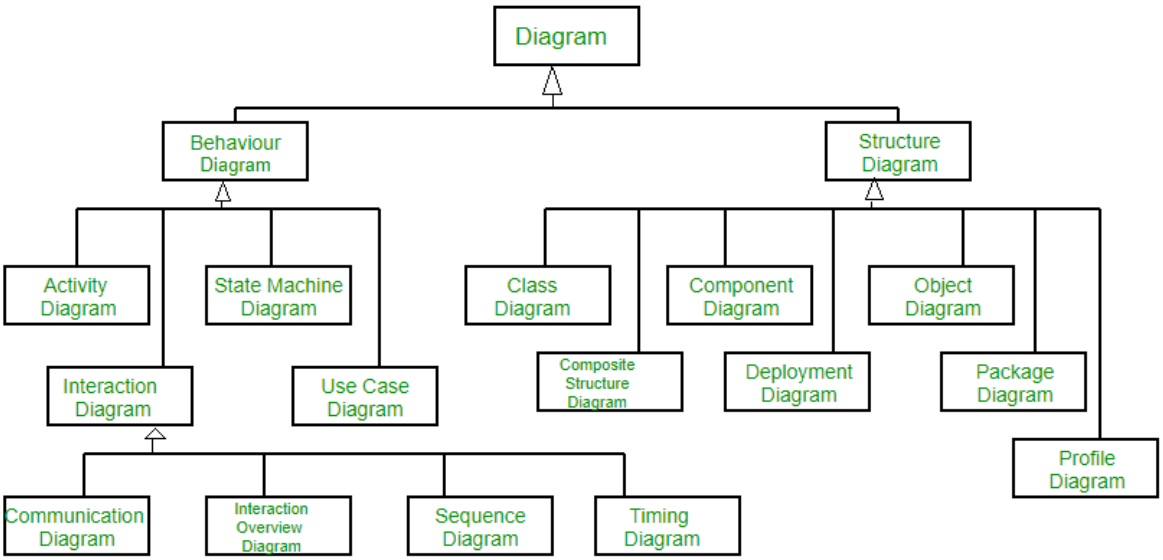
UML

Unified Modeling Language



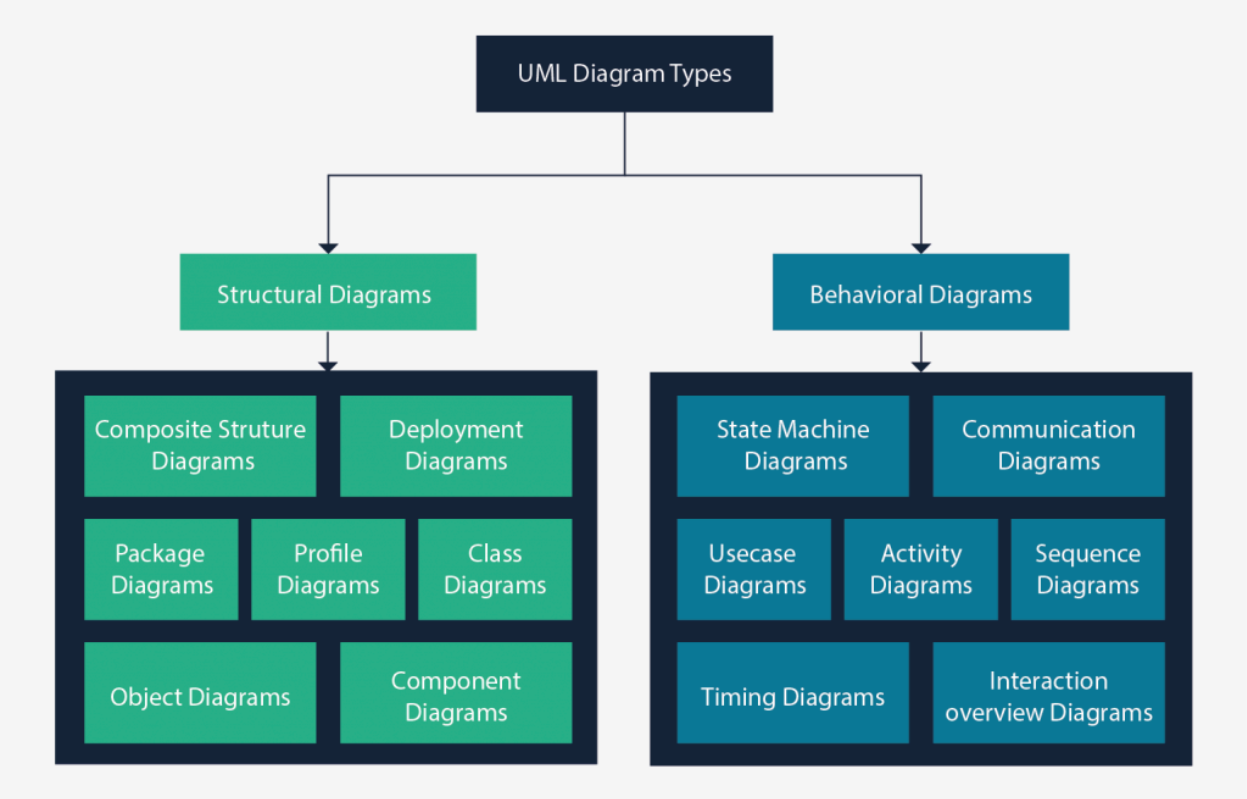
Types UML

Structure Diagrams

1. Class Diagram
2. Component Diagram
3. Deployment Diagram
4. Object Diagram
5. Package Diagram
6. Profile Diagram
7. Composite Structure Diagram

Behavioral Diagrams

1. Use Case Diagram
2. Activity Diagram
3. State Machine Diagram
4. Sequence Diagram
5. Communication Diagram
6. Interaction Overview Diagram
7. Timing Diagram



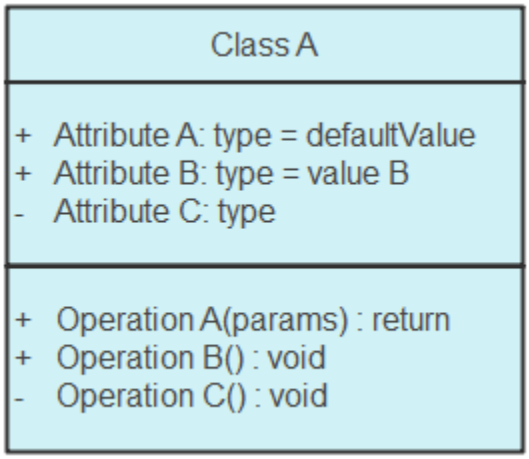
Structure Diagrams

1-Class Diagram

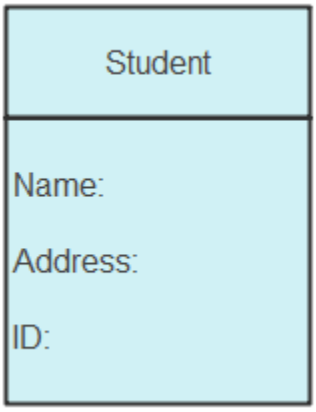
Class Notation

There are three major parts of a class diagram as shown in the image below:

1. **Class Name**
2. **Class Attributes**
3. **Class Operations**



Attributes



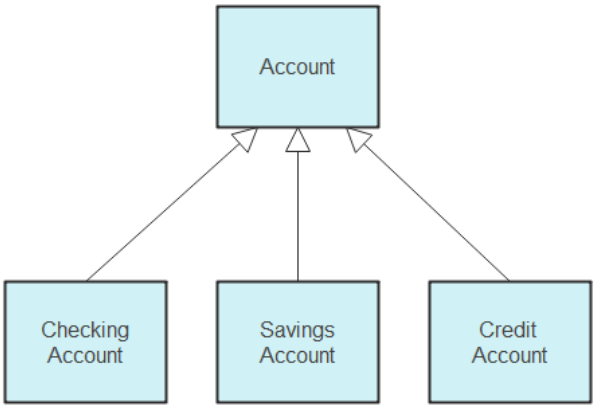
Class Relationships

To create a class diagram, the next step is building relationships. There are three main types of relationships here:

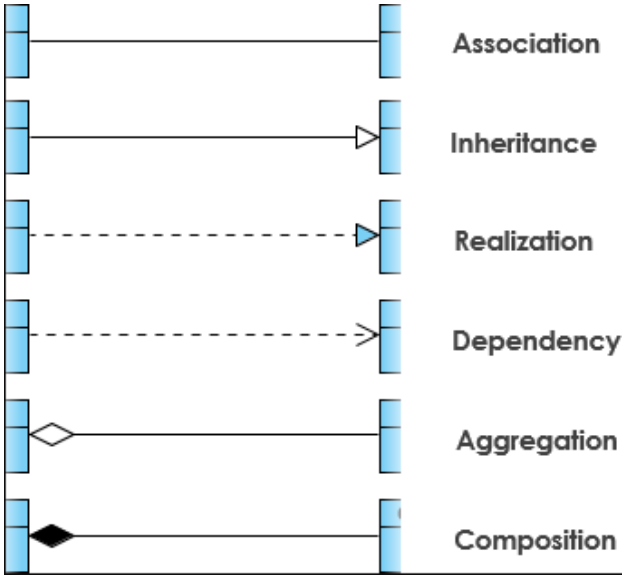
1. **Generalizations**
2. **Associations**
3. **Dependencies**

Generalizations

Generalizations are often known as **Inheritance** because it links a subclass to its superclass. The class diagram allows a subclass to inherit from multiple superclasses but it can't be used to model interface implementation. Checking, Savings, and Credit Accounts are *generalized* by Account



Associations



Association

```
1. class Car {
2. private Engine engine;
3. private Wheel[] wheels;  }
4. class Engine { // engine }
5. class Wheel { // wheel }
```

Aggregation

```
1. class Library {
2. private List<Book> books;
3. // constructors, methods, etc.
4. }
5. class Book {
6.  // book implementation
7. }
```

Composition

```
1. class House {
2. private Room livingRoom;
3. // constructors, methods, etc.
4. }
5. class Room {
6. // room implementation
7. }
```

Inheritance

```
1. class Animal {
2. // animal implementation }
3. class Dog extends Animal {
4. // additional dog-specific properties and
   behaviors
5. }
```

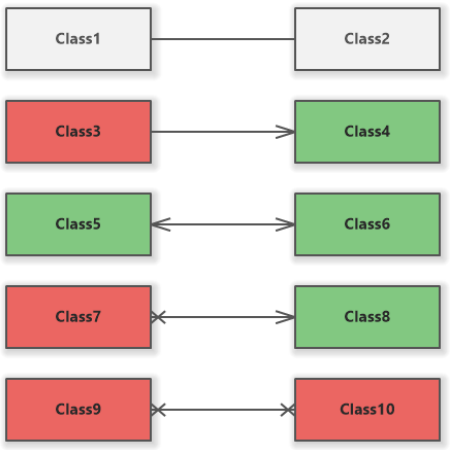
Realization

```
1. interface Printable {
2. void print();  }
3. class Document implements Printable {
4. public void print() { // printing }}
```

Dependency

```
1. class Order {
2. private Customer customer;
3. // constructors, methods, etc. }
4. class Customer {
5. // customer implementation }
```

Association Navigability



Unspecified navigability

Navigable in one direction

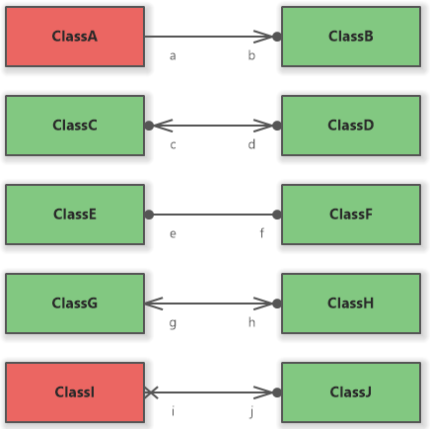
Navigable in both directions

Navigable in one direction (explicit)

Non-navigable

---

ssociation Navigability with Class Ownership



Class-owned on one side (navigable in one direction)

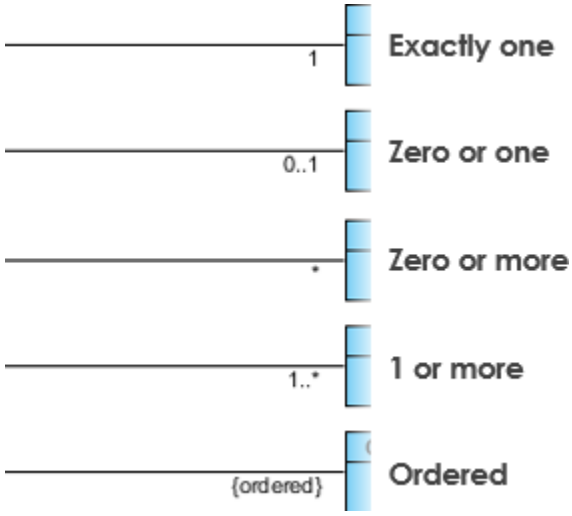
Class-owned on both sides (explicitly navigable)

Class-owned on both sides (implicitly navigable)

Class-owned on one side (navigable in both directions)

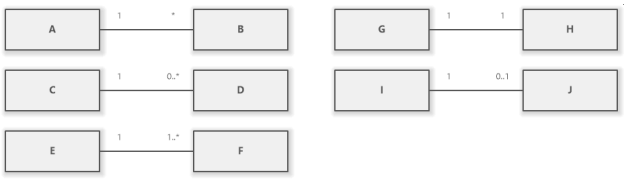
Class-owned on one side (explicitly non-navigable)

Association Cardinality



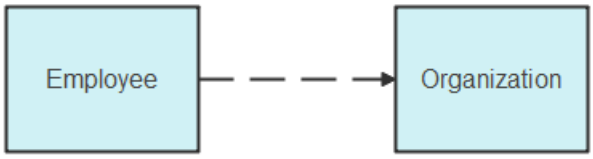
The association ends can have various cardinalities:

- One
- Zero or one
- Zero to many
- One to many
- Many
- A specific number
- A specific range of numbers



Dependencies

Dependency shows that one class depends on another. Change in one class will create change in another class. For example, an employee is dependent on the organization.



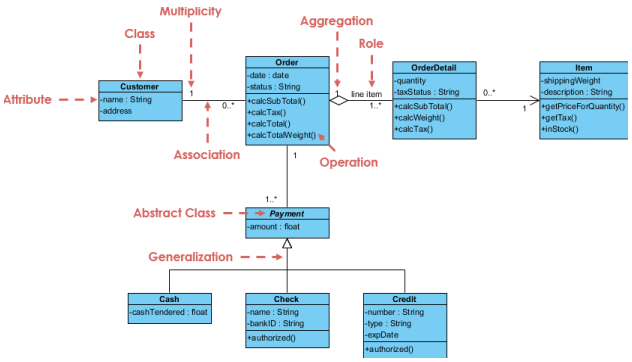

```
1. class Organization {
2. // organization implementation
3. }
4. class Employee {
5. private Organiz organiz;
6. public Employee(Organiz organiz){
7. this.organiz=organiz;
8. } }
```

Member access modifiers

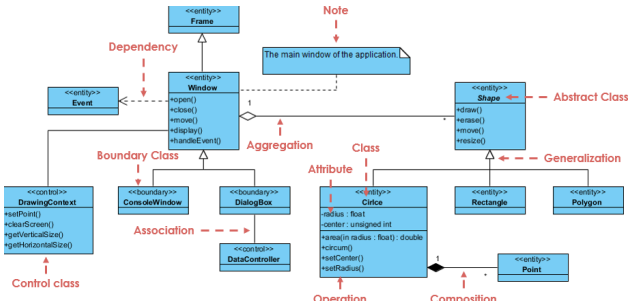
All classes have different access levels depending on the access modifier (visibility). Here are the access levels with their corresponding symbols:

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

Class Diagram Example: Order System

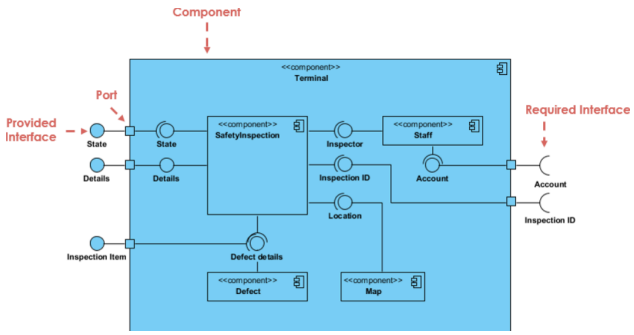


Class Diagram Example: GUI



2-Component Diagram

Component Diagram at a Glance

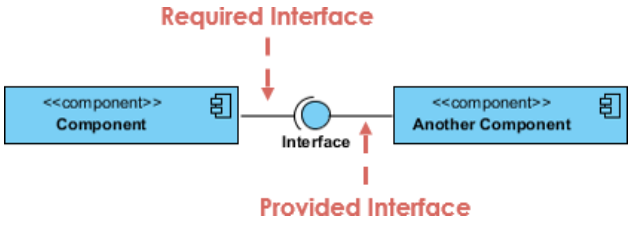


Basic Concepts of Component Diagram

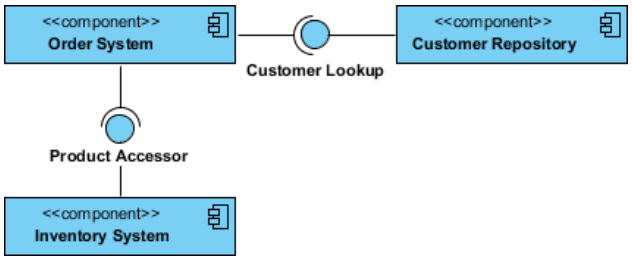
1. A rectangle with the component's name
2. A rectangle with the component icon
3. A rectangle with the stereotype text and/or icon

Interface

In the example below shows two type of component interfaces:



Component Diagram Example - Using Interface (Order System)



Port

Ports are represented using a square along the edge of the system or a component. A port is often used to help expose required and provided interfaces of a component.

Association: _____

- An association specifies a semantic relationship that can occur between typed instances.
- It has at least two ends represented by properties, each of which is connected to the type of the end. More than one end of the association may have the same type.

Composition:

- Composite aggregation is a strong form of aggregation that requires a part instance be included in at most one composite at a time.
- If a composite is deleted, all of its parts are normally deleted with it

---

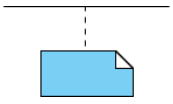
Aggregation

- A kind of association that has one of its end marked shared as kind of aggregation, meaning that it has a shared aggregation.

Dependency

- A dependency is a relationship that signifies that a single or a set of model elements requires other model elements for their specification or implementation.
- This means that the complete semantics of the depending elements is either semantically or structurally dependent on the definition of the supplier element(s)

Constraint



- A condition or restriction expressed in natural language text or in a machine readable language for the purpose of declaring some of the semantics of an element.

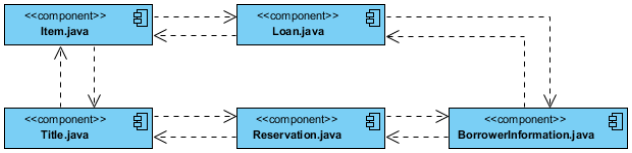
Links:

- A generalization is a taxonomic relationship between a more general classifier and a more specific classifier.
- Each instance of the specific classifier is also an indirect instance of the general classifier.
- Thus, the specific classifier inherits the features of the more general classifie

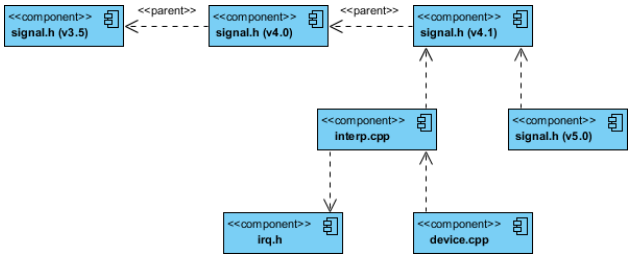
---

Modeling Source Code

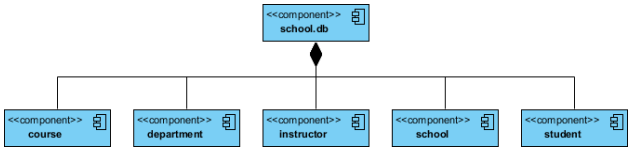
Component Example - Java Source Code



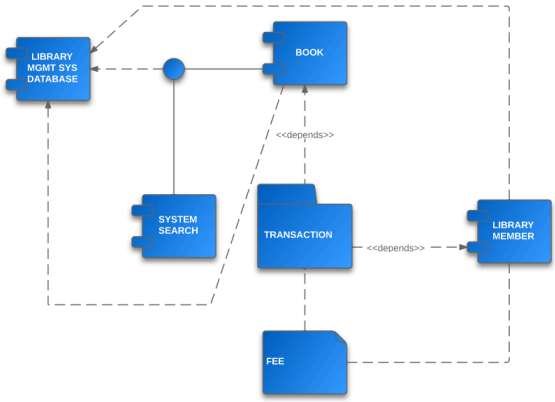
Component Diagram Example - C++ Code with versioning



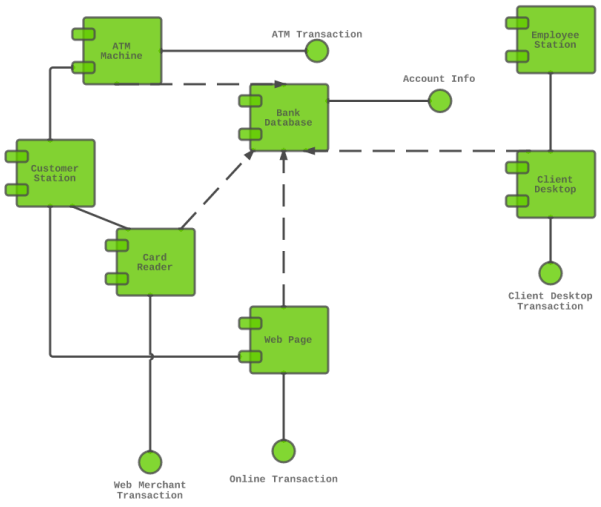
Modeling a Physical Database



Component diagram for a library management system



Component diagram for an ATM system



Provided interfaces:

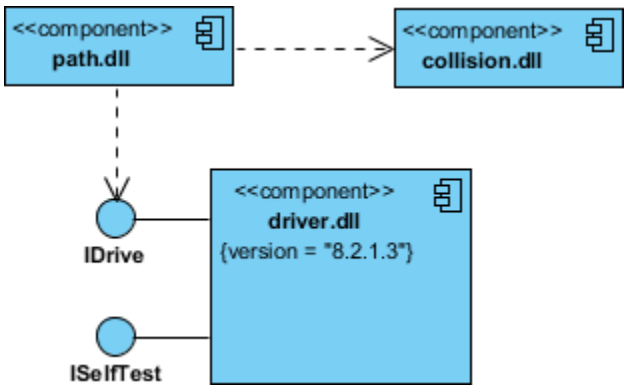
A straight line from the component box with an attached circle. These symbols represent the interfaces where a component produces information used by the required interface of another component.

Required interfaces:

A straight line from the component box with an attached half circle (also represented as a dashed arrow with an open arrow). These symbols represent the interfaces where a component requires information in order to perform its proper function.

Port symbol

Dependency symbol



3-Deployment Diagram

When to Use Deployment Diagram

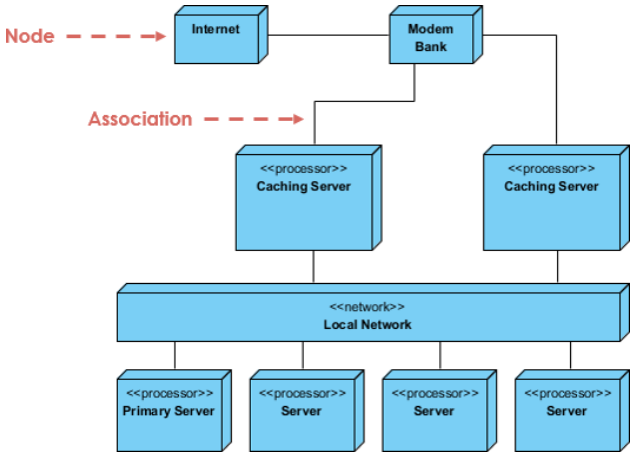
- What existing systems will the newly added system need to interact or integrate with?
- How robust does system need to be (e.g., redundant hardware in case of a system failure)?
- What and who will connect to or interact with system, and how will they do it
- What middleware, including the operating system and communications approaches and protocols, will system use?
- What hardware and software will users directly interact with (PCs, network computers, browsers, etc.)?
- How will you monitor the system once deployed?
- How secure does the system needs to be (needs a firewall, physically secure hardware, etc.)?

Nodes

- 3-D box represents a node, either software or hardware
- HW node can be signified with <<stereotype>>
- Connections between nodes are represented with a line, with optional <<stereotype>>
- Nodes can reside within a node

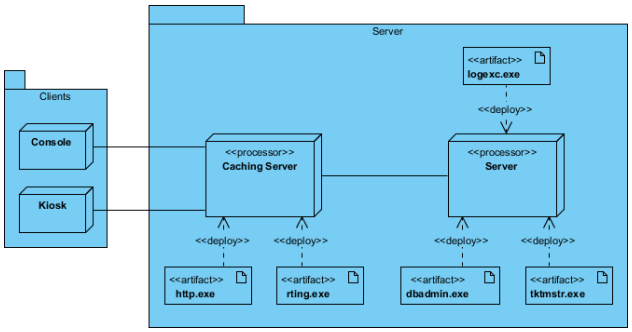
Other Notations

- Dependency
- Association relationships.
- May also contain notes and constraints.

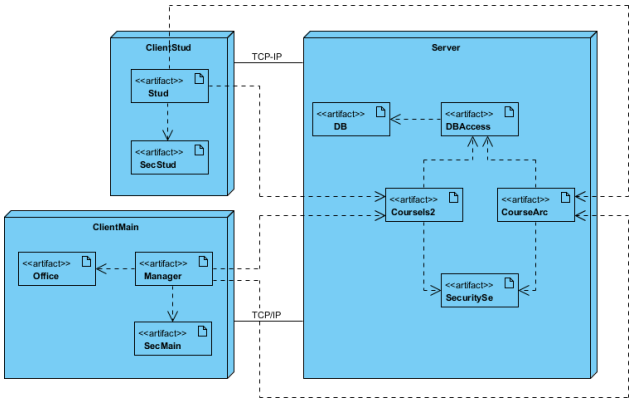


Steps for Modeling a Client/Server System

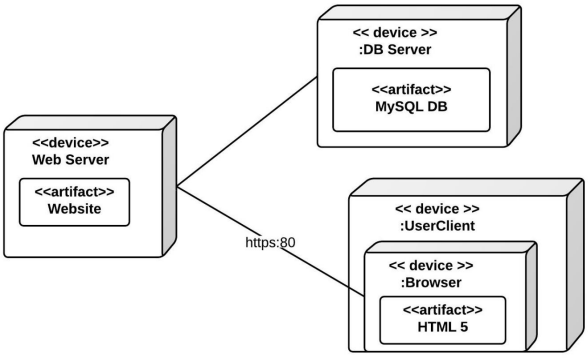
The example shows the topology of a human resources system, which follows a classical client/server architecture.



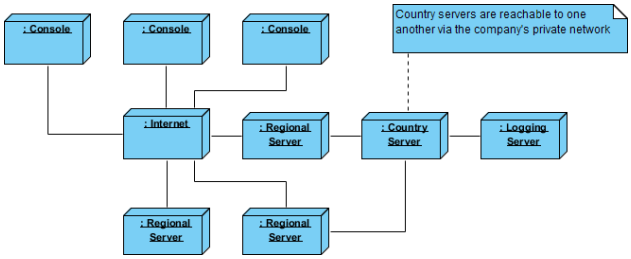
TCP/IP Client / Server Example



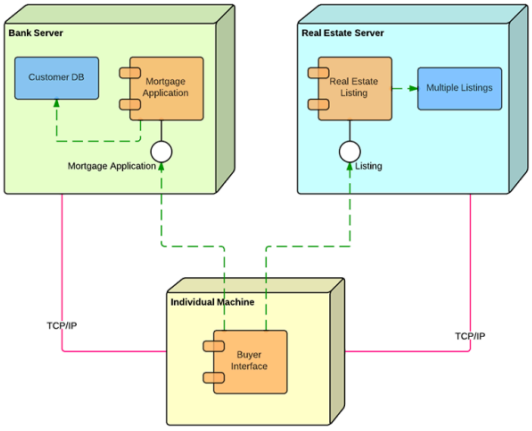
Deployment diagram example



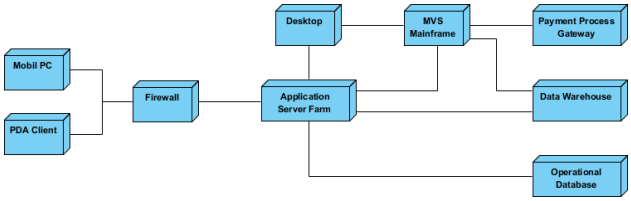
The Example shows the topology of a fully distributed system.



Deployment diagram elements

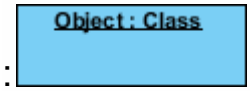


Deployment Diagram Example - Corporate Distributed System



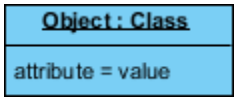
4-Object Diagram

Basic Object Diagram Symbols and Notations



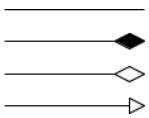
Object Names

- Every object is actually symbolized like a rectangle, that offers the name from the object and its class underlined as well as divided with a colon.



Object Attributes:

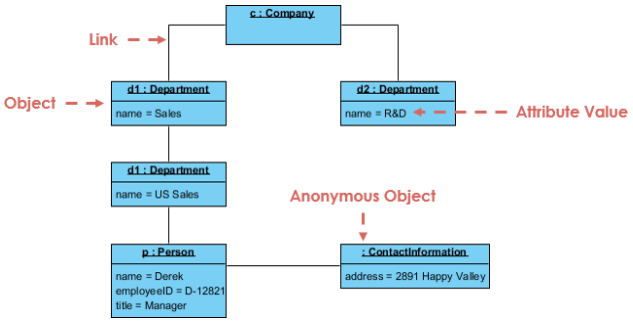
- Similar to classes, you are able to list object attributes inside a separate compartment. However, unlike classes, object attributes should have values assigned for them.



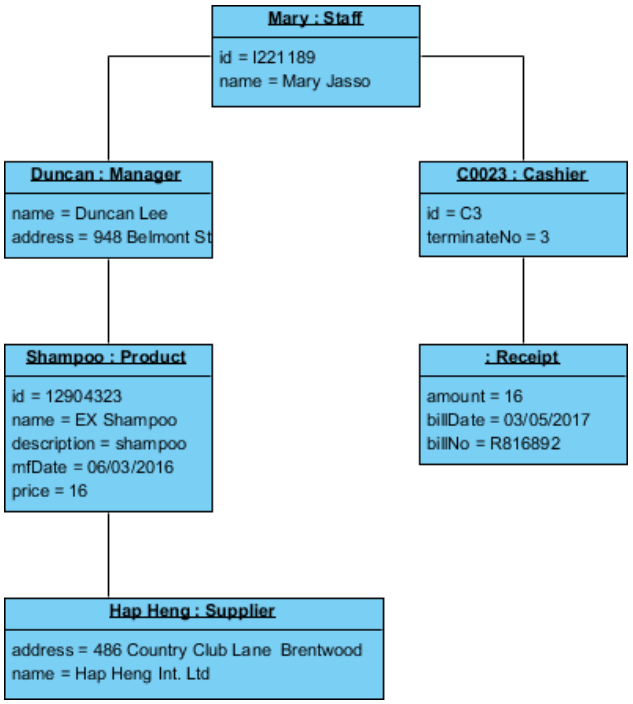
Links:

- Links tend to be instances associated with associations. You can draw a link while using the lines utilized in class diagrams.

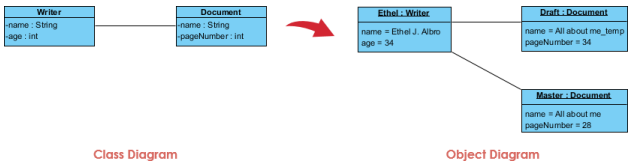
Object Diagram Example I - Company Structure

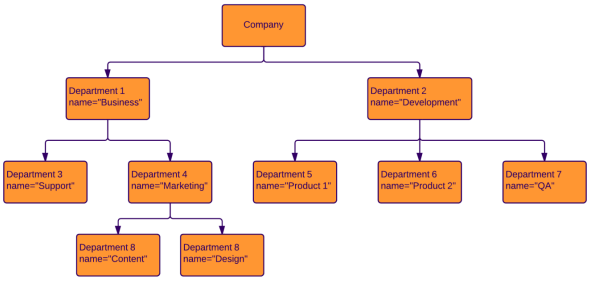


Object Diagram Example II - POS



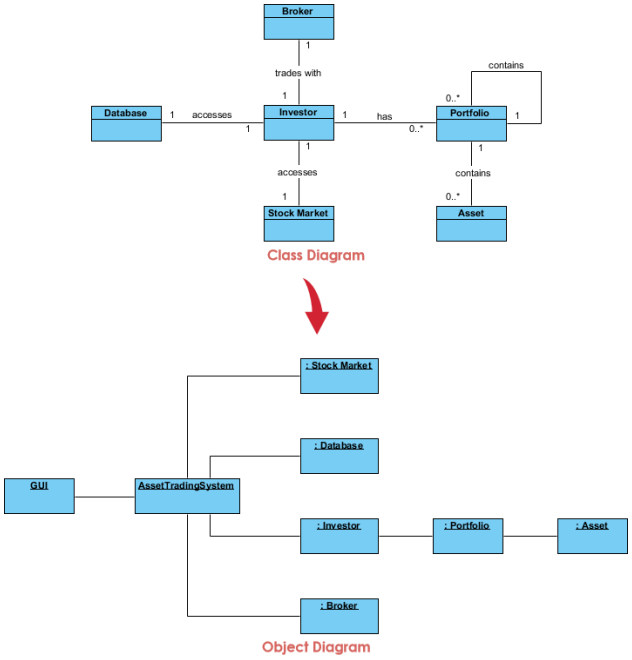
Object Diagram Example III - Writer

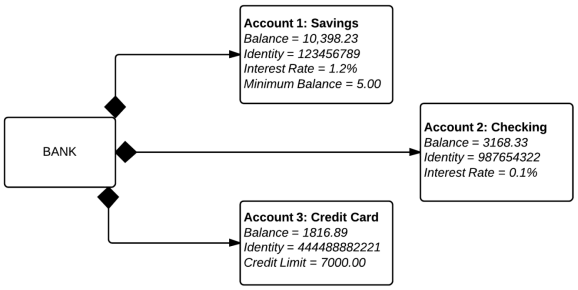


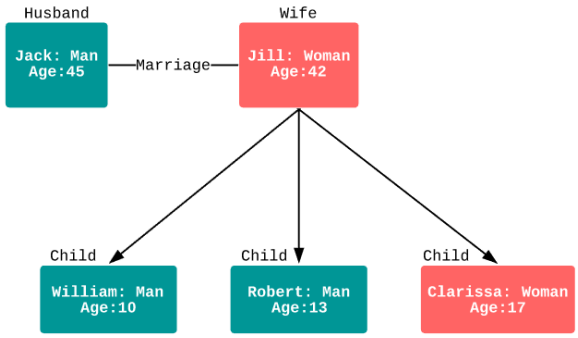


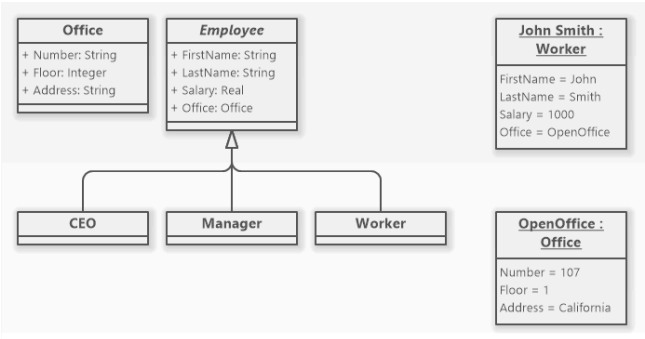
Object Diagram Example V - Deriving an Object Structure Similar to Communication Diagram

:

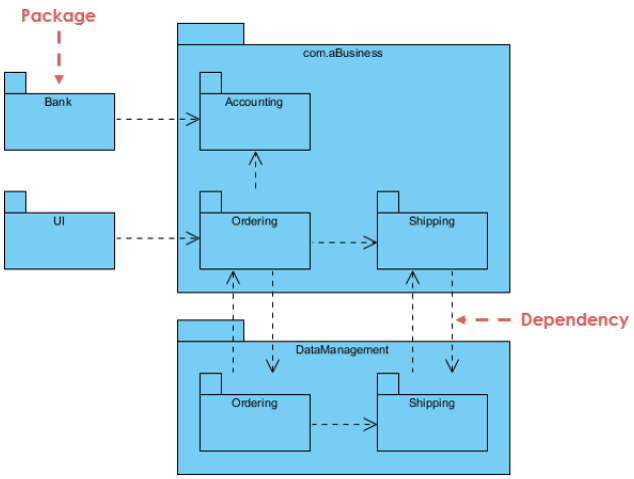




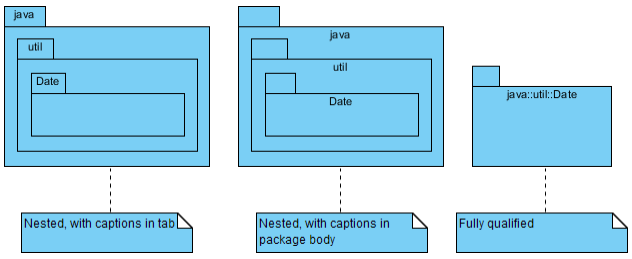




5-Package Diagram



Packages can be represented by the notations with some examples shown below:

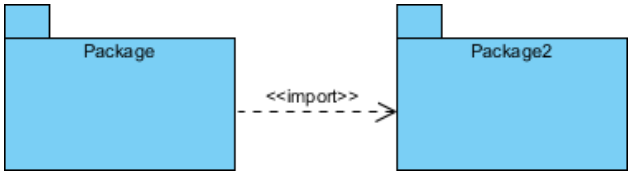


Package Diagram - Dependency Notation

There are two sub-types involved in dependency. They are <<import>> & <<access>>. Though there are two stereotypes users can use their own stereotype to represent the type of dependency between two packages.

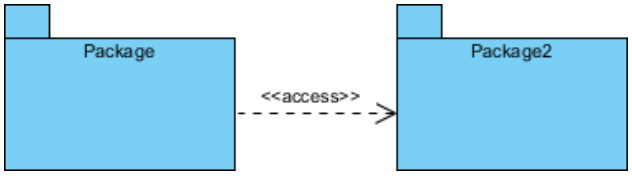
Package Diagram Example - Import

<<import>> - one package imports the functionality of other package



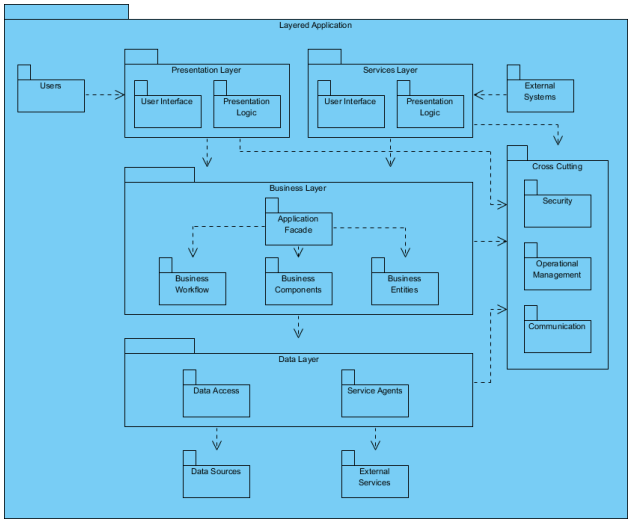
Package Diagram Example - Access

<<access>> - one package requires help from functions of other package.

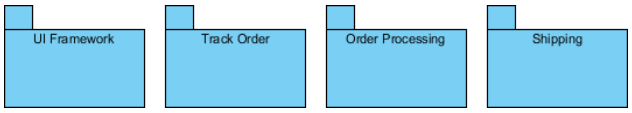


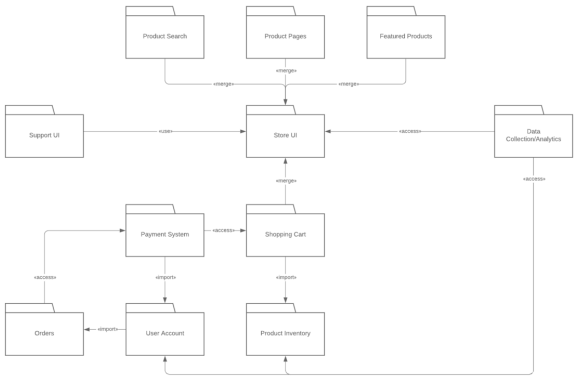
Modeling Complex Grouping

Package Diagram Example - Layering Structure

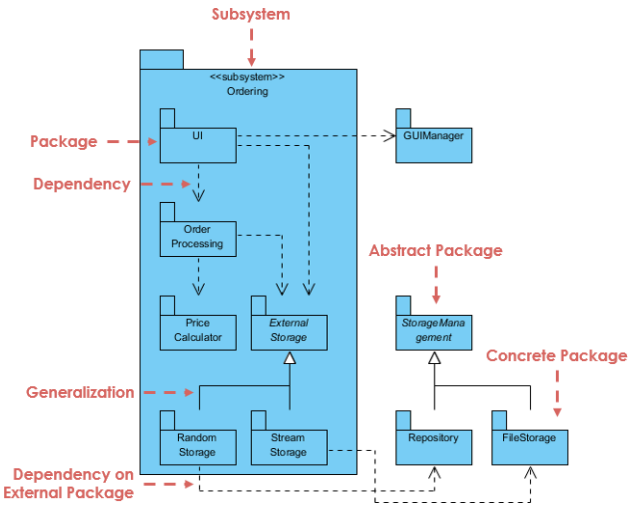


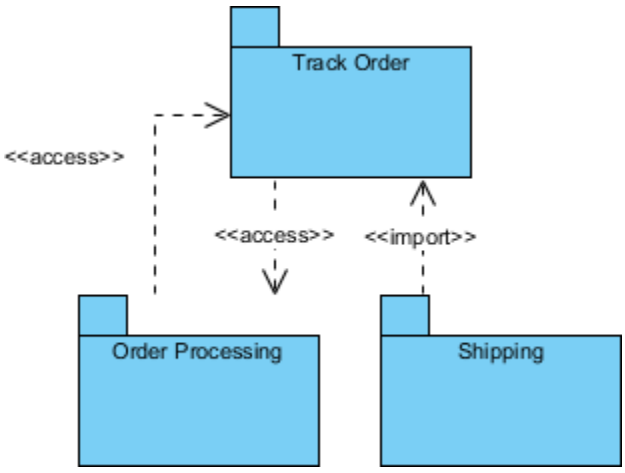
Package Diagram Example - Order Processing System





Package Diagram Example - Order Subsystem





6-Profile Diagram

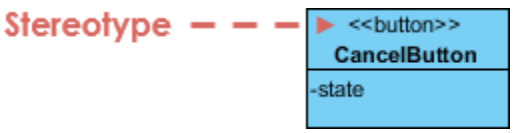
Basic Concepts of Profile Diagram

Profile diagram is basically an extensibility mechanism that allows you to extend and customize UML by adding new building blocks, creating new properties and specifying new semantics in order to make the language suitable to your specific problem domain.
Profile diagram has three types of extensibility mechanisms:

- Stereotypes
- Tagged Values
- Constraints

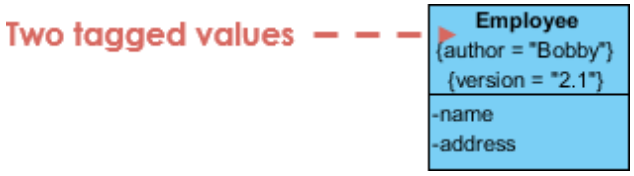
Stereotypes

Stereotypes allow you to increase vocabulary of UML. You can add, create new model elements, derived from existing ones but that have specific properties that are suitable to your problem domain. Stereotypes are used to introduce new building blocks that speak the language of your domain and look primitive. It allows you to introduce new graphical symbols. For example: When modeling a network you might need to have symbols for <<router>>, <<switches>>, <<hub>> etc. A stereotype allows you to make these things appear as primitive.



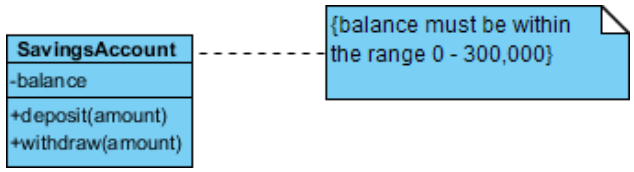
Tagged Values Tagged Value can be useful for adding properties to the model for some useful purposes:

- Code generation
- Version control
- Configuration management
- Authorship
- Etc



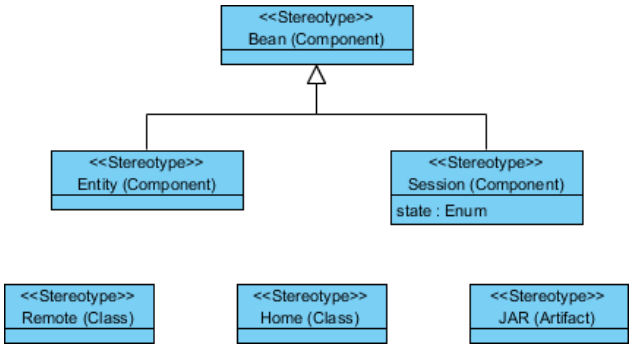
Constraints

For example: In development of a real time system it is necessary to adorn the model with some necessary information such as response time. A constraint defines a relationship between model elements that must be use *{subset}* or *{xor}*. Constraints can be on attributes, derived attributes and associations. It can be attached to one or more model elements shown as a note as well.



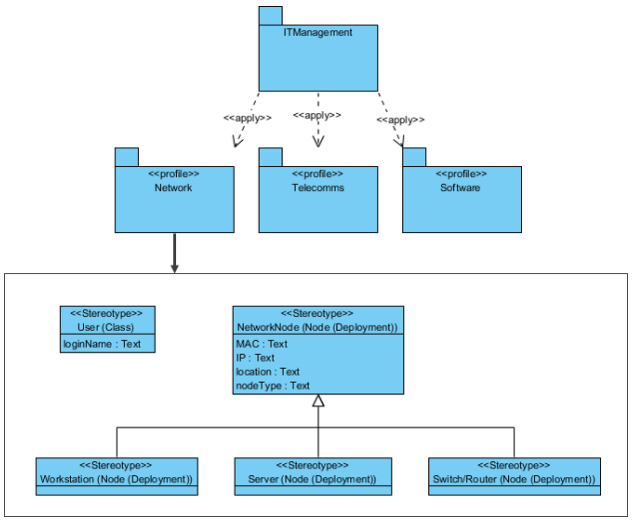
Profile Diagram at a Glance

In the figure below, we define a profile of EJB as a package. The bean itself is extended from component metamodel as an abstract bean. The abstract bean can be concreted as either an Entity Bean or Session Bean. An EJB has two types of remote and home interfaces. An EJB also contains a special kind of artifact called JAR file for storing a collection of Java code.



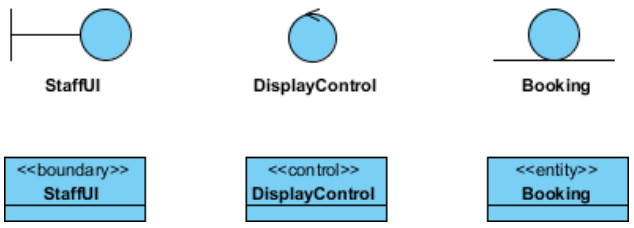
Profile Diagram Example I - IT Management

A profile is applied to another package in order to make the stereotypes in the profile available to that package. The illustration below shows the Network, Telecomms and Software profiles being applied to the ITManagement package.



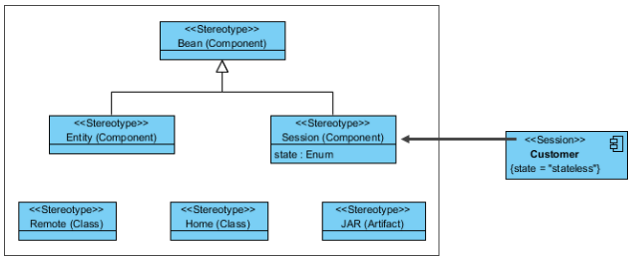
Textual vs Graphic Icon Stereotype

Stereotypes can be in textual or graphical representation. The icon can also replace the normal class box. For Example: People often use these 3 stereotyped class representations to model the software MVC framework:



Profile Diagram Example

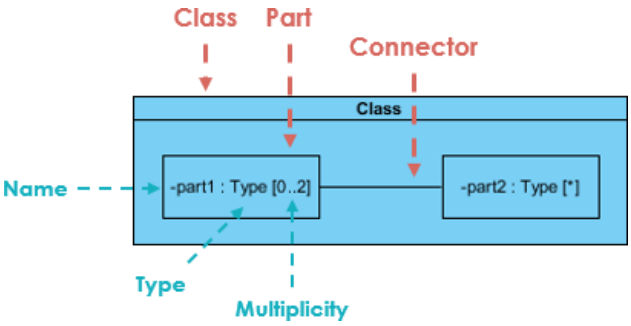
In the figure below, we define a profile of EJB as a package. The bean itself is extended from component metamodel as an abstract bean. The abstract bean can be concreted as either an Entity Bean or Session Bean. An EJB has two types of remote and home interfaces. An EJB also contains a special kind of artifact called JAR file for storing a collection of Java code.



7-Composite Structure Diagram

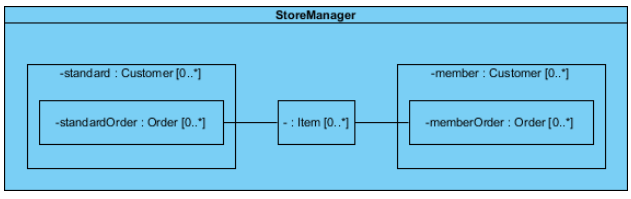
Composite Structure Diagram at a Glance

- Composite Structure Diagrams show the internal parts of a class.
- Parts are named: partName:partType[multiplicity]
- Aggregated classes are parts of a class but parts are not necessarily classes, a part is any element that is used to make up the containing class.



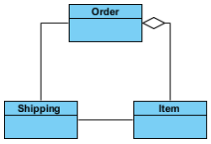
Everything looks like it ends up inside StoreManager,

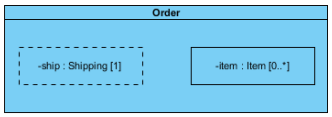
so we can create a composite structure diagram to really see what it's made of.



References to External parts

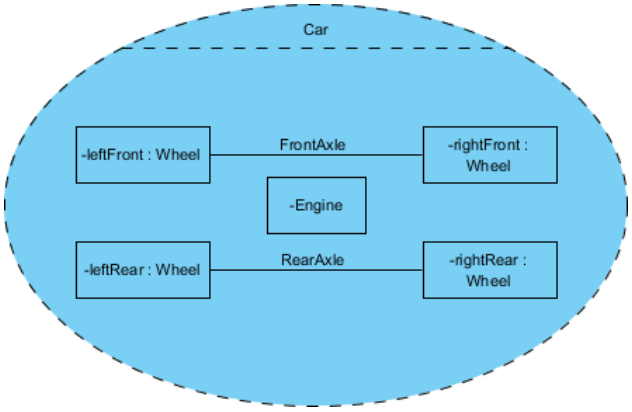
We have seen examples of how Composite Structure diagrams are great at describing aggregation, but your models will also need to contain references to objects outside of the class you are modeling.
But what about the referencing an external object with Composite Structure Diagram like the example below?





Example

In this example the Wheels and the Engine are the Parts of the Collaboration and the FrontAxle and the RearAxle are the Connectors. The Car is the Composite Structure that shows the parts and the connections between the parts.



Parts

A part is a diagram element that represents a set of one or more instances that a containing structured classifier owns. A part describes the role of an instance in a classifier. You can create parts in the structure compartment of a classifier, and in several UML diagrams such as composite structure, class, object, component, deployment, and package diagrams.

Port

A port defines the interaction point between a classifier instance and its environment or between the behavior of the classifier and its internal parts.

Interface

Composite Structure diagram supports the ball-and-socket notation for the provided and required interfaces. Interfaces can be shown or hidden in the diagram as needed.

Connector

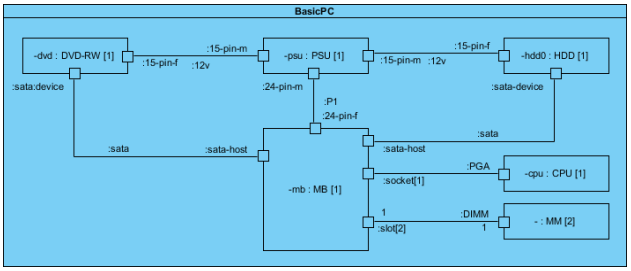
A line that represents a relationship in a model. When you model the internal structure of a classifier, you can use a connector to indicate a link between two or more instances of a part or a port. The connector defines the relationship between the objects or instances that are bound to roles in the same structured classifier and it identifies the communication between those roles. The product automatically specifies the kind of connector to create.

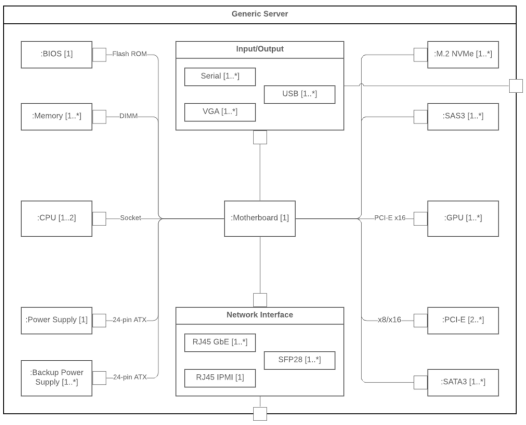
Composite Structure Diagram Example - Computer System

Let's develop the composite structure diagram for a computer system which include the following a list of the components:

- Power Supply Unit (PSU)
- Hard Disk Drive (HDD)
- Mainboard (MB)
- Optical Drive (DVD-RW)
- Memory Module (MM)

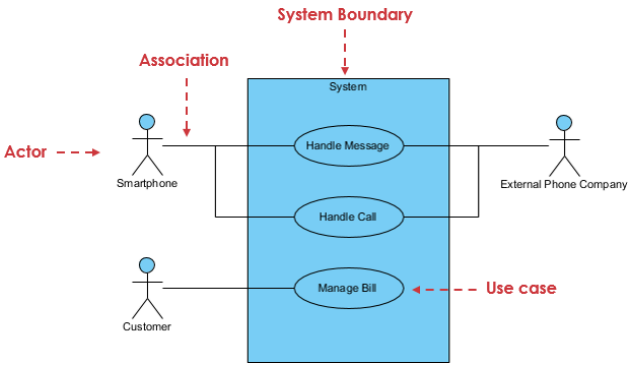
We will assume for the moment that the mainboard is of the type that has a sound card and display adapter built in:



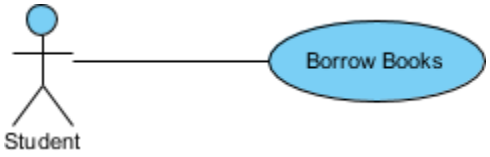


Behavioral Diagrams

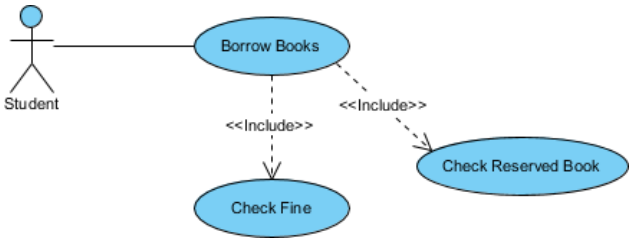
8-Use Case Diagram



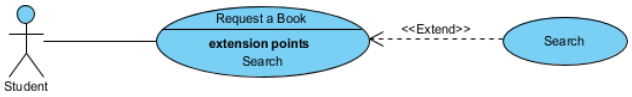
Association Link



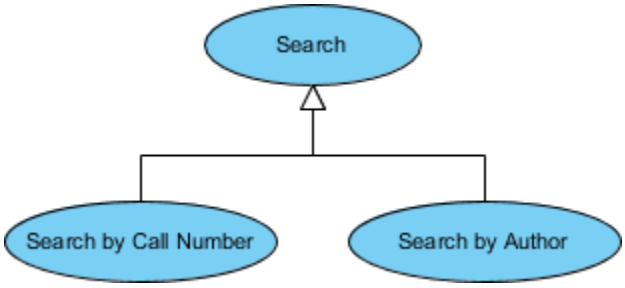
Include Relationship



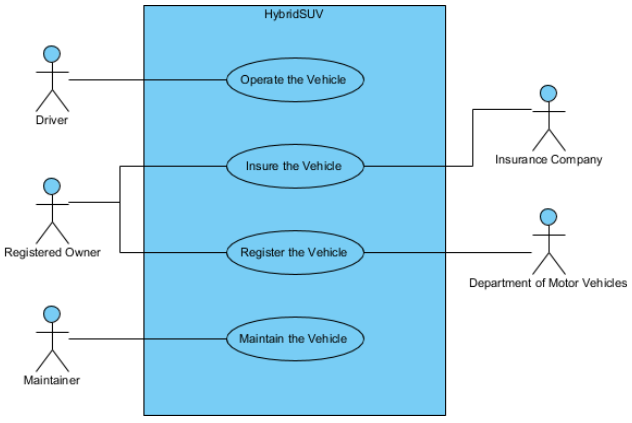
Extend Relationship



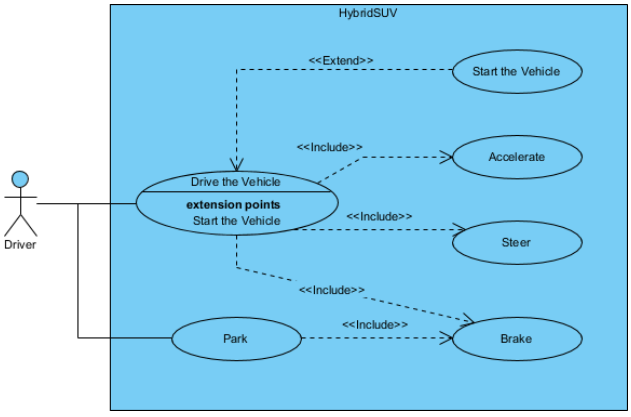
Generalization Relationship



Top Level Use Cases



Operational Use Cases



Actor

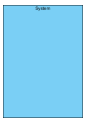
- Someone interacts with the use case (system function).
- Named by a noun.
- Actor plays a role in the business
- Similar to the concept of a user, but a user can play different roles
- For example:
- A prof. can be an instructor and also a researcher
- plays 2 roles with two systems
- Actor triggers use case(s).
- Actor has a responsibility toward the system (inputs), and Actor has expectations from the system (outputs).

Use Case

- System function (process – automated or manual)
- Named by verb + Noun (or Noun Phrase).
- i.e. Do something
- Each Actor must be linked to a use case, while some use cases may not be linked to actors.

Association Link

- The participation of an actor in a use case is shown by connecting an actor to a use case by a solid link.
- Actors may be connected to use cases by associations, indicating that the actor and the use case communicate with one another using messages.
-

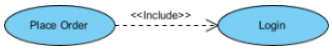


Boundary of system

- The system boundary is potentially the entire system as defined in the requirements document.
- For large and complex systems, each module may be the system boundary.
- For example, for an ERP system for an organization, each of the modules such as personnel, payroll, accounting, etc.
- can form a system boundary for use cases specific to each of these business functions.
- The entire system can span all of these modules depicting the overall system boundary

Extends

- Indicates that an “Invalid Password” use case may include (subject to specified in the extension) the behavior specified by base use case “Login Account”.
- Depict with a directed arrow having a dotted line. The tip of arrowhead points to the base use case and the child use case is connected at the base of the arrow.
- The stereotype “<<extends>>” identifies as an extend relationship



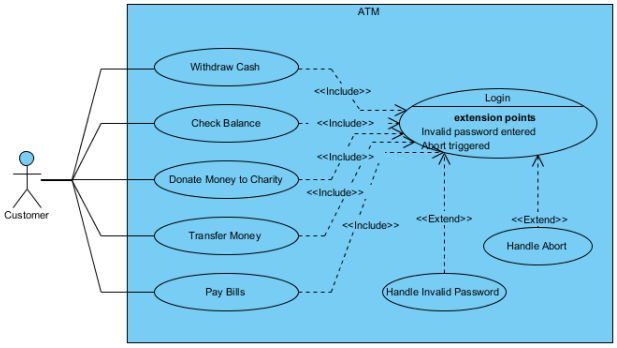
Include

- When a use case is depicted as using the functionality of another use case, the relationship between the use cases is named as include or uses relationship.
- A use case includes the functionality described in another use case as a part of its business process flow.
- A uses relationship from base use case to child use case indicates that an instance of the base use case will include the behavior as specified in the child use case.
- An include relationship is depicted with a directed arrow having a dotted line. The tip of arrowhead points to the child use case and the parent use case connected at the base of the arrow.
- The stereotype “<<include>>” identifies the relationship as an include relationship.

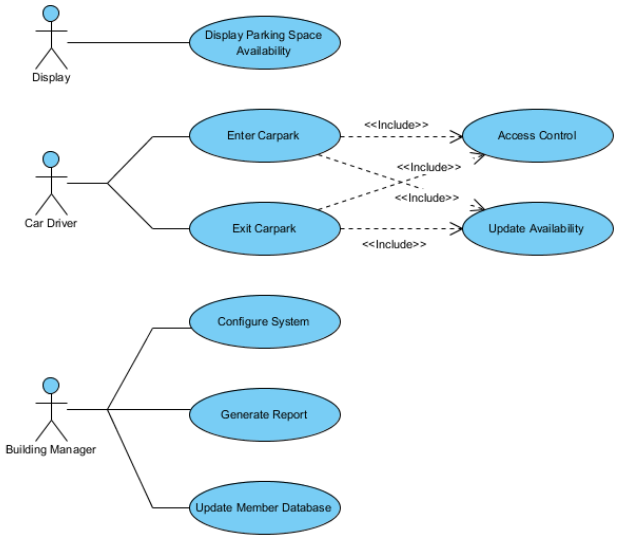
Generalization

- A generalization relationship is a parent-child relationship between use cases.
- The child use case is an enhancement of the parent use case.
- Generalization is shown as a directed arrow with a triangle arrowhead.
- The child use case is connected at the base of the arrow. The tip of the arrow is connected to the parent use case.
-

Use Case Diagram Examples



Carpark system

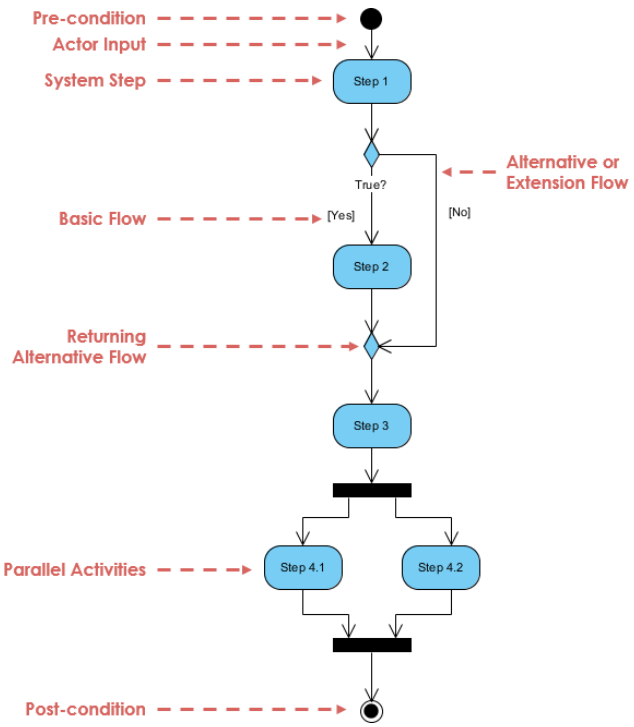


9-Activity Diagram

Activity diagram is another important behavioral diagram in UML diagram to describe dynamic aspects of the system. Activity diagram is essentially an advanced version of flow chart that modeling the flow from one activity to another activity.

Activity Diagram - Learn by Examples

A basic activity diagram - flowchart like



Activity Diagram Notation Summary

Activity

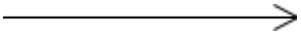
Is used to represent a set of actions

Action

A task to be performed

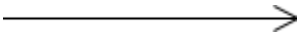
Control Flow

Shows the sequence of execution



Object Flow

Show the flow of an object from one activity (or action) to another activity (or action).



Initial Node

Portrays the beginning of a set of actions or activities

Activity Final Node

Stop all control flows and object flows in an activity (or action)

Object Node

Represent an object that is connected to a set of Object Flows

Decision Node

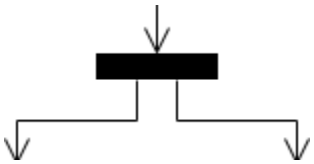
Represent a test condition to ensure that the control flow or object flow only goes down one path

Merge Node

Bring back together different decision paths that were created using a decision-node.

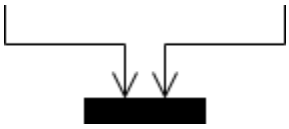
Fork Node

Split behavior into a set of parallel or concurrent flows of activities (or actions)



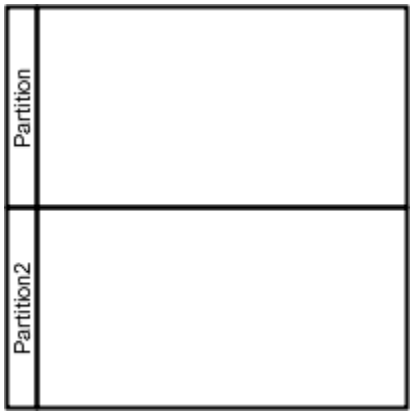
Join Node

Bring back together a set of parallel or concurrent flows of activities (or actions).

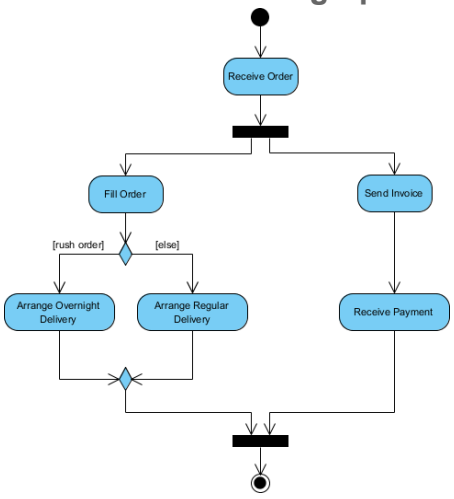


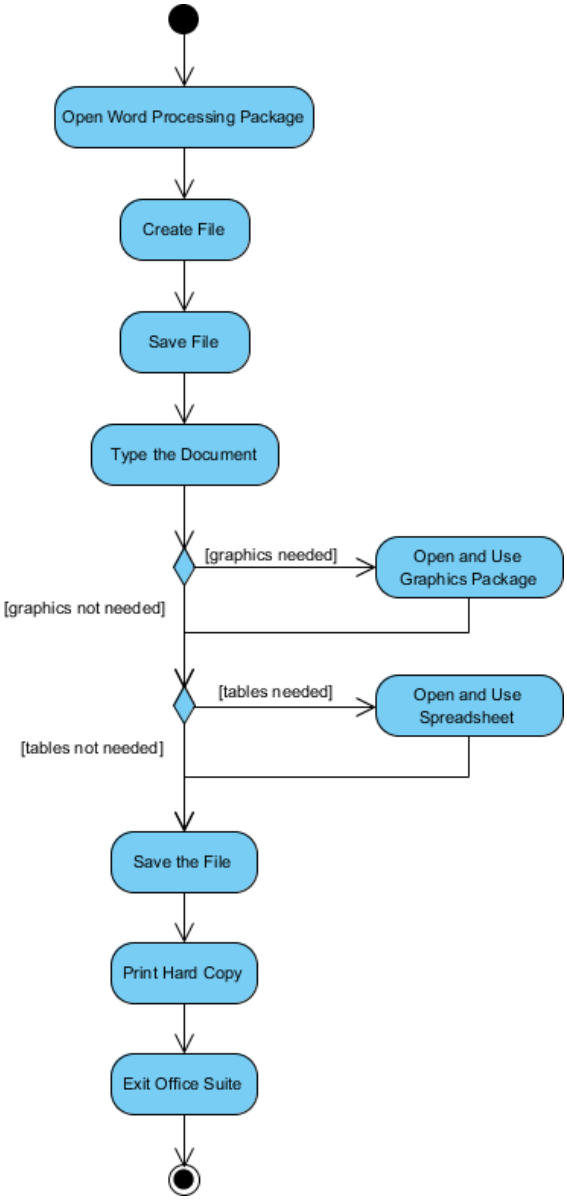
Swimlane and Partition

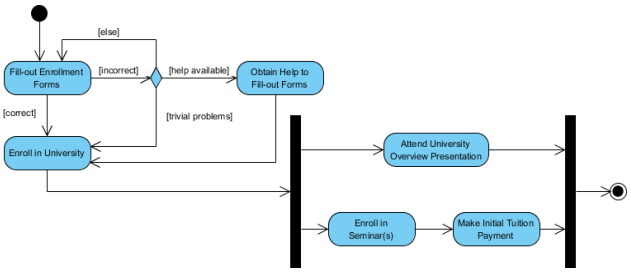
A way to group activities performed by the same actor on an activity diagram or to group activities in a single thread

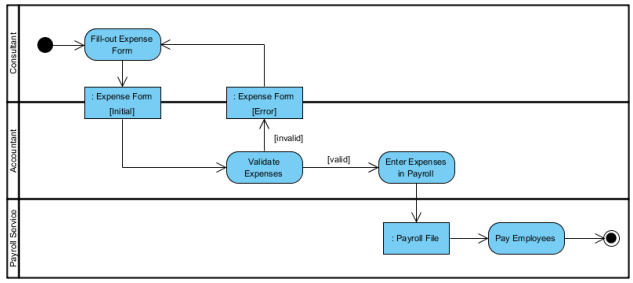


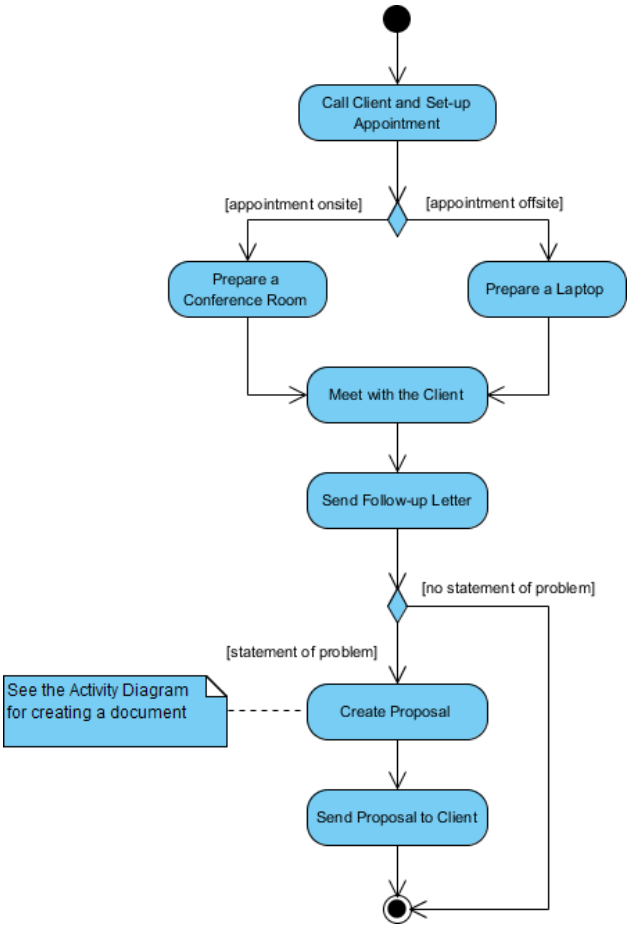
The activity diagram example below visualize the flow in graphical form.




Open Word Processing Package
Create File
Save File
Type the Document
[graphics needed]
Open and Use Graphics Package
[graphics not needed]
[tables needed]
Open and Use Spreadsheet
[tables not needed]
Save the File
Print Hard Copy
Exit Office Suite

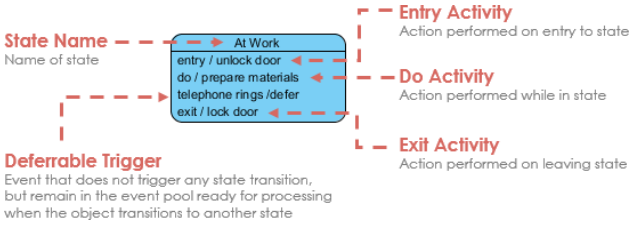

[else]
Fill-out Enrollment Forms
[incorrect]
[help available]
Obtain Help to Fill-out Forms
[correct]
[trivial problems]
Enroll in University
Attend University Overview Presentation
Enroll in Seminar(s)
Make Initial Tuition Payment


Consultant
Accountant
Payroll Service
Fill-out Expense Form
: Expense Form [Initial]
: Expense Form [Error]
[invalid]
Validate Expenses
[valid]
Enter Expenses in Payroll
: Payroll File
Pay Employees


Call Client and Set-up Appointment
[appointment onsite]
[appointment offsite]
Prepare a Conference Room
Prepare a Laptop
Meet with the Client
Send Follow-up Letter
[no statement of problem]
[statement of problem]
See the Activity Diagram for creating a document
Create Proposal
Send Proposal to Client

10-State Machine Diagram

State Notation

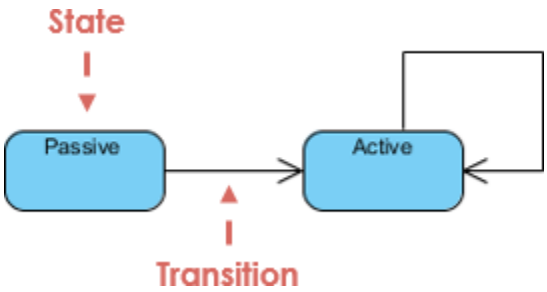


State

A state is a constraint or a situation in the life cycle of an object, in which a constraint holds, the object executes an activity or waits for an event.
A state machine diagram is a graph consisting of:

- States (simple states or composite states)
- State transitions connecting the states

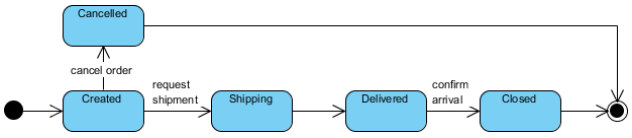
Example:



Initial and Final States

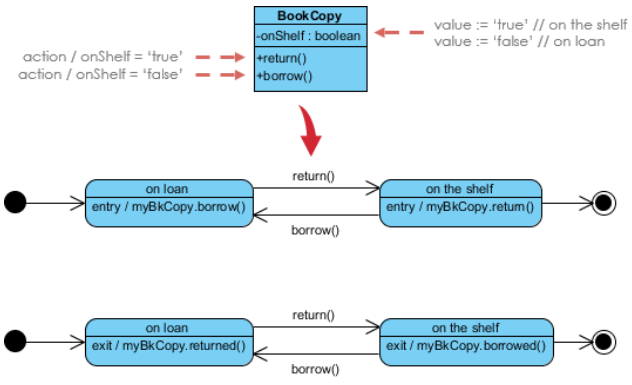
- The **initial state** of a state machine diagram, known as an initial pseudo-state, is indicated with a solid circle. A transition from this state will show the first real state
- The **final state** of a state machine diagram is shown as concentric circles. An open loop state machine represents an object that may terminate before the system terminates, while a closed loop state machine diagram does not have a final state; if it is the case, then the object lives until the entire system terminates.

Example:



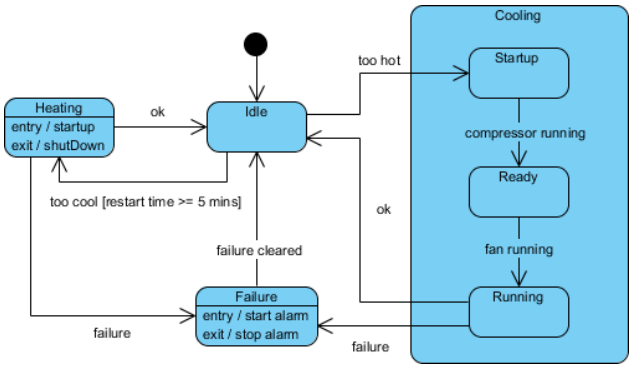
Example - Entry / Exit Action (Check Book Status)

This example illustrates a sta
te machine diagram derived from a Class - "BookCopy":



Substates

A simple state is one which has no substructure. A state which has substates (nested states) is called a composite state. Substates may be nested to any level. A nested state machine may have at most one initial state and one final state. Substates are used to simplify complex flat state machines by showing that some states are only possible within a particular context (the enclosing state).
Substate Example - Heater

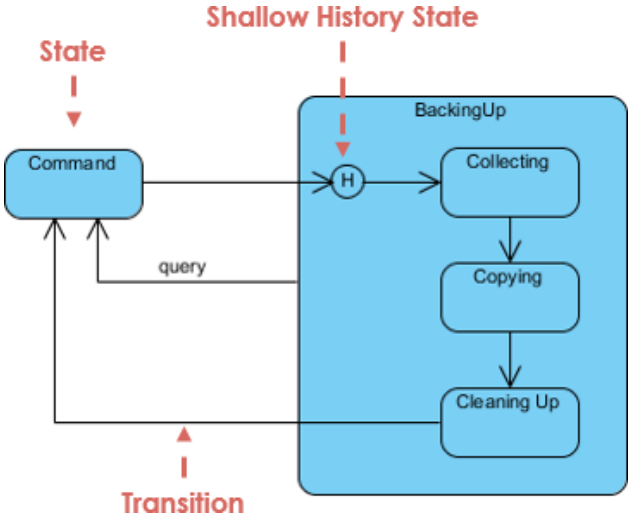


State Machine Diagrams are often used for deriving testing cases, here is a list of possible test ideas:

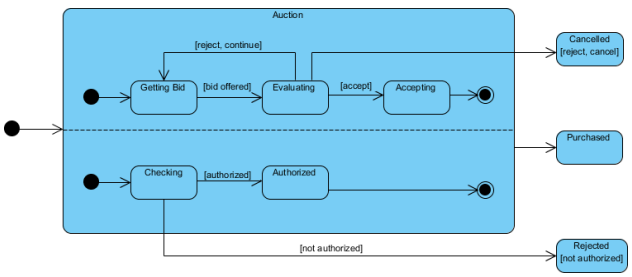
- Idle state receives Too Hot event
- Idle state receives Too Cool event
- Cooling/Startup state receives Compressor Running event
- Cooling/Ready state receives Fan Running event
- Cooling/Running state receives OK event
- Cooling/Running state receives Failure event
- Failure state receives Failure Cleared event
- Heating state receives OK event
- Heating state receives Failure event

History States

Unless otherwise specified, when a transition enters a composite state, the action of **the nested state machine starts over again at the initial state** (unless the transition targets a substate directly). History states allow the state machine to **re-enter the last substate that was active prior to leaving** the composite state. An example of history state usage is presented in the figure below.



In this example, the state machine first entering the Auction requires a fork at the start into two separate start threads. Each substate has an exit state to mark the end of the thread. Unless there is an abnormal exit (Canceled or Rejected), the exit from the composite state occurs when both substates have exited.

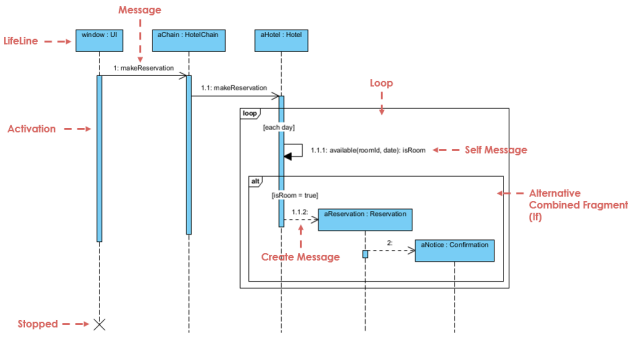


11-Sequence Diagram

Sequence Diagrams captures:

- the interaction that takes place in a collaboration that either realizes a use case or an operation (instance diagrams or generic diagrams)
- high-level interactions between user of the system and the system, between the system and other systems, or between subsystems (sometimes known as system sequence diagrams)

Below is a sequence diagram for making a hotel reservation. The object initiating the sequence of messages is a Reservation window.

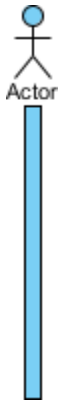


Actor

- a type of role played by an entity that interacts with the subject (e.g., by exchanging signals and data)
- external to the subject (i.e., in the sense that an instance of an actor is not a part of the instance of its corresponding subject).
- represent roles played by human users, external hardware, or other subjects.

Note that:

- An actor does not necessarily represent a specific physical entity but merely a particular role of some entity
- A person may play the role of several different actors and, conversely, a given actor may be played by multiple different person.

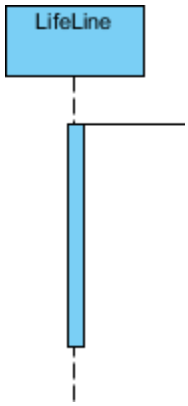


Lifeline

- A lifeline represents an individual participant in the Interaction.

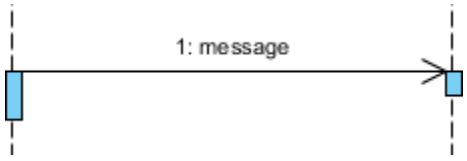
Activations

- A thin rectangle on a lifeline) represents the period during which an element is performing an operation.
- The top and the bottom of the of the rectangle are aligned with the initiation and the completion time respectively



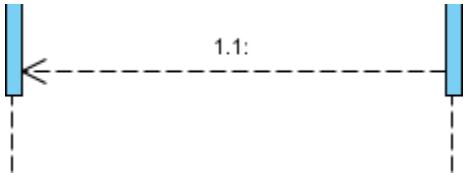
Call Message

- A message defines a particular communication between Lifelines of an Interaction.
- Call message is a kind of message that represents an invocation of operation of target lifeline.



Return Message

- A message defines a particular communication between Lifelines of an Interaction.
- Return message is a kind of message that represents the pass of information back to the caller of a corresponded former message.



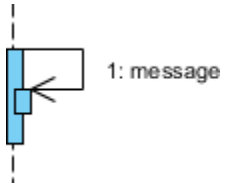
Self Message

- A message defines a particular communication between Lifelines of an Interaction.
- Self message is a kind of message that represents the invocation of message of the same lifeline.

Recursive Message

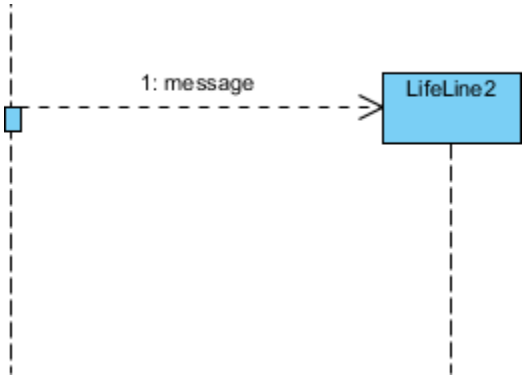
- A message defines a particular communication between Lifelines of an Interaction.
- Recursive message is a kind of message that represents the

invocation of message of the same lifeline. It's target points to an activation on top of the activation where the message was invoked from.



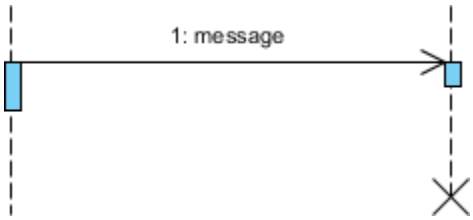
Create Message

- A message defines a particular communication between Lifelines of an Interaction.
- Create message is a kind of message that represents the instantiation of (target) lifeline.



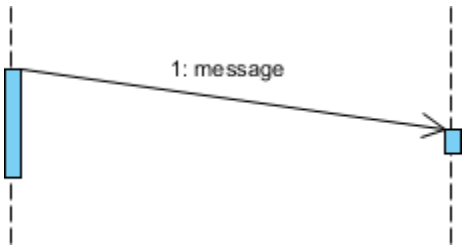
Destroy Message

- A message defines a particular communication between Lifelines of an Interaction.
- Destroy message is a kind of message that represents the request of destroying the lifecycle of target lifeline.



Duration Message

- A message defines a particular communication between Lifelines of an Interaction.
- Duration message shows the distance between two time instants for a message invocation.

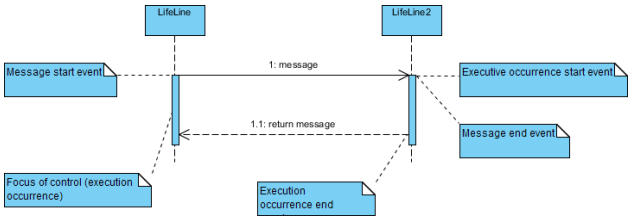


Note

A note (comment) gives the ability to attach various remarks to elements. A comment carries no semantic force, but may contain information that is useful to a modeler.

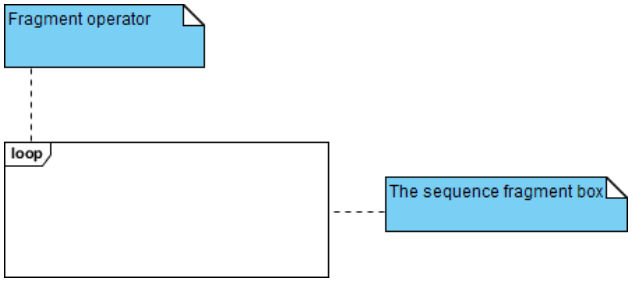
Message and Focus of Control

- An Event is any point in an interaction where something occurs.
- Focus of control: also called execution occurrence, an execution occurrence
- It shows as tall, thin rectangle on a lifeline)
- It represents the period during which an element is performing an operation. The top and the bottom of the rectangle are aligned with the initiation and the completion time respectively.



Sequence Fragments

- UML 2.0 introduces sequence (or interaction) fragments. Sequence fragments make it easier to create and maintain accurate sequence diagrams
- A sequence fragment is represented as a box, called a combined fragment, which encloses a portion of the interactions within a sequence diagram
- The fragment operator (in the top left cornet) indicates the type of fragment
- Fragment types: ref, assert, loop, break, alt, opt, neg

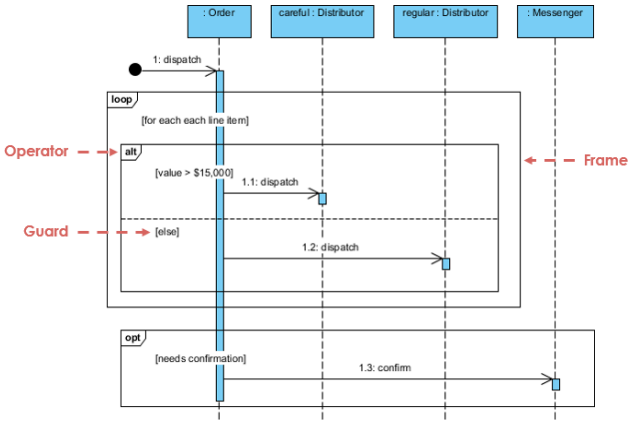


Operator	Fragment Type
alt	Alternative multiple fragments: only the one whose condition is true will execute.
opt	Optional: the fragment executes only if the supplied condition is true. Equivalent to an alt only with one trace.
par	Parallel: each fragment is run in parallel.
loop	Loop: the fragment may execute multiple times, and the guard indicates the basis of iteration.
region	Critical region: the fragment can have only one thread executing it at once.
neg	Negative: the fragment shows an invalid interaction.
ref	Reference: refers to an interaction defined on another diagram. The frame is drawn to cover the lifelines involved in the interaction. You can define parameters and a return value.
sd	Sequence diagram: used to surround an entire sequence diagram.

Note That:

- It is possible to combine frames in order to capture, e.g., loops or branches.
- **Combined fragment** keywords: alt, opt, break, par, seq, strict, neg, critical, ignore, consider, assert and loop.
- Constraints are usually used to show timing constraints on messages. They can apply to the timing of one message or intervals between messages.

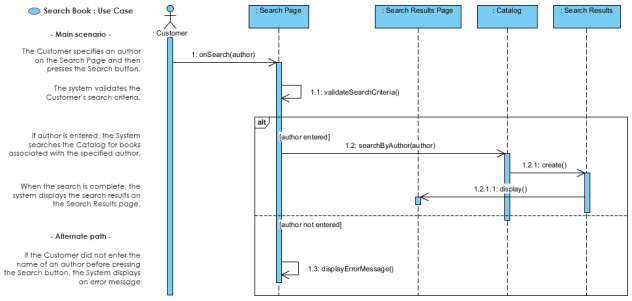
Combined Fragment Example



Sequence Diagram for Modeling Use Case Scenarios

User requirements are captured as use cases that are refined into scenarios. A use case is a collection of interactions between external actors and a system. In UML, a use case is:
"the specification of a sequence of actions, including variants, that a system (or entity) can perform, interacting with actors of the system."

A scenario is one path or flow through a use case that describes a sequence of events that occurs during one particular execution of a system which is often represented by a sequence diagram.

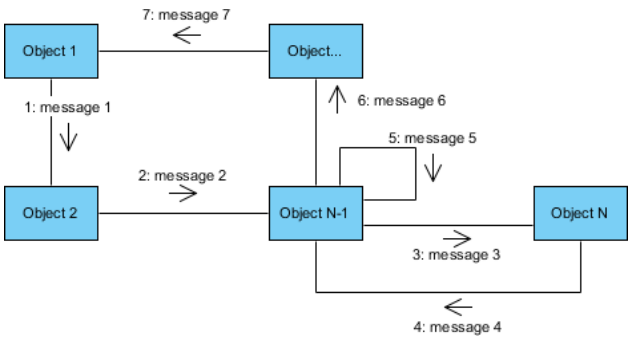


12-Communication Diagram

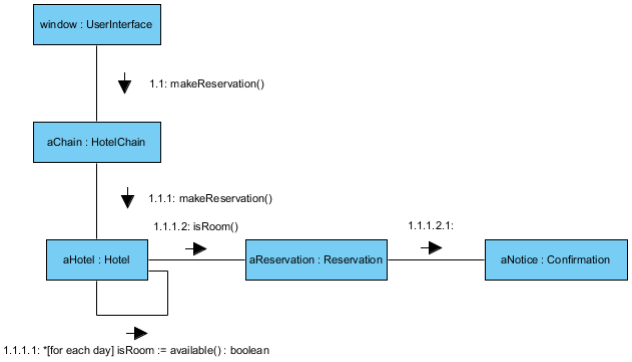
Communication Diagram at a Glance

In the example of the notation for a communication diagram, objects (actors in use cases) are represented by rectangles. In the example (generic communication diagram):

- The objects are Object1, Object2, Object..., ObjectN-1 ..., and ObjectN.
- Messages passed between objects are represented by labeled arrows that start with the sending object (actor) and end with the receiving object.
- The sample messages passed between objects are labeled 1: message1, 2: message2, 3: message3, etc., where the numerical prefix to the message name indicates its order in the sequence.
- Object1 first sends Object2 the message message1, Object2 in turn sends ObjectN-1 the message message2, and so on.
- Messages that objects send to themselves are indicated as loops (e.g., message message5).

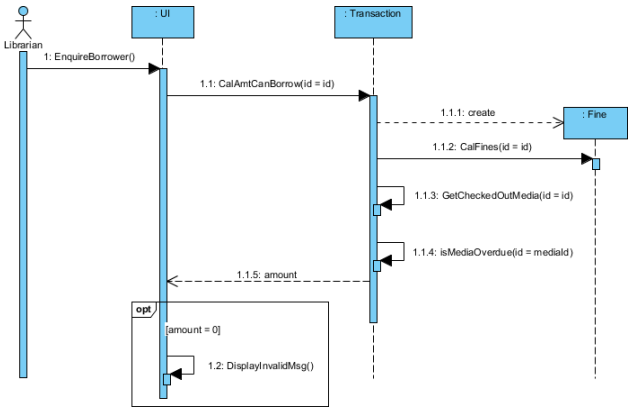


Communication Example - Hotel Reservation

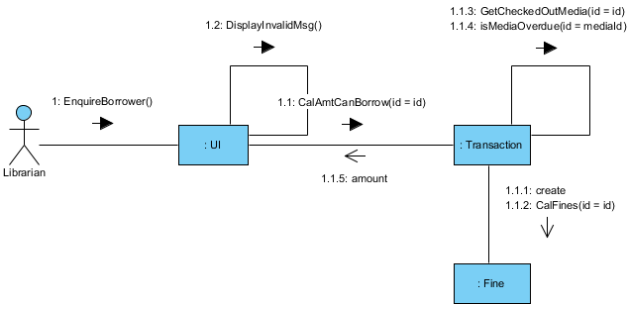


- Each message in a communication diagram has a sequence number.
- The top-level message is numbered 1.
- Messages sent during the same call have the same decimal prefix, but suffixes of 1, 2, etc. according to when they occur.

Example - Sequence diagram vs Communication (Library Item Overdue)



If you open this sequence diagram in Visual Paradigm you can automatically generate the communication diagram shown in figure below:

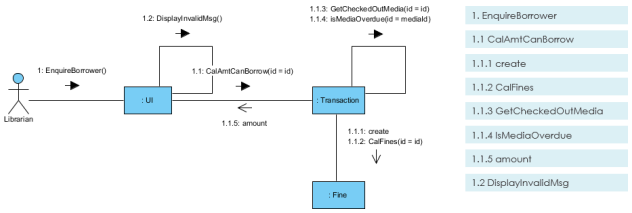


Note: If you compare the two diagrams, you'll see they both contain objects and messages. It also becomes clear that it's much easier to determine the time ordering of messages by looking at the sequence diagram and it's easier to see the relationships between objects by looking at the communication diagram.

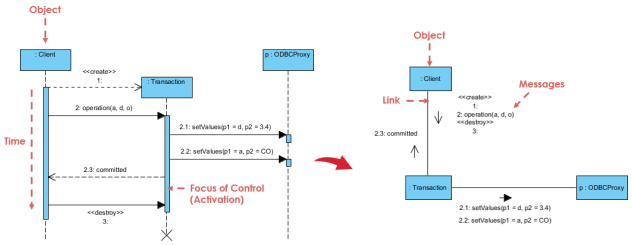
Messages in communication diagrams are shown as arrows pointing from the Client object to the Supplier object. Typically, messages represent a client invoking an operation on a supplier object. They can be modeled along with the objects in the following manner:

- Message icons have one or more messages associated with them.
- Messages are composed of message text prefixed by a sequence number.
- This sequence number indicates the time-ordering of the message.

For example, in the communication diagram in the figure below, you can follow the sequence numbers to determine the order of messages between objects:



Example - From Sequence Diagram to Communication Diagram



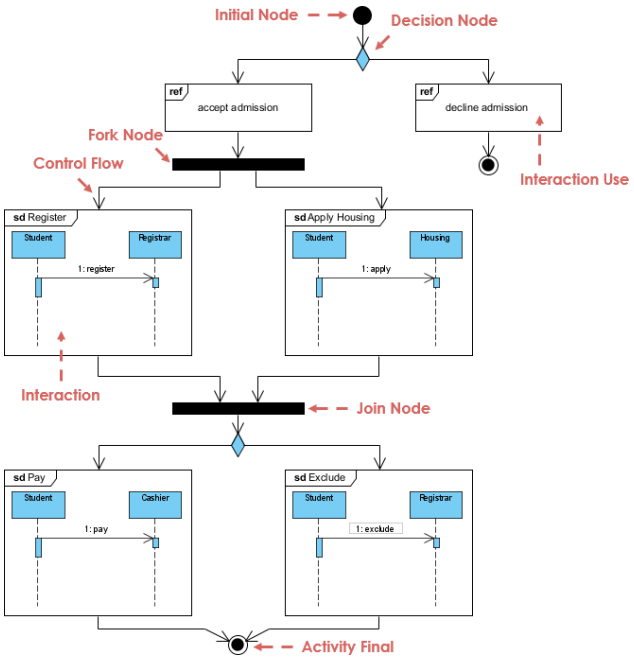
Note that:

- Focus of control: also called execution occurrence/activation. It shows as tall, thin rectangle on a lifeline that represents the period during which an element is performing an operation.
- The top and the bottom of the rectangle are aligned with the initiation and the completion time respectively.
- In communication diagram focus of control is explicit and thus, can be represented by the message nest numbering.

13-Interaction Overview Diagram

Interaction Overview Diagram at a Glance

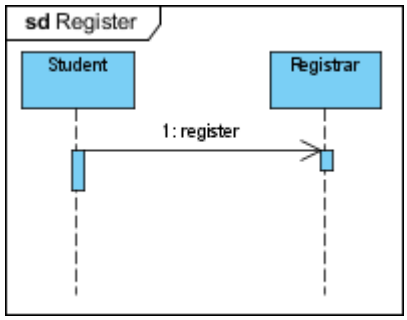
Interaction Overview Diagram is one of the fourteen types of diagrams of the Unified Modeling Language (UML), which can picture a control flow with nodes that can contain interaction diagrams which show how a set of fragments might be initiated in various scenarios. Interaction overview diagrams focus on the overview of the flow of control where the nodes are interactions (sd) or interaction use (ref).
The other notation elements for interaction overview diagrams are the same as for activity and sequence diagrams. These include initial, final, decision, merge, fork and join nodes.



**extracted from UML 2.0 Reference Manual*
The example above shows a student who has been accepted into a university. First the student must be accept or decline admission. After accepting, the student must both register for classes and apply for housing. After both of those are complete, the student must pay the registrar. If payment is not received in time the student is excluded by the registrar.

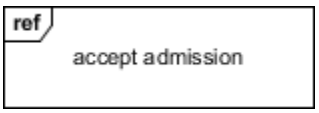
Interaction

An Interaction diagram of any kind may appear inline as an Activity Invocation



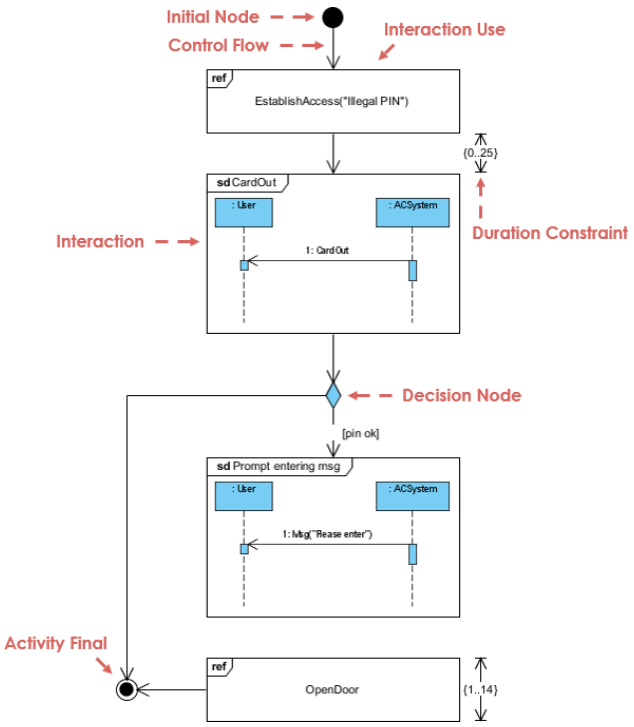
Interaction Use

Large and complex sequence diagrams could be simplified with interaction uses. It is also common to reuse some interaction between several other interactions.

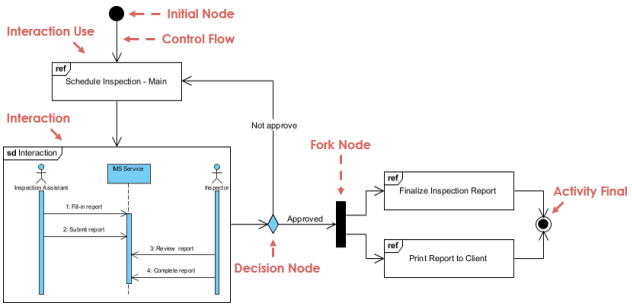


Interaction Diagram Example - Access Control System

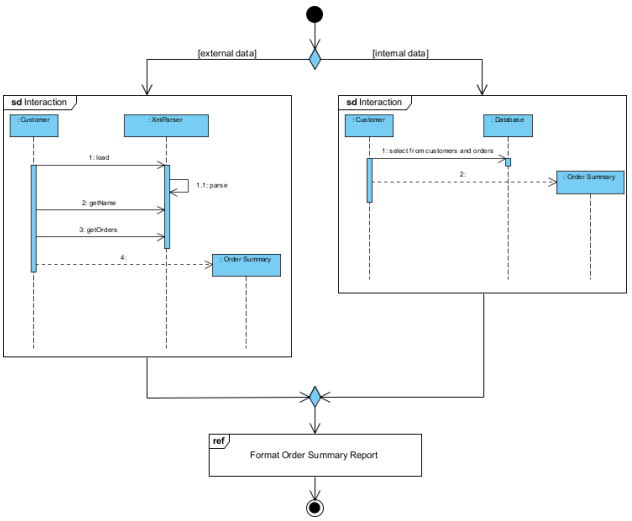
- The Interaction EstablishAccess occurs first with argument"Illegal PIN" followed by an interaction with the message CardOut which is shown in an inline Interaction.
- Then there is an alternative as we find a decision node with an InteractionConstraint on one of the branches.
- Along that control flow we find another inline Interaction and an InteractionUse in the sequence.



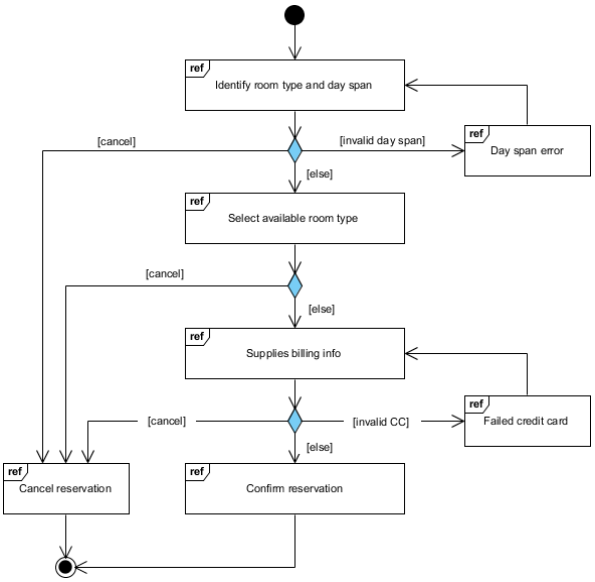
Interaction Diagram Example - Scheduling System



Interaction Diagram Example - Order Reporting



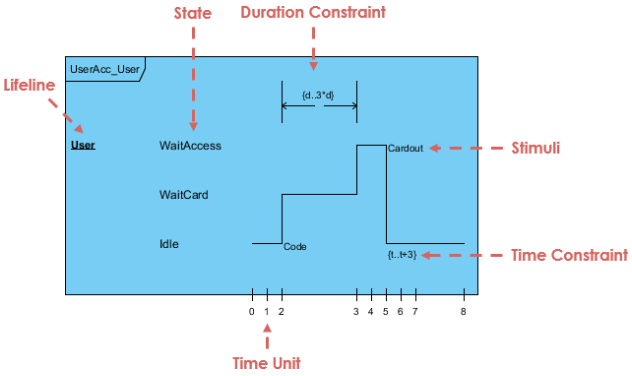
Interaction Diagram Example - Room Reservation



14-Timing Diagram

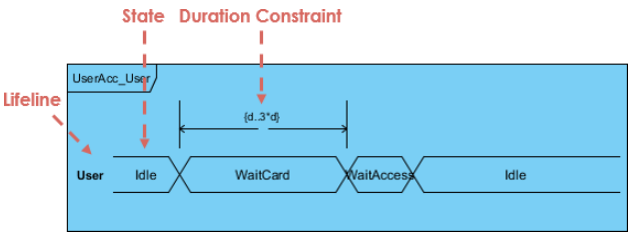
State Timeline Representation

Changes from one state to another are represented by a change in the level of the lifeline. For the period of time when the object is a given state, the timeline runs parallel to that state. A change in state appears as a vertical change from one level to another. The cause of the change, as is the case in a state or sequence diagram, is the receipt of a message, an event that causes a change, a condition within the system, or even just the passage of time.



Value lifeline Representation

The figure below shows an alternative notation of UML Timing diagram. It shows the state of the object between two horizontal lines that cross with each other each time the state changes.

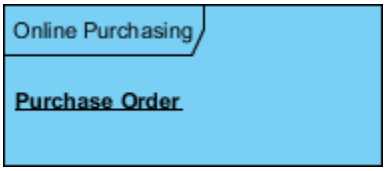


Basic Concepts of Timing Diagrams

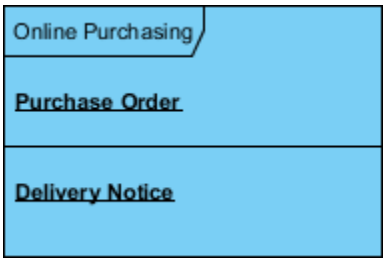
Major elements of timing UML diagram - lifeline, timeline, state or condition, message, duration constraint, timing ruler.

Lifeline

A lifeline in a Timing diagram forms a rectangular space within the content area of a frame. Lifeline is a named element which represents an individual participant in the interaction. It is typically aligned horizontally to read from left to right.

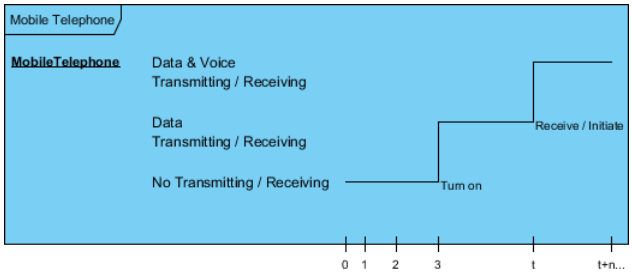


Multiple lifelines may be stacked within the same frame to model the interaction between them.



State Timeline in Timing Diagram

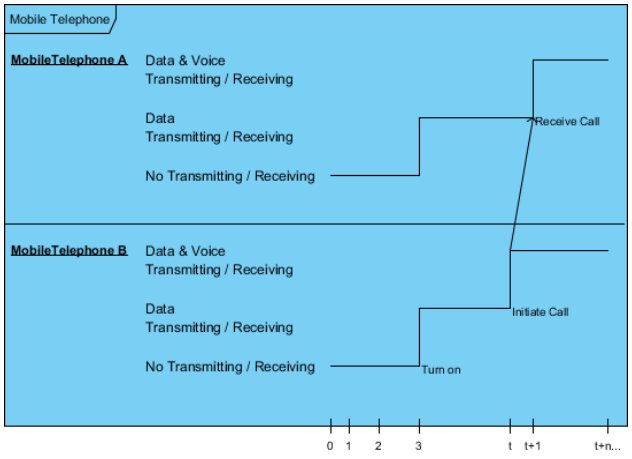
A state or condition timeline represents the set of valid states and time. The states are stacked on the left margin of the lifeline from top to bottom.



The cause of the change, as is the case in a state or sequence diagram, is the receipt of a message, an event that causes a change, a condition within the system, or even just the passage of time.

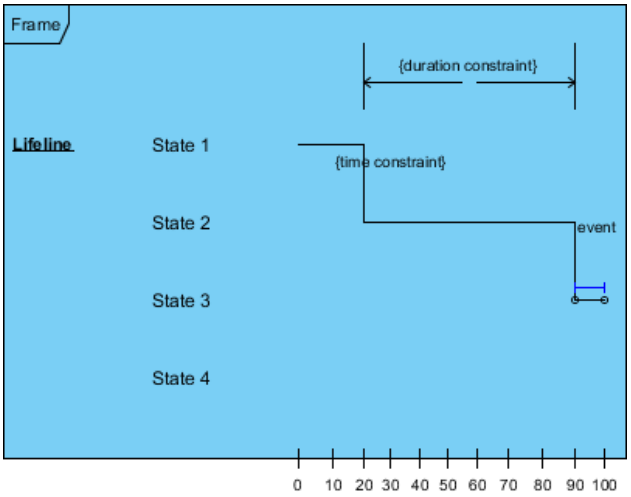
Multiple Compartments

It is possible to stack several life lines of different objects in the same timing diagram. One life line above the other. Messages sent from one object to another can be depicted using simple arrows. The start and the end points of each arrow indicate when each message was sent and when it was received.



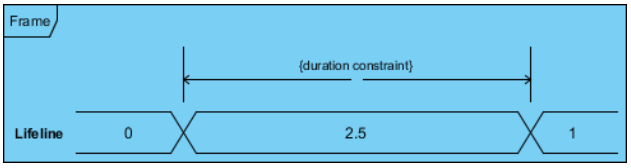
State Lifeline

A state lifeline shows the change of state of an item over time. The X-axis displays elapsed time in whatever units are chosen while the Y-axis is labelled with a given list of states. A state lifeline is shown below:



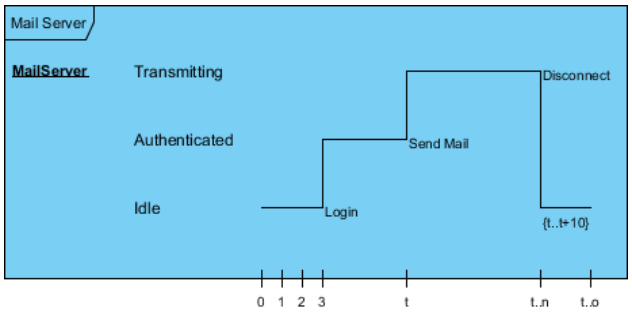
Value Lifeline

A value lifeline shows the change of value of an item over time. The X-axis displays elapsed time in whatever units are chosen, the same as for the state lifeline. The value is shown between the pair of horizontal lines which crosses over at each change in value.



Timeline and Constraints

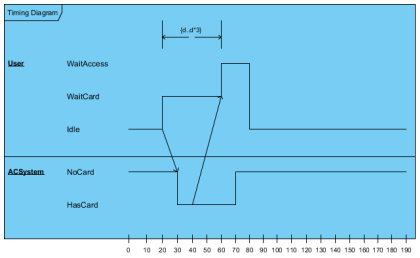
We can use the length of a timeline to indicate how long the object remains in a particular state by reading it from left to right. To associate time measurements, you show tick marks online the bottom part of the frame.
The example below shows that the Login event is received three time units after the start of the sequence. To show relative times, you can mark a specific instance in time using a variable name. The figure marks the time the sendMail event is received as time

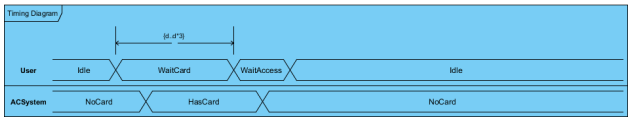


You can use relative time marks in constraints to indicate that a message must be received within a specified amount of time.

State and Value Lifeline Side-by-Side

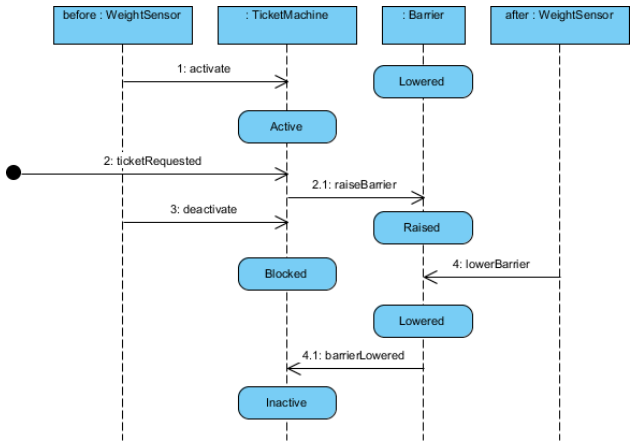
State and Value Lifelines can be put one after the other in any combination. Messages can be passed from one lifeline to another. Each state or value transition can have a defined event, a time constraint which indicates when an event must occur, and a duration constraint which indicates how long a state or value must be in effect for.





Model Consistency among Interaction Diagrams

Timing diagram should always be consistent with the relevant sequence diagram and state machine. To do this 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.



The figure above shows a sequence diagarm of the car park example, while the figure below presents the corresponding timing diagram of the car park example. The various parts of the timing diagram refer to the content of sequence diagram.