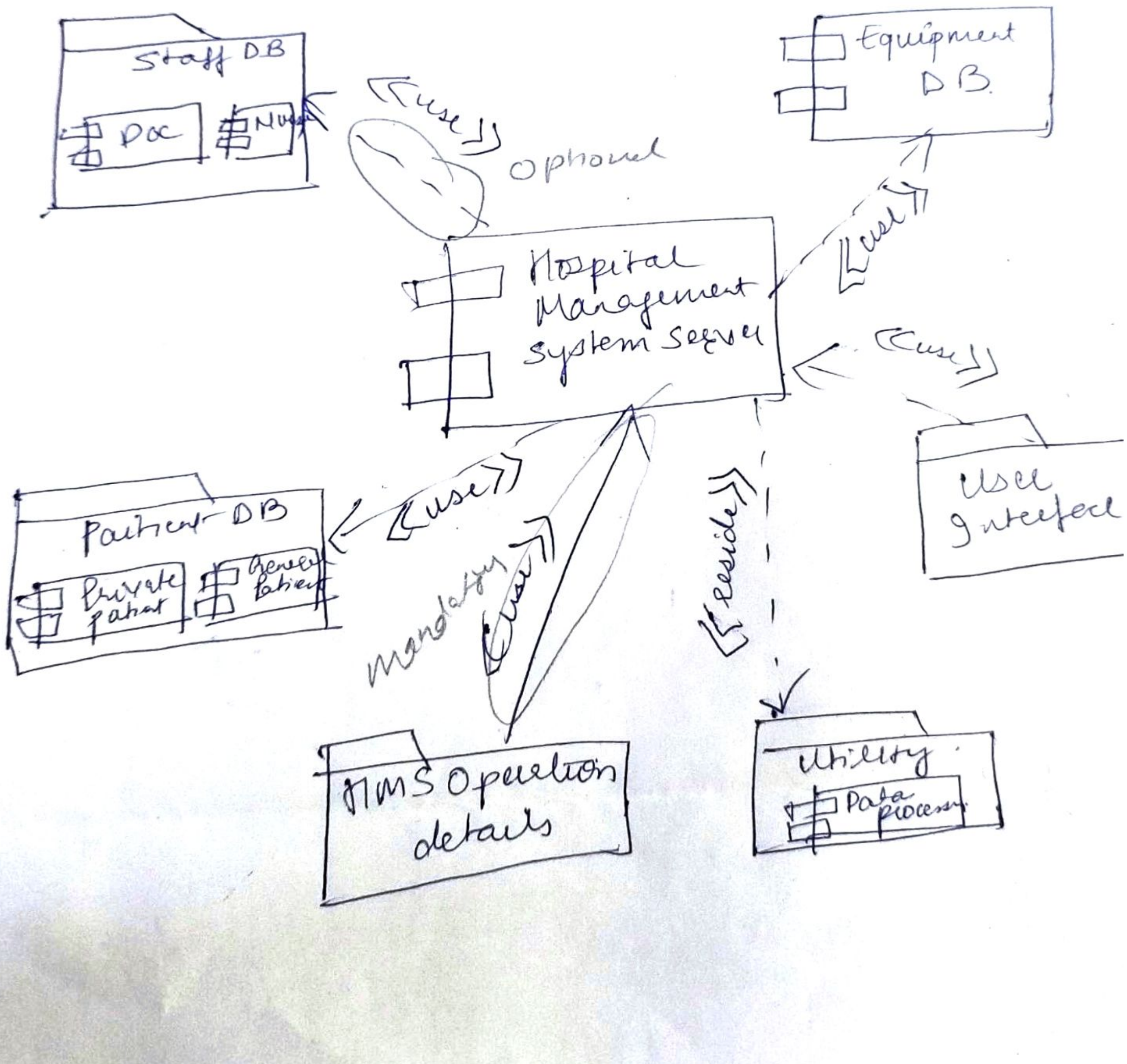
It represents various physical components of a system at runtime. It is helpful in visualizing the structure and the organization of a system. It describes how individual components can together form a single system. Following are some reasons, which tells when to use component diagram:

1. To divide a single system into multiple components according to the functionality.
2. To represent the component organization of the system.

Component Diagram

Component
Package

Hospital M.S



Deployment Diagram

Deployment Diagram is a type of diagram that specifies the physical hardware on which the software system will execute. It also determines how the software is deployed on the underlying hardware. It maps software pieces of a system to the device that are going to execute it.

The deployment diagram maps the software architecture created in design to the physical system architecture that executes it.

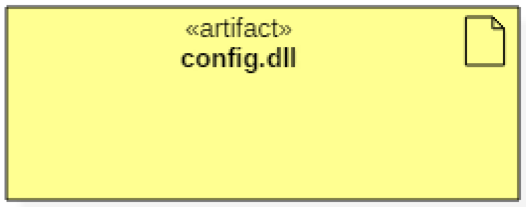
The software systems are manifested using various **artifacts**, and then they are mapped to the execution environment that is going to execute the software such as **nodes**. Many nodes are involved in the deployment diagram; hence, the relation between them is represented using communication paths.

What is an artifact?

An artifact represents the specification of a concrete real-world entity related to software development. You can use the artifact to describe a framework which is used during the software development process or an executable file. Artifacts are deployed on the nodes. The most common artifacts are as follows,

1. Source files
2. Executable files
3. Database tables
4. Scripts
5. DLL files
6. User manuals or documentation
7. Output files

Artifacts are labeled with the stereotype **<<artifact>>,** and it may have an artifact icon on the top right corner.



What is a node?

Node is a computational resource upon which artifacts are deployed for execution. A node is a physical thing that can execute one or more artifacts.

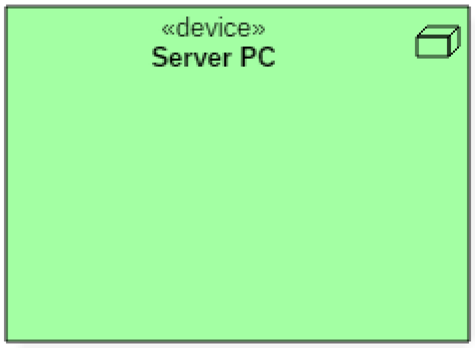
Generally, a node has two stereotypes as follows:

- **<< device >>**

  It is a node that represents a physical machine capable of performing computations. A device can be a router or a server PC. It is represented using a node with stereotype <<device>>.

  In the UML model, you can also nest one or more devices within each other.

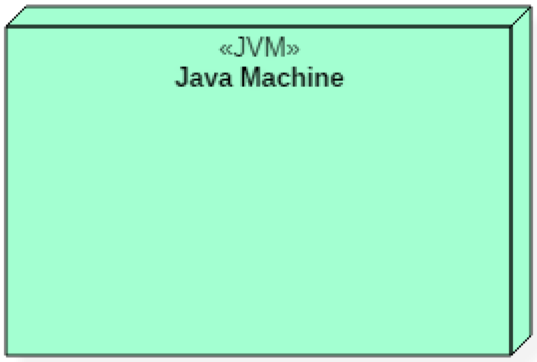
  Following is a representation of a device in UML:



device node

- **<< execution environment >>**

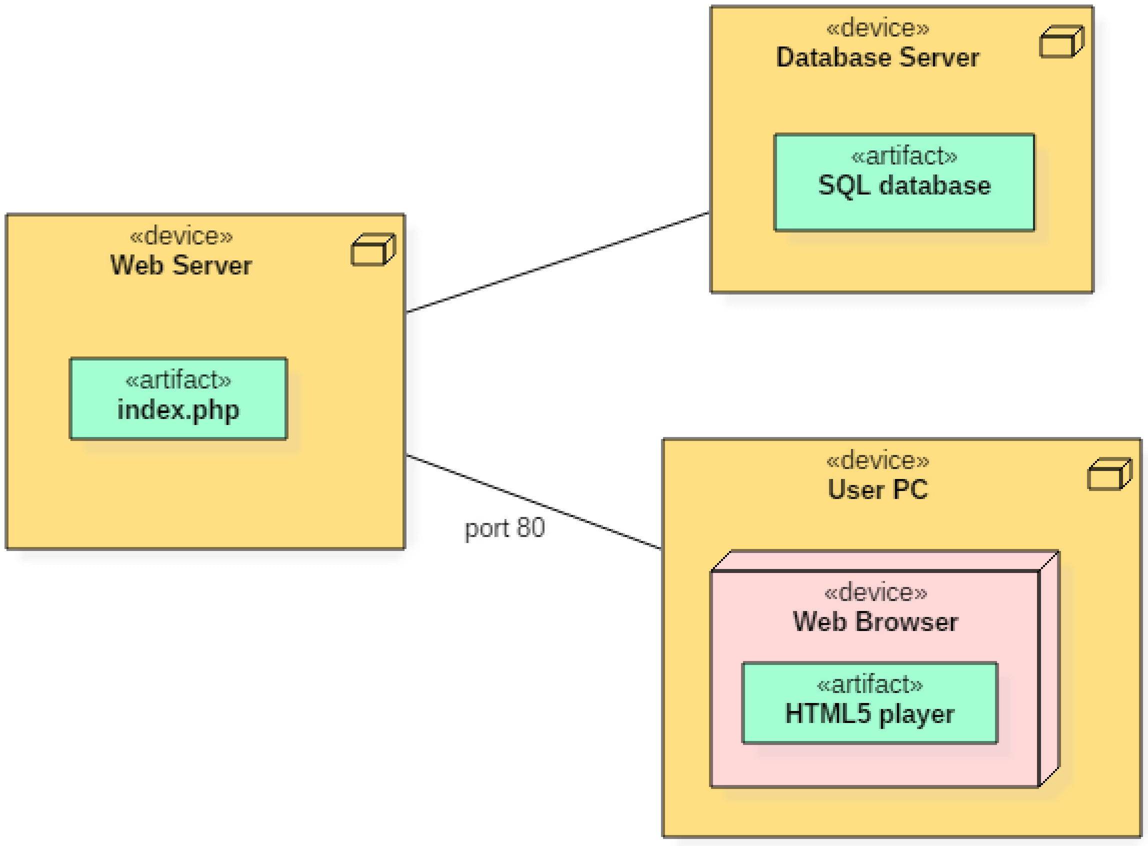
  It is a node that represents an environment in which software is going to execute. For example, Java applications are executed in java virtual machine (JVM). JVM is considered as an execution environment for Java applications. We can nest an execution environment into a device node. You can net more than one execution environments in a single device node.
  Following is a representation of an execution environment in UML:



execution environment node

Example of a Deployment diagram

Following deployment diagram represents the working of HTML5 video player in the browser:



Deployment Diagram

What is a package diagram?

Package diagrams are structural diagrams used to show the organization and arrangement of various model elements in the form of packages. A package is a grouping of related UML elements, such as diagrams, documents, classes, or even other packages. Each element is nested within the package, which is depicted as a file folder within the diagram, then arranged hierarchically within the diagram. Package diagrams are most commonly used to provide a visual organization of the layered architecture within any UML classifier, such as a software system.

Benefits of a package diagram

A well-designed package diagram provides numerous benefits to those looking to create a visualization of their UML system or project.

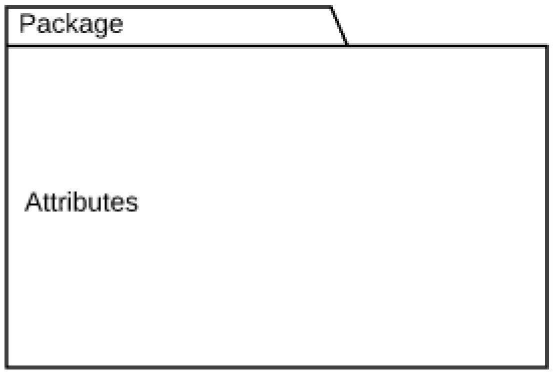
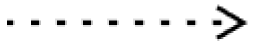
- They provide a clear view of the hierarchical structure of the various UML elements within a given system.

- These diagrams can simplify complex class diagrams into well-ordered visuals.
- They offer valuable high-level visibility into large-scale projects and systems.
- Package diagrams can be used to visually clarify a wide variety of projects and systems.
- These visuals can be easily updated assystems and projects evolve.

Basic components of a package diagram

The makeup of a package diagram is relatively simple. Each diagram includes only two symbols:

Symbol Image	**Symbol Name**	**Description**
	Package	Groups common elements based on data, behavior, or user interaction

		
	Dependenc y	Depicts the relationship between one element (package, named element, etc) and another

These symbols can be used in a variety of ways to represent different iterations of packages, dependencies, and other elements within a system. Here are the basic components you'll find within a package diagram:

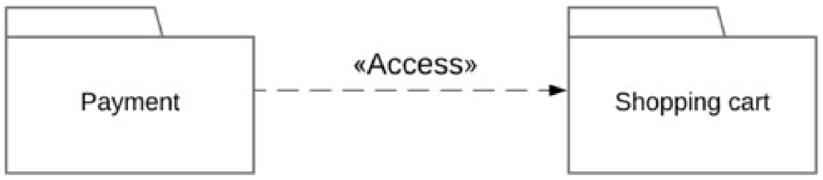
- **Package**: A namespace used to group together logically related elements within a system. Each element contained within the package should be a packageable element and have a unique name.
- **Packageable element**: A named element, possibly owned directly by a package. These can include events, components, use cases, and packages themselves.
  Packageable elements can also be rendered as a rectangle within a package, labeled with the appropriate name.
- **Dependencies**: A visual representation of how one element (or set of elements) depends on or influences another. Dependencies are divided into two groups: access and import dependencies. (See next section for more info.)
- **Element import**: A directed relationship between an importing namespace and an imported packageable element. This is used to import select individual elements without resorting to a package import and without making it public within the namespace.
- **Package import**: A directed relationship between and importing namespace and an imported package. This type of directed relationship adds the names of the members of the imported package to its own namespace
- **Package merge**: A directed relationship in which the contents of one package are extended by the contents of another. Essentially, the content of two packages are combined to produce a new package.

Dependency notations in a package diagram

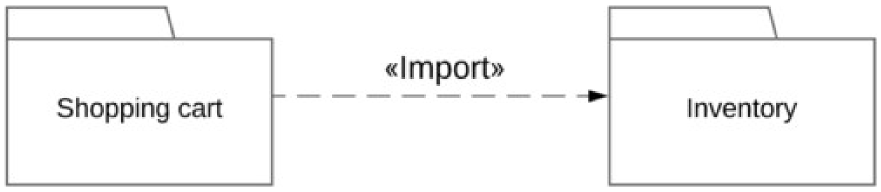
Package diagrams are used, in part, to depict import and access dependencies between packages, classes, components, and other named elements within your system. Each dependency is rendered as a connecting line with an arrow representing the type of relationship between the two or more elements.

There are two main types of dependencies:

Access: Indicates that one package requires assistance from the functions of another package. Example:

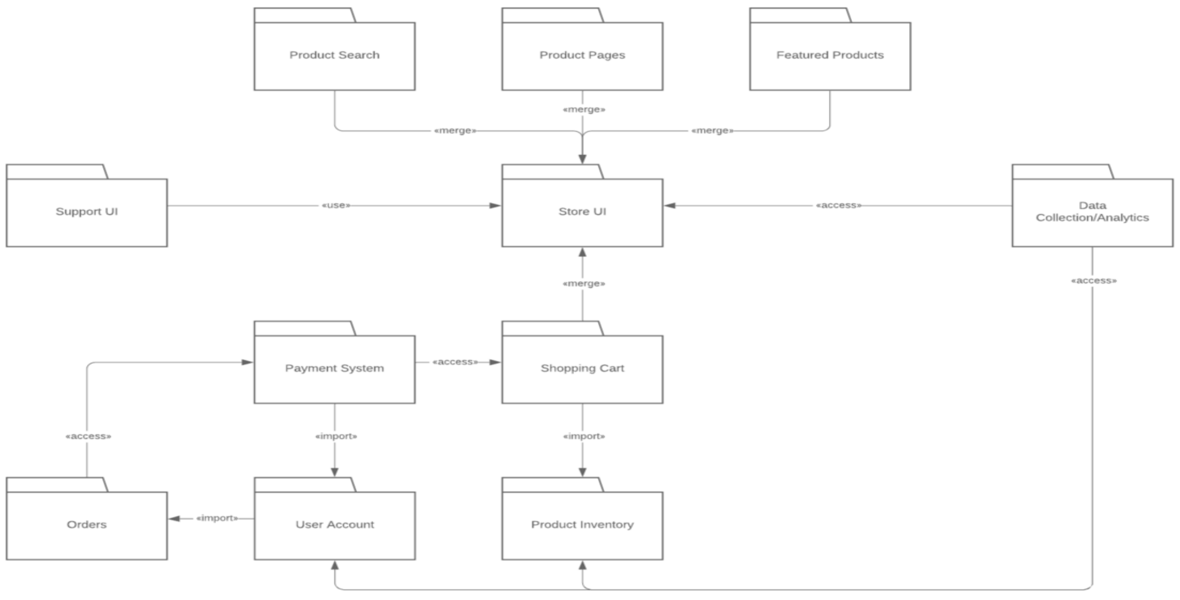


Import: Indicates that functionality has been imported from one package to another. Example:



Package diagram example

Take a look at the following template to see how a package diagram models the packages within a basic e-commerce web app. Click on the template to modify it and explore how you can show the structure of any designed system at a package level.


Product Search
Product Pages
Featured Products
«merge»
«merge»
«merge»
Support UI
«use»
Store UI
«access»
Data Collection/Analytics
«merge»
«access»
Payment System
«access»
Shopping Cart
«access»
«import»
«import»
Orders
«import»
User Account
Product Inventory