UML Tutorial

The Unified Modeling Language has quickly become the de-facto standard for building Object-Oriented software.

The **OMG** specification states: *"The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components."*

The important point to note here is that UML is a 'language' for specifying and not a method or procedure. The UML is used to define a software system; to detail the artifacts in the system, to document and construct - it is the language that the blueprint is written in. The UML may be used in a variety of ways to support a software development methodology' but in itself it does not specify that methodology or process.

UML defines the notation and semantics for the following domains:

- The User Interaction or Use Case Model - describes the boundary and interaction between the system and users. Corresponds in some respects to a requirements model.
- The Interaction or Communication Model - describes how objects in the system will interact with each other to get work done.
- The State or Dynamic Model - State charts describe the states or conditions that classes assume over time. Activity graphs describe the workflow's the system will implement.
- The Logical or Class Model - describes the classes and objects that will make up the system.
- The Physical Component Model - describes the software (and sometimes hardware components) that make up the system.
- The Physical Deployment Model - describes the physical architecture and the deployment of components on that hardware architecture.

UML 2.0

UML 2 builds on the already highly successfull UML 1.x standard, which has become an industry standard for modeling, design and construction of software systems as well as more generalized business and scientific processes. UML 2 defines 13 basic diagram types, divided into two general sets:

1. Structural Modeling Diagrams

Structure diagrams define the static architecture of a model. They are used to model the 'things' that make up a model - the classes, objects, interfaces and physical components. In addition they are used to model the relationships and dependencies between elements.

- Package diagrams are used to divide the model into logical containers or 'packages' and describe the interactions between them at a high level
- Class or Structural diagrams define the basic building blocks of a model: the types, classes and general materials that are used to construct a full model
- Object diagrams show how instances of structural elements are related and used at run-time.
- Composite Structure diagrams provide a means of layering an element's structure and focusing on inner detail, construction and relationships
- Component diagrams are used to model higher level or more complex structures, usually built up from one or more classes, and providing a well defined interface
- Deployment diagrams show the physical disposition of significant artefacts within a real-world setting.

2. Behavioral Modeling Diagrams

Behavior diagrams capture the varieties of interaction and instantaneous state within a model as it 'executes' over time.

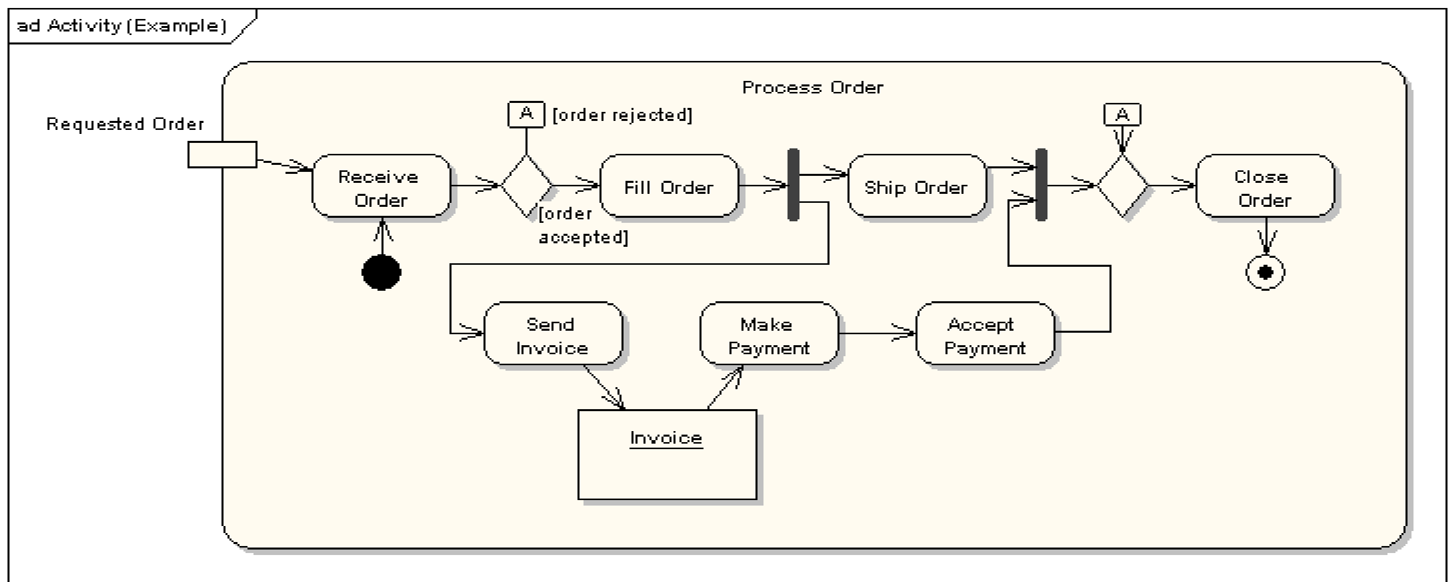
- Use Case diagrams are used to model user/system interactions. They define behavior, requirements and constraints in the form of scripts or scenarios
- Activity diagrams have a wide number of uses, from defining basic program flow, to capturing the decision points and actions within any generalized process
- State Machine diagrams are essential to understanding the instant to instant condition or "run state" of a model when it executes
- Communication diagrams show the network and sequence of messages or communications between objects at run-time during a collaboration instance
- Sequence diagrams are closely related to Communication diagrams and show the sequence of messages passed between objects using a vertical timeline
- Timing diagrams fuse Sequence and State diagrams to provide a view of an object's state over time and messages which modify that state
- Interaction Overview diagrams fuse Activity and Sequence diagrams to provide allow interaction fragments to be easily combined with decision points and flows

UML 2 Activity Diagram

Activity Diagrams

In UML an activity diagram is used to display the sequence of activities. Activity Diagrams show the workflow from a start point to the finish point detailing the many decision paths that exist in the progression of events contained in the activity. They may be use to detail situations where parallel processing may occur in the execution of some activities. Activity Diagrams are useful for Business Modelling where they are used for detailing the processes involved in business activities.

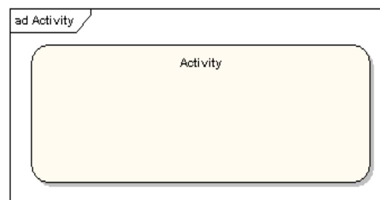
An Example of an Activity Diagram is shown here



The following sections describe the elements that constitute an Activity diagram.

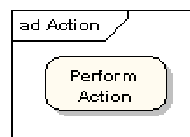
Activities

An activity is the specification of a parameterized sequence of behaviour. An activity is shown as a round-cornered rectangle enclosing all the actions, control flows and other elements that make up the activity.



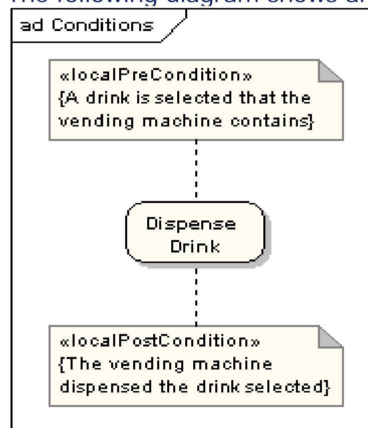
Actions

An action represents a single step within an activity. Actions are denoted by round-cornered rectangles.



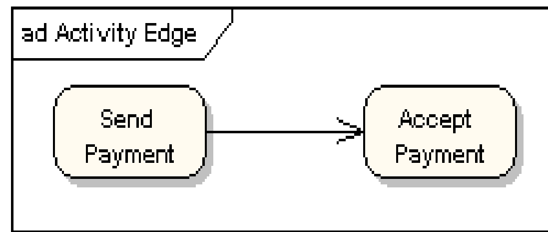
Action Constraints

Constraints can be attached to an action. The following diagram shows an action with local pre- and post-conditions.



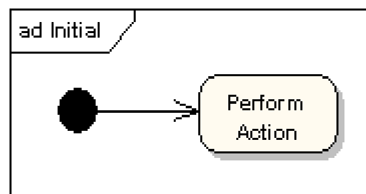
Control Flow

A control flow shows the flow of control from one action to the next. Its notation is a line with an arrowhead.



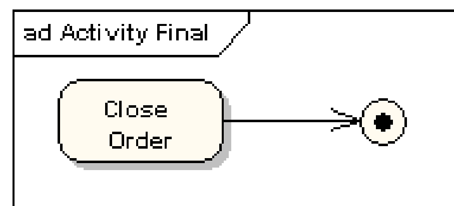
Initial Node

An initial or start node is depicted by a large black spot, as depicted below.

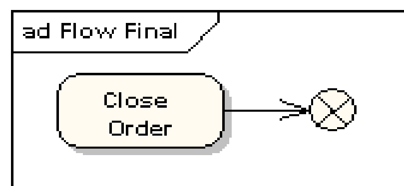


Final Node

There are two types of final node: activity and flow final nodes. The activity final node is depicted as a circle with a dot inside.



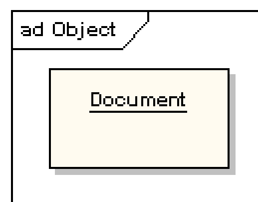
The flow final node is depicted as a circle with a cross inside.



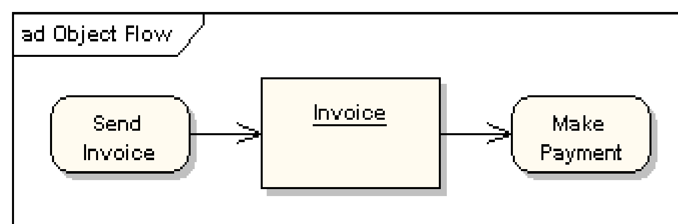
The difference between the two node types is that the flow final node denotes the end of a single control flow; the activity final node denotes the end of all control flows within the activity.

Objects and Object Flows

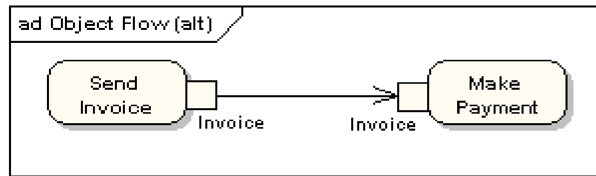
An object flow is a path along which objects or data can pass. An object is shown as a rectangle.



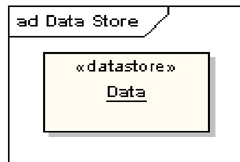
An object flow is shown as a connector with an arrowhead denoting the direction the object is being passed.



An object flow must have an object on at least one of its ends. A shorthand notation for the above diagram would be to use input and output pins.

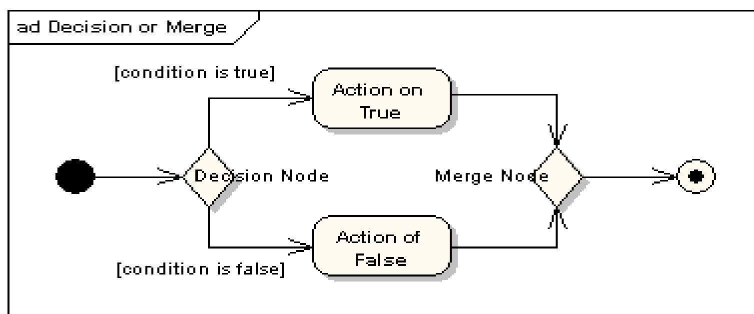


A data store is shown as an object with the «datastore» keyword.



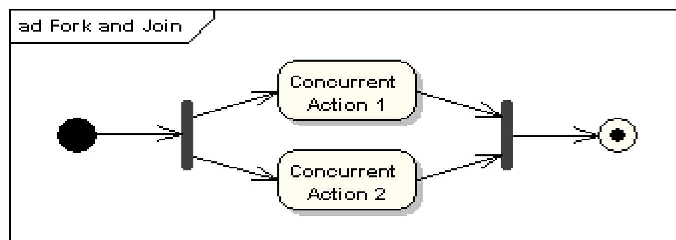
Decision and Merge Nodes

Decision nodes and merge nodes have the same notation: a diamond shape. They can both be named. The control flows coming away from a decision node will have guard conditions which will allow control to flow if the guard condition is met. The following diagram shows use of a decision node and a merge node.



Fork and Join Nodes

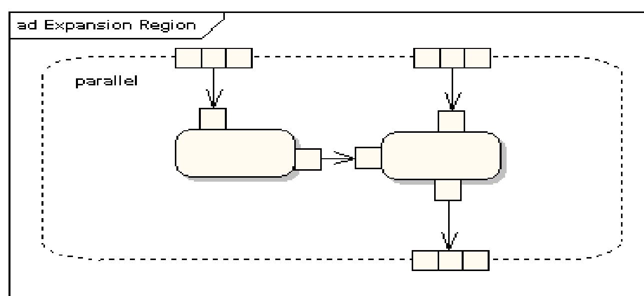
Forks and joins have the same notation: either a horizontal or vertical bar (the orientation is dependent on whether the control flo is running left to right or top to bottom). They indicate the start and end of concurrent threads of control. The following diagram shows an example of their use.



A join is different from a merge in that the join synchronises two inflows and produces a single outflow. The outflow from a join cannot execute until all inflows have been received. A merge passes any control flows straight through it. If two or more inflows ai received by a merge symbol, the action pointed to by its outflow is executed two or more times.

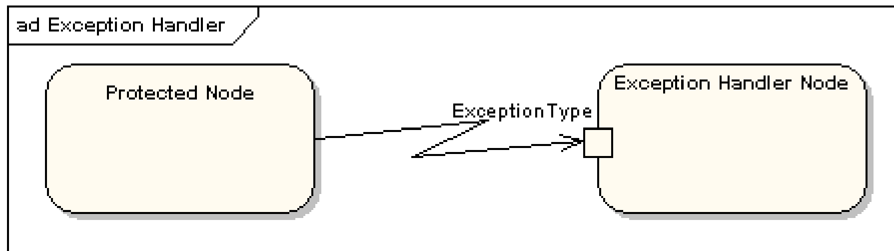
Expansion Region

An expansion region is a structured activity region that executes multiple times. Input and output expansion nodes are drawn as a group of three boxes representing a multiple selection of items. The keyword iterative, parallel or stream is shown in the top left corner of the region.



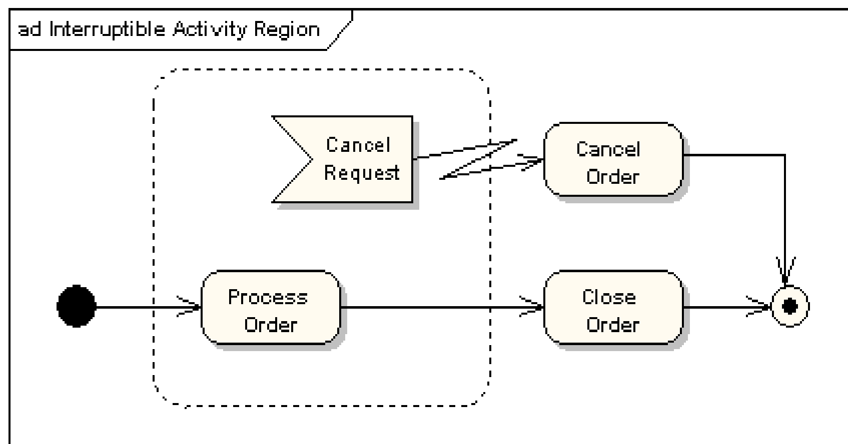
Exception Handlers

Exception Handlers can be modelled on activity diagrams as in the example below.



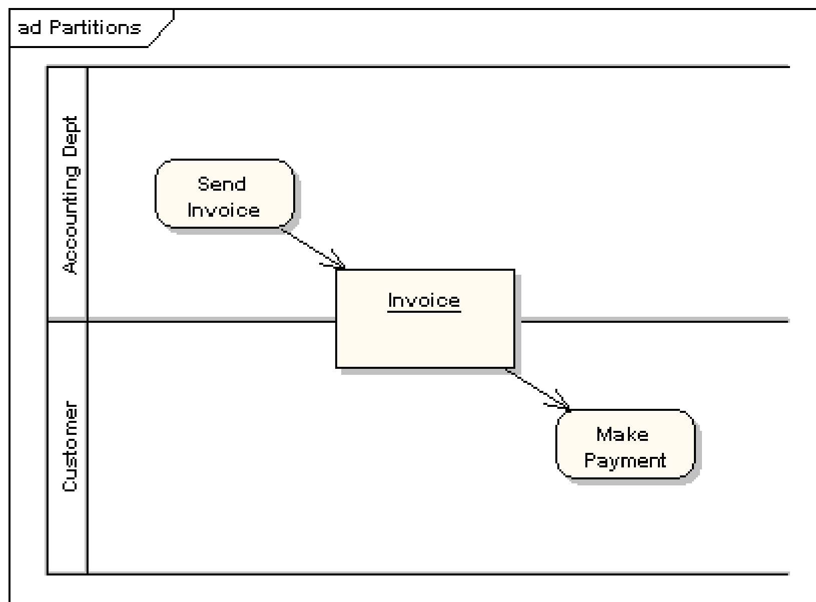
Interruptible Activity Region

An interruptible activity region surrounds a group of actions that can be interrupted. In the very simple example below, the Proces Order action will execute until completion, when it will pass control to the Close Order action, unless a Cancel Request interrupt is received which will pass control to the Cancel Order action.



Partition

An activity partition is shown as either horizontal or vertical swimlanes. In the following diagram, the partitions are used to separate actions within an activity into those performed by the accounting department and those performed by the customer.

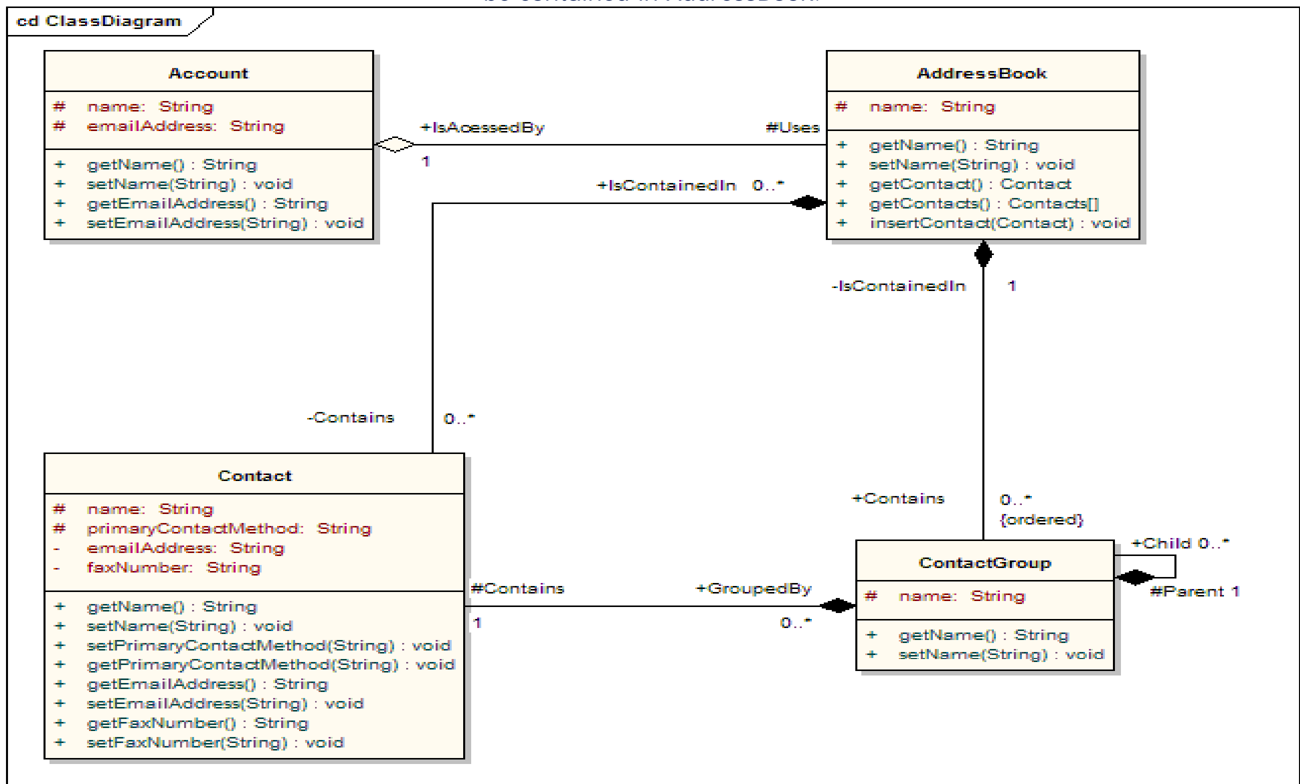


UML 2 Class Diagram

Class Diagrams

The Class diagram shows the building blocks of any object-orientated system. Class diagrams depict the static view of the model or part of the model, describing what attributes and behaviour it has rather that detailing the methods for achieving operations. Class diagrams are most useful to illustrate relationships between classes and interfaces. Generalizations, aggregations, and associations are all valuable in reflecting inheritance, composition or usage, and connections, respectively.

The diagram below illustrates aggregation relationships between classes. The lighter aggregation indicates that the class Account uses AddressBook, but does not necessarily contain an instance of it. The strong, composite aggregations by the other connectors indicate ownership or containment of the source classes by the target classes, for example Contact and ContactGroup values will be contained in AddressBook.



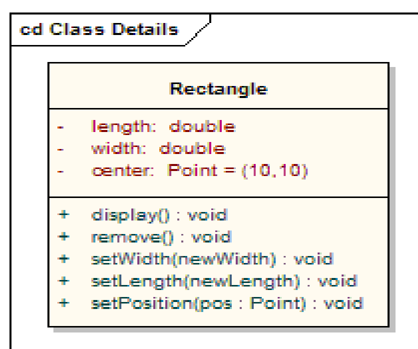
Classes

A class is an element that defines the attributes and behaviours that an object is able to generate. The behaviour is the described by the possible messages the class is able to understand along with operations that are appropriate for each message. Classes may also contain definitions of constraints tagged values and stereotypes.

Class Notation

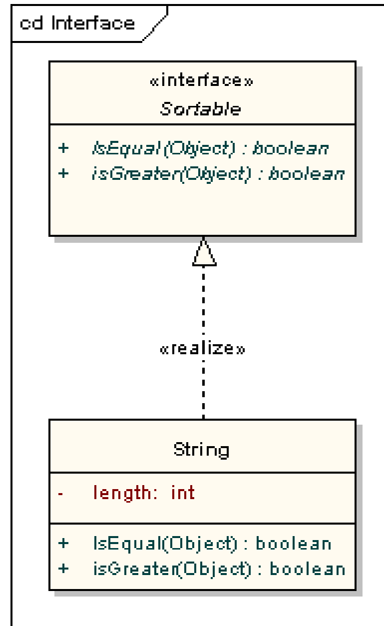
Classes are represented by rectangles which show the name of the class and optionally the name of the operations and attributes. Compartments are used to divide the class name, attributes and operations. Additionally constraints, initial values and parameters may be assigned to classes.

In the diagram below the class contains the class name in the topmost compartment, the next compartment details the attributes, with the "center" attribute showing initial values. The final compartment shows the operations, the setWidth, setLength and setPosition operations showing their parameters. The notation that precedes the attribute or operation name indicates the visibility of the element, if the + symbol is used the attribute or operation has a public level of visibility, if a - symbol is used the attribute or operation is private. In addition the # symbol allows an operation or attribute to be defined as protected and the ~ symbol indicates package visibility.

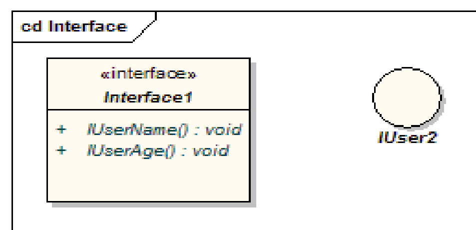


Interfaces

An interface is a specification of behaviour that implementers agree to meet. It is a contract. By realizing an interface, classes are guaranteed to support a required behaviour, which allows the system to treat non-related elements in the same way – i.e. through the common interface.

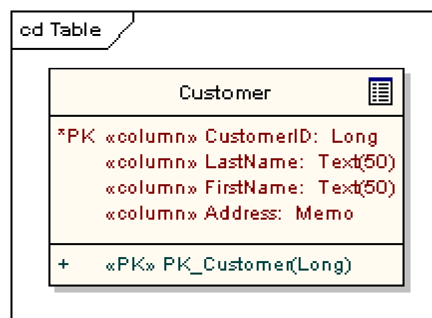


Interfaces may be drawn in a similar style to a class, with operations specified, as shown below. They may also be drawn as a circle with no explicit operations detailed. When drawn as a circle, realization links to the circle form of notation are drawn without target arrows.



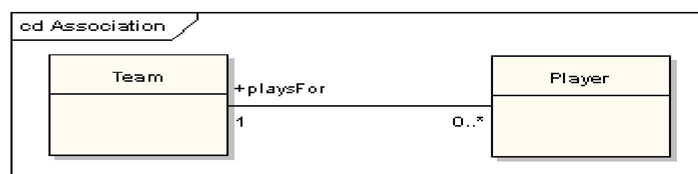
Tables

A table is a stereotyped class. It is drawn with a small table icon in the upper right corner. Table attributes are stereotyped «column». Most tables will have a primary key, being one or more fields that form a unique combination used to access the table, plus a primary key operation which is stereotyped «PK». Some tables will have one or more foreign keys, being one or more fields that together map onto a primary key in a related table, plus a foreign key operation which is stereotyped «FK».



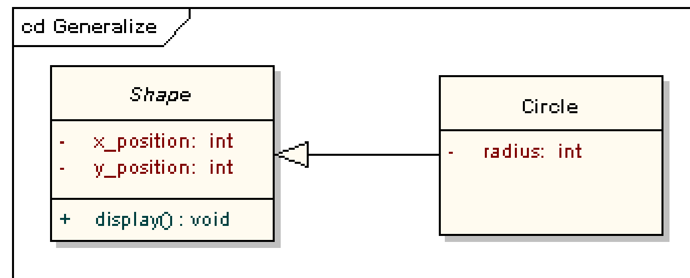
Associations

An association implies two model elements have a relationship - usually implemented as an instance variable in one class. This connector may include named roles at each end, cardinality, direction and constraints. Association is the general relationship type between elements. For more than two elements, a diagonal representation toolbox element can be used as well. When code is generated for class diagrams, associations become instance variables in the target class.

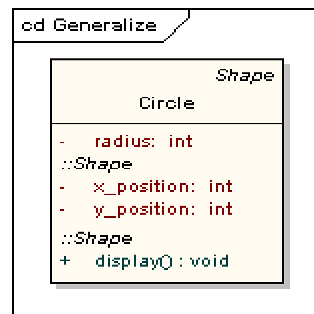


Generalizations

A generalization is used to indicate inheritance. Drawn from the specific classifier to a general classifier, the generalize implication is that the source inherits the target's characteristics. The following diagram shows a parent class generalizing a child class. Implicitly, an instantiated object of the Circle class will have attributes x_position, y_position and radius and a method display(). Note that the class Shape is abstract, shown by the name being italicized.



The following diagram shows an equivalent view of the same information.

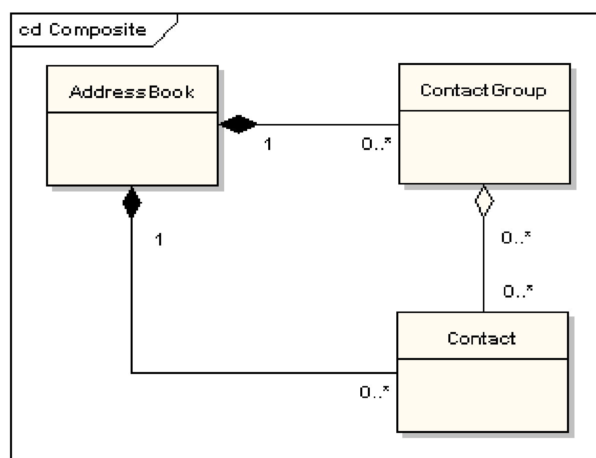


Aggregations

Aggregations are used to depict elements which are made up of smaller components. Aggregation relationships are shown by a white diamond-shaped arrowhead pointing towards the target or parent class.

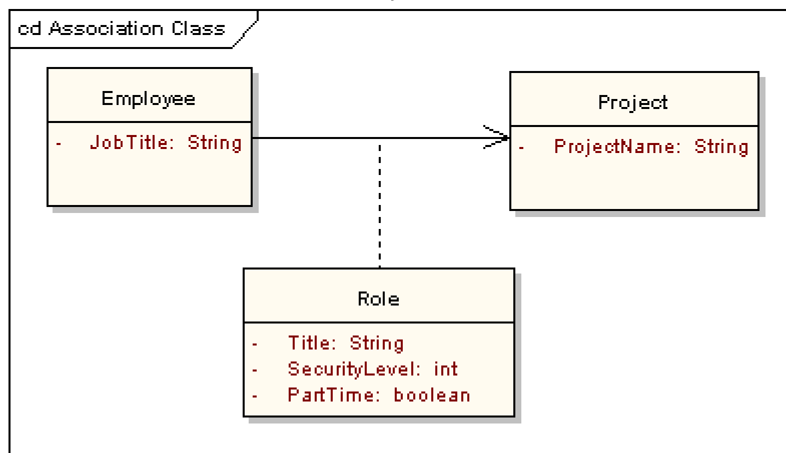
A stronger form of aggregation - a composite aggregation - is shown by a black diamond-shaped arrowhead and is used where components can be included in a maximum of one composition at a time. If the parent of a composite aggregation is deleted, usually all of its parts are deleted with it; however a part can be individually removed from a composition without having to delete the entire composition. Compositions are transitive, asymmetric relationships and can be recursive.

The following diagram illustrates the difference between weak and strong aggregations. An address book is made up of a multiplicity of contacts and contact groups. A contact group is a virtual grouping of contacts; a contact may be included in more than one contact group. If you delete an address book, all the contacts and contact groups will be deleted too; if you delete a contact group, no contacts will be deleted.



Association Classes

An association class is a construct that allows an association connection to have operations and attributes. The following example shows that there is more to allocating an employee to a project than making a simple association link between the two classes: the role that the employee takes up on the project is a complex entity in its own right and contains detail that does not belong in the employee or project class. For example, an employee may be working on several projects at the same time and have different job titles and security levels on each.



Dependencies

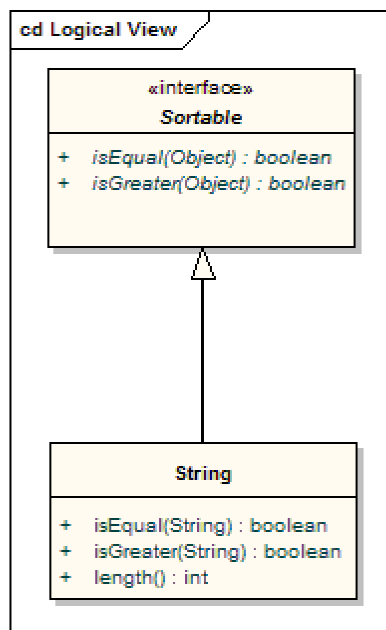
A dependency is used to model a wide range of dependent relationships between model elements. It would normally be used early in the design process where it is known that there is some kind of link between two elements but it is too early to know exactly what the relationship is. Later in the design process, dependencies will be stereotyped (stereotypes available include «instantiate» «trace», «import» and others) or replaced with a more specific type of connector.

Traces

The trace relationship is a specialization of a dependency, linking model elements or sets of elements that represent the same idea across models. Traces are often used to track requirements and model changes. As changes can occur in both directions, the order of this dependency is usually ignored. The relationship's properties can specify the trace mapping, but the trace is usually bi-directional, informal and rarely computable.

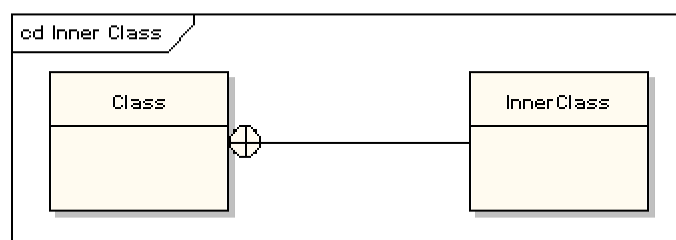
Realizations

The source object implements or realizes the destination. Realize is used to express traceability and completeness in the model - a business process or requirement is realized by one or more use cases which are in turn realized by some classes, which in turn are realized by a component, etc. Mapping requirements, classes, etc. across the design of your system, up through the levels of modelling abstraction, ensures the big picture of your system remembers and reflects all the little pictures and details that constrain and define it. A realization is shown as a dashed line with a solid arrowhead and the «realize» stereotype.



Nestings

A nesting is connector that shows that the source element is nested within the target element. The following diagram shows the definition of an inner class although in EA it is more usual to show them by their position in the Project View hierarchy.



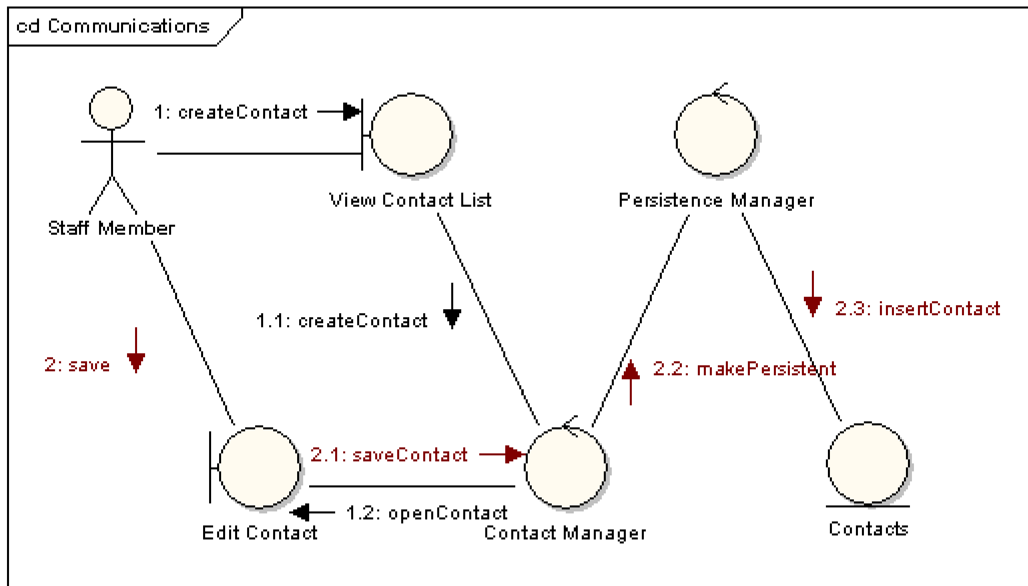
UML 2 Communication Diagram

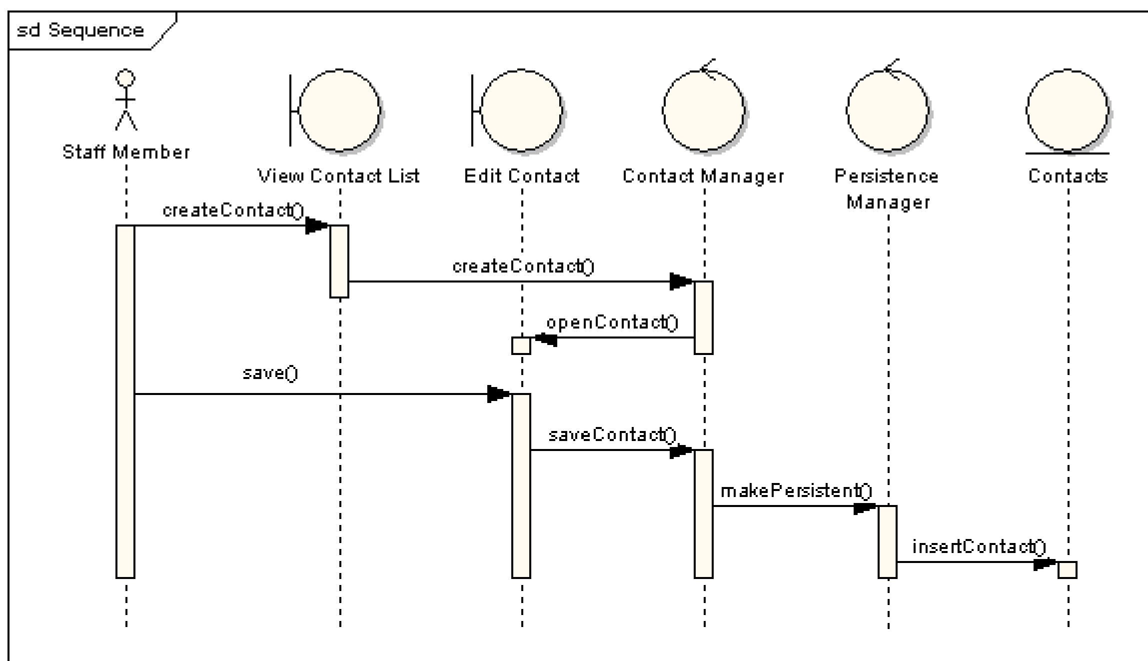
Communication Diagrams

A communication diagram, formerly called a collaboration diagram, is an interaction diagram that shows similar information to sequence diagrams but its primary focus in on object relationships.

On communication diagrams, objects are shown with association connectors between them. Messages are added to the associations and show as short arrows pointing in the direction of the message flow. The sequence of messages is shown through a numbering scheme.

The following two diagrams show a communication diagram and the sequence diagram that shows the same information. Although it is possible to derive the sequencing of messages in the communication diagram from the numbering scheme, it isn't immediately visible. What the communication diagram does show quite clearly though is the full set of messages passed between adjacent objects.



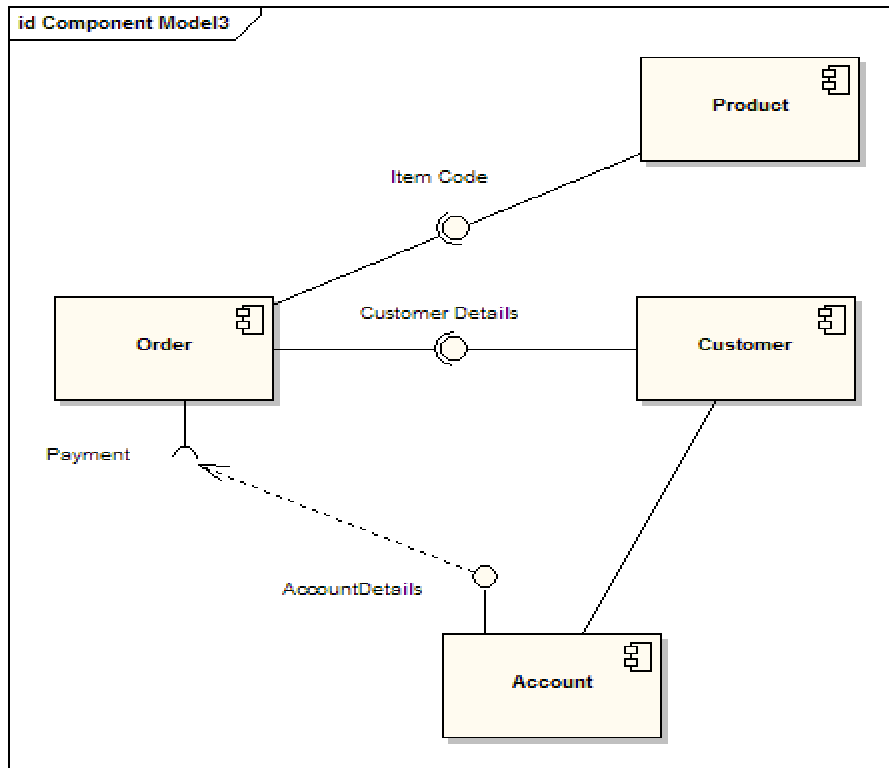


UML 2 Component Diagram

Component Diagrams

Component Diagrams illustrate the pieces of software, embedded controllers, etc. that will make up a system. A Component diagram has a higher level of abstraction than a Class diagram - usually a component is implemented by one or more classes (or objects) at runtime. They are building blocks, such that eventually a component can encompass a large portion of a system.

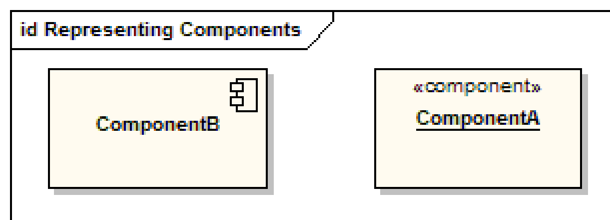
The diagram below demonstrates some components and their inter-relationships. Assembly connectors 'link' the provided interfaces supplied by Product and Customer to the required interfaces specified by Order. A dependency relationship maps a customer's associated account details to the required interface, 'Payment', indicated by Order.



Components are similar in practice to package diagrams as the define boundaries and are used to group elements into logical structures. The difference between Package Diagrams and Component diagrams is that Component Diagrams offer a more semantically rich grouping mechanism. With Component Diagrams all of the model elements are private whereas Package diagrams only display public items.

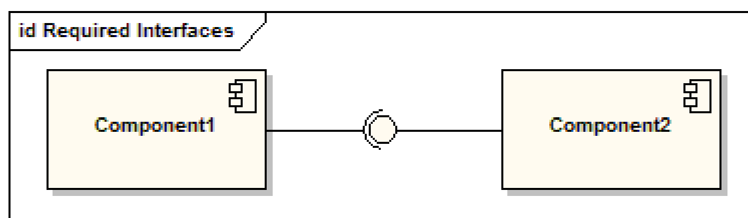
Representing Components

Components are represented as a rectangular classifier with the keyword «component», optionally the component may be displayed as a rectangle with a component icon in the right-hand upper corner.



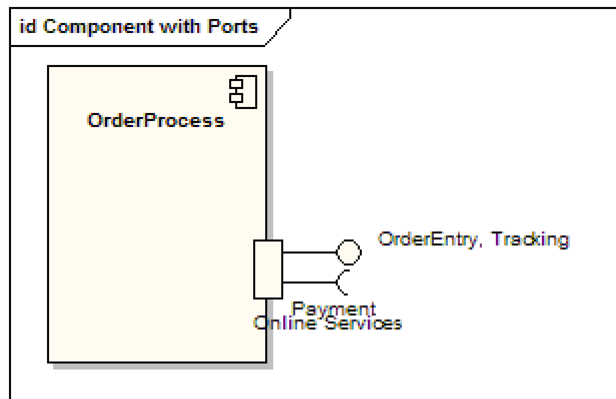
Required Interfaces

The Assembly connector bridges a component's required interface (Component1) with the provided interface of another component (Component2); this allows one component to provide the services that another component requires. Interfaces are collections of one or more methods which may or may not contain attributes.



Components with Ports

Using Ports with Component Diagrams allows for a service or behavior to be specified to its environment as well as a service or behavior that a Component requires. Ports may specify inputs, outputs as well as operating bi-directionally. The following diagram details a component with a port for Online services along with two provided interfaces Order Entry and Tracking as well as a required interface Payment.

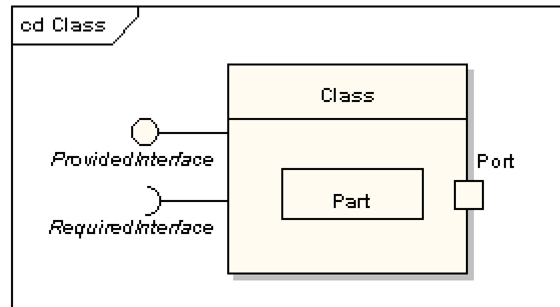


UML 2 Composite Structure Diagram

Composite Diagrams

A composite structure diagram is a diagram that shows the internal structure of a classifier, including its interaction points to other parts of the system. It shows the configuration and relationship of parts that together perform the behaviour of the containing classifier.

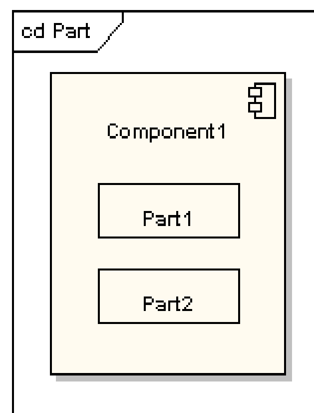
Class elements have been described in great detail in the section on class diagrams. This section describes the way that classes can be displayed as composite elements exposing interfaces and containing ports and parts.



Part

A part is an element that represents a set of one or more instances which are owned by a containing classifier instance. So for example, if a diagram instance owned a set of graphical elements, then the graphical elements could be represented as parts, if it were useful to do so to model some kind of relationship between them. Note that a part can be removed from its parent before the parent is deleted, so that the part isn't deleted at the same time.

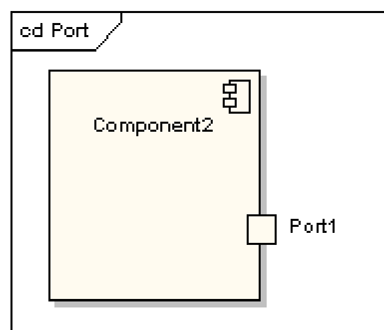
A part is shown as an unadorned rectangle contained within the body of a class or component element.



Port

A port is a typed element that represents an externally visible part of a containing classifier instance. Ports define the interaction between a classifier and its environment. A port can appear on the boundary of a contained part, a class or a composite structure. A port may specify the services a classifier provides as well as the services that it requires of its environment.

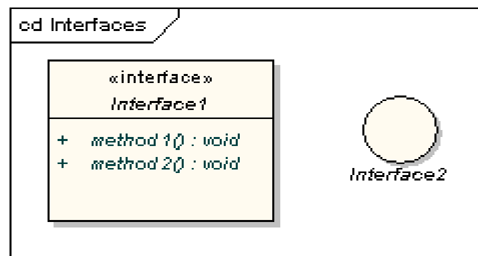
A port is shown as a named rectangle on the boundary edge of its owning classifier.



Interfaces

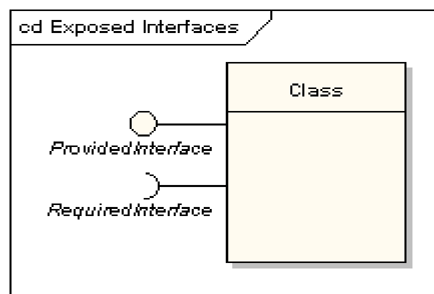
An interface is similar to a class but with a number of restrictions. All interface operations are public and abstract, and do not provide any default implementation. All interface attributes must be constants. However, while a class may only inherit from a single super-class, it may implement multiple interfaces.

An interface, when standing alone in a diagram, is either shown as a class element rectangle with the «interface» keyword and with its name italicised to denote it is abstract, or it is shown as a circle.



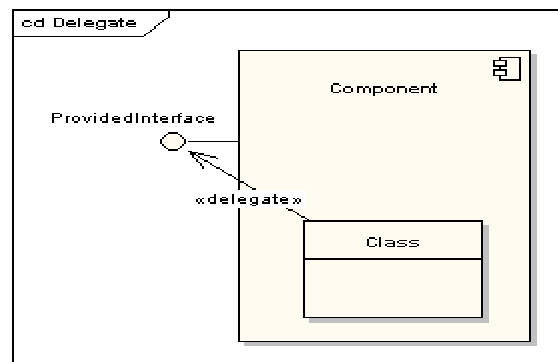
Note that the circle notation does not show the interface operations. When interfaces are shown as being owned by classes, they are referred to as exposed interfaces. An exposed interface can be defined as either provided or required. A provided interface is an affirmation that the containing classifier supplies the operations defined by the named interface element and is defined by drawing a realisation link between the class and the interface. A required interface is a statement that the classifier is able to communicate with some other classifier which provides operations defined by the named interface element and is defined by drawing a dependency link between the class and the interface.

A provided interface is shown as a "ball on a stick" attached to the edge of a classifier element. A required interface is shown as a "cup on a stick" attached to the edge of a classifier element.



Delegate

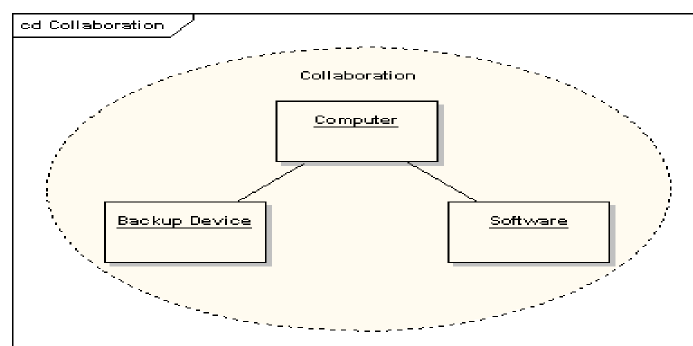
A delegate connector is used for defining the internal workings of a component's external ports and interfaces. A delegate connector is shown as an arrow with a «delegate» stereotype. It connects an external contract of a component as shown by its ports to the internal realisation of the behaviour of the component's part.



Collaboration

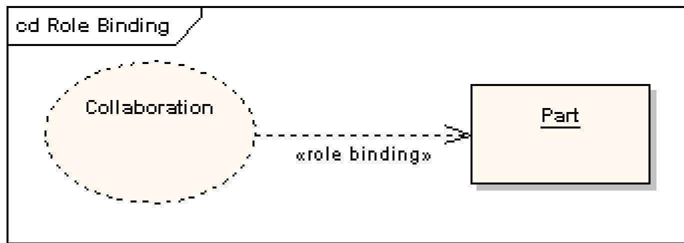
A collaboration defines a set of co-operating roles used collectively to illustrate a specific functionality. A collaboration should only show the roles and attributes required to accomplish its defined task or function. Isolating the primary roles is an exercise in simplifying the structure and clarifying the behaviour, and also provides for re-use. A collaboration often implements a pattern.

A collaboration element is shown as an ellipse.



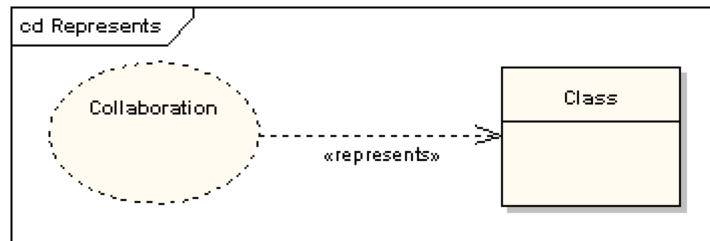
Role Binding

A role binding connector is drawn from a collaboration to the classifier that fulfils the role. It is shown as a dashed line with arrowhead and the stereotype «role binding».



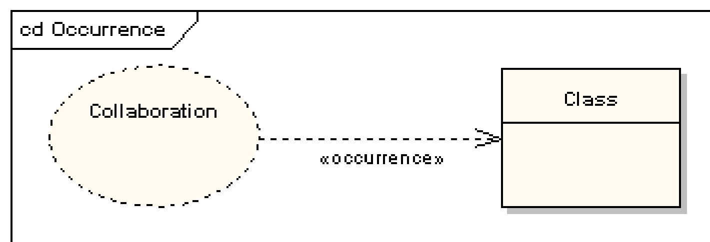
Represents

A represents connector may be drawn from a collaboration to a classifier to show that a collaboration is used in the classifier. It is shown as a dashed line with arrowhead and the stereotype «represents».



Occurrence

An occurrence connector may be drawn from a collaboration to a classifier to show that a collaboration represents(sic) the classifier. It is shown as a dashed line with arrowhead and the stereotype «occurrence».



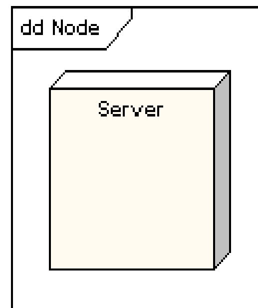
UML 2 Deployment Diagram

Deployment Diagrams

A Deployment Diagram models the run-time architecture of a system. It shows the configuration of the hardware elements (nodes) and shows how software elements and artifacts are mapped onto those nodes.

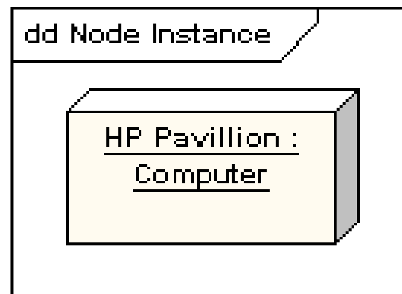
Node

A Node is either a hardware or software element. It is shown as a 3-dimensional box shape, as below



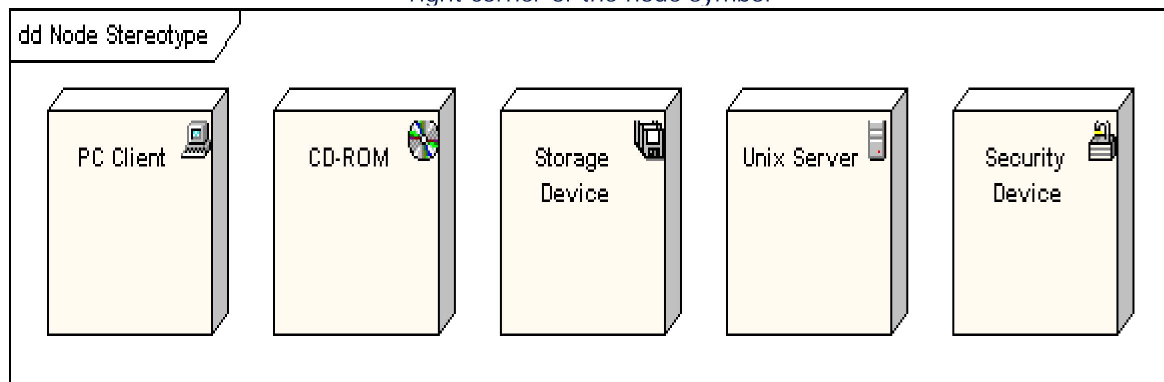
Node Instance

A node instance can be shown on a diagram. An instance can be distinguished from a node by the fact that its name is underlined and has a colon before its base node type. An instance may or may not have a name before the colon. The following diagram shows a named instance of a computer.



Node Stereotypes

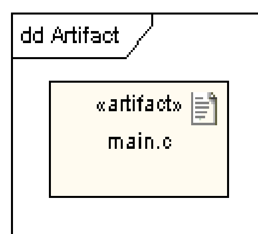
A number of standard stereotypes are provided for nodes, namely «cdrom», «cd-rom», «computer», «disk array», «pc», «pc client», «pc server», «secure», «server», «storage», «unix server», «user pc». These will display an appropriate icon in the top right corner of the node symbol



Artifact

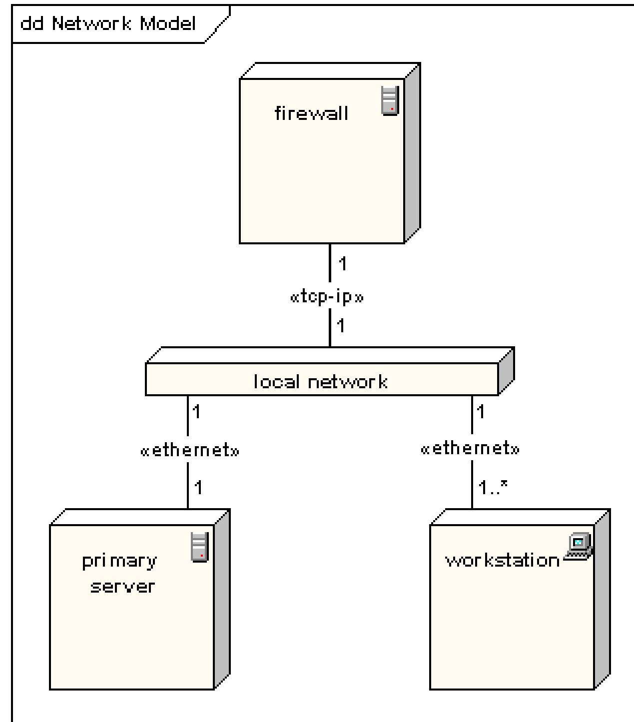
An Artifact is a product of the software development process. That may include process models (e.g. Use Case models, Design models etc), source files, executables, design documents, test reports, prototypes, user manuals and so on.

An artifact is denoted by a rectangle showing the artifact name, the «artifact» stereotype and a document icon, as follows.



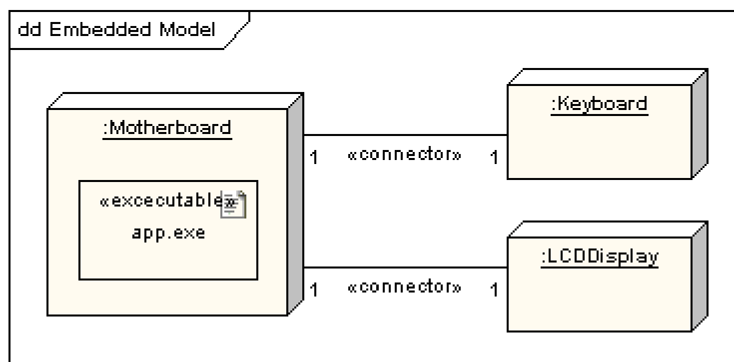
Association

In the context of a deployment diagram, an association represents a communication path between nodes. The following diagram shows a deployment diagram for a network, showing network protocols as stereotypes and also showing multiplicities at the association ends.



Node as Container

A node can contain other elements, such as components or artifacts. The following diagram shows a deployment diagram for part of an embedded system and showing an executable artifact as being contained by the motherboard node.



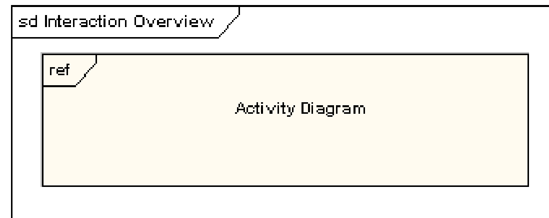
UML 2 Interaction Overview Diagram

Interaction Overview Diagrams

An Interaction Overview Diagram is a form of activity diagram in which the nodes represent interaction diagrams. Interaction diagrams can include sequence, communication, interaction overview and timing diagrams. Most of the notation for interaction overview diagrams is the same as for activity diagrams, for example initial, final, decision, merge, fork and join nodes are all the same. However, interaction overview diagrams introduce two new elements, interaction occurrences and interaction elements.

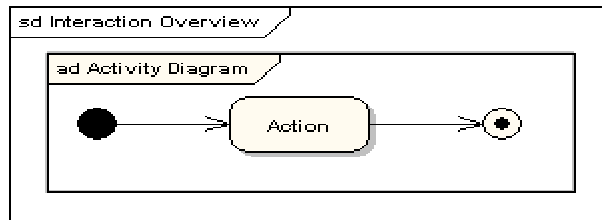
Interaction Occurrence

Interaction Occurrences are references to existing interaction diagrams. An interaction occurrence is shown as a reference frame, i.e. a frame with "ref" in the top-left corner. The name of the diagram being referenced is shown in the center of the frame.



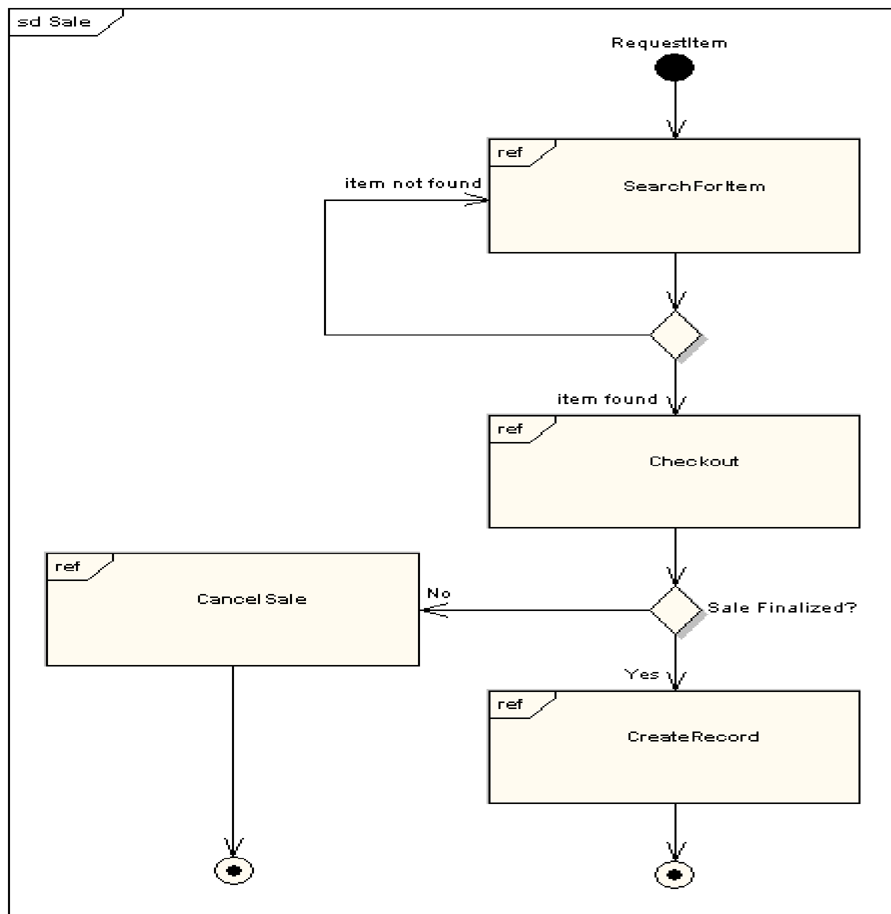
Interaction Element

Interaction Elements are similar to interaction occurrences in that they display a representation of existing interaction diagrams within a rectangular frame. They differ in that they display the contents of the references diagram inline.



Putting it all together

All the same controls from activity diagrams (fork, join, merge etc) can be used on Interaction Overview diagrams to put the control logic around the lower level diagrams. The following example depicts a sample sale process with sub-processes abstracted within interaction occurrences.



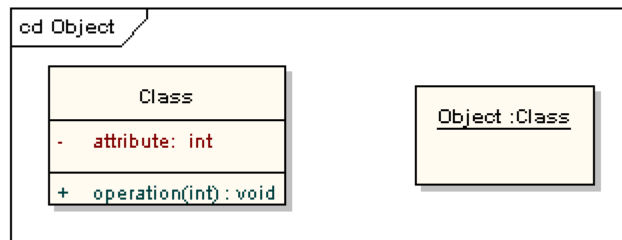
UML 2 Object Diagrams

Object Diagrams

An object diagram may be considered a special case of a class diagram. Object diagrams use a subset of the elements of a class diagram in order to emphasize the relationship between instances of classes at some point in time. They are useful in understanding class diagrams. They don't show anything architecturally different to class diagrams, but reflect multiplicity and roles.

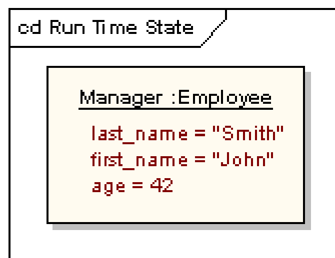
Class and Object Elements

The following diagram shows the differences in appearance between a class element and an object element. Note that the class element consists of three parts, being divided into name, attribute and operation compartments; by default, object elements don't have compartments. The display of names is also different: object names are underlined and may show the name of the classifier from which the object is instantiated.



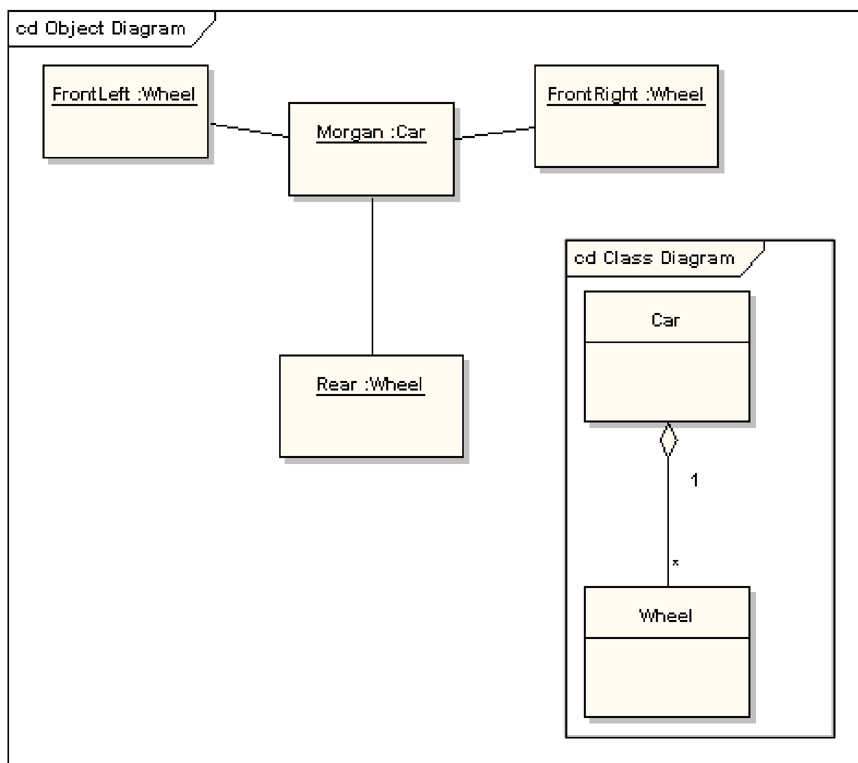
Run Time State

A classifier element can have any number of attributes and operations. These aren't shown in an object instance. It is possible, however, to define an object's run time state, showing the set values of attributes in the particular instance.



Example Class and Object Diagrams

The following diagram shows an object diagram with its defining class diagram inset, and it illustrates the way in which an object diagram may be used to test the multiplicities of assignments in class diagrams. The car class has a 1-to-many multiplicity to the wheel class, but if a 1-to-4 multiplicity had been chosen instead, that wouldn't have allowed for the three-wheeled car shown in the object diagram.

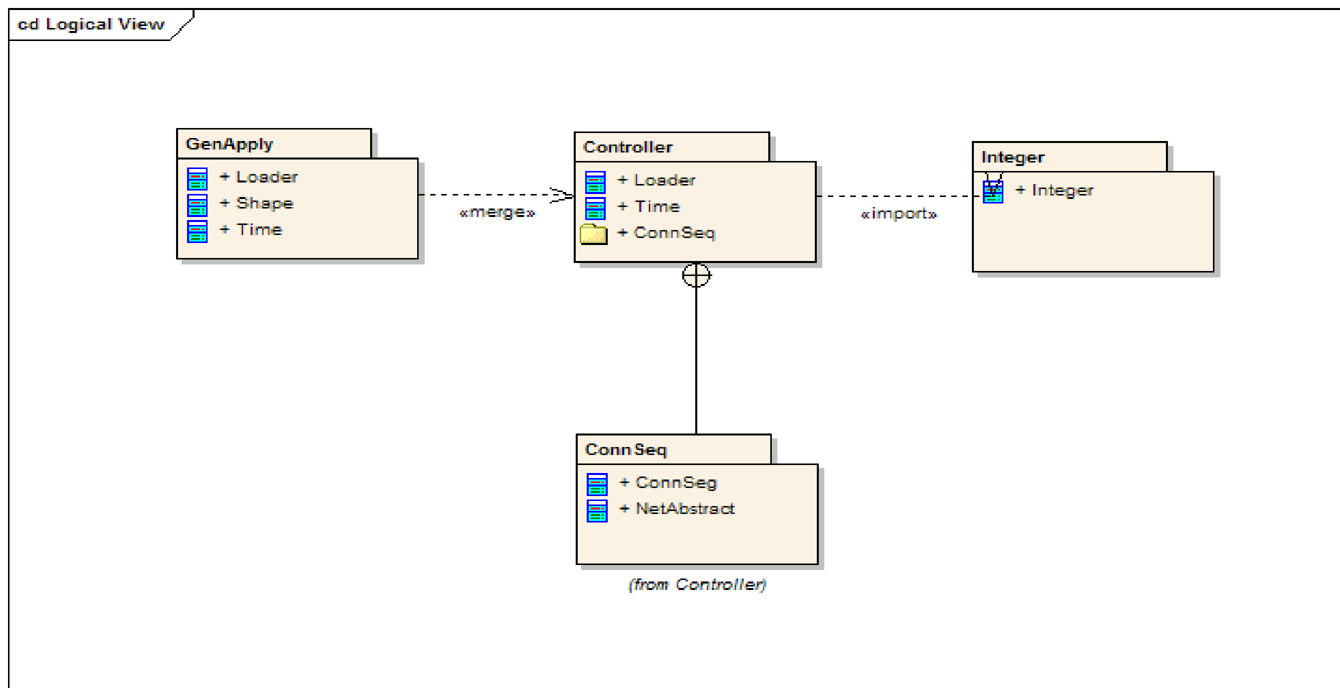


UML 2 Package Diagram

Package Diagrams

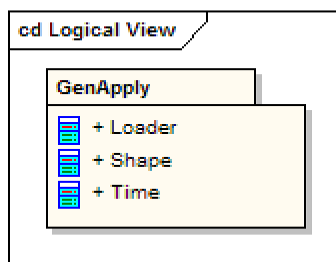
Package Diagrams are used to reflect the organization of packages and their elements. When used to represent class elements package diagrams are used to provide a visualization of the namespaces. The most common uses for Package diagrams is to use them to organize Use-Case Diagrams and Class diagrams, although the use of Package Diagrams is not limited to these UML elements.

The following is an example of a package diagram.



Elements contained in a Package share the same namespace, this sharing of namespace requires the elements contained in a specific namespace to have unique names.

Packages can be built to represent either physical or logical relationships. When choosing to include classes to specific packages, it is useful to assign the classes with the same inheritance hierarchy to packages, classes that are related via composition and classes that collaborate with also have a strong argument for being included into the same package..



Packages are represented in UML 2.0 as folders and contain the elements that share a namespace; all elements within a package must have a unique identifier. The Package must show the Package name and can optionally show the elements within the Packag in extra compartments.

Package Merge

When a «merge» connector is used on a package, the source of the merge imports the target's nested and imported contents. If a element exists within the source and in the target the sources element's definitions will be expanded to will be expanded to includ the element definitions contained in the target. All of the elements added or updated by a merge are noted by a generalization relationship from the source to the target.

Package Import

The «import» connector indicates that the elements within the target package, which in this example is a single class, the target package, will be imported into the source package. The Source Package's namespace will gain access to the Target's class/s; the Target's namespace is not affected.

Nesting Connectors

The nesting connector between the target package and source packages reflect what the package contents reveal.

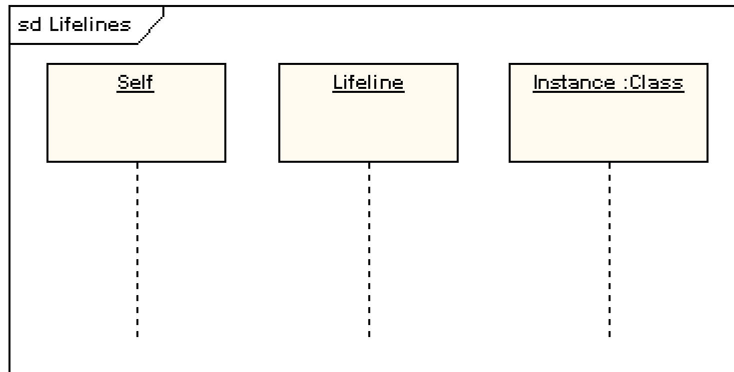
UML 2 Sequence Diagram

Sequence Diagrams

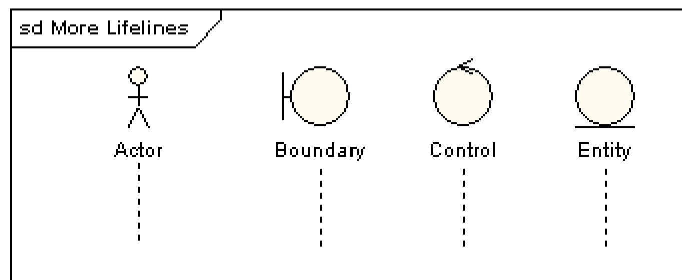
A sequence diagram is a form of interaction diagram which shows objects as lifelines running down the page and with their interactions over time represented as messages drawn as arrows from the source lifeline to the target lifeline. Sequence diagrams are good at showing which objects communicate with which other objects and what messages trigger those communications. Sequence diagrams are not intended for showing complex procedural logic..

Lifelines

A lifeline represents an individual participant in a sequence diagram. A lifeline will usually have a rectangle containing its object name. If its name is self then that indicates that the lifeline represents the classifier which owns the sequence diagram..

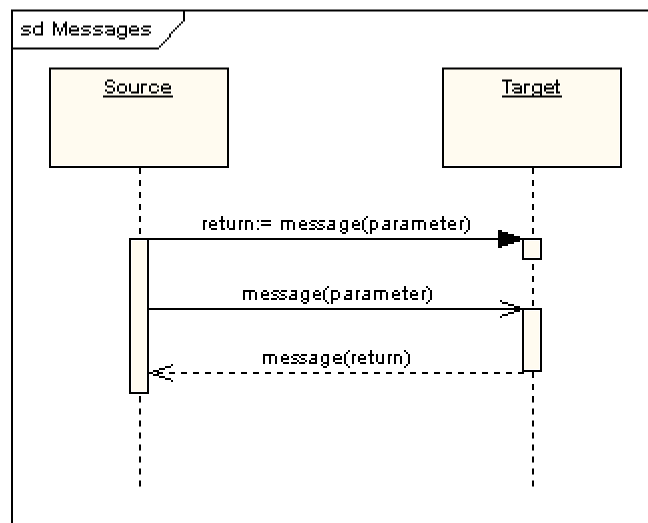


Sometimes a sequence diagram will have a lifeline with an actor element symbol at its head. This will usually be the case if the sequence diagram is owned by a use case. Boundary, control and entity elements from robustness diagrams can also own lifelines



Messages

Messages are displayed as arrows. Messages can be complete, lost or found; synchronous or asynchronous; call or signal. In the following diagram, the first message is a synchronous message (denoted by the solid arrowhead) complete with an implicit return message; the second message is asynchronous (denoted by line arrowhead) and the third is the asynchronous return message (denoted by the dashed line).

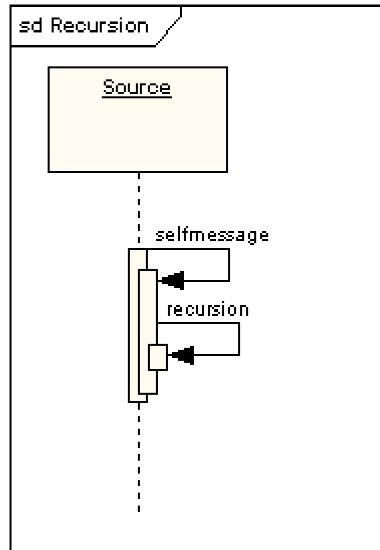


Execution Occurrence

A thin rectangle running down the lifeline denotes the execution occurrence or activation of a focus of control. In the previous diagram, there are three execution occurrences. The first is the source object sending two messages and receiving two replies; the second is the target object receiving a synchronous message and returning a reply; and the third is the target object receiving an asynchronous message and returning a reply.

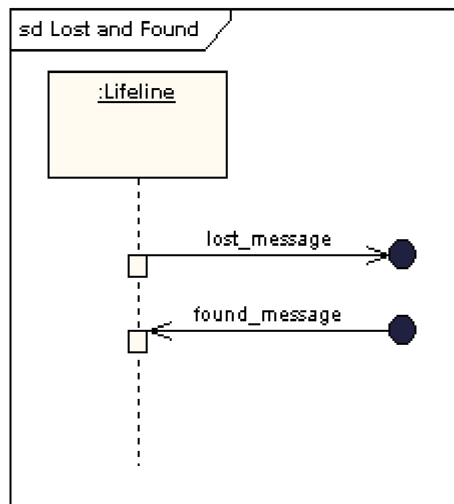
Self Message

A self message can represent a recursive call of an operation, or one method calling another method belonging to the same object It is shown as creating a nested focus of control in the lifeline's execution occurrence.



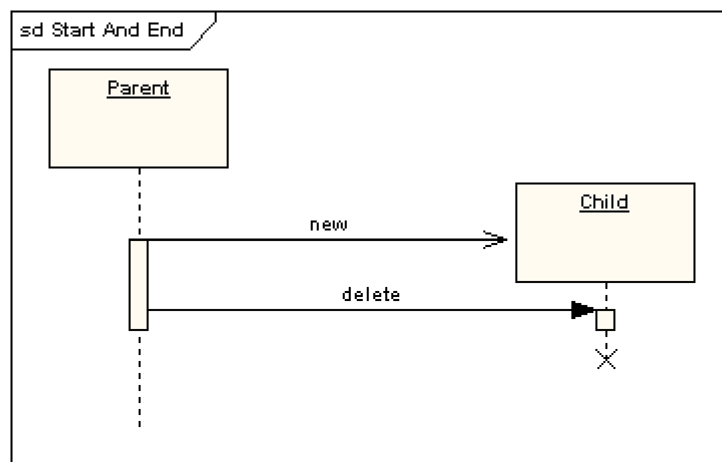
Lost and Found Messages

Lost messages are those that are either sent but do not arrive at the intended recipient, or which go to a recipient not shown on the current diagram. Found messages are those that arrive from an unknown sender, or from a sender not shown on the current diagram. They are denoted going to or coming from an endpoint element.



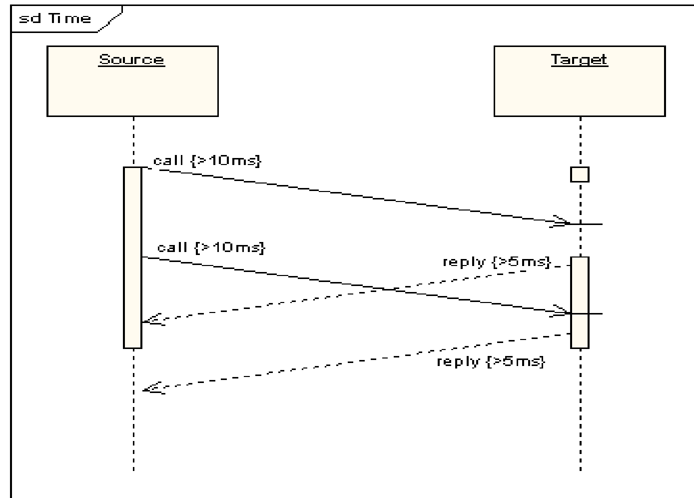
Lifeline Start and End

A lifeline may be created or destroyed during the timescale represented by a sequence diagram. In the latter case, the lifeline is terminated by a stop symbol, represented as a cross. In the former case, the symbol at the head of the lifeline is shown at a lower level down the page than the symbol of the object that caused the creation. The following diagram shows an object being created and destroyed.



Duration and Time Constraints

By default, a message is shown as a horizontal line. Since the lifeline represents the passage of time down the screen, when modelling a real-time system, or even a time-bound business process, it can be important to consider the length of time it takes to perform actions. By setting a duration constraint for a message, the message will be shown as a sloping line.

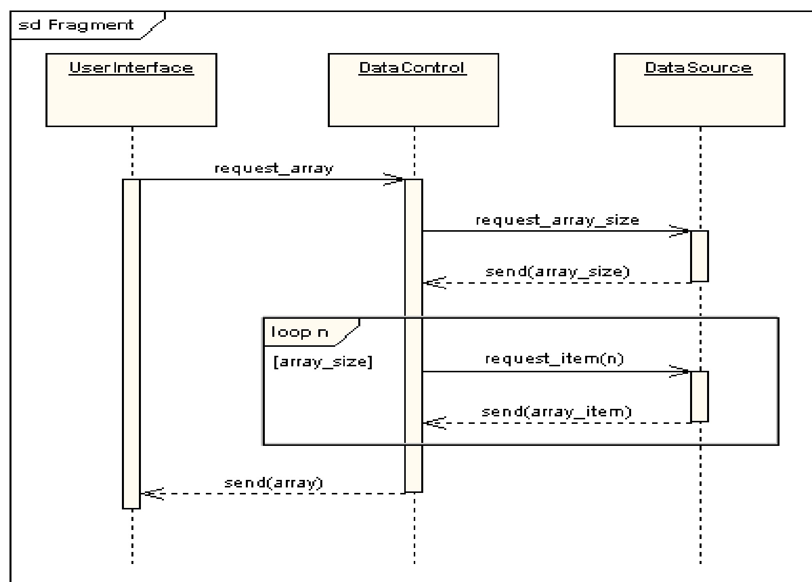


Combined Fragments

It was stated earlier that Sequence diagrams are not intended for showing complex procedural logic. While this is the case, there are a number of mechanisms that do allow for adding a degree of procedural logic to diagrams and which come under the heading of combined fragments. A combined fragment is one or more processing sequence enclosed in a frame and executed under specific named circumstances. The fragments available are:

- Alternative fragment (denoted "alt") models if...then...else constructs.
- Option fragment (denoted "opt") models switch constructs.
- Break fragment models an alternative sequence of events that is processed instead of the whole of the rest of the diagram.
- Parallel fragment (denoted "par") models concurrent processing.
- Weak sequencing fragment (denoted "seq") encloses a number of sequences for which all the messages must be processed in a preceding segment before the following segment can start, but which does not impose any sequencing within a segment on messages that don't share a lifeline.
- Strict sequencing fragment (denoted "strict") encloses a series of messages which must be processed in the given order.
- Negative fragment (denoted "neg") encloses an invalid series of messages.
- Critical fragment encloses a critical section.
- Ignore fragment declares a message or message to be of no interest if it appears in the current context.
- Consider fragment is in effect the opposite of the ignore fragment: any message not included in the consider fragment should be ignored.
- Assertion fragment (denoted "assert") designates that any sequence not shown as an operand of the assertion is invalid.
- Loop fragment encloses a series of messages which are repeated.

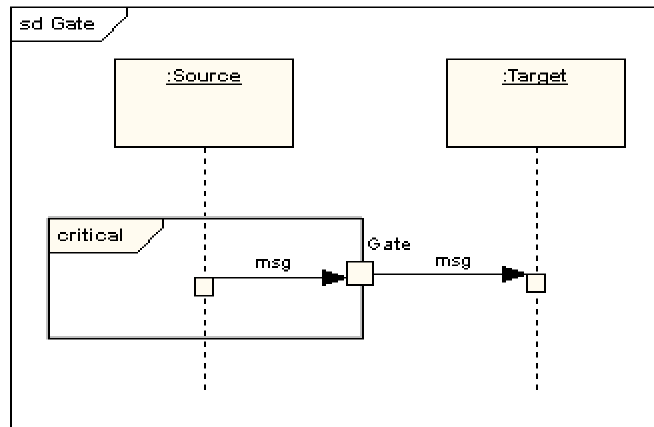
The following diagram shows a loop fragment.



There is also an interaction occurrence, which is similar to a combined fragment. An interaction occurrence is a reference to anoth diagram which has the word "ref" in the top left corner of the frame, and has the name of the referenced diagram shown in the middle of the frame.

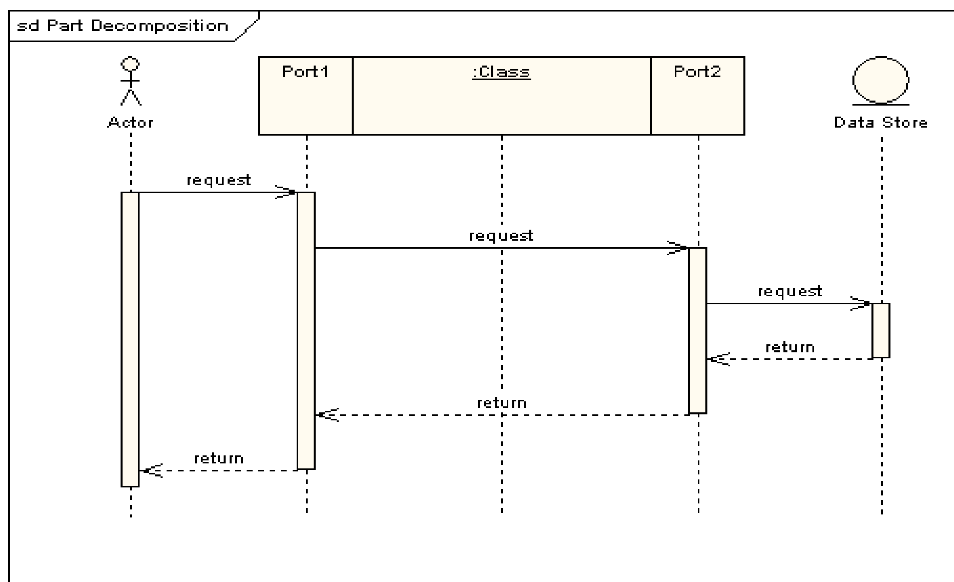
Gate

A gate is a connection point for connecting a message inside a fragment with a message outside a fragment. EA shows a gate as a small square on a fragment frame.



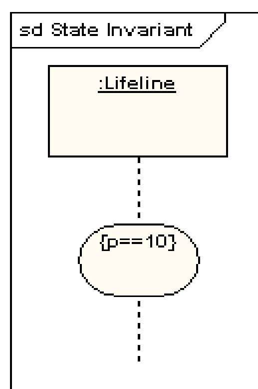
Part Decomposition

An object can have more than one lifeline coming from it. This allows for inter- and intra-object messages to be displayed on the same diagram.



State Invariant / Continuations

A state invariant is a constraint placed on a lifeline that must be true at run-time. It is shown as a rectangle with semi-circular ends.



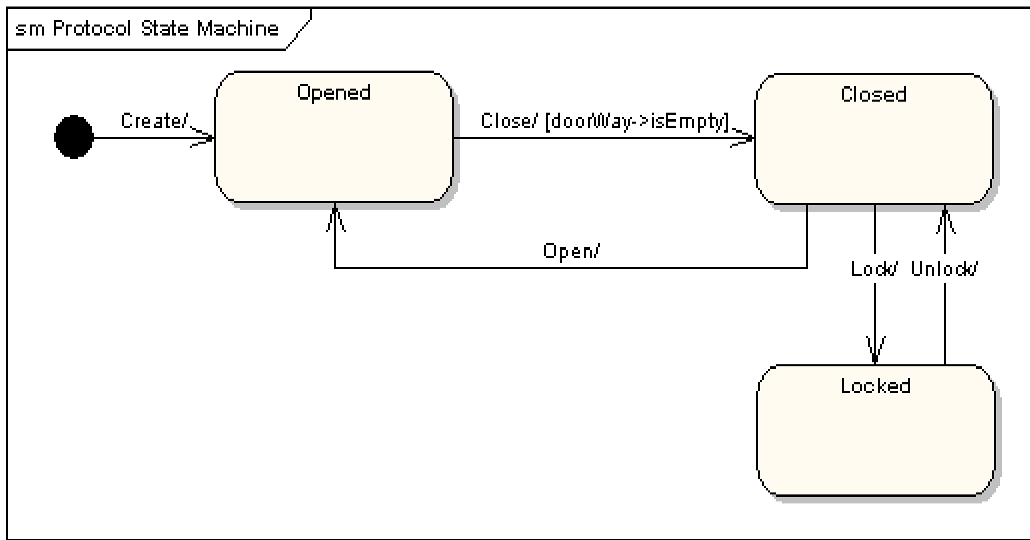
A Continuation has the same notation as a state invariant but is used in combined fragments and can stretch across more than on lifeline.

UML 2 State Machine Diagram

State Machine Diagrams

A State Machine Diagram models the behaviour of a single object, specifying the sequence of events that an object goes through during its lifetime in response to events.

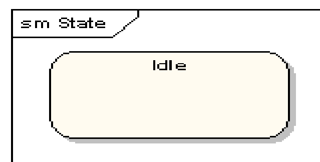
As an example, the following State Machine Diagram shows the states that a door goes through during its lifetime.



The door can be in one of three states: Opened, Closed or Locked. It can respond to the events Open, Close, Lock and Unlock. Notice that not all events are valid in all states: for example, if a door is Opened, you cannot lock it until you close it. Also notice that a state transition can have a guard condition attached: if the door is Opened, it can only respond to the Close event if the condition doorWay->isEmpty is fulfilled. The syntax and conventions used in State Machine Diagrams will be discussed in full in th following sections.

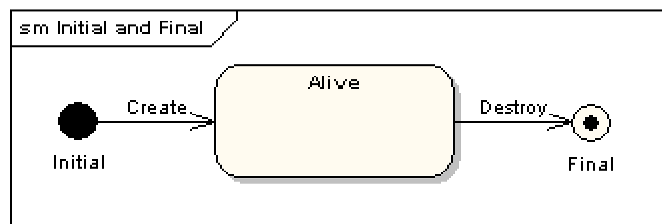
States

A State is denoted by a round-cornered rectangle with the name of the state written inside it.



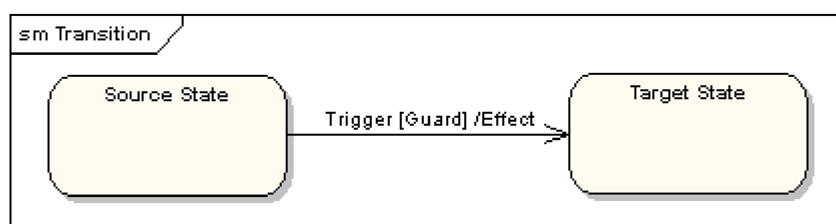
Initial and Final States

The Initial State is denoted by a filled black circle and may be labelled with a name. The Final State is denoted by a circle with a d inside and may also be labelled with a name.



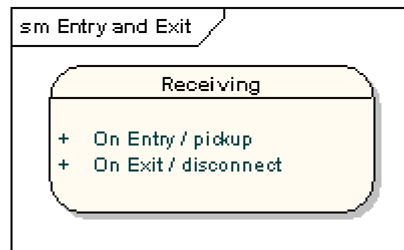
Transitions

Transitions from one state to the next are denoted by lines with arrowheads. A transition may have a trigger, a guard and an effect, as below.



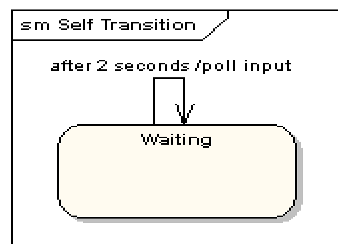
"Trigger" is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time. "Guard" is a condition which must be true in order for the trigger to cause the transition. "Effect" is an action which will be invokec directly on the object that owns the state machine as a result of the transition.

State Actions
In the transition example above, an Effect was associated with the transition. If the target state had many transitions arriving at it and each transition had the same effect associated with it, it would be better to associate the effect with the target state rather than the transitions. This can be done by defining an entry action for the state. The diagram below shows a state with an entry action and an exit action.

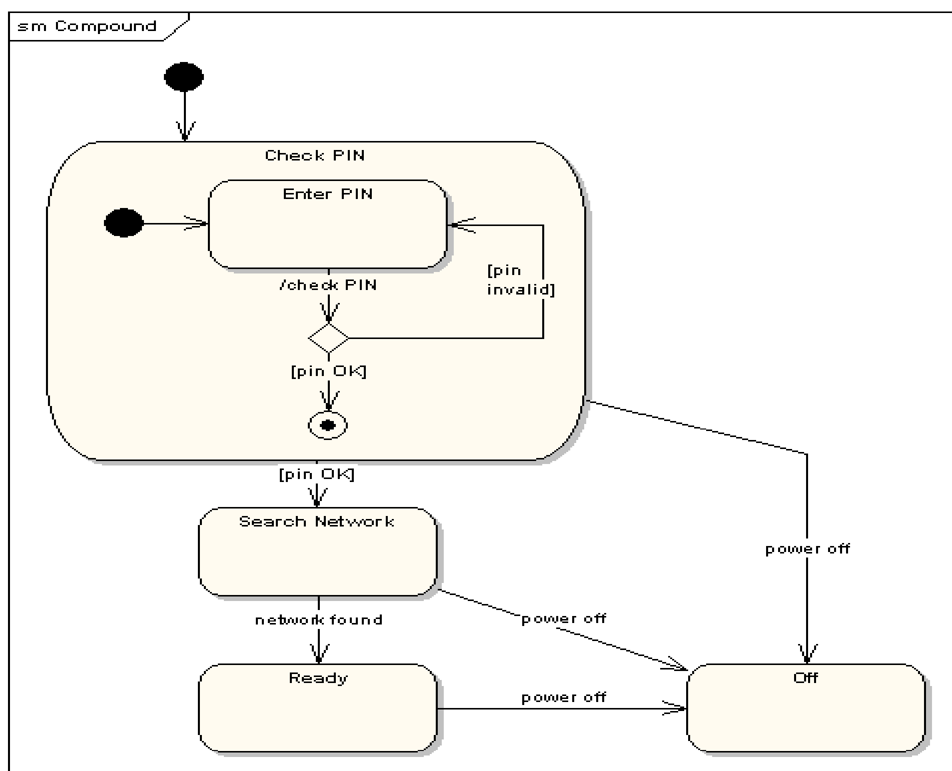


It is also possible to define actions that occur on events, or actions that always occur. It is possible to define any number of action of each type.

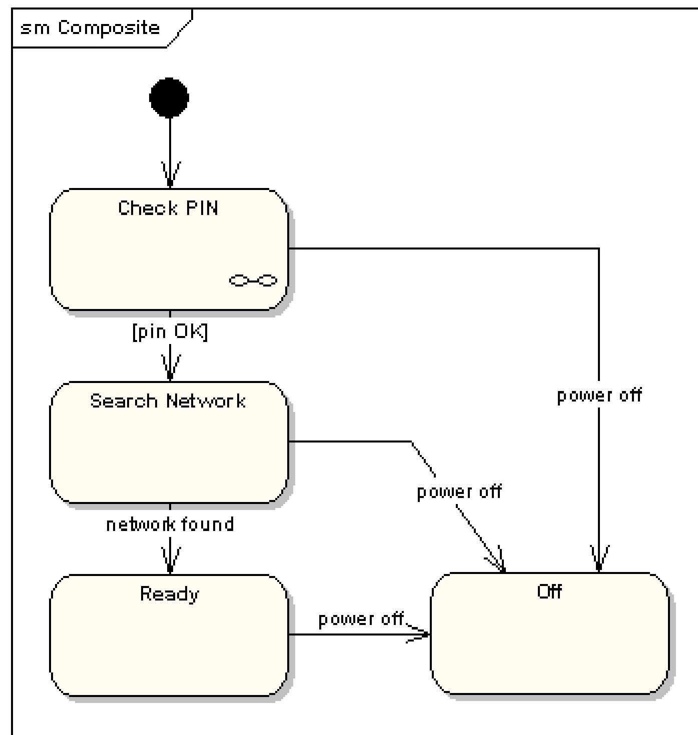
Self-Transitions
A state can have a transition that returns to itself, as in the following diagram. This is most useful when an effect is associated wit the transition.



Compound States
A state machine diagram may include sub-machine diagrams, as in the example below.



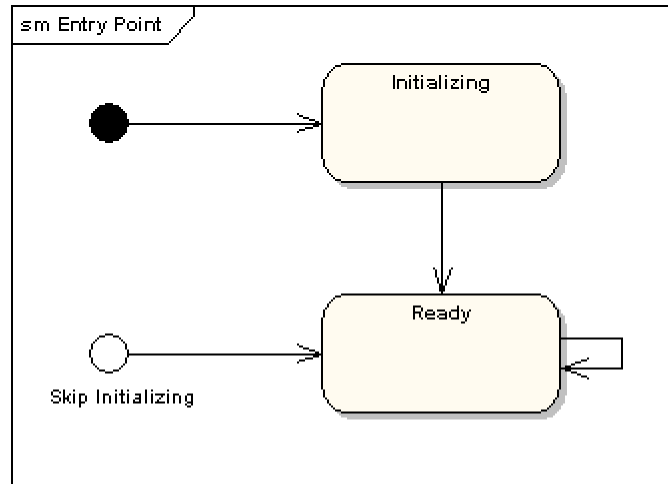
The alternative way to show the same information is as follows.



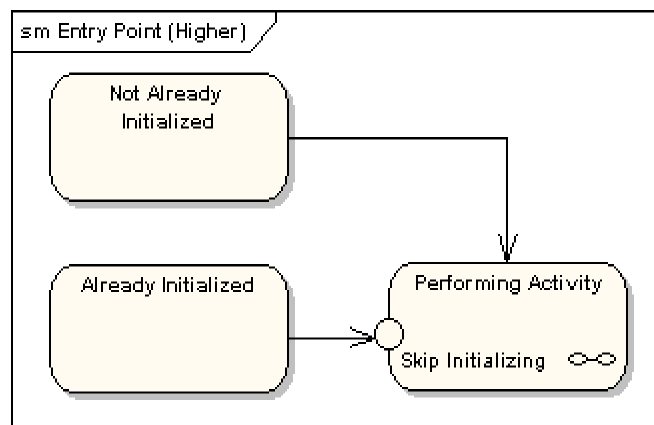
The notation in the above version indicates that the details of the Check PIN sub-machine are shown in a separate diagram.

Entry Point

Sometimes you won't want to enter a sub-machine at the normal Initial State. For example, in the following sub-machine it would be normal to begin in the Initializing state, but if for some reason it wasn't necessary to perform the initialization, it would be possible to begin in the Ready state by transitioning to the named Entry Point.

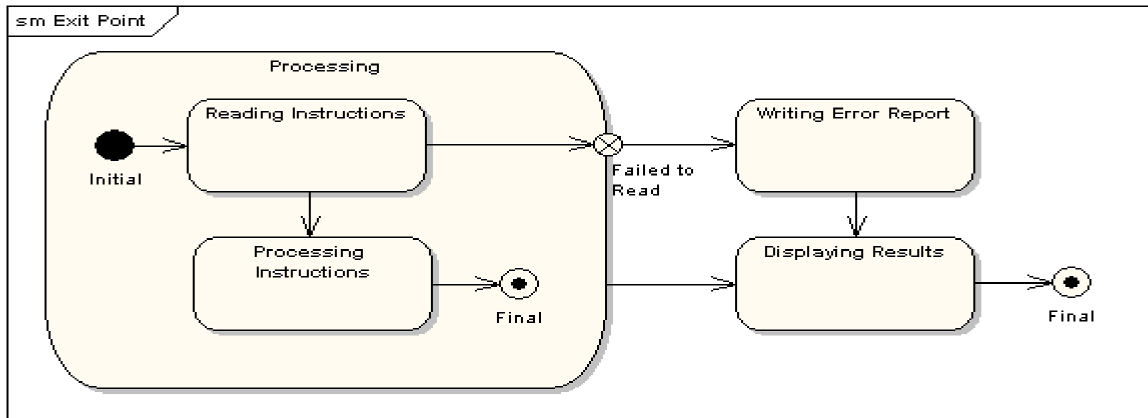


The following diagram shows the state machine one level up:



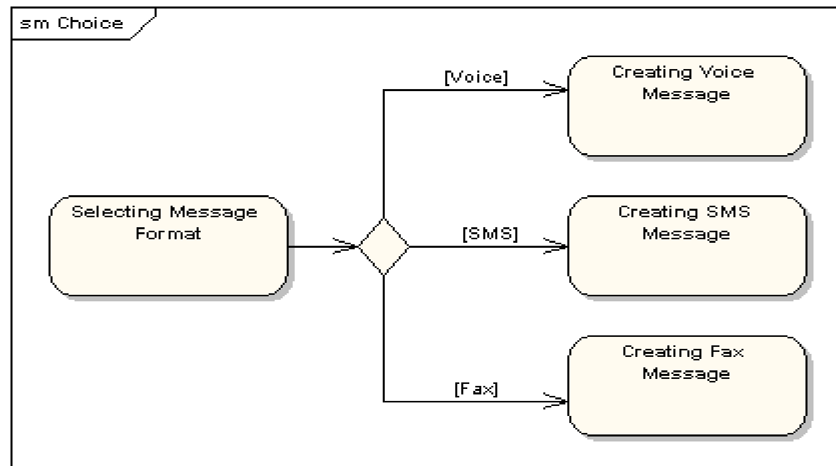
Exit Point

In a similar manner to Entry Points, it is possible to have named alternative Exit Points. The following diagram gives an example where the state executed after the main processing state depends on which route is used to transition out of the state.



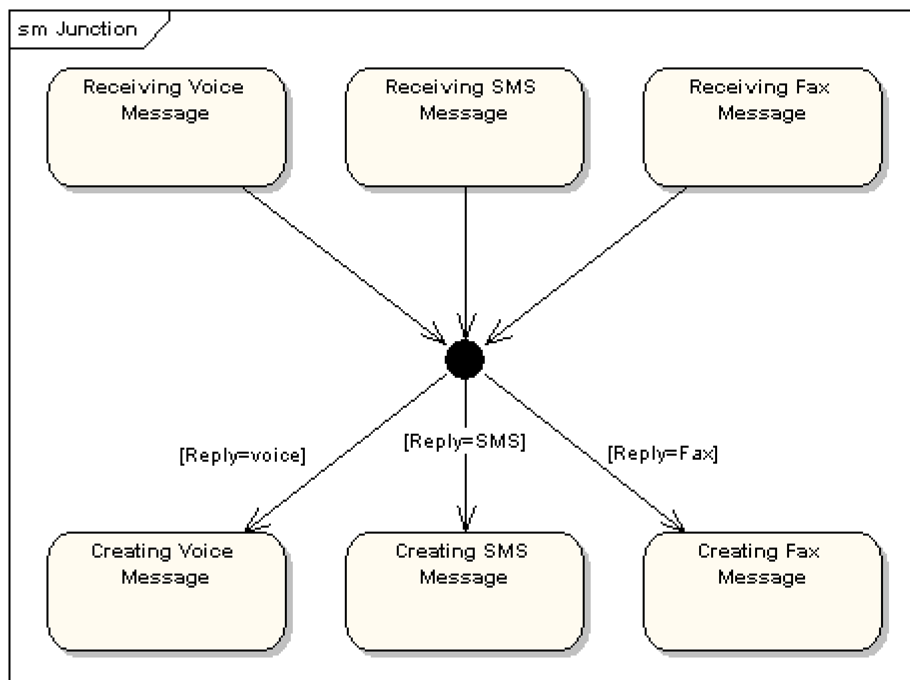
Choice Pseudo-State

A choice pseudo-state is shown as a diamond with one transition arriving and two or more transitions leaving. The following diagram shows that whichever state is arrived at after the choice pseudo-state is dependent on the message format selected durir execution of the previous state.



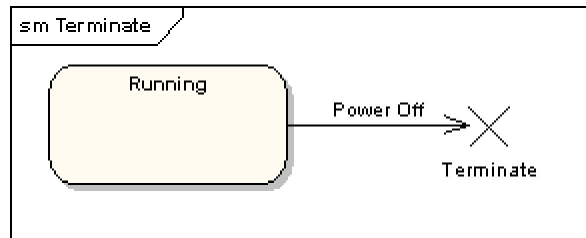
Junction Pseudo-State

Junction pseudo-states are used to chain together multiple transitions. A single junction can have one or more incoming and one c more outgoing transitions and a guard can be applied to each transition. Junctions are semantic-free; a junction which splits an incoming transition into multiple outgoing transitions realizes a static conditional branch as opposed to a choice pseudo-state whic realizes a dynamic conditional branch.



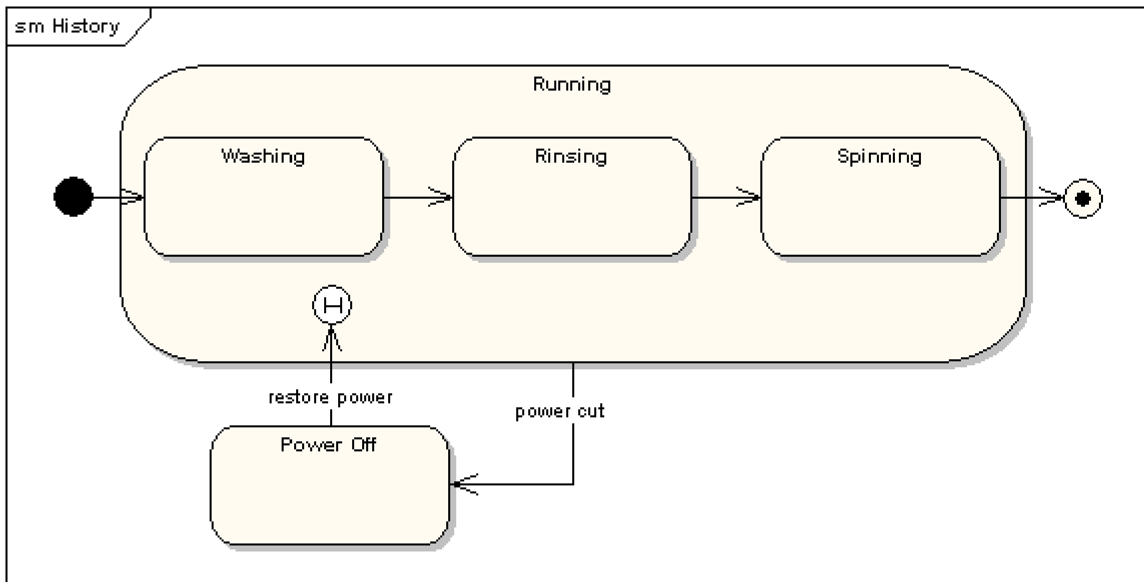
Terminate Pseudo-State

Entering a terminate pseudo-state indicates that the lifeline of the state machine has ended. A terminate pseudo-state is notated a
a cross.



History States

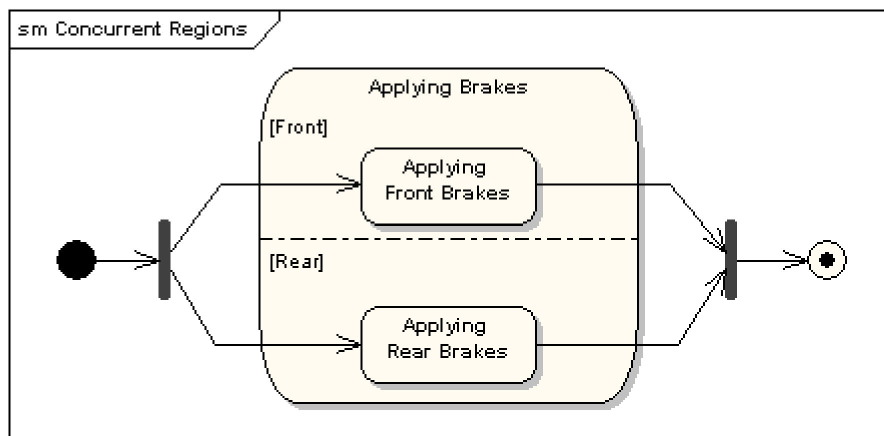
A History State is used to remember the previous state of a state machine when it was interrupted. The following diagram illustrates the use of history states. The example is a state machine belonging to a washing machine.



In this state machine, when a washing machine is running it will progress from Washing through Rinsing to Spinning. If there is a power cut, the washing machine will stop running and will go to the Power Off state. Then when the power is restored, the Runnin state is entered at the History State symbol meaning that it should resume where it last left-off.

Concurrent Regions

A state may be divided into regions containing sub-states that exist and execute concurrently. The example below shows that within the state "Applying Brakes", the front and rear brakes will be operating simultaneously and independently. Notice the use o fork and join pseudo-states rather than choice and merge pseudo-states. These symbols are used to synchronize the concurrent threads.



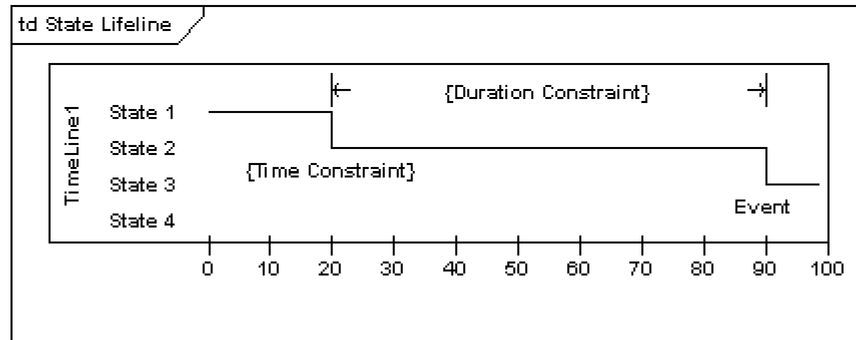
UML 2 Timing Diagram

Timing Diagrams

UML timing diagrams are used to display the change in state or value of one or more elements over time. It can also show the interaction between timed events and the time and duration constraints that govern them.

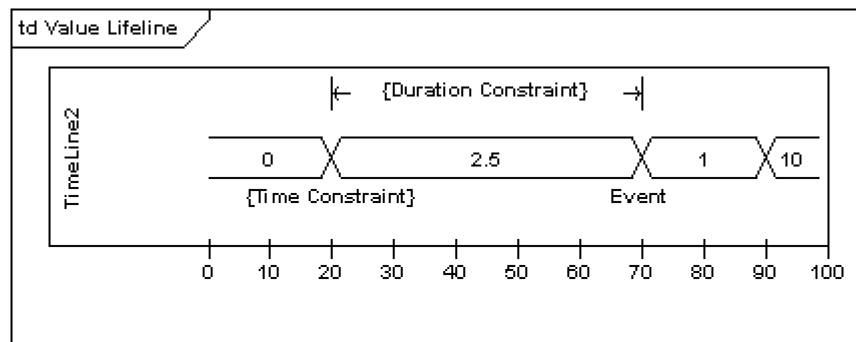
State Lifeline

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



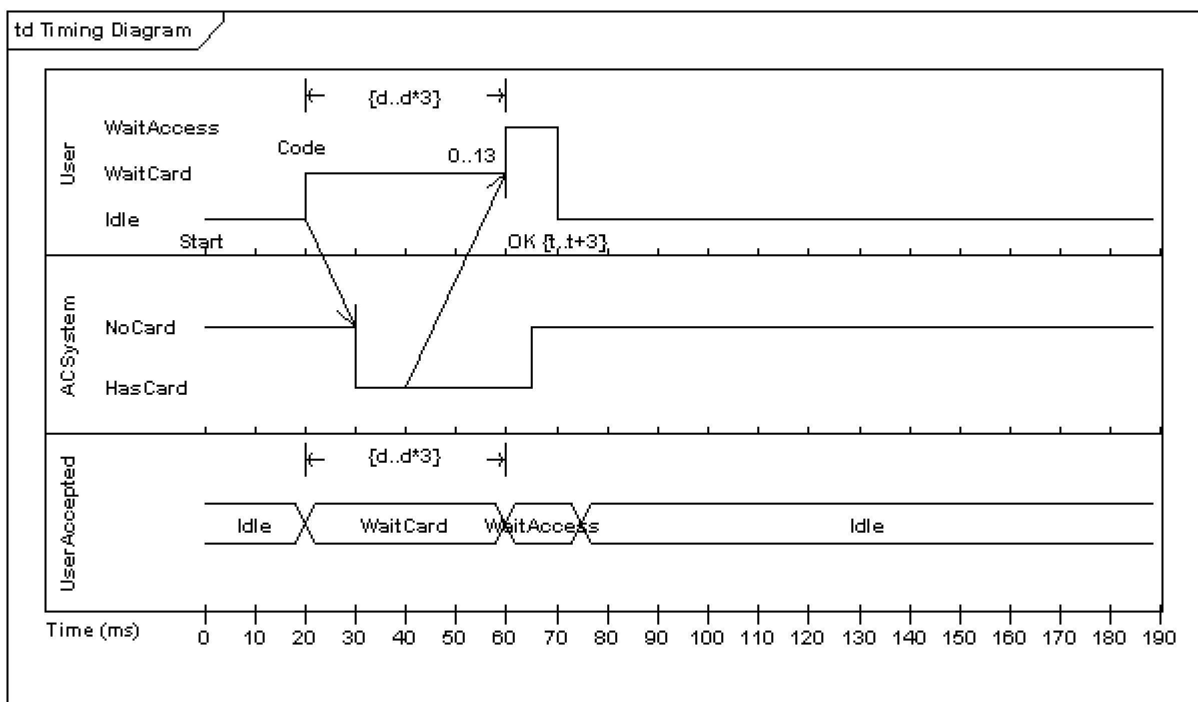
Value Lifeline

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. A value lifeline is shown below.



Putting it all together

State and Value Lifelines can be stacked one on top of another in any combination. They must have the same X-axis. Messages ca 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. Once these have all been applied, a timing diagram may look like the following.



UML 2 Use Case Diagram

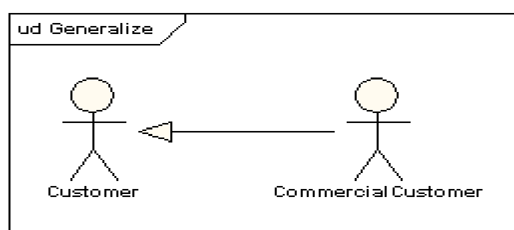
Use Case Model

The Use Case Model captures the requirements of a system. Use cases are a means of communicating with users and other stakeholders about what the system is intended to do.

Actors

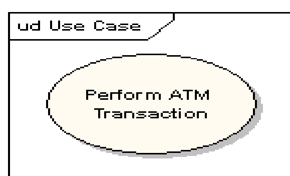
A Use Case Diagram shows the interaction between the system and entities external to the system. These external entities are referred to as Actors. Actors represent roles which may include human users, external hardware or other systems. An actors is usually drawn as a named stick figure, or alternatively as a class rectangle with the «actor» keyword.

Actors can generalize other actors as detailed in the following diagram:

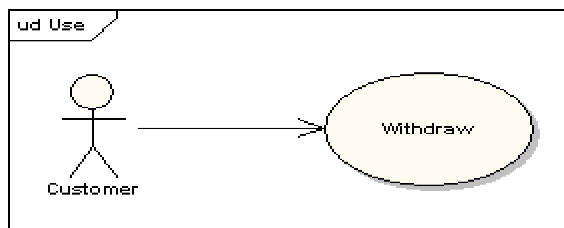


Use Cases

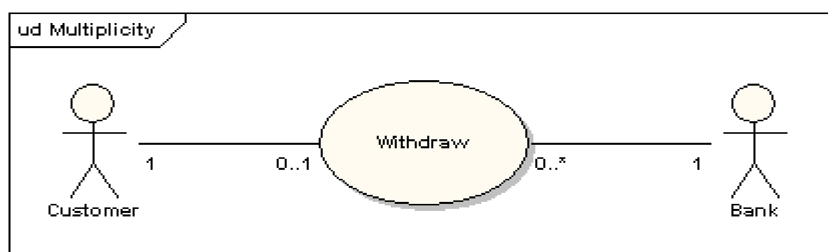
A use case is a single unit of meaningful work. It provides a high-level view of behavior observable to someone or something outside the system. The notation for a use case is an ellipse.



The notation for using a use case is a connecting line with an optional arrowhead showing the direction of control. The following diagram indicates that the actor Customer uses the Withdraw use case.



The uses connector can optionally have multiplicity values at each end, as in the following diagram which shows that a customer may only have one withdrawal session at a time, but a bank may have any number of customers making withdrawals concurrently



Use Case Definition

A Use Case Typically Includes:

- Name and Description
- Requirements
- Constraints
- Scenarios
- Scenario Diagrams
- Additional information.

Name and Description

A use case is normally named as a verb-phrase and given a brief informal textual description.

Requirements

The requirements define the formal functional requirements that a use case must supply to the end user. They correspond to the functional specifications found in structured methodologies. A requirement is a contract or promise that the Use Case will perform an action or provide some value to the system.

Constraints

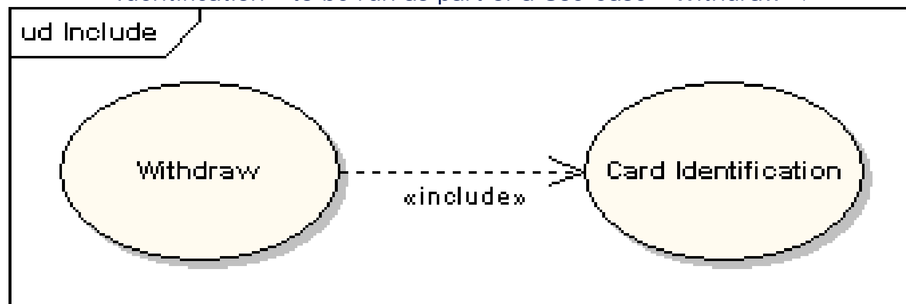
A constraint is a condition or restriction that a Use Case operates under and includes pre, post and invariant conditions. A precondition specifies the conditions that need to be met before the Use Case can proceed. A post condition is used to document the change in conditions that must be true after the execution of the Use Case. An invariant condition specifies the conditions that are true throughout the execution of the Use Case

Scenarios

A Scenario is a formal description of the flow of events that occur during the execution of a Use Case instance. It defines the specific sequence of events between the system and the external Actors. It is normally described in text and corresponds to the textual representation of the Sequence Diagram.

Including Use Cases

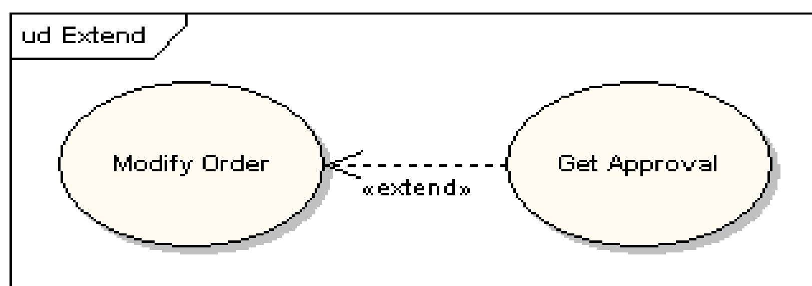
Uses Cases may contain the functionality of another Use Case as part of their normal processing. In general it is assumed that an included use case will be called every time the basic path is run. An example of this is to have the execution of the Use Case <Car Identification> to be run as part of a Use Case <Withdraw>.



Use Cases may be included by one or more Use Case, helping to reduce the level of duplication of functionality by factoring out common behavior into Use Cases that are re-used many times.

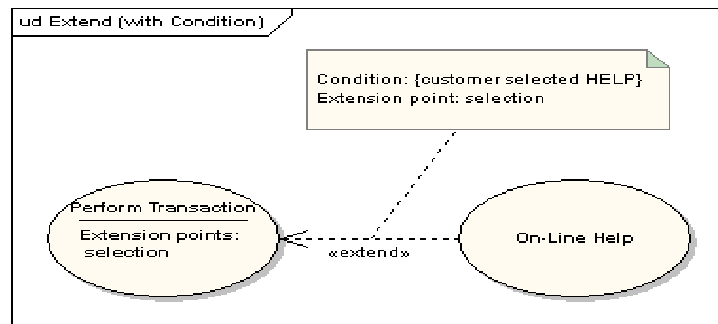
Extending Use Cases

One Use Case may be used to extend the behavior of another; this is typically used in exceptional circumstances. For example, if before modifying a particular type of customer order, a user must get approval from some higher authority, then the <Get Approval> Use Case may optionally extend the regular <Modify Order> Use Case.



Extension Points

The point at which an extending use case is added can be defined by means of an extension point.



System Boundary

It is usual to display use cases as being inside the system and actors as being outside the system.