UML

The Unified Modeling Language (UML) has quickly become the de-facto standard for building Object-Oriented software. This topic provides a technical overview of the 13 UML diagrams supported by Enterprise Architect.

Firstly... What is UML?

The Object Management Group (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 (such as the Rational Unified Process) - but in itself it does not specify that methodology or process.

UML defines the notation and semantics for the following domains:

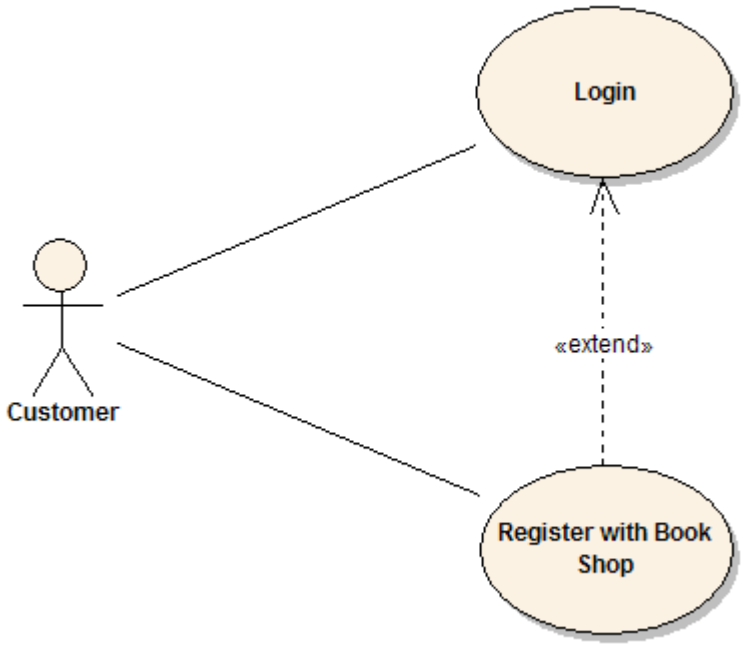
- The User Interaction or Use Case Model - describes the boundary and interaction between the system and users. Corresponds in some respects to a requirements model.
- The Interaction or Communication Model - describes how objects in the system will interact with each other to get work done.
- The State or Dynamic Model - State charts describe the states or conditions that classes assume over time. Activity graphs describe the workflows 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.

The UML also defines extension mechanisms for extending the UML to meet specialized needs (for example Business Process Modeling extensions).

The Use Case Model

A Use Case Model describes the proposed functionality of a new system. A Use Case represents a discrete unit of interaction between a user (human or machine) and the system. This interaction is a single unit of meaningful work, such as Create Account or View Account Details.

Each Use Case describes the functionality to be built in the proposed system, which can include another Use Case's functionality or extend another Use Case with its own behavior.



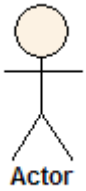
A Use Case description will generally includes:

- General comments and notes describing the use case.
- Requirements - The formal functional requirements of things that a Use Case must provide to the end user, such as <ability to update order>. These correspond to the functional specifications found in structured methodologies, and form a contract that the Use Case performs some action or provides some value to the system.
- Constraints - The formal rules and limitations a Use Case operates under, defining what can and cannot be done. These include:
    - Pre-conditions that must have already occurred or be in place before the use case is run; for example, <create order> must precede <modify order>
    - Post-conditions that must be true once the Use Case is complete; for example, <order is modified and consistent>

    - Invariants that must always be true throughout the time the Use Case operates; for example, an order must always have a customer number.
- Scenarios – Formal, sequential descriptions of the steps taken to carry out the use case, or the flow of events that occur during a Use Case instance. These can include multiple scenarios, to cater for exceptional circumstances and alternative processing paths. These are usually created in text and correspond to a textual representation of the Sequence Diagram.
- Scenario diagrams - Sequence diagrams to depict the workflow; similar to Scenarios but graphically portrayed.
- Additional attributes, such as implementation phase, version number, complexity rating, stereotype and status.

Actors

Use Cases are typically related to 'actors', which are human or machine entities that use or interact with the system to perform a piece of meaningful work that helps them to achieve a goal. The set of Use Cases an actor has access to defines their overall role in the system and the scope of their action.



Includes and Extends relationships between Use Cases

One Use Case could include the functionality of another as part of its normal processing. Generally, it is assumed that the included Use Case is called every time the basic path is run. For example, when listing a set of customer orders to choose from before modifying a selected order, the <list orders> Use Case would be included every time the <modify order> Use Case is run.

A Use Case can be included by one or more other Use Cases, so it helps to reduce duplication of functionality by factoring out common behavior into Use Cases that are re-used many times.

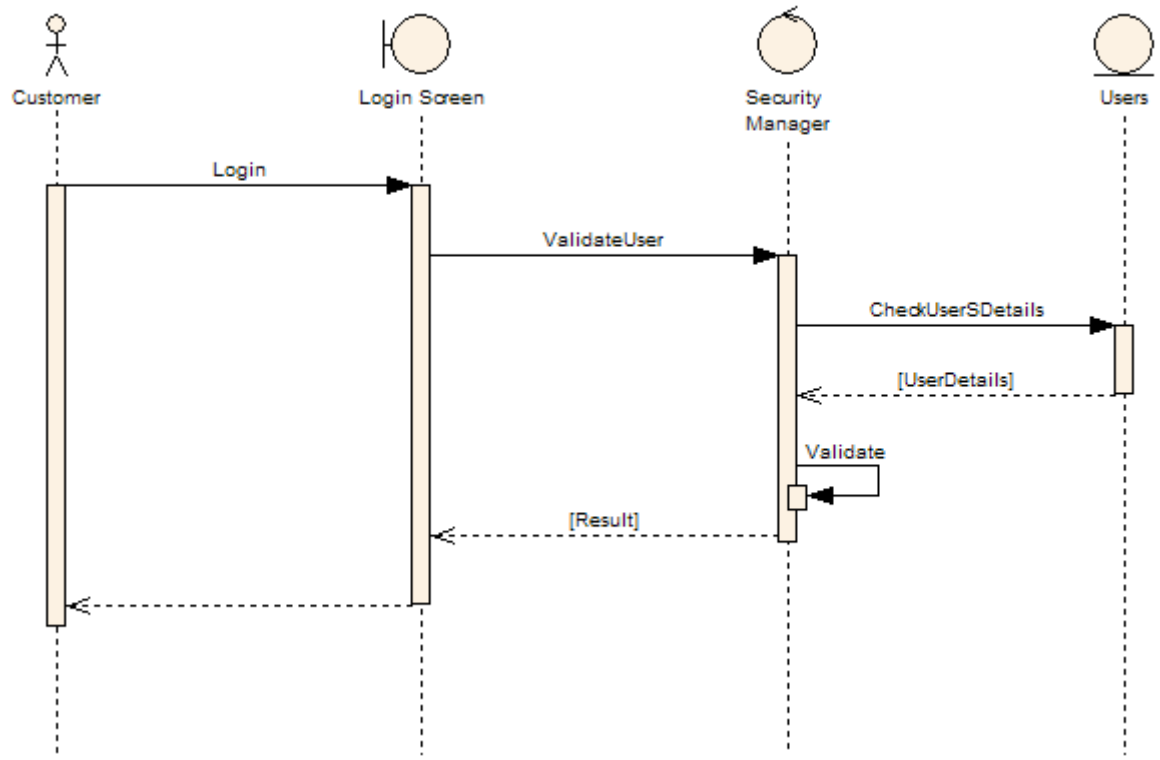
One Use Case can extend the behavior of another, typically when exceptional circumstances are encountered. For example, if a user must get approval from some higher authority before modifying a particular type of customer order, then the <get approval> Use Case could optionally extend the regular <modify order> Use Case.

Sequence Diagrams

Sequence diagrams provide a graphical representation of object interactions over time. These typically show a user or actor, and the objects and components they interact with in the execution of a use case. One sequence diagram typically represents a single Use Case 'scenario' or flow of events.

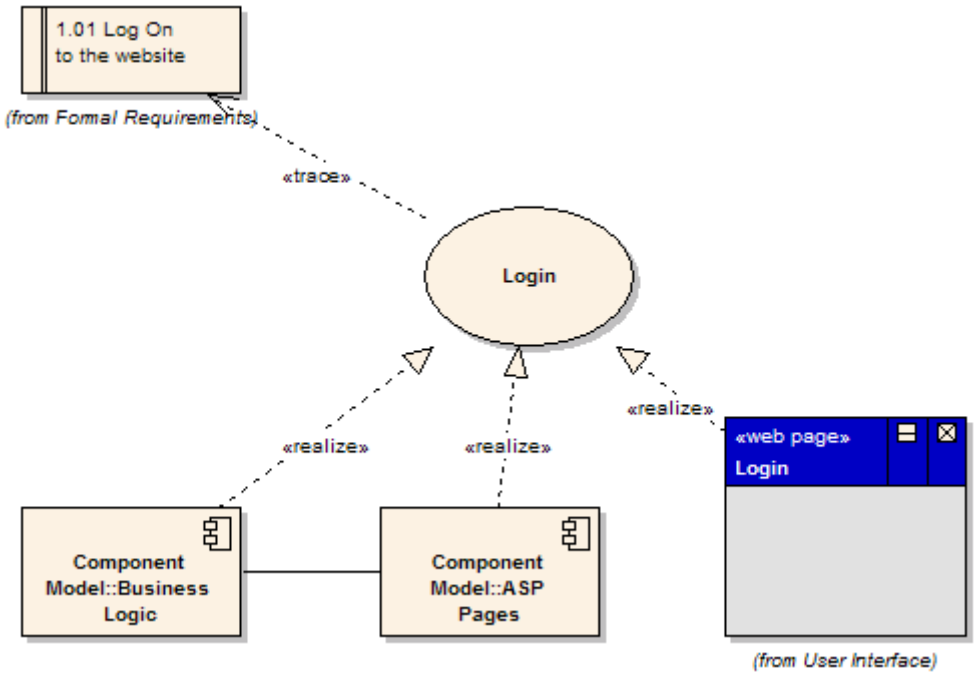
Sequence diagrams are an excellent way of documenting usage scenarios and both capturing required objects early in analysis and verifying object use later in design. The diagrams show the flow of messages from one object to another, and as such correspond to the methods and events supported by a class/object.

The following example of a sequence diagram shows the user or actor on the left initiating a flow of events and messages that correspond to the Use Case scenario. The messages that pass between objects become class operations in the final model.



Implementation Diagram

A Use Case is a formal description of functionality that the system will have when constructed. An implementation diagram is typically associated with a Use Case to document which design elements (for example, components and classes) implement the Use Case functionality in the new system. This provides a high level of traceability for the system designer, the customer and the team that will actually build the system. The list of Use Cases that a component or class is linked to documents the minimum functionality that must be implemented by the component.



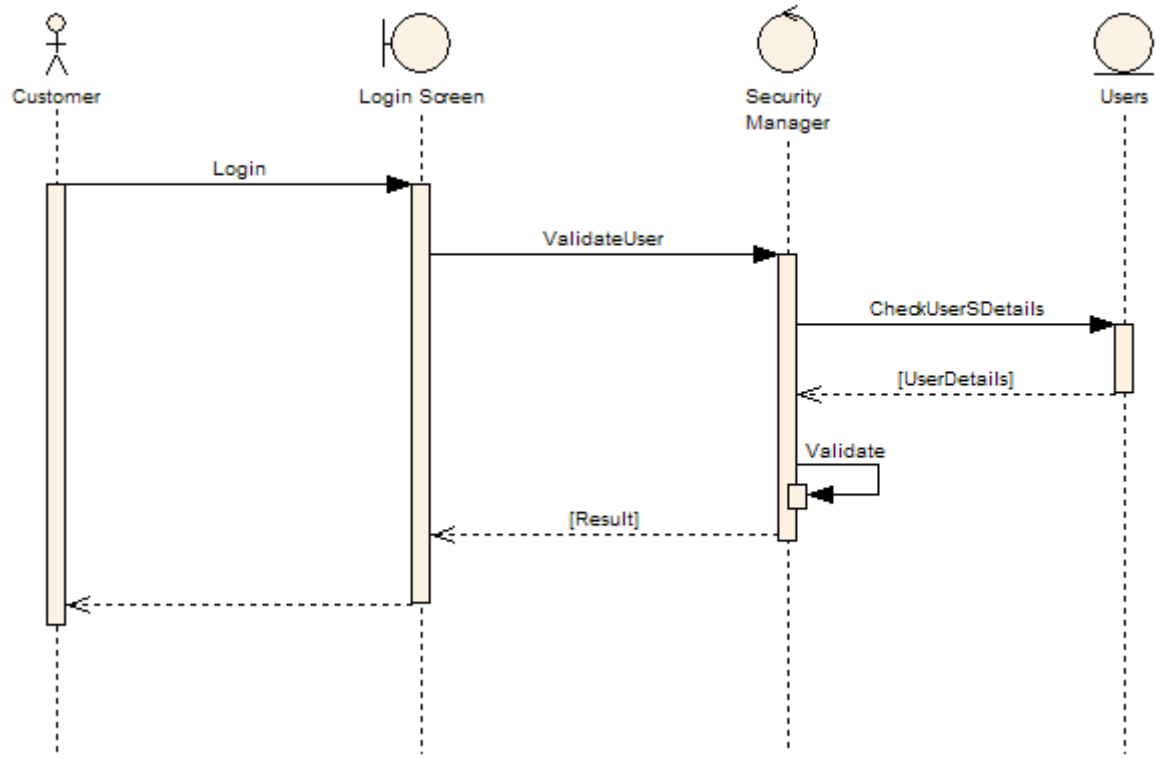
The example above shows that the use case 'Login' implements the formal requirement '1.01 Log On to the website'. It also shows that the 'Business Logic' component and 'ASP Pages' component implement some or all of the 'Login' functionality. A further refinement is to show the 'Login' screen (a web page) as implementing the 'Login' use case. These implementation or realization links define the traceability from the formal requirements, through use cases on to components and screens.

The Dynamic Model

The dynamic model is used to express and model the behaviour of the system over time. It includes support for activity diagrams, state diagrams, sequence diagrams and extensions including business process modelling.

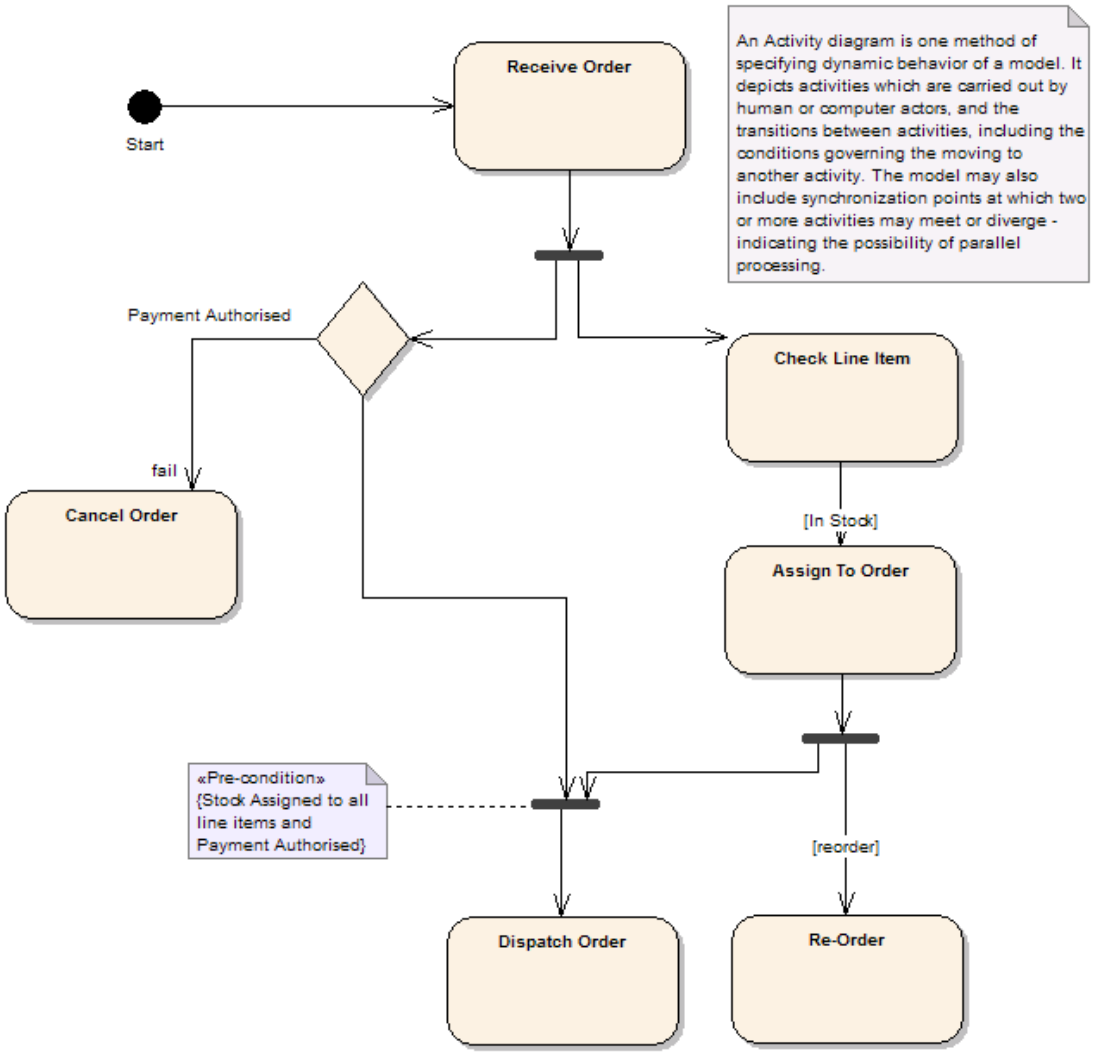
Sequence Diagrams

Sequence diagrams are used to display the interaction between users, screens, objects and entities within the system. It provides a sequential map of message passing between objects over time. Frequently these diagrams are placed under Use Cases in the model to illustrate the use case scenario - how a user will interact with the system and what happens internally to get the work done. Often, the objects are represented using special stereotyped icons, as in the example below. The object labelled Login Screen is shown using the User Interface icon. The object labelled SecurityManager is shown using the Controller icon. The Object labelled users is shown using the Entity icon.



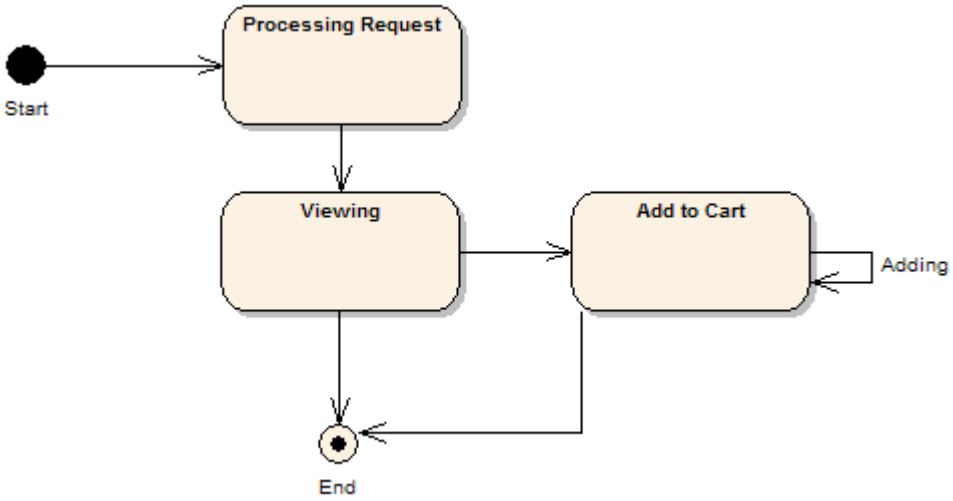
Activity Diagrams

Activity diagrams are used to show how different workflows in the system are constructed, how they start and the possibly many decision paths that can be taken from start to finish. They may also illustrate the where parallel processing may occur in the execution of some activities.



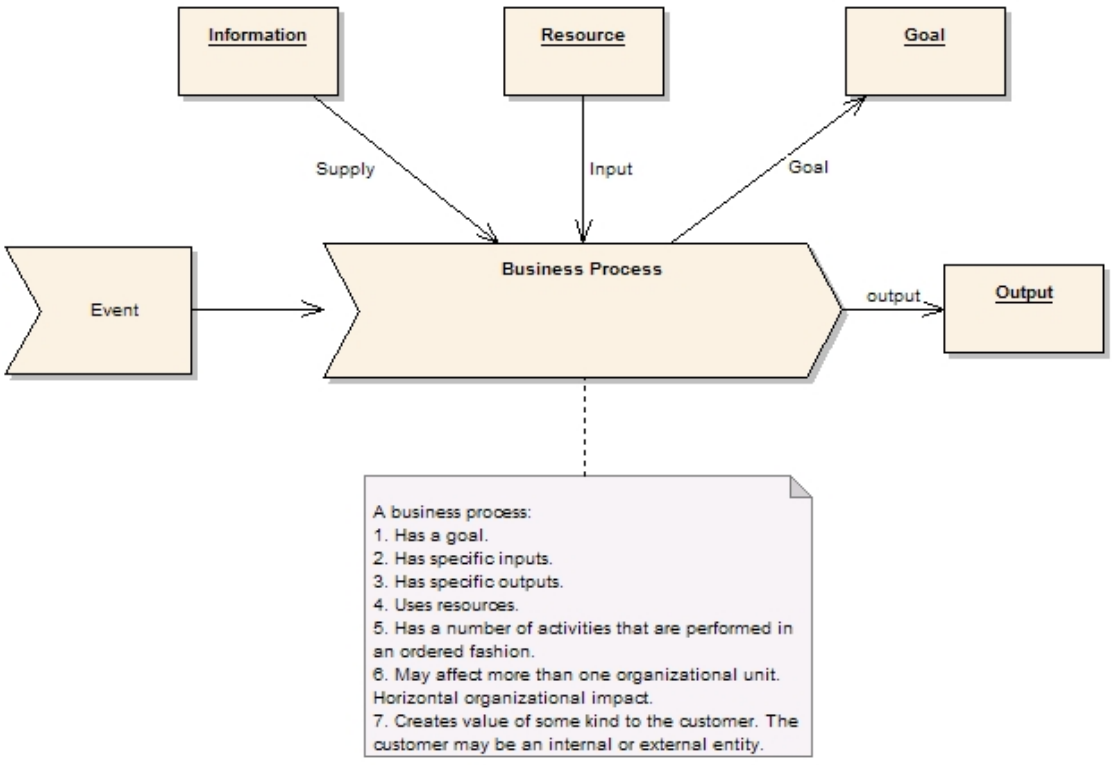
State Charts

State charts are used to detail the transitions or changes of state an object can go through in the system. They show how an object moves from one state to another and the rules that govern that change. State charts typically have a start and end condition.



Process Model

A process model is a UML extension of an activity diagram used to model a business process - this diagram shows what goal the process has, the inputs, outputs, events and information that are involved in the process.



The Logical Model

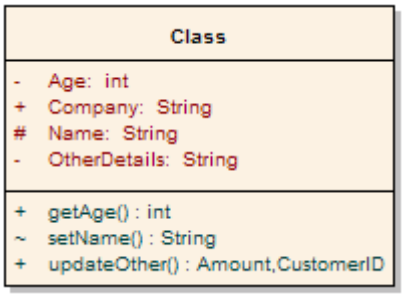
A logical model is a static view of the objects and classes that make up the design/analysis space. Typically, a Domain Model is a looser, high level view of Business Objects and entities, while the Class Model is a more rigorous and design focused model. This discussion relates mainly to the Class Model

The Class Model

A Class is a standard UML construct used to detail the pattern from which objects will be produced at run-time. A class is a specification - an object an instance of a class. Classes may be inherited from other classes (that is they inherit all the behavior and state of their parent and add new functionality of their own), have other classes as attributes, delegate responsibilities to other classes and implement abstract interfaces.

The Class Model is at the core of object-oriented development and design - it expresses both the persistent state of the system and the behavior of the system. A class encapsulates state (attributes) and offers services to manipulate that state (behavior). Good object-oriented design limits direct access to class attributes and offers services which manipulate attributes on behalf of the caller. This hiding of data and exposing of services ensures data updates are only done in one place and according to specific rules - for large systems the maintenance burden of code which has direct access to data elements in many places is extremely high.

The class is represented as below:



Note that the class has three distinct areas:

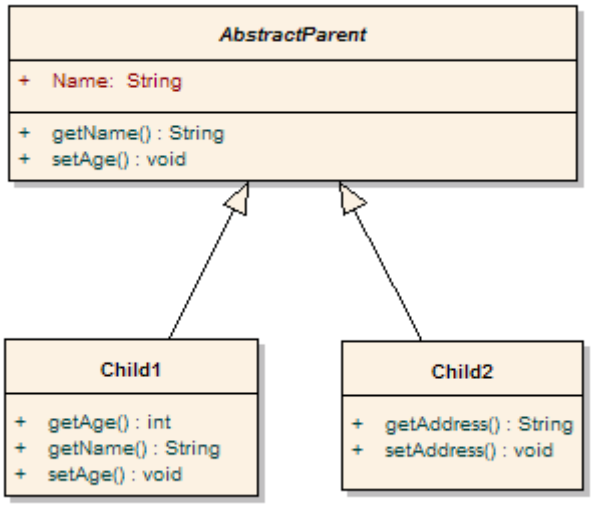
1. The class name (and stereotype if applied)
2. The class attributes area (that is internal data elements)
3. The behavior - both private and public

Attributes and methods may be marked as

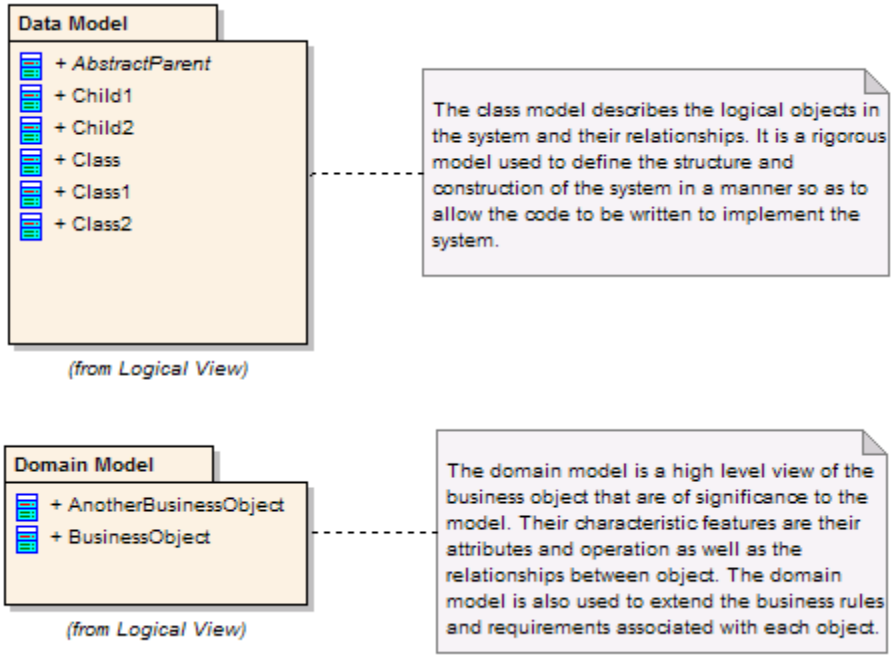
- Private, indicating they are not visible to callers outside the class
- Protected, they are only visible to children of the class
- Public, they are visible to all

Inheritance

Class inheritance is shown as below: an abstract class in this case, is the parent of two children, each of which inherits the base class features and extends it with their own behavior.



Class models may be collected into packages of related behavior and state. The diagram below illustrates this.

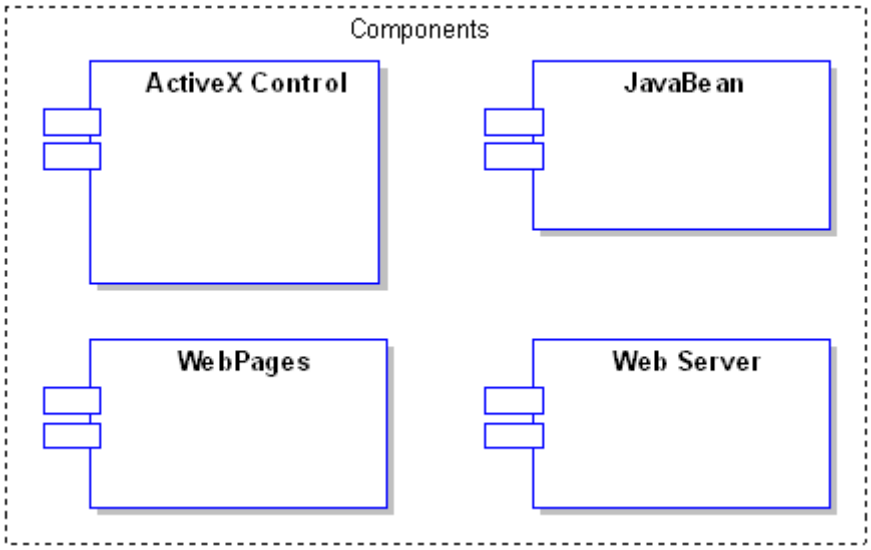


The Component Model

The component model illustrates the software components that will be used to build the system. These may be built up from the class model and written from scratch for the new system, or may be brought in from other projects and 3rd party vendors. Components are high level aggregations of smaller software pieces, and provide a 'black box' building block approach to software construction.

Component Notation

A component may be something like an ActiveX control - either a user interface control or a business rules server. Components are drawn as the following diagram shows:

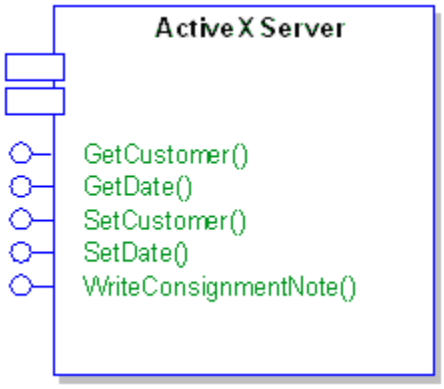


The Component Diagram

The component diagram shows the relationship between software components, their dependencies, communication, location and other conditions.

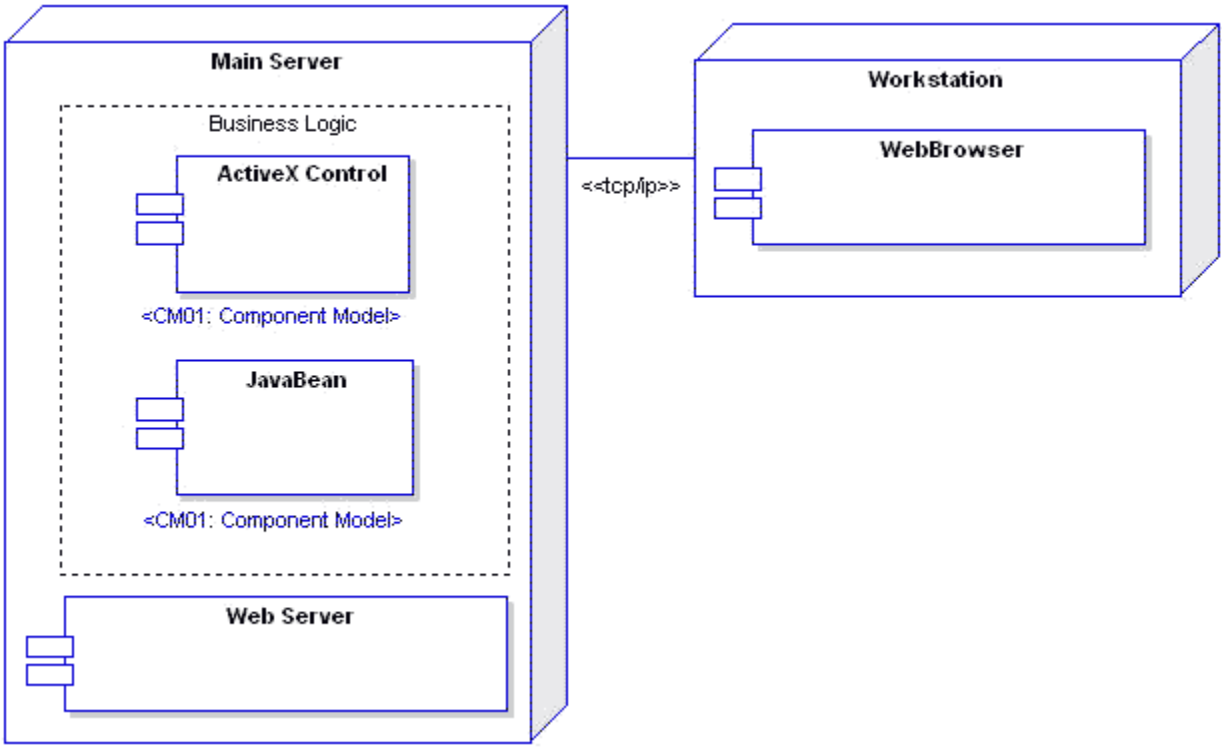
Interfaces

Components may also expose interfaces. These are the visible entry points or services that a component is advertising and making available to other software components and classes. Typically a component is made up of many internal classes and packages of classes. It may even be assembled from a collection of smaller components.



Components and Nodes

A deployment diagram illustrates the physical deployment of the system into a production (or test) environment. It shows where components will be located, on what servers, machines or hardware. It may illustrate network links, LAN bandwidth & etc.



Requirements

Components may have requirements attached to indicate their contractual obligations - that is, what service they will provide in the model. Requirements help document the functional behaviour of software elements.

Constraints

Components may have constraints attached which indicate the environment in which they operate. Pre-conditions specify what must be true before a component can perform some function; post-conditions indicate what will be true after a component has done some work and Invariants specify what must remain true for the duration of the components lifetime.

Scenarios

Scenarios are textual/procedural descriptions of an object's actions over time and describe the way in which a component works. Multiple scenarios may be created to describe the basic path (a perfect run through) as well as exceptions, errors and other conditions.

Traceability

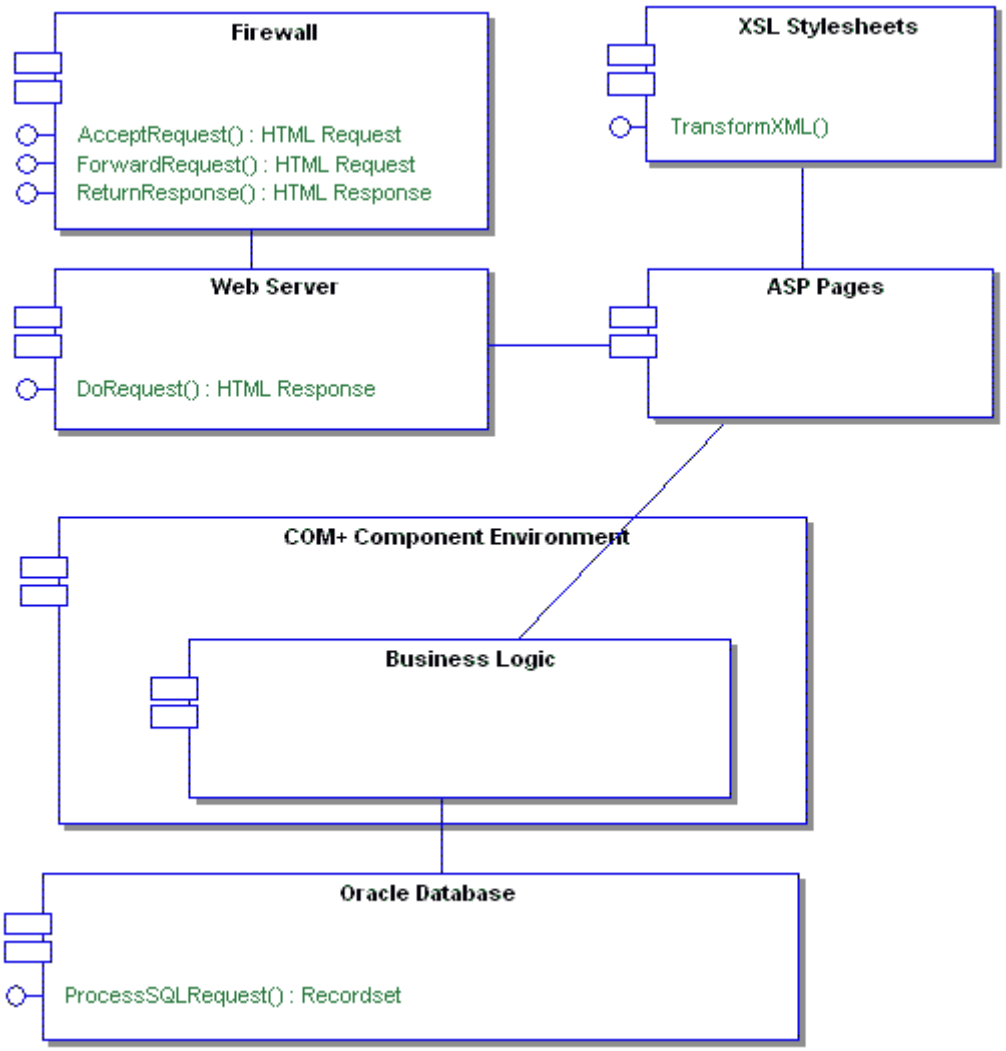
You may indicate traceability through realisation links. A component may implement another model element (eg. a use case) or a component may be implemented by another element (eg. a package of classes). By providing realisation links to and from components you can map the dependencies amongst model elements and the traceability from the initial requirements to the final implementation.

An Example

The following example shows how components may be linked to provide a conceptual/logical view of a systems construction. This example is concerned with the server and security elements of an on-line book store. It includes such elements as the web server, firewall, ASP pages & etc.

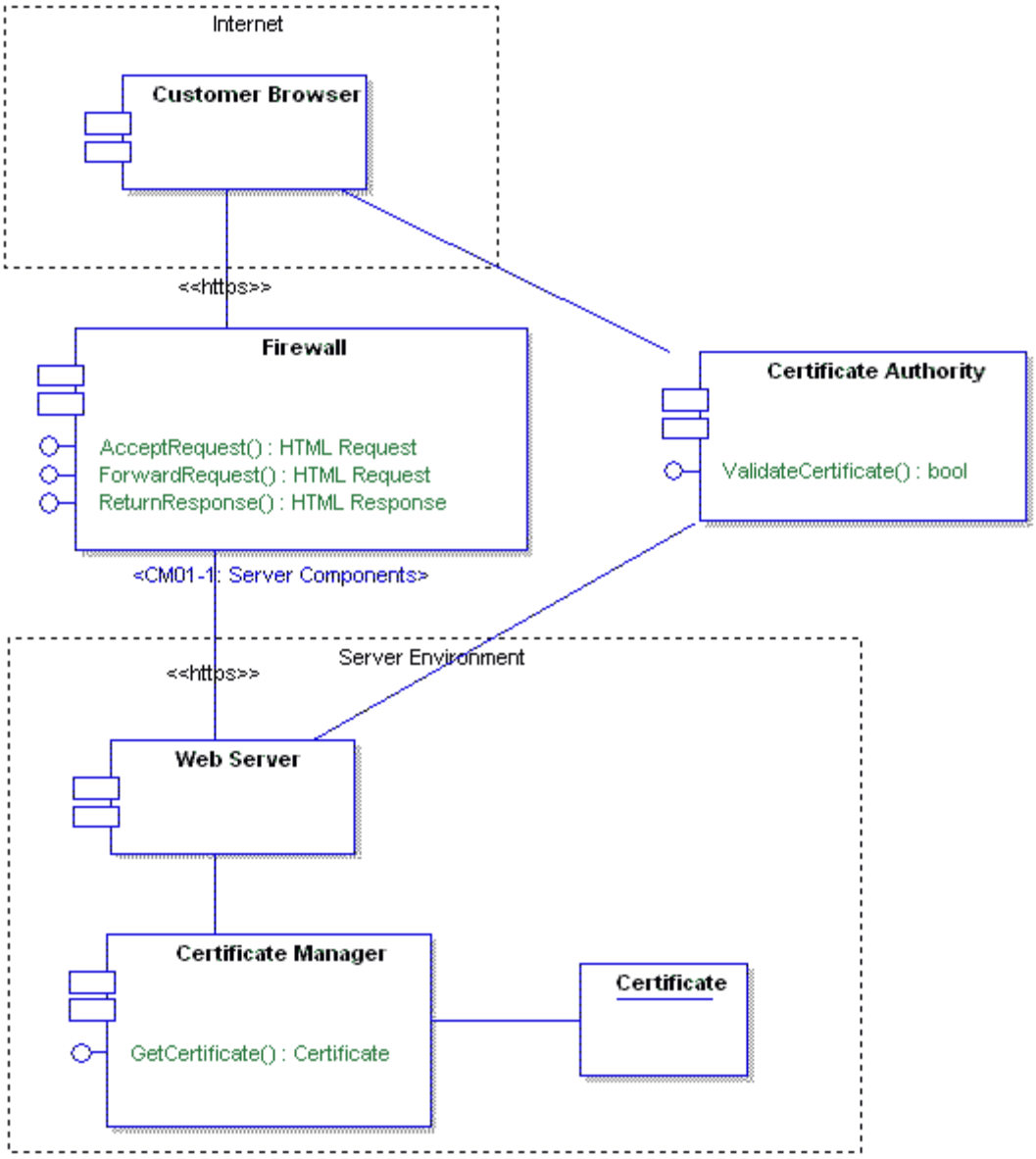
Server Components

This diagram illustrates the layout of the main server side components that will require building for an on-line book store. These components are a mixture of custom built and purchased items which will be assembled to provide the required functionality.



Security Components

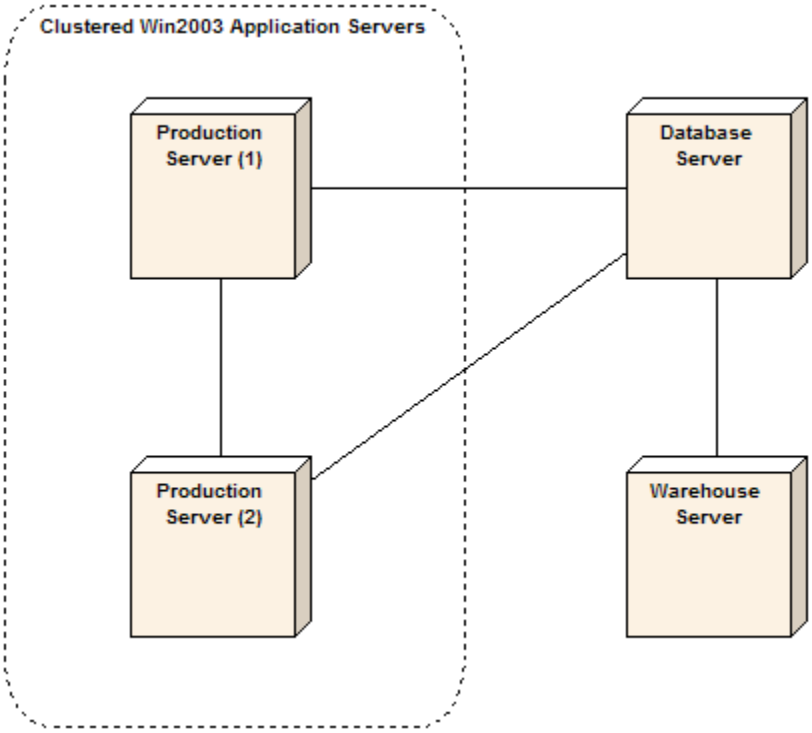
The security components diagram shows how security software such as the Certificate Authority, Browser, Web server and other model elements work together to assure security provisions in the proposed system.



The Physical Model

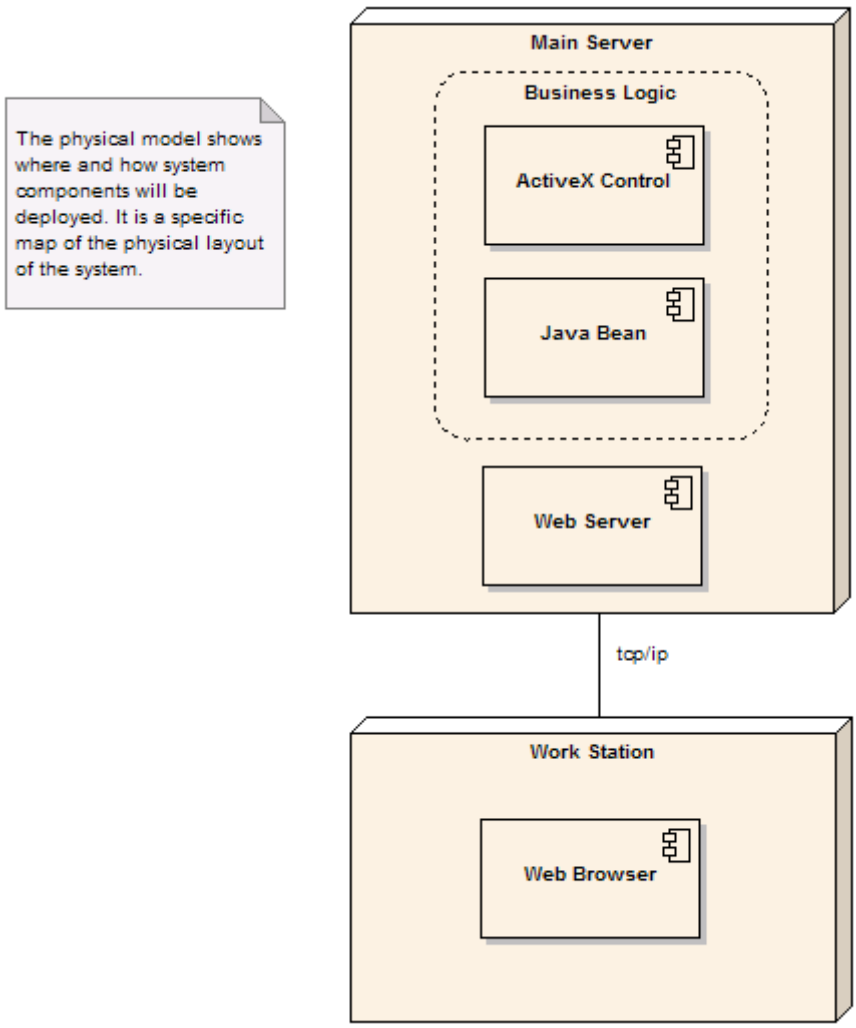
The Physical or Deployment Model provides a detailed model of the way components will be deployed across the system infrastructure. It details network capabilities, server specifications, hardware requirements and other information related to deploying the proposed system.

Deployment View



Physical Model

The physical model shows where and how system components will be deployed. It is a specific map of the physical layout of the system. A deployment diagram illustrates the physical deployment of the system into a production (or test) environment. It shows where components will be located, on what servers, machines or hardware. It may illustrate network links, LAN bandwidth & etc.



A node is used to depict any server, workstation or other host hardware used to deploy components into the production environment. You may also specify the links between nodes and assign stereotypes (such as TCP/IP) and requirements to them. Nodes may also have performance characteristics, minimum hardware standards, operating system levels & etc. documented. The screen below illustrates the common properties you can set for a node.

So... How do you use the UML?

The UML is typically used as a part of a software development process, with the support of a suitable CASE tool, to define the requirements, the interactions and the elements of the proposed software system. The exact nature of the process depends on the development methodology used. An example process might look something like the following:

1. A captured Business Process Model will be used to define the high level business activities and processes that occur in an organization and to provide a foundation for the Use Case model. The Business Process Model will typically capture more than a software system will implement (ie. it includes manual and other processes).
2. Map a Use Case Model to the Business Process Model to define exactly what functionality you are intending to provide from the business user perspective. As each

Use Case is added, create a traceable link from the appropriate business processes to the Use Case (ie. a realisation connection). This mapping clearly states what functionality the new system will provide to meet the business requirements outlined in the process model. It also ensures no Use Cases exist without a purpose.

3. Refine the Use Cases - include requirements, constraints, complexity rating, notes and scenarios. This information unambiguously describes what the Use Case does, how it is executed and the constraints on its execution. Make sure the Use Case still meets the business process requirements. Include the definition of system tests for each use case to define the aceptance criteria for each use case. Also include some user acceptance test scripts to define how the user will test this functionality and what the acceptance criteria are.
4. From the inputs and outputs of the Business Process Model and the details of the use cases, begin to construct a domain model (high level business objects), sequence diagrams, collaboration diagrams and user interface models. These describe the 'things' in the new system, the way those things interact and the interface a user will use to execute use case scenarios.
5. From the domain model, the user interface model and the scenario diagrams create the Class Model. This is a precise specification of the objects in the system, their data or attributes and their behaviour or operations. Domain objects may be abstracted into class hierarchies using inheritance. Scenario diagram messages will typically map to class operations. If an existing framework or design pattern is to be used, it may be possible to import existing model elements for use in the new system. For each class define unit tests and integration tests to thoroughly test i) that the class functions as specified internally and that ii) the class interacts with other related classes and components as expected.
6. As the Class Model develops it may be broken into discrete packages and components. A component represents a deployable chunk of software that collects the behaviour and data of one or more classes and exposes a strict interface to other consumers of its services. So from the Class Model a Component Model is built to define the logical packaging of classes. For each component define integration tests to confirm that the component's interface meets the specifcation given it in relation to other software elements.
7. Concurrent with the work you have already done, additional requirements should have been captured and documented. For example - Non Functional requirements, Performance requirements, Security requirements, responsibilities, release plans & etc. Collect these within the model and keep up to date as the model matures.
8. The Deployment model defines the physical architecture of the system. This work can be begun early to capture the physical deployment characteristics - what hardware, operating systems, network capabilities, interfaces and support software will make up the new system, where it will be deployed and what parameters apply to disaster recovery, reliability, back-ups and support. As the model develops the physical architecture will be updated to reflect the actual system being proposed.
9. Build the system: Take discrete pieces of the model and assign to one or more developers. In a Use Case driven build this will mean assigning a Use Case to the development team, having them build the screens, business objects, database tables, and related components necessary to execute that Use Case. As each Use Case is built it should be accompanied by completed unit, integration and system tests. A Component driven build may see discrete software components assigned to development teams for construction.

10. Track defects that emerge in the testing phases against the related model elements - eg. System test defects against Use Cases, Unit Test defects against classes & etc. Track any changes against the related model elements to manage 'scope creep'.
11. Update and refine the model as work proceeds - always assessing the impact of changes and model refinements on later work. Use an iterative approach to work through the design in discrete chunks, always assessing the current build, the forward requirements and any discoveries that come to light during development.
12. Deliver the complete and tested software into a test then production environment. If a phased delivery is being undertaken, then this migration of built sofware from test to production may occur several times over the life of the project.

Note: the above process is necessarily brief in description, and leaves much unsaid and while it may not be how you work or follow the process you have adopted. It is given as an example of how the UML may be used to support a software development project.

What is UML 2?

UML 2 advances the successful UML specification, and is quickly becoming the accepted standard for specifying, documenting and visualizing software systems. The Unified Modeling Language (UML) is also used for the modeling of non-software systems, and is extensively implemented in most industry sectors including finance, military and engineering.

UML is divided into two general sets and includes fourteen basic diagram types:

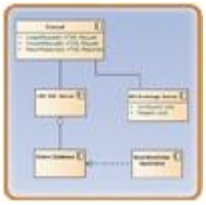
1. Structural Modeling Diagrams

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.

1. Package Diagrams

Package diagrams are used to divide the model into logical containers, or 'packages', and describe the interactions between them at a high level.

2. Component Diagrams



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.

3. Class or Structural Diagrams

Class or Structural diagrams define the basic building blocks of a model: the types, classes and general materials used to construct a full model.

4. Deployment Diagrams

Deployment diagrams Deployment diagrams show the physical disposition of significant artifacts within a real-world setting.

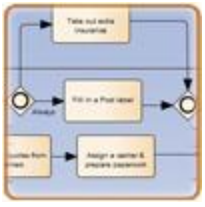
5. Composite Structure Diagrams

Composite Structure diagrams provide a means of layering an element's structure and focusing on inner detail, construction and relationships.

6. Object Diagrams

Object diagrams show how instances of structural elements are related and used at run-time.

7. Profile Diagrams



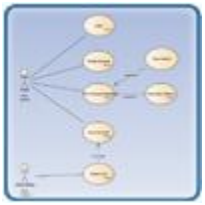
Profile diagrams provide a visual way of defining light-weight extensions to the UML specification. UML Profiles are often used to define a group of constructs with domain-specific or platform-specific properties and

constraints, which extend the underlying UML elements.

2. Behavioral Modeling Diagrams

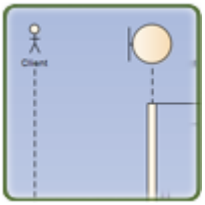
Behavior diagrams capture the varieties of interaction and instantaneous states within a model as it 'executes' over time; tracking how the system will act in a real-world environment, and observing the effects of an operation or event, including its results.

8. Use Case Diagrams



Use Case diagrams are used to model user/system interactions. They define behavior, requirements and constraints in the form of scripts or scenarios.

9. Sequence Diagrams

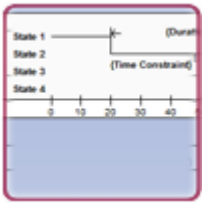


Sequence diagrams are closely related to communication diagrams and show the sequence of messages passed between objects using a vertical timeline.

10. Activity Diagrams

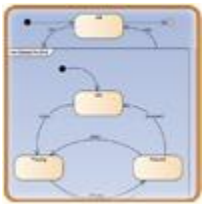
Activity diagrams have a wide number of uses, from defining basic program flow, to capturing the decision points and actions within any generalized process.

11. Timing Diagrams



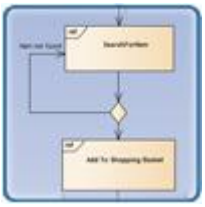
Timing diagrams fuse sequence and state diagrams to provide a view of an object's state over time, and messages which modify that state.

12. State Machine Diagrams



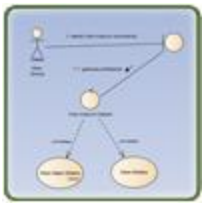
State Machine diagrams are essential to understanding the instant to instant condition, or "run state" of a model when it executes.

13. Interaction Overview Diagrams



Interaction Overview diagrams fuse activity and sequence diagrams to allow interaction fragments to be easily combined with decision points and flows.

14. Communication Diagrams



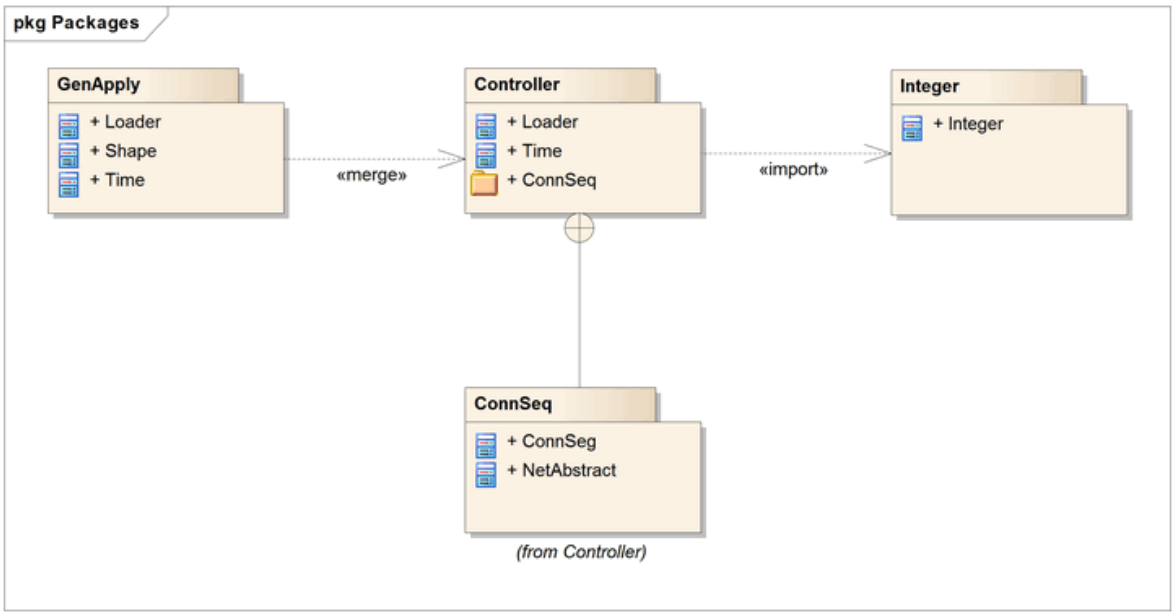
Communication diagrams show the network, and sequence, of messages or communications between objects at run-time, during a collaboration instance.

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 provide a visualization of the namespaces. The most common use for package diagrams is to organize use case diagrams and class diagrams, although the use of package diagrams is not limited to these UML elements.

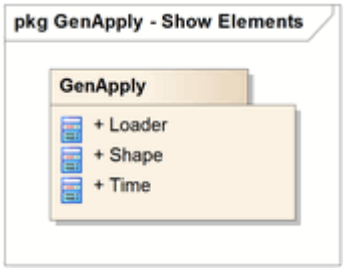
The following is an example of a package diagram.



Elements contained in a package share the same namespace. Therefore, the elements contained in a specific namespace must have unique names.

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

Packages are represented in UML 2.1 as folders and contain the elements that share a namespace; all elements within a package must be identifiable, and so have a unique name or type. The package must show the package name and can optionally show the elements within the package in extra compartments.



Package Merge

A «merge» connector between two packages defines an implicit generalization between elements in the source package, and elements with the same name in the target package. The source element definitions are expanded to include the element definitions contained in the target. The target element definitions are unaffected, as

are the definitions of source package elements that don't match names with any element in the target package.

Package Import

The «import» connector indicates that the elements within the target package, which in this example is a single class, use unqualified names when being referred to from the source package. The source package's namespace gains access to the target classes; the target's namespace is not affected.

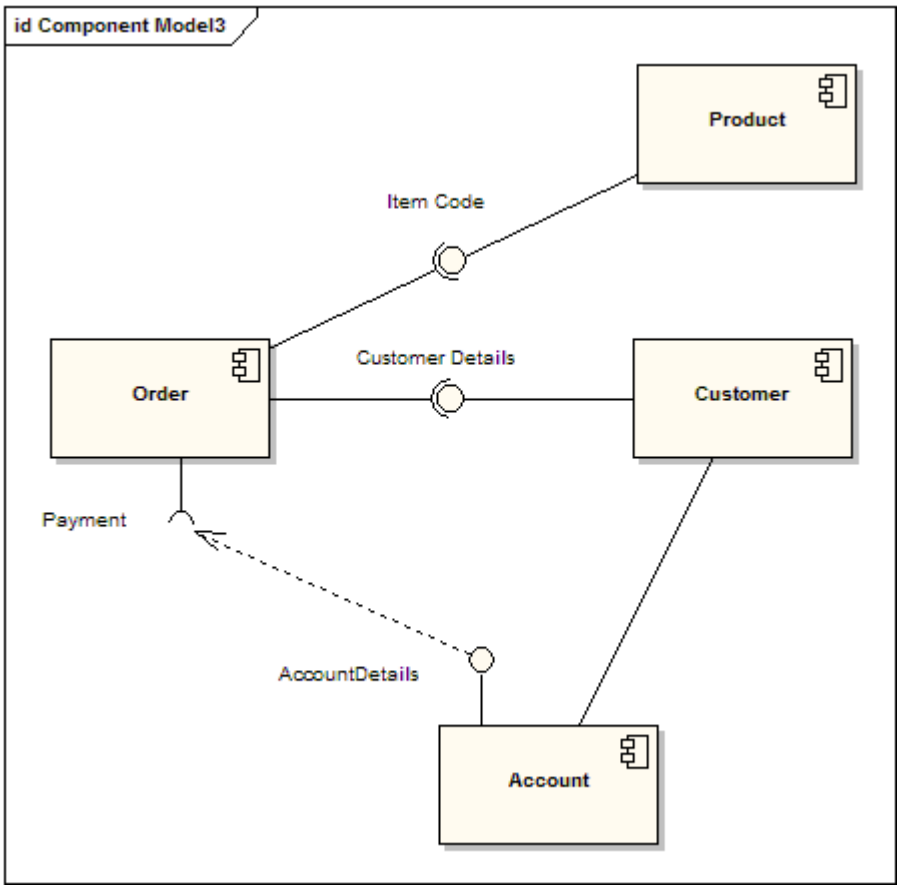
Nesting Connectors

The nesting connector between the target package and source packages shows that the source package is fully contained in the target package.

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 so a component can eventually encompass a large portion of a system.

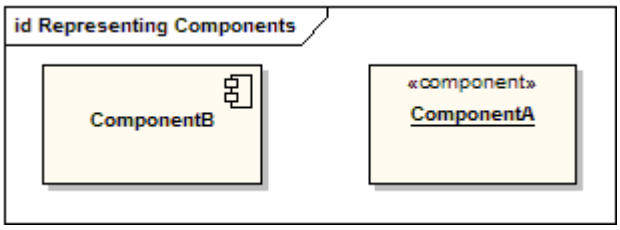


The diagram above demonstrates some components and their inter-relationships. Assembly connectors "link" the provided interfaces supplied by "Product" and "Customer" to the required interfaces specified by "Order". A dependency relationship maps a customer's associated account details to the required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

Components are similar in practice to package diagrams, as they 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.

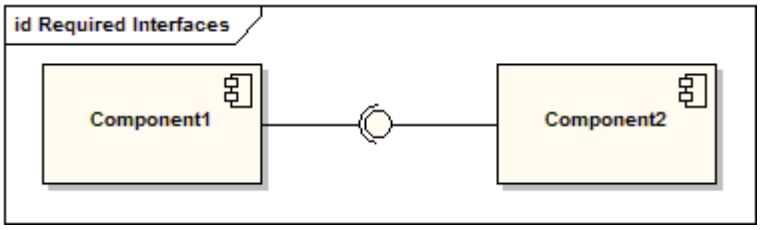
Representing Components

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.



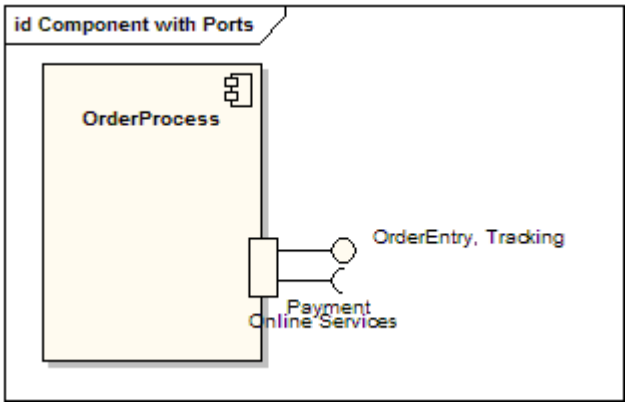
Assembly Connector

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.



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 and outputs as they can operate 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.

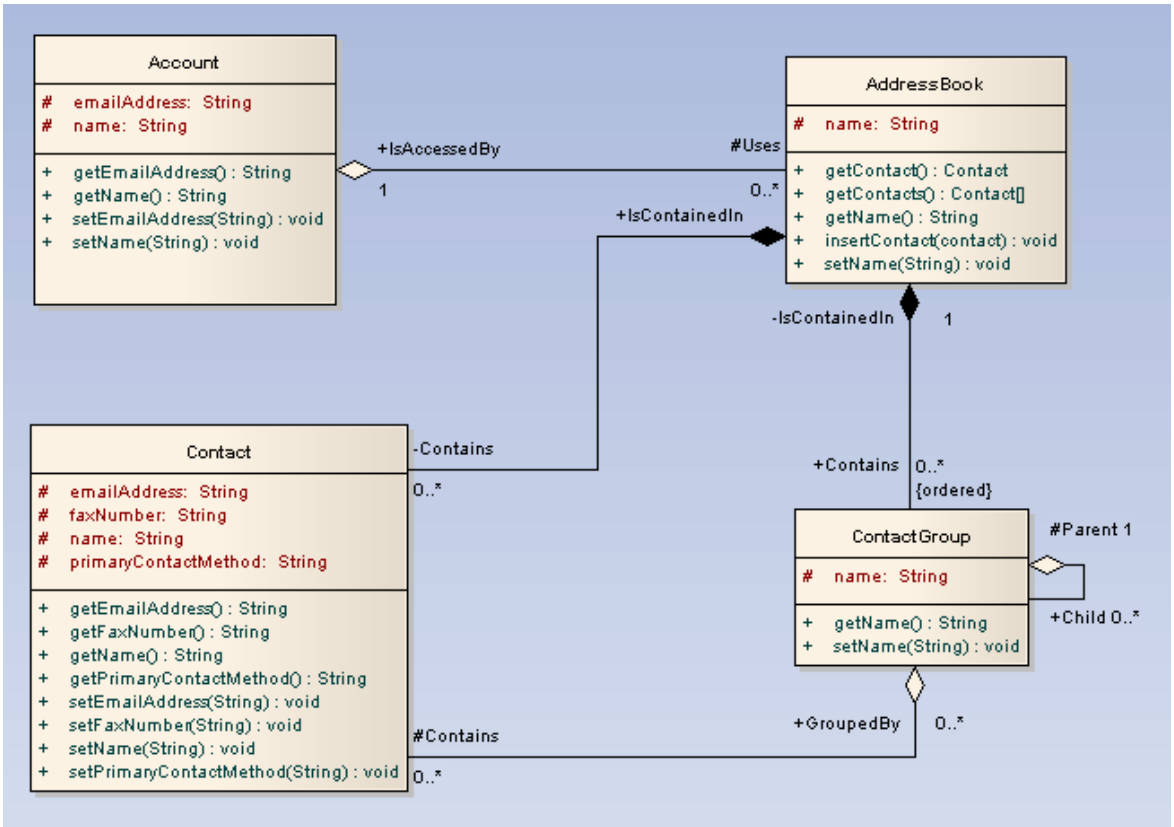


Class Diagram

Class Diagrams

The class diagram shows the building blocks of any object-orientated system. Class diagrams depict a static view of the model, or part of the model, describing what attributes and behavior it has rather than detailing the methods for achieving operations. Class diagrams are most useful in illustrating 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.



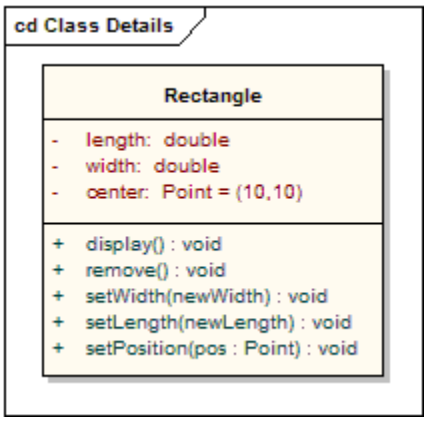
Classes

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

Class Notation

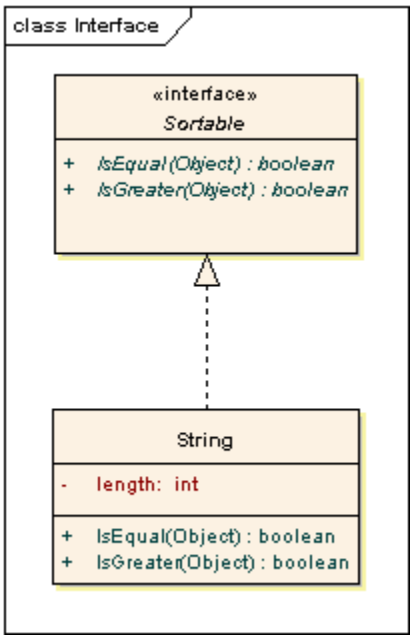
Classes are represented by rectangles which show the name of the class and optionally the name of the operations and attributes. Compartments are used to divide the class name, attributes and operations.

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 setWidth, setLength and setPosition and 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, while the ~ symbol indicates package visibility.

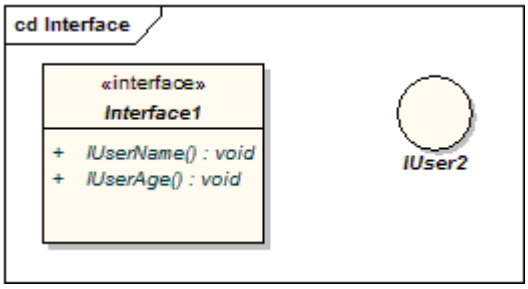


Interfaces

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

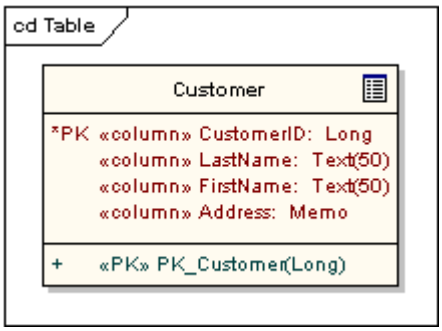


Interfaces may be drawn in a similar style to a class, with operations specified, as shown below. They may also be drawn as a circle with no explicit operations detailed. When drawn as a circle, realization links to the circle form of notation are drawn without target arrows.



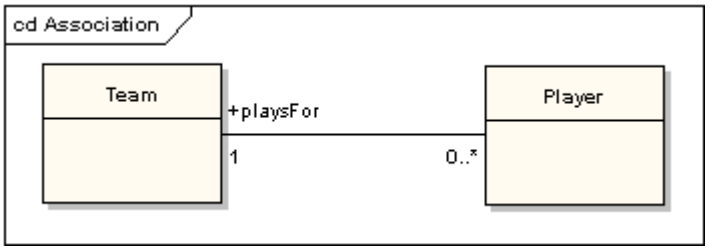
Tables

Although not a part of the base UML, a table is an example of what can be done with stereotypes. 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».



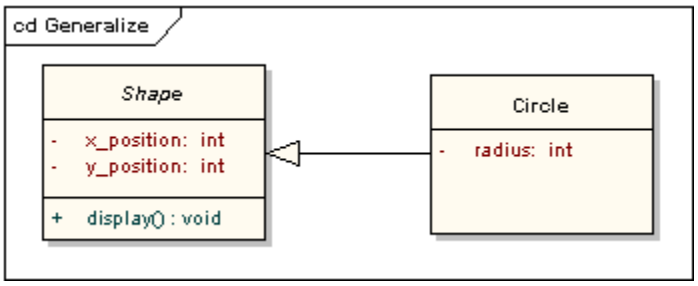
Associations

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 diamond representation toolbox element can be used as well. When code is generated for class diagrams, named association ends become instance variables in the target class. So, for the example below, "playsFor" will become an instance variable in the "Player" class.

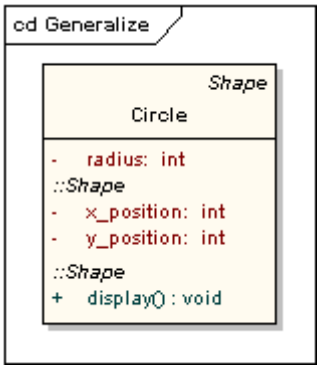


Generalizations

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.

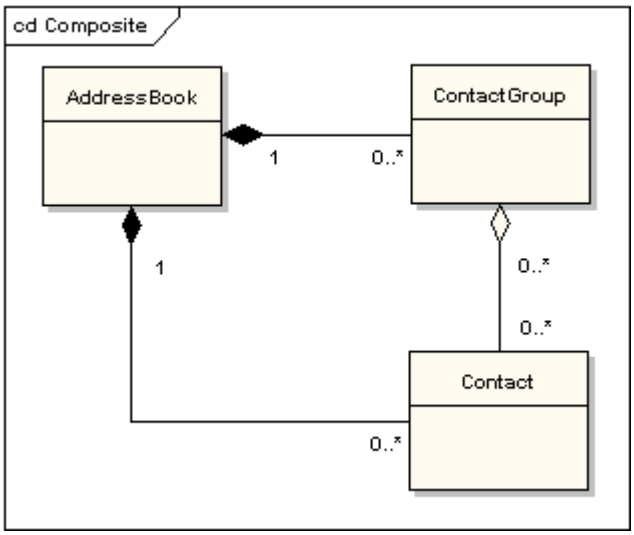


Aggregations

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.

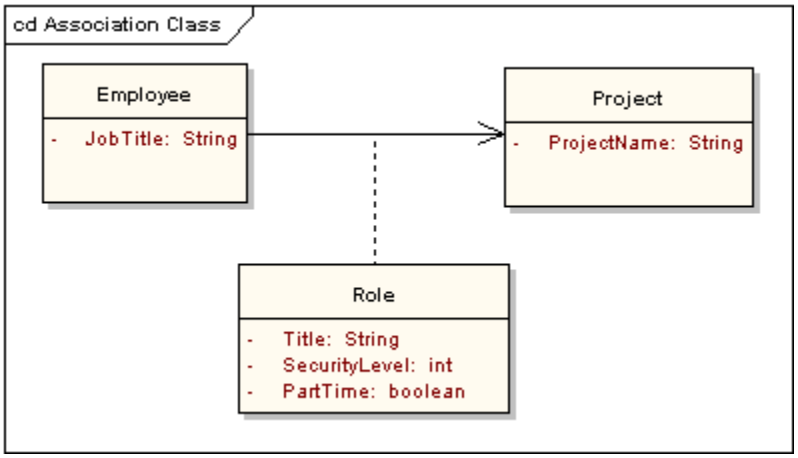
A stronger form of aggregation - a composite aggregation - is shown by a black diamond-shaped arrowhead and is used where components can be included in a maximum of one composition at a time. If the parent of a composite aggregation is deleted, usually all of its parts are deleted with it; however a 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 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

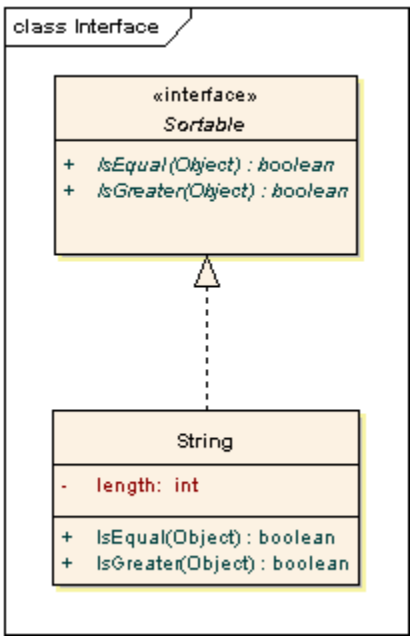
know exactly what the relationship is. Later in the design process, dependencies will be stereotyped (stereotypes available include «instantiate», «trace», «import», and others), or replaced with a more specific type of connector.

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.

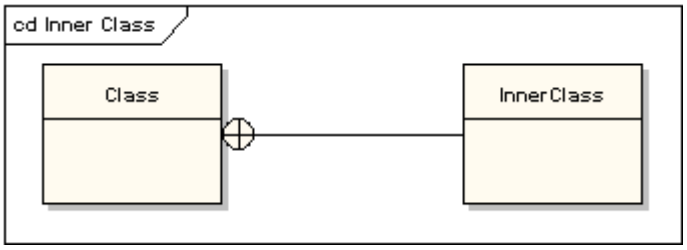
Realizations

The source object implements or realizes the destination. Realizations are 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 modeling 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.



Nestings

A nesting is connector that shows 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.



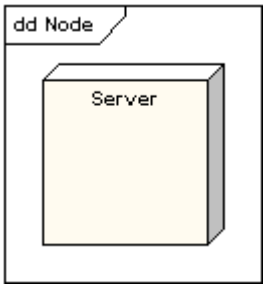
Deployment Diagram

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

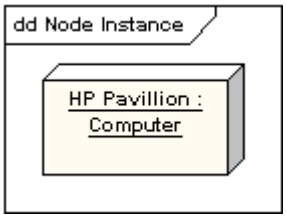
A Node is either a hardware or software element. It is shown as a three-dimensional box shape, as shown below.



Node Instance

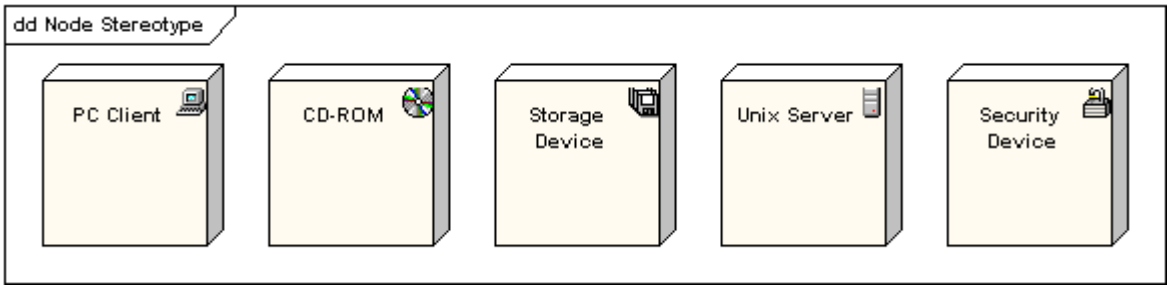
A node instance can be shown on a diagram. An instance can be distinguished from a node by the fact that its name is underlined and has a colon before its base node

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

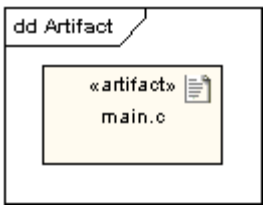
A number of standard stereotypes are provided for nodes, namely «cdrom», «cd-rom», «computer», «disk array», «pc», «pc client», «pc server», «secure», «server», «storage», «unix server», «user pc». These will display an appropriate icon in the top right corner of the node symbol



Artifact

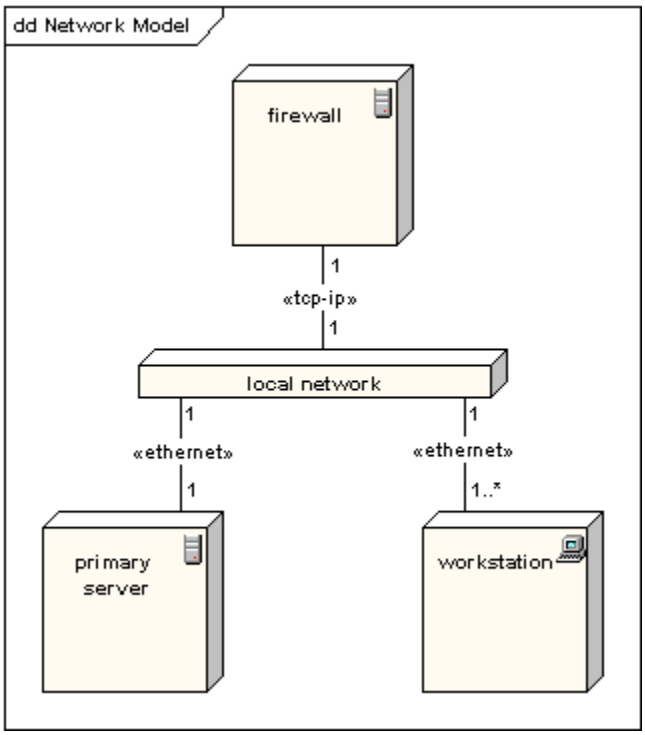
An artifact is a product of the software development process. That may include process models (e.g. use case models, design models etc), source files, executables, design documents, test reports, prototypes, user manuals, etc.

An artifact is denoted by a rectangle showing the artifact name, the «artifact» keyword and a document icon, as shown below.



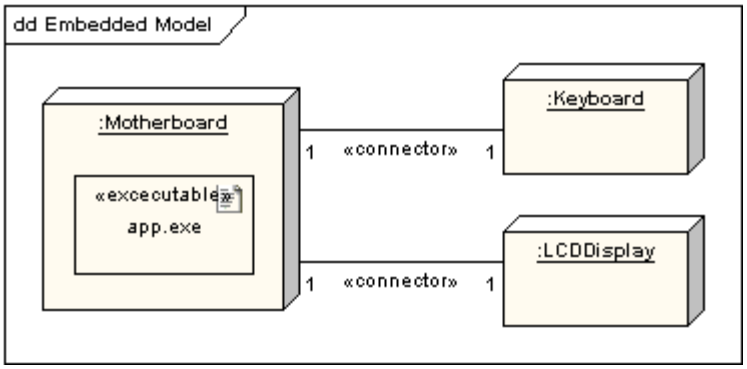
Association

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, depicting network protocols as stereotypes, and 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, depicting an executable artifact as being contained by the motherboard node.

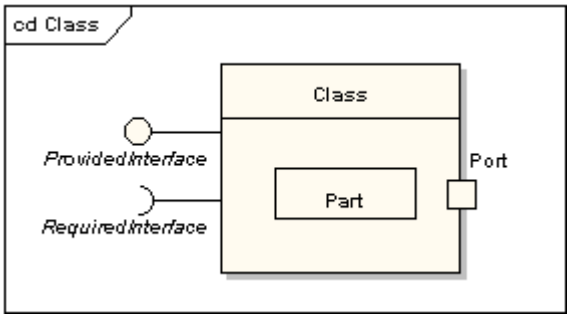


Composite Structure Diagram

Composite Structure 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 behavior of the containing classifier.

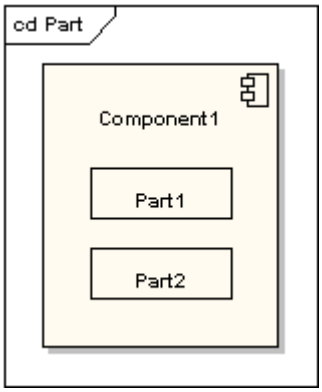
Class elements have been described in great detail in the section on class diagrams. This section describes the way classes can be displayed as composite elements exposing interfaces and containing ports and parts.



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.

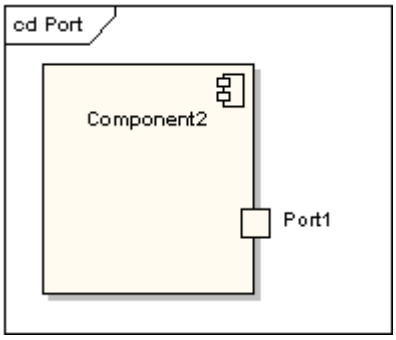
A part is shown as an unadorned rectangle contained within the body of a class or component element.



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.

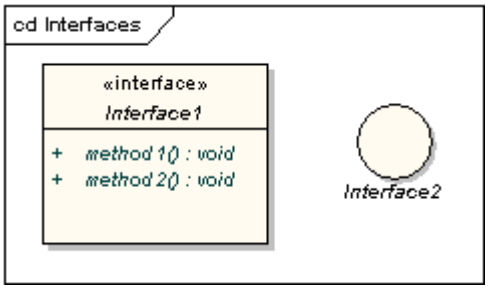
A port is shown as a named rectangle on the boundary edge of its owning classifier.



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.

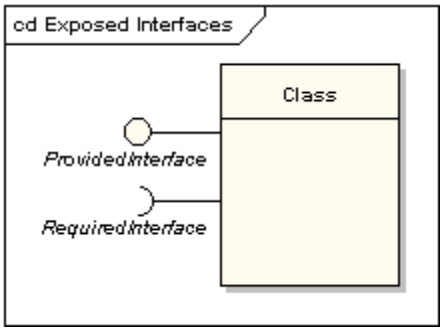
An interface, when standing alone in a diagram, is either shown as a class element rectangle with the «interface» keyword and with its name italicized to denote it is abstract, or it is shown as a circle.



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

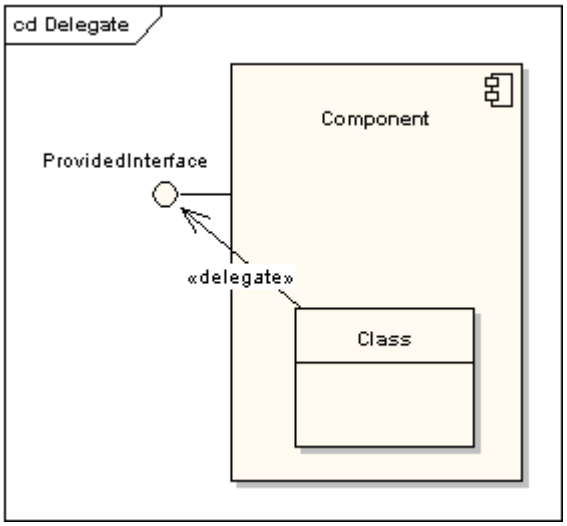
defined by the named interface element and is defined by drawing a realization link between the class and the interface. A required interface is a statement that the classifier is able to communicate with some other classifier which provides operations defined by the named interface element and is defined 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» keyword. It connects an external contract of a component as shown by its ports to the internal realization of the behavior of the component's part.

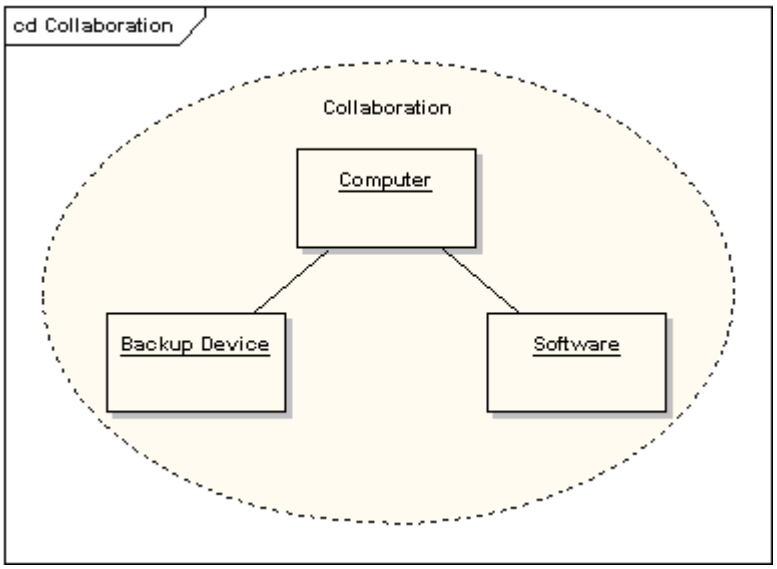


Collaboration

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

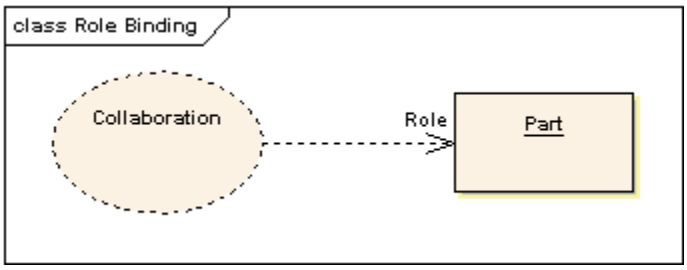
exercise in simplifying the structure and clarifying the behavior, and also provides for re-use. A collaboration often implements a pattern.

A collaboration element is shown as an ellipse.



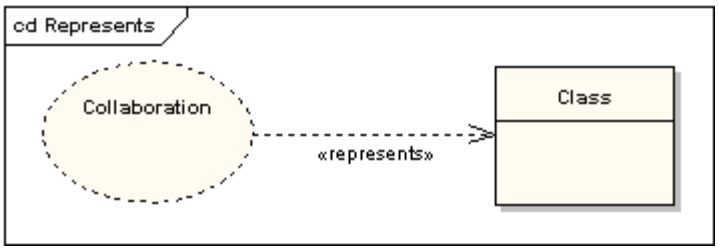
Role Binding

A role binding connector is drawn from a collaboration to the classifier that fulfils the role. It is shown as a dashed line with the name of the role at the classifier end.



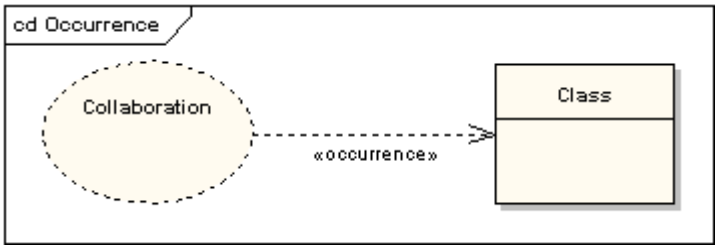
Represents

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 keyword «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 keyword «occurrence».



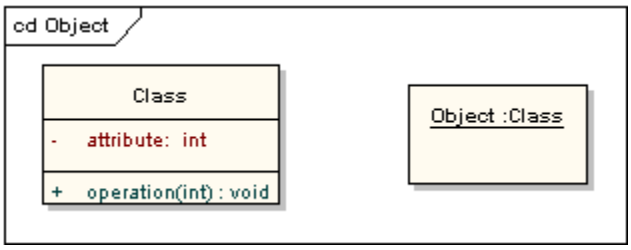
Object Diagram

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.

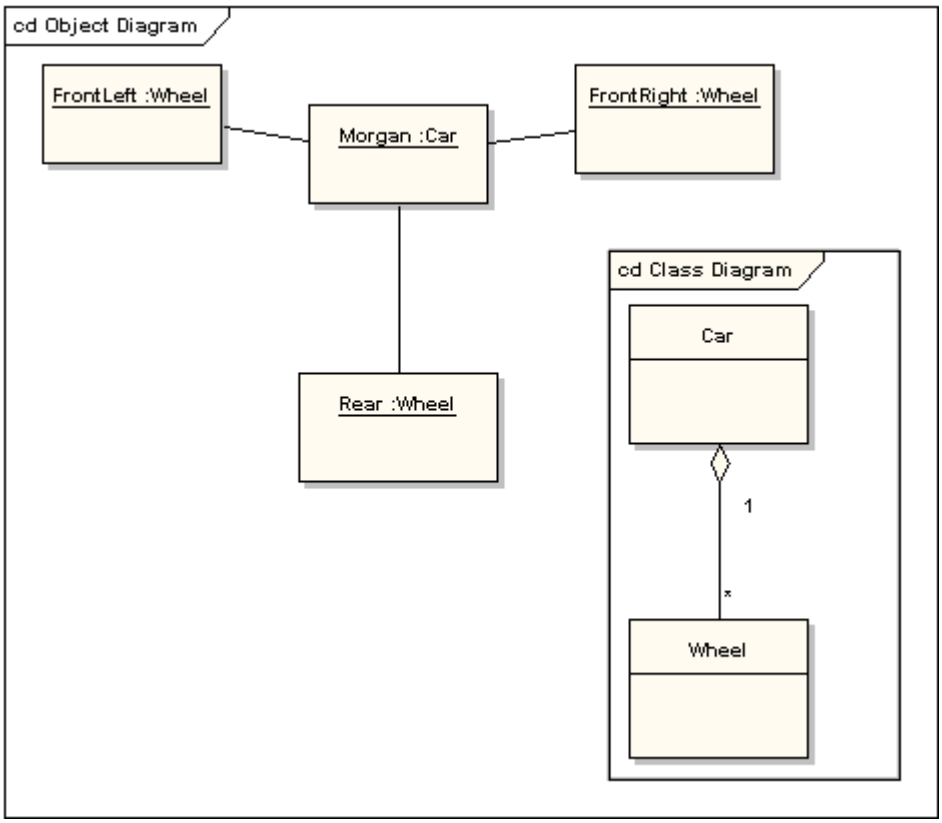


Run Time State

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 Profiles

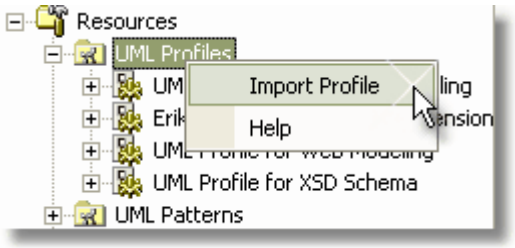
UML Profiles provide a generic extension mechanism for building UML models in particular domains. They are based on additional Stereotypes and Tagged values that are applied to Elements, Attributes, Methods, Links, Link Ends and more. A profile is a collection of such extensions that together describe some particular modeling problem and facilitate modeling constructs in that domain. For example the UML Profile for XML as defined by David Carlson in the book "Modeling XML Applications with UML" pp. 310, describes a set of extensions to basic UML model elements to enable accurate modeling of XSD Schemas.

Enterprise Architect has a generic UML Profile mechanism for loading and working with different Profiles. UML Profiles for Enterprise Architect are specified in XML files, with a specific format - see the examples below. These XML files may be imported into EA in the Resource page of the project browser. Once imported, you may drag and drop Profile elements onto the current diagram. EA will attach the stereotype, tagged values and default values, notes and even metafile if one is specified, to the new element. You can also drag and drop attributes and operations onto existing classes and have them immediately added with the specified stereotype, values etc.

To get you started, some profiles are supplied below for downloading and importing into EA. Over time we will expand the range of Profiles, the content of each profile and the degree of customization possible in each profile. Remember, you can always create your own profiles to describe modeling scenarios peculiar to your development environment. Some more details on working with Profiles is provided below also.

Working with UML Profiles in Enterprise Architect

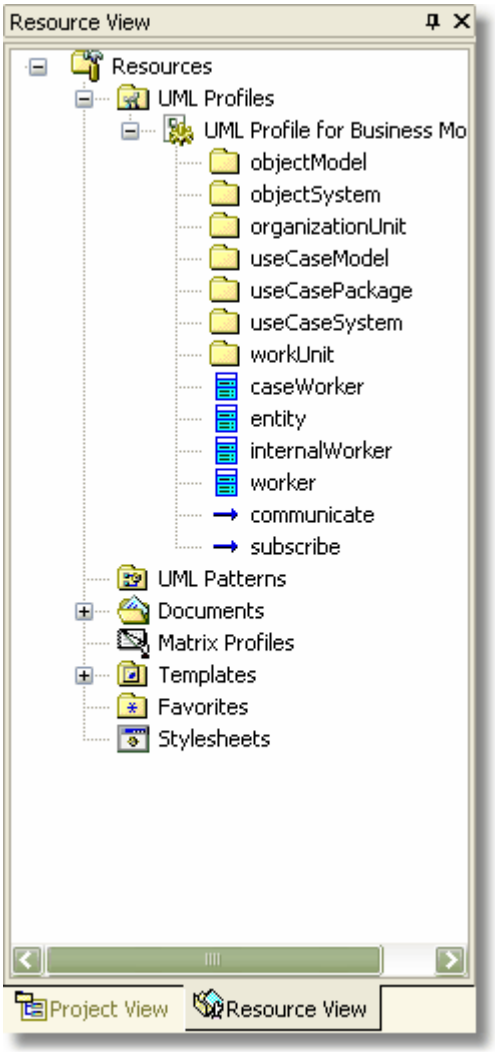
Importing a UML Profile



To import a profile you will need a suitable Profile XML file (as in the examples supplied above). If the Profile includes references to any Metafiles, they should be in the same directory as the XML profile.
To import a profile, follow the steps below:

1. Right click on the UML Profiles tree node in the Resources View and select *Import Profile* from the context menu - as in the to the right.
2. The Import UML Profile dialog will open.
3. Locate the XML Profile file to import using the *Browse [...]* button.
4. Set the required import options for all stereotypes defined in the profile - you can select to import:
   - Element Size yes/no - check this to import the element size attributes.
   - Color and Appearance yes/no - check this to import the color (background, border and font) and appearance (border thickness) attributes.
   - Alternate Image yes/no - check this to import the metafile image.
   - Code Templates yes/no - check this to import the code templates if they exist.
   - Overwrite Existing Templates yes/no - check this to overwrite any existing code templates defined in the current project.
5. Press *Import*.

Importing a UML Profile



Using Profile Elements

The image displayed to the right details the profiles branch for the UML Business Profile and the available stereotyped UML elements

You use the profile elements in the following manner:

- Elements such as classes and interfaces can be dragged directly from the resource window to the current diagram
- Attributes can be dragged over a host element (EG. Class) - they will automatically be added to the element feature list
- Operations are like Attributes - drag over a host element to add the operation
- Links such as Associations, Generalization, Dependency are added by selecting them in the browser, then click on start object in a diagram and drag to the end object (in the same manner as adding normal links. The link will be added with the new stereotype and tagged value information.
- Association Ends can be added by dragging the link end element over the end of an Association in the diagram

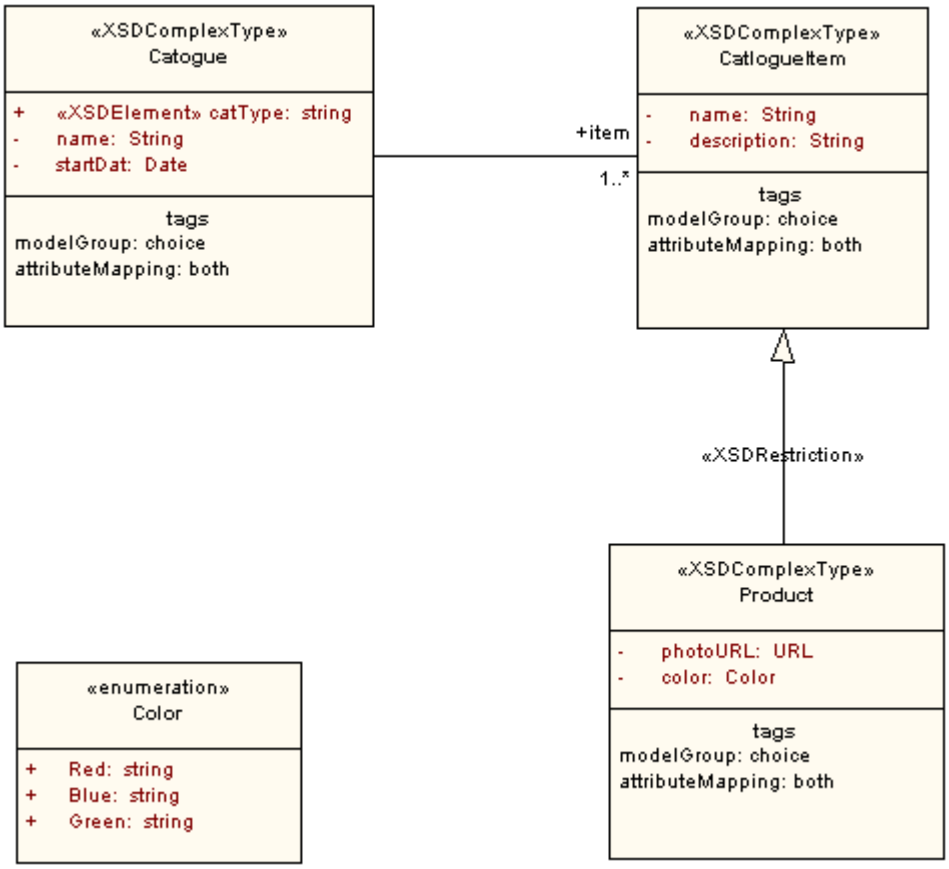
Deleting a Profile

To delete a profile, right click on the profile to remove and select the 'Delete Profile' context menu option. Note that this will not adversely affect elements already defined using this profile. If a stereotype that was imported using the Profile is in use, it will not be deleted from the model when you delete the profile.

Reloading a Profile

To reload a profile, you first delete the profile as above and then import again. A future version of EA will include the ability to refresh a profile.

An example diagram built with Profile elements illustrates the display of stereotypes and tagged values:



Use Case Diagram

Use Case Diagrams

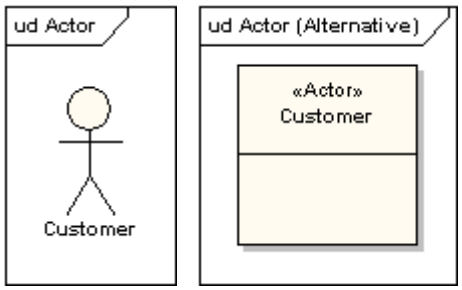
Use Case Model

The use case model captures the requirements of a system. Use cases are a means of communicating with users and other stakeholders what the system is intended to do.

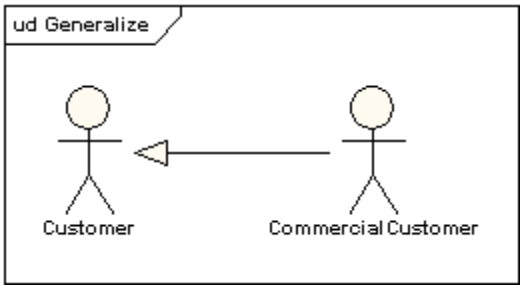
Actors

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 actor is usually drawn as a named stick figure, or alternatively as a class rectangle with the «actor» keyword.

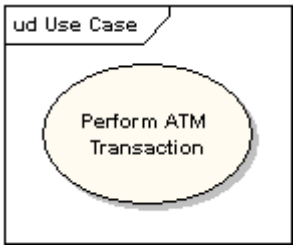


Actors can generalize other actors as detailed in the following diagram:

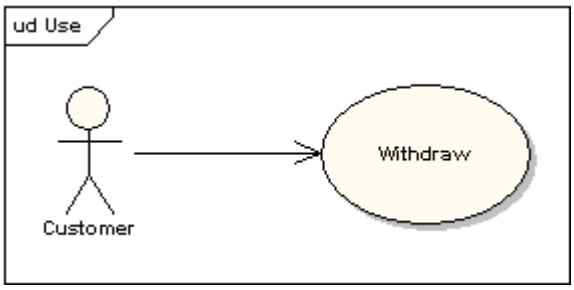


Use Cases

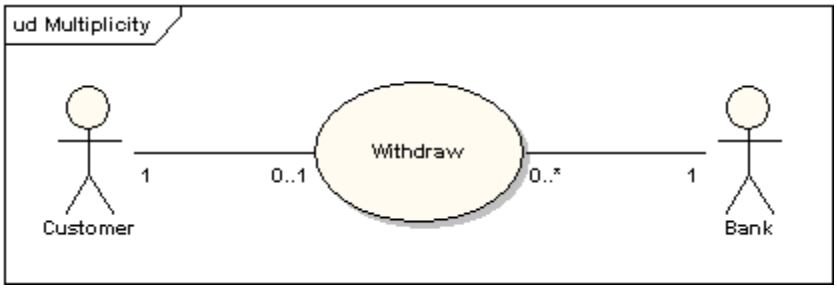
A use case is a single unit of meaningful work. It provides a high-level view of behavior observable to someone or something outside the system. The notation for a use case is an ellipse.



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 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

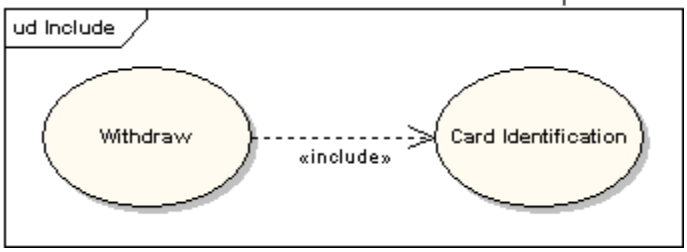
A constraint is a condition or restriction that a use case operates under and includes pre-, post- and invariant conditions. A precondition specifies the conditions that need to be met before the use case can proceed. A post-condition is used to document the change in conditions that must be true after 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

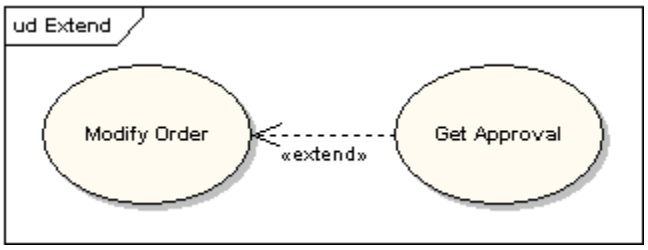
Use cases may contain the functionality of another use case as part of their normal processing. In general it is assumed that any included use case will be called every time the basic path is run. An example of this is to have the execution of the use case <Card Identification> to be run as part of a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

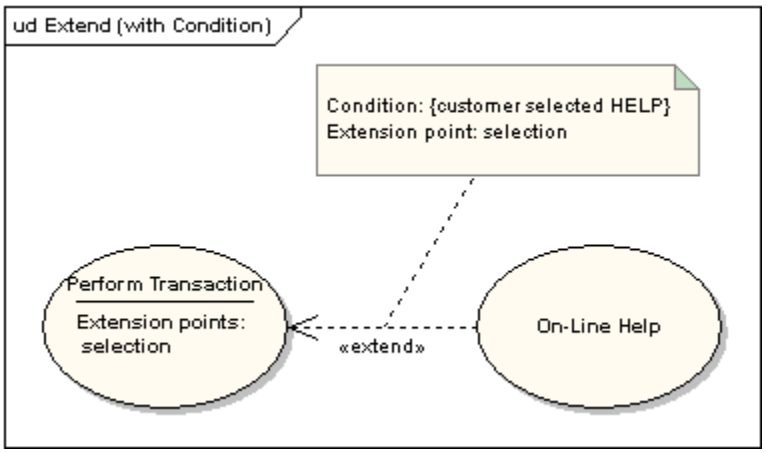
Extending Use Cases

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.



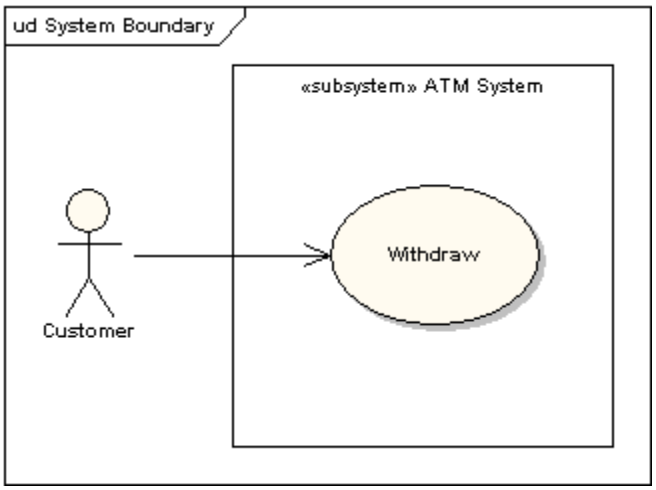
Extension Points

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.



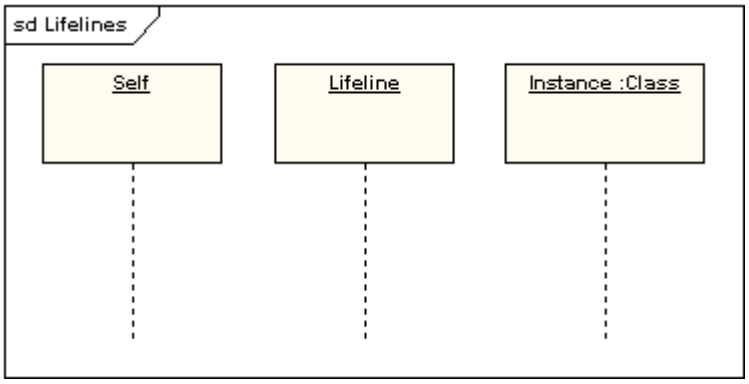
Sequence Diagram

Sequence Diagrams

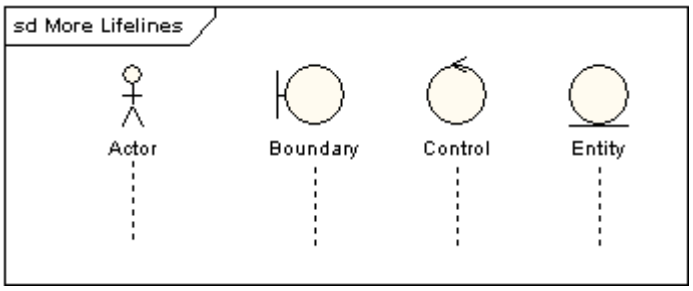
A sequence diagram is a form of interaction diagram which shows objects as lifelines running down the page, with their interactions over time represented as messages drawn as arrows from the source lifeline to the target lifeline. Sequence diagrams are good at showing which objects communicate with which 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", that indicates that the lifeline represents the classifier which owns the sequence diagram.

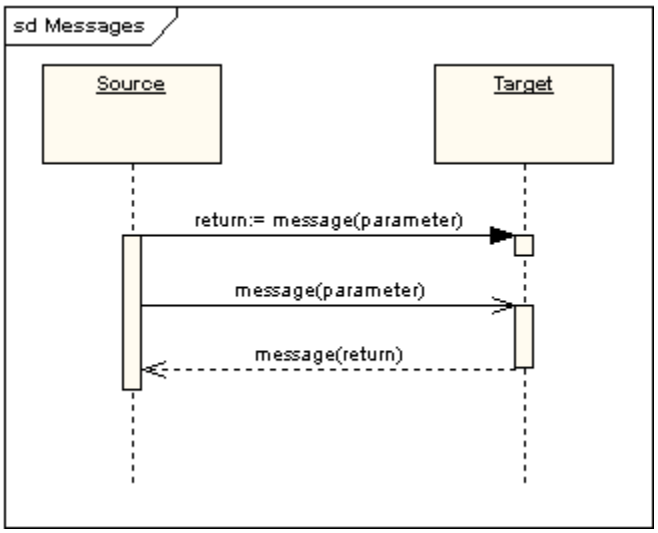


Sometimes a sequence diagram will have a lifeline with an actor element symbol at its head. This will usually be the case if the sequence diagram is owned by a use case. Boundary, control and entity elements from robustness diagrams can also own lifelines.



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).



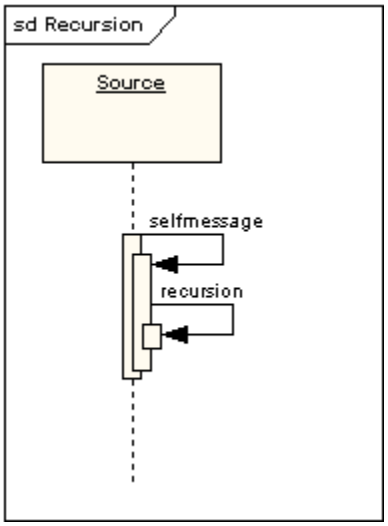
Execution Occurrence

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.

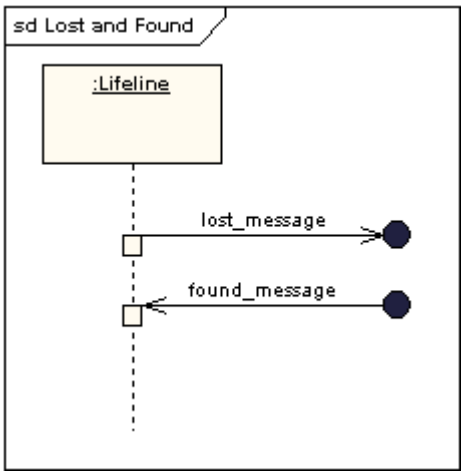
Self Message

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.



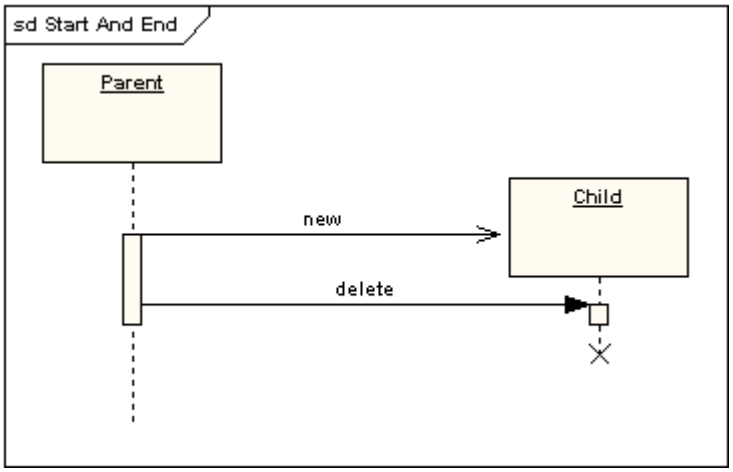
Lost and Found Messages

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.



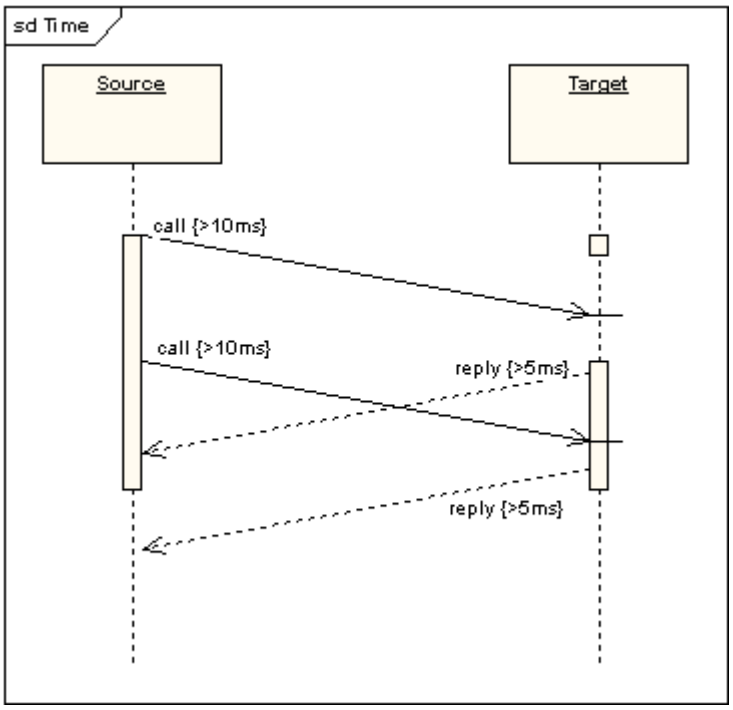
Lifeline Start and End

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.



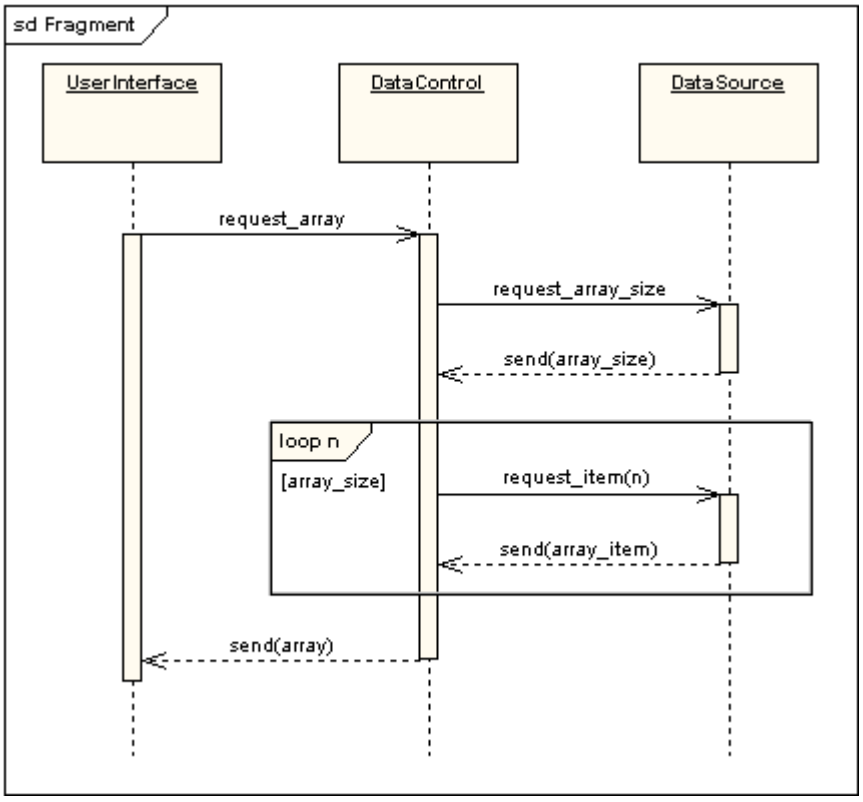
Combined Fragments

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.

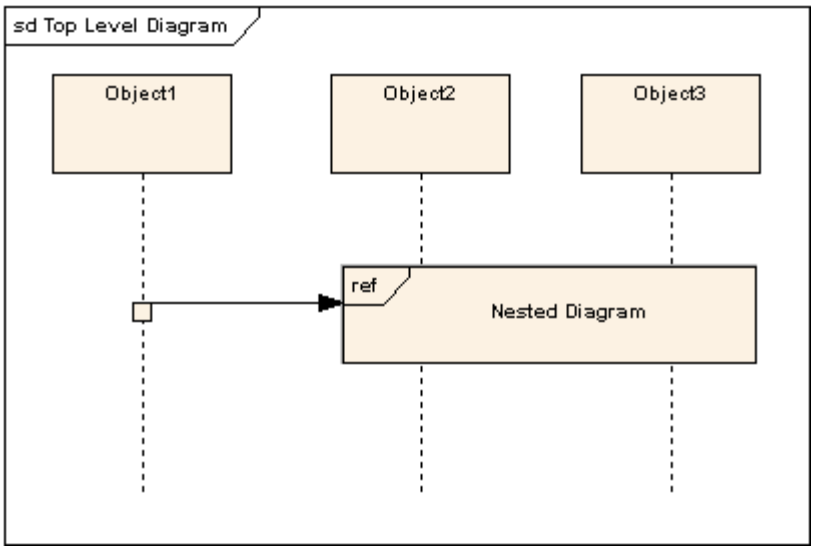
The following diagram shows a loop fragment.

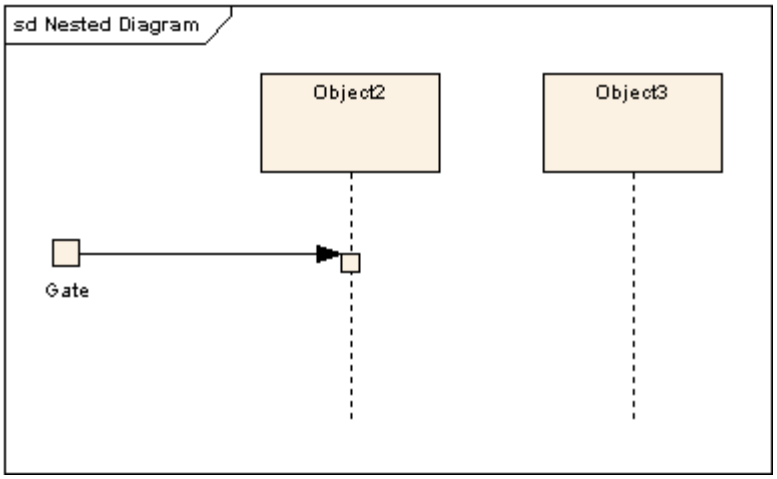


There is also an interaction occurrence, which is similar to a combined fragment. An interaction occurrence is a reference to another diagram which has the word "ref" in the top left corner of the frame, and has the name of the referenced diagram shown in the middle of the frame.

Gate

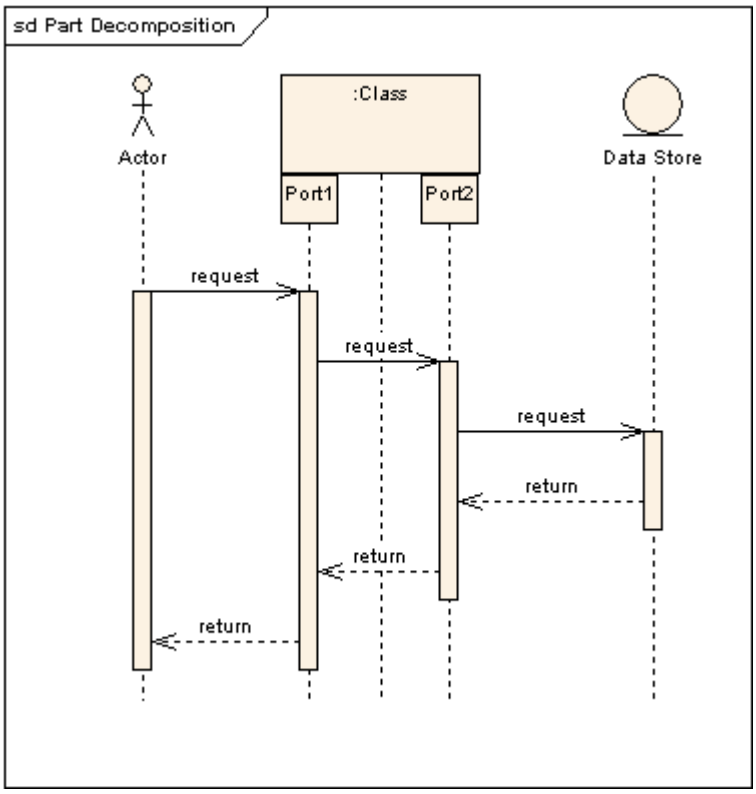
A gate is a connection point for connecting a message inside a fragment with a message outside a fragment. EA shows a gate as a small square on a fragment frame. Diagram gates act as off-page connectors for sequence diagrams, representing the source of incoming messages or the target of outgoing messages. The following two diagrams show how they might be used in practice. Note that the gate on the top level diagram is the point at which the message arrowhead touches the reference fragment - there is no need to render it as a box shape.





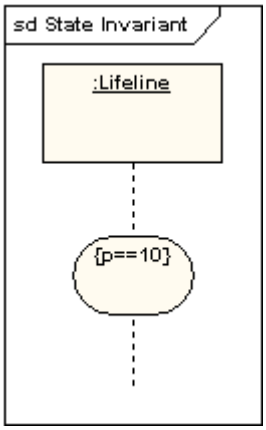
Part Decomposition

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.



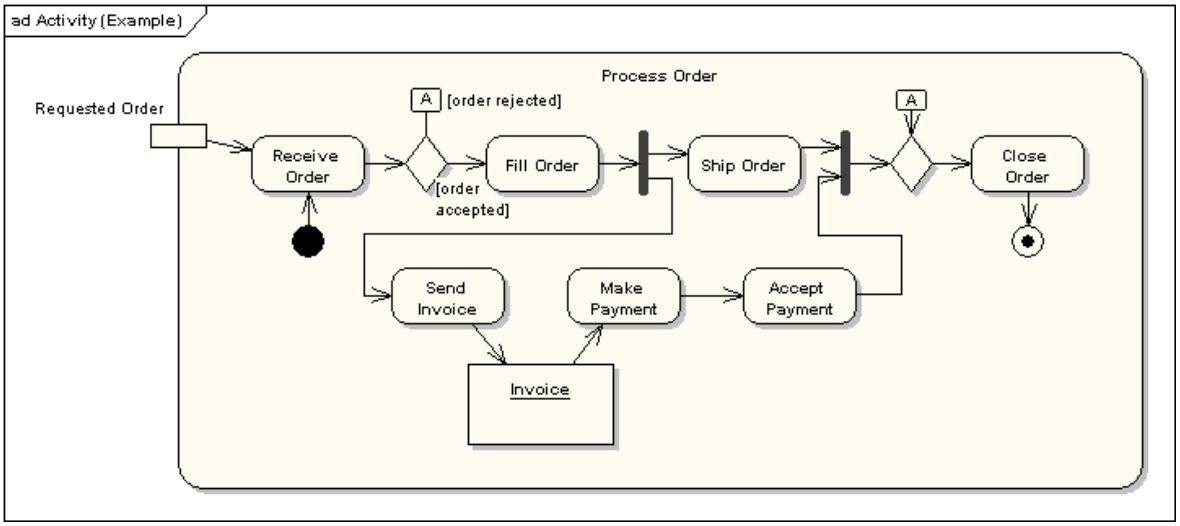
A continuation has the same notation as a state invariant, but is used in combined fragments and can stretch across more than one lifeline.

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 used 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.

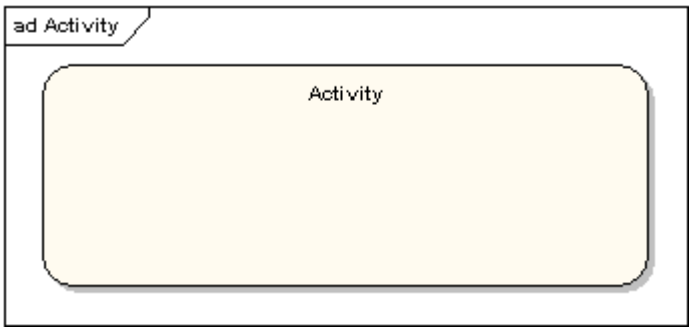
An Example of an activity diagram is shown below.



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

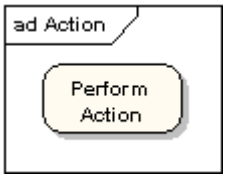
Activities

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.



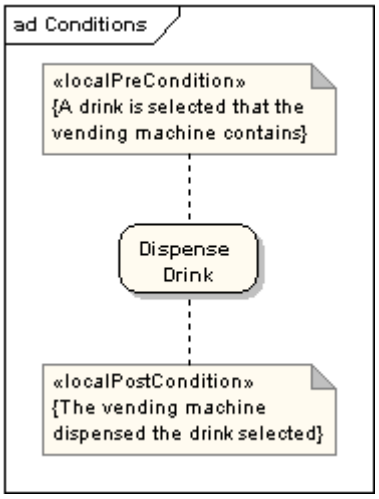
Actions

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



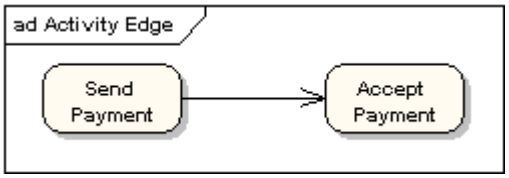
Action Constraints

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



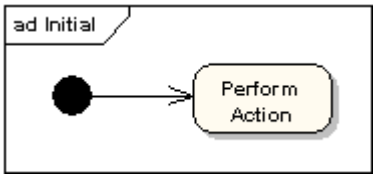
Control Flow

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



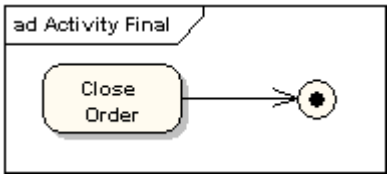
Initial Node

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

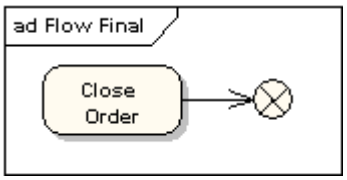


Final Node

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.



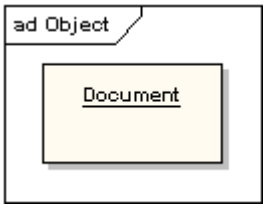
The flow final node is depicted as a circle with a cross inside.



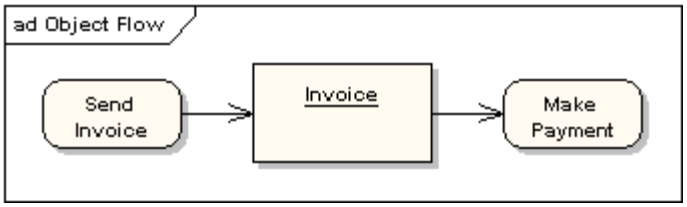
The difference between the two node types is that the flow final node denotes the end of a single control flow; the activity final node denotes the end of all control flows within the activity.

Objects and Object Flows

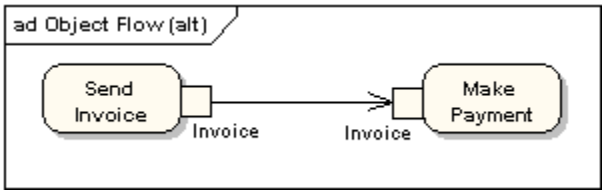
An object flow is a path along which objects or data can pass. An object is shown as a rectangle.



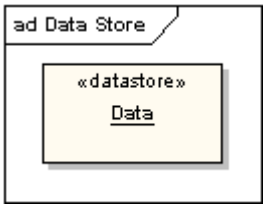
An object flow is shown as a connector with an arrowhead denoting the direction the object is being passed.



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.

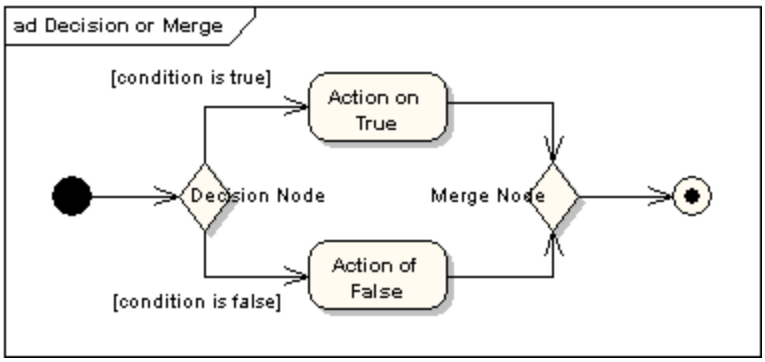


A data store is shown as an object with the «datastore» keyword.



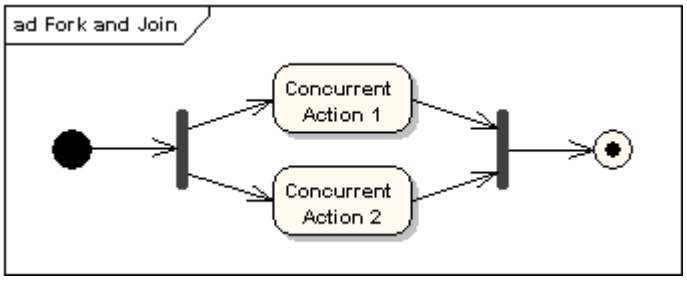
Decision and Merge Nodes

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

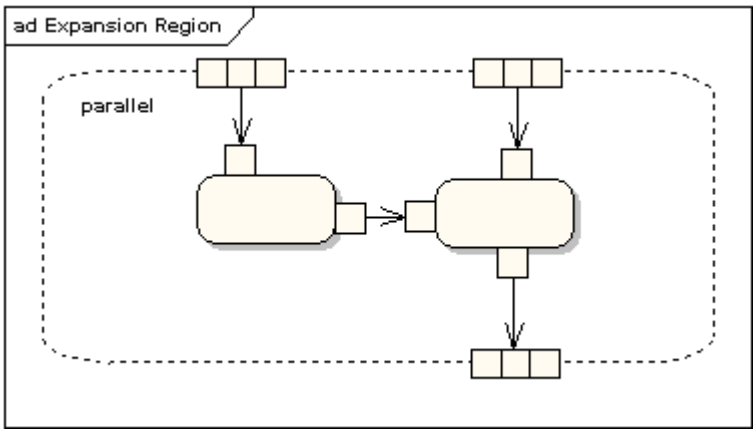
Forks and joins have the same notation: either a horizontal or vertical bar (the orientation is dependent on whether the control flow is running left to right or top to bottom). They indicate the start and end of concurrent threads of control. The following diagram shows an example of their use.



A join is different from a merge in that the join synchronizes 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 are received by a merge symbol, the action pointed to by its outflow is executed two or more times.

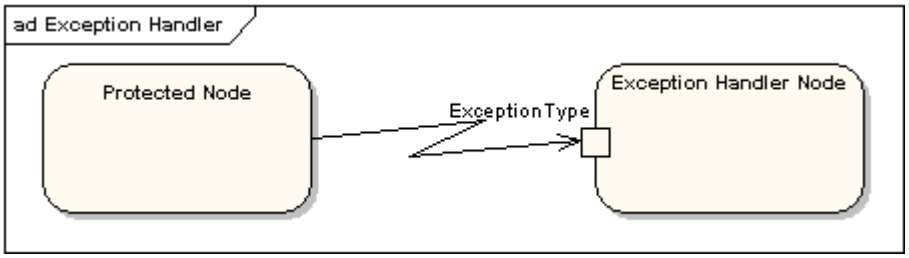
Expansion Region

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.



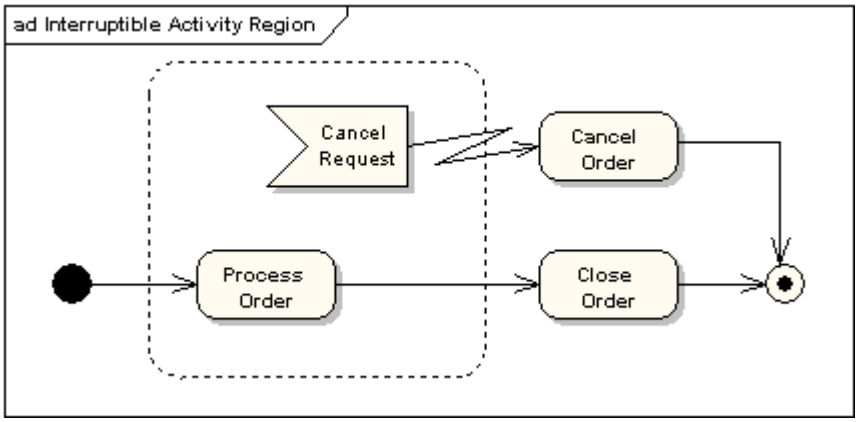
Exception Handlers

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



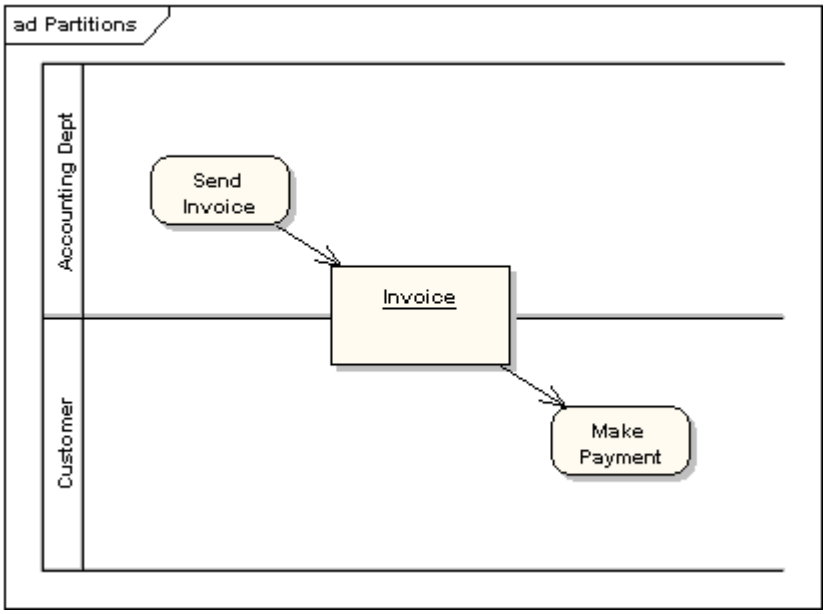
Interruptible Activity Region

An interruptible activity region surrounds a group of actions that can be interrupted. In the very simple example below, the "Process 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 a horizontal or vertical swimlane. 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.



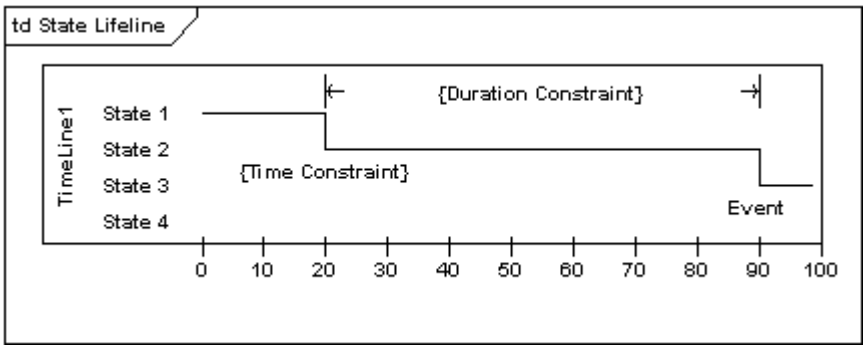
Timing Diagram

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.

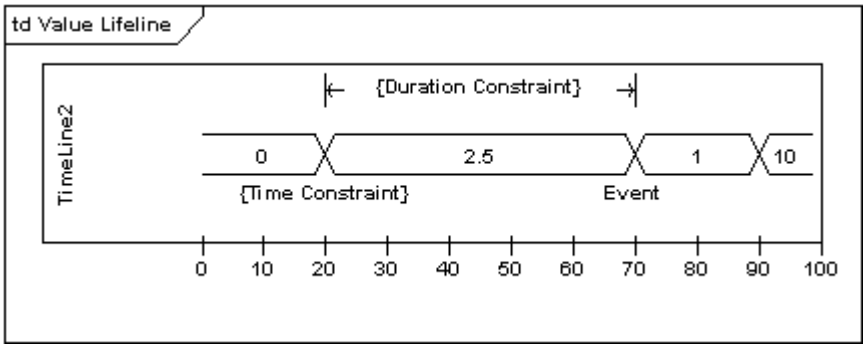
State Lifeline

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



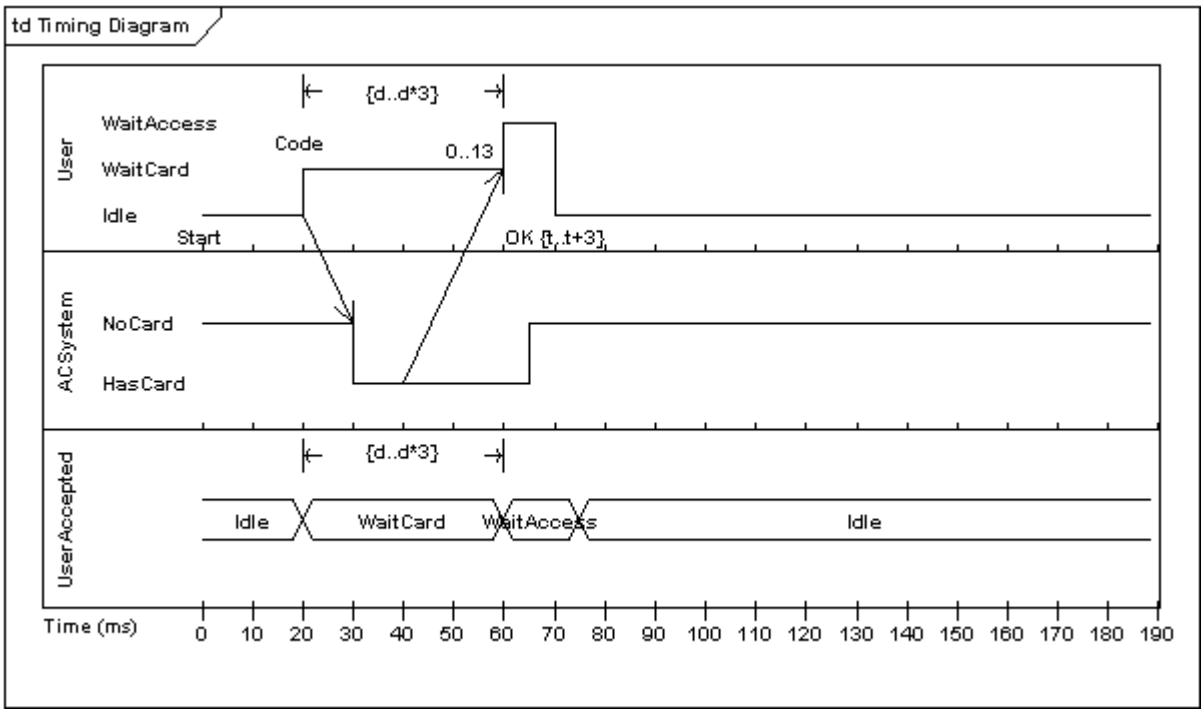
Value Lifeline

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 cross 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 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. Once these have all been applied, a timing diagram may look like the following.

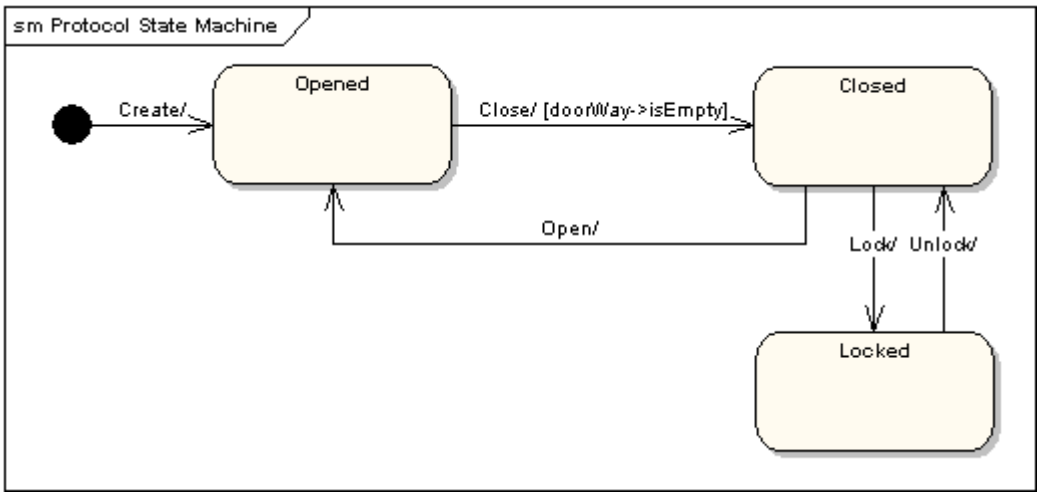


State Machine Diagram

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.

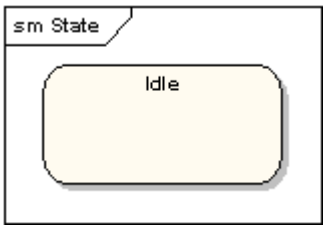
As an example, the following state machine diagram shows the states that a door goes through during its lifetime.



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 the following sections.

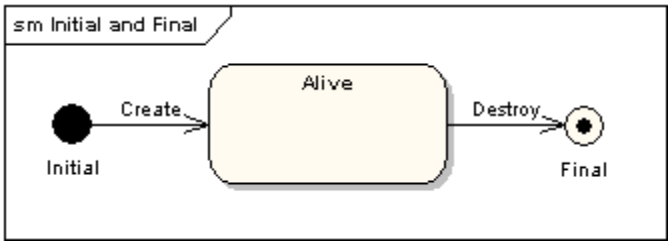
States

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



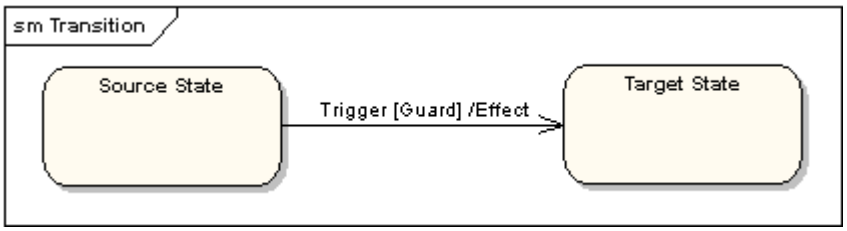
Initial and Final States

The initial state is denoted by a filled black circle and may be labeled with a name. The final state is denoted by a circle with a dot inside and may also be labeled with a name.



Transitions

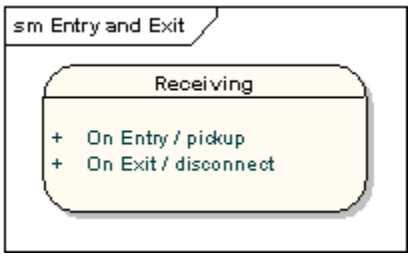
Transitions from one state to the next are denoted by lines with arrowheads. A transition may have a trigger, a guard and an effect, as below.



"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 invoked directly on the object that owns the state machine as a result of the transition.

State Actions

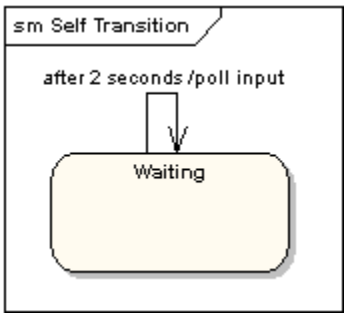
In the transition example above, an effect was associated with the transition. If the target state had many transitions arriving at it, and each transition had the same effect associated with it, it would be better to associate the effect with the target state rather than the transitions. This can be 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 actions of each type.

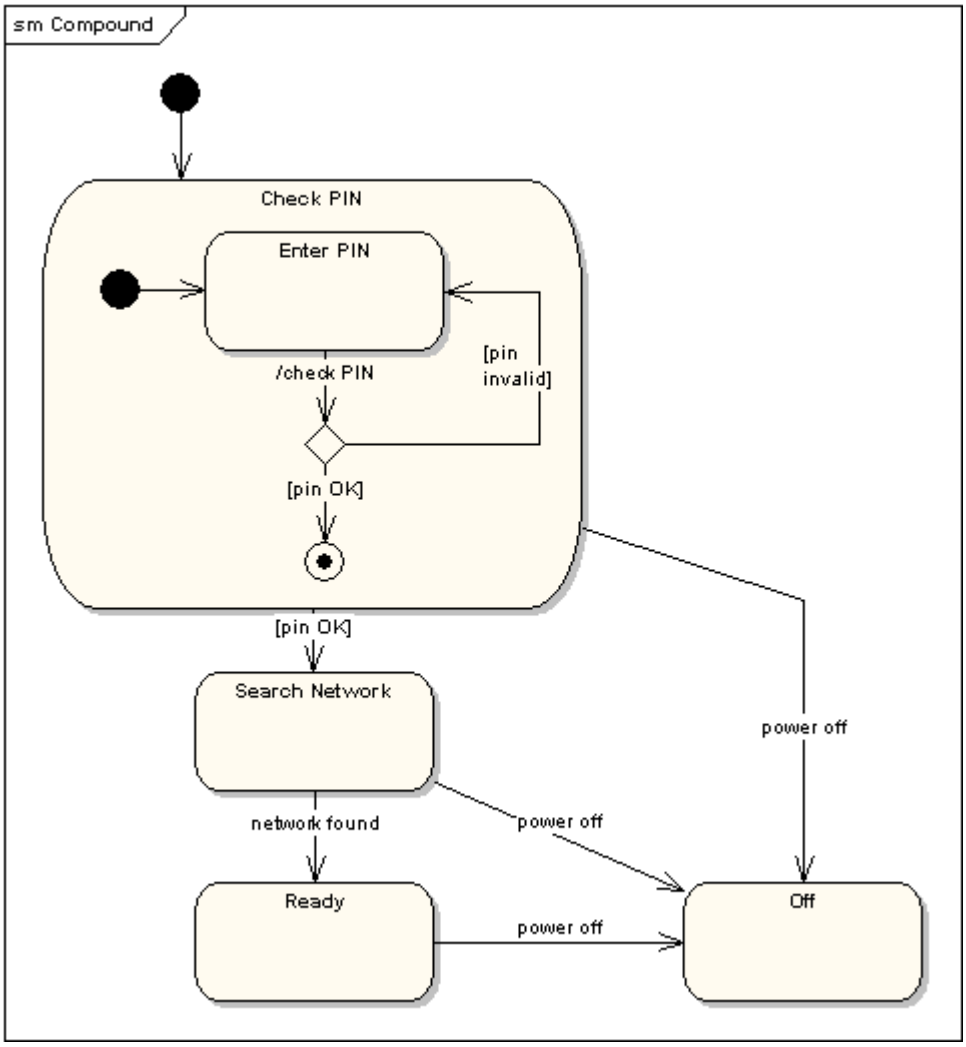
Self-Transitions

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

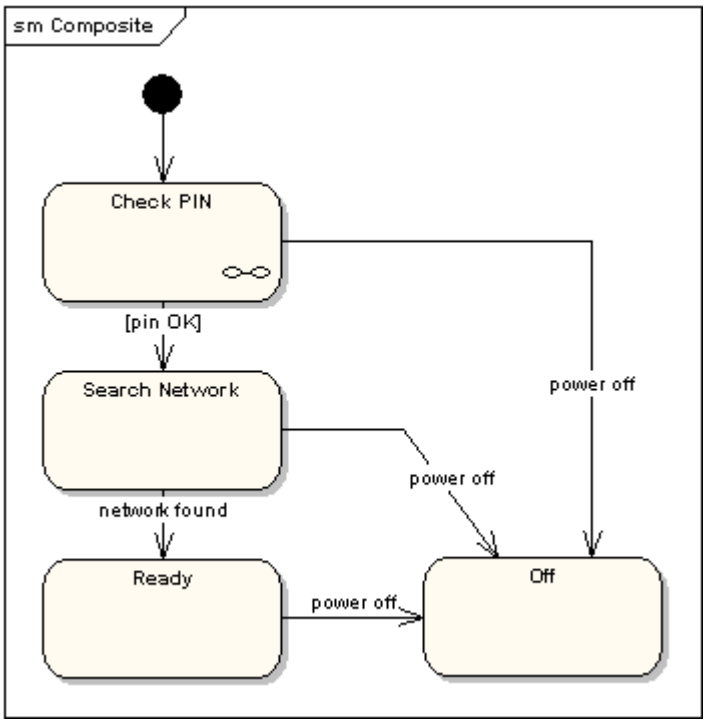


Compound States

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



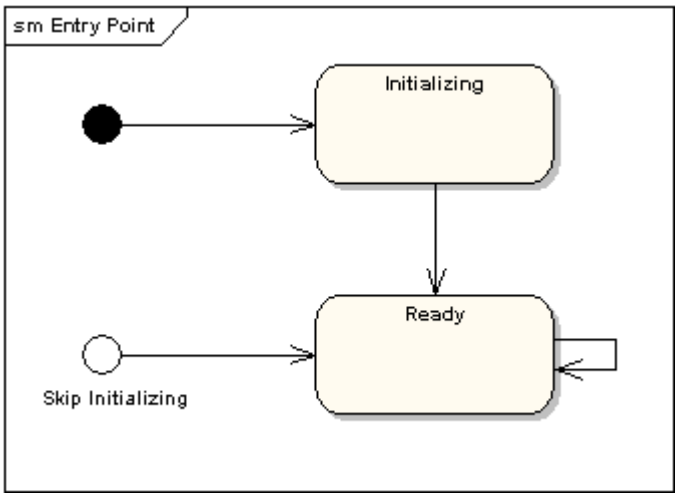
The alternative way to show the same information is as follows.



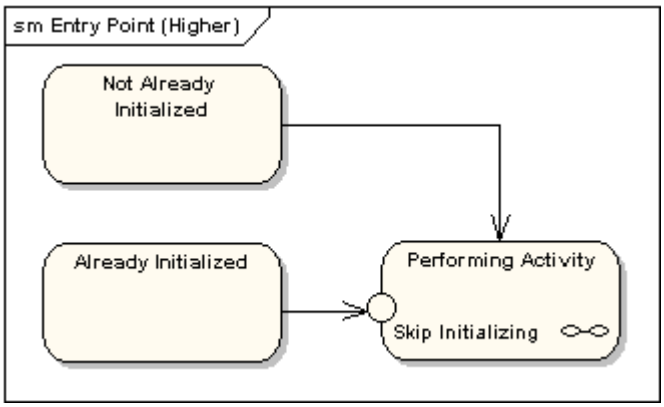
The notation in the above version indicates that the details of the Check PIN sub-machine are shown in a separate diagram.

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.

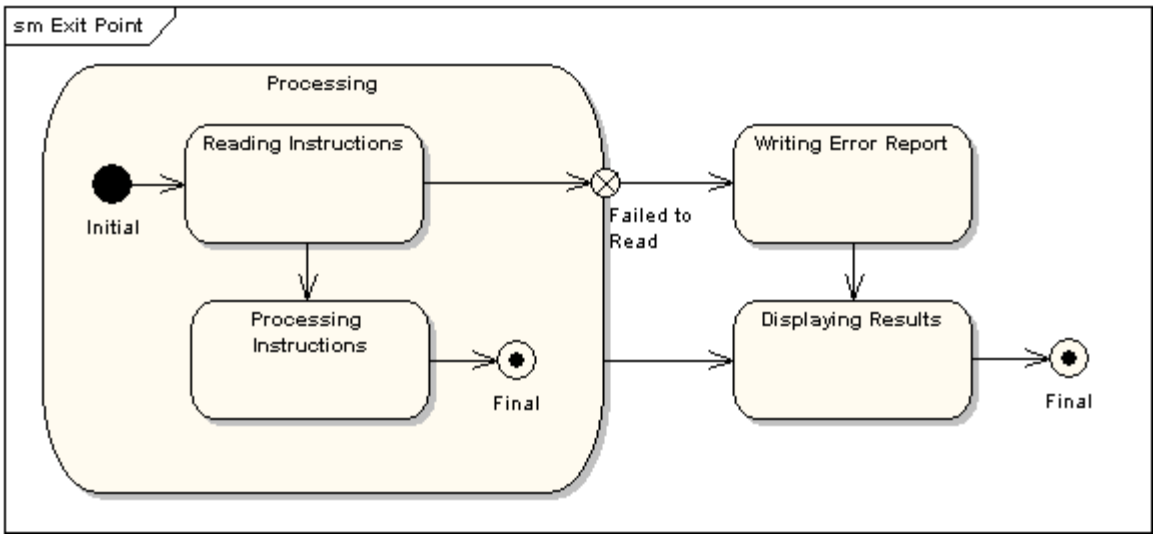


The following diagram shows the state machine one level up.



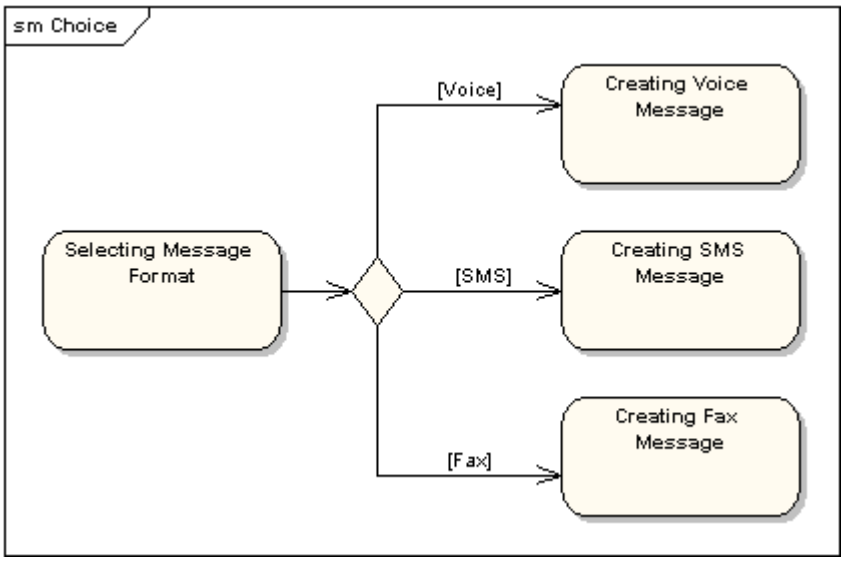
Exit Point

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.



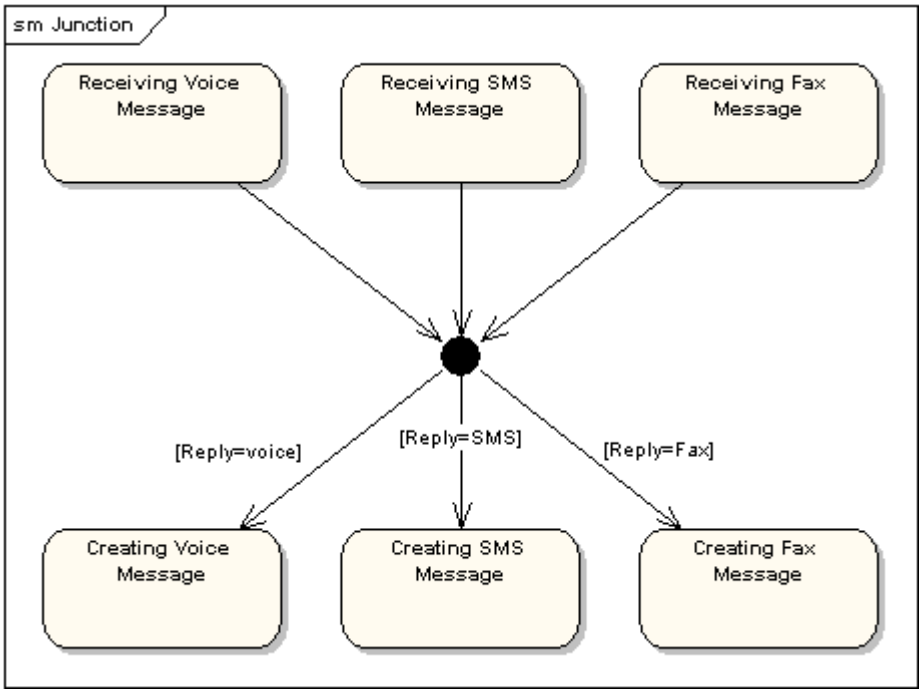
Choice Pseudo-State

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 during execution of the previous state.



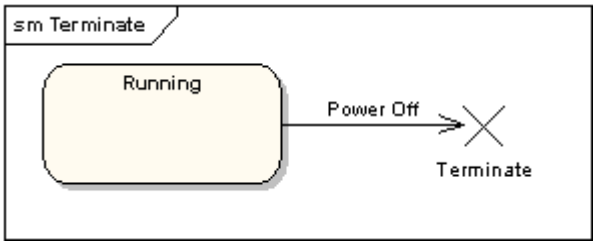
Junction Pseudo-State

Junction pseudo-states are used to chain together multiple transitions. A single junction can have one or more incoming, and one or more outgoing, transitions; 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 which realizes a dynamic conditional branch.



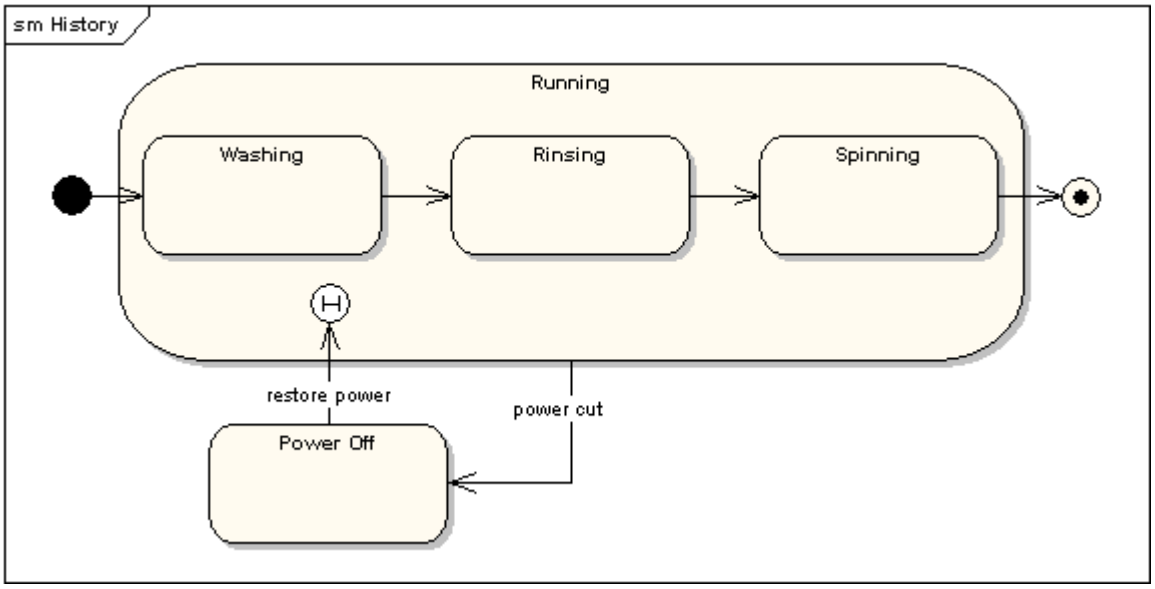
Terminate Pseudo-State

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



History States

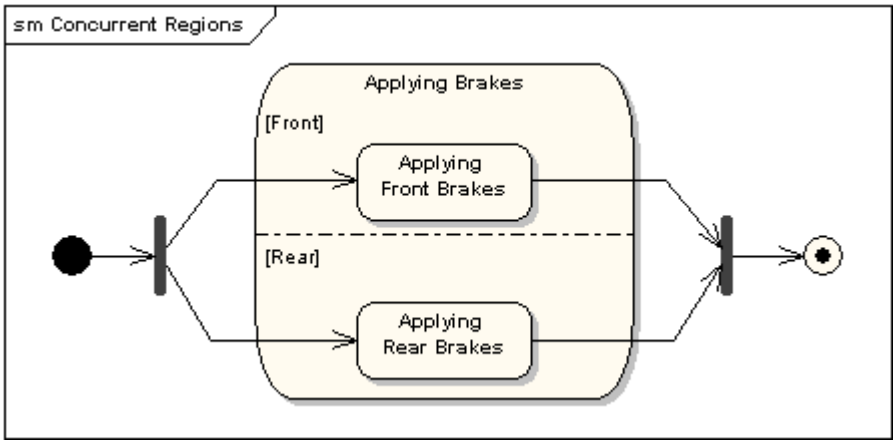
A history state is used to remember the previous state of a state machine when it was interrupted. The following diagram illustrates the use of history states. The example is a state machine belonging to a washing machine.



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 Running 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 of fork and join pseudo-states, rather than choice and merge pseudo-states. These symbols are used to synchronize the concurrent threads.



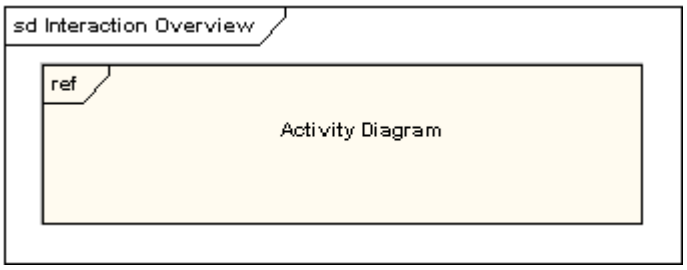
Interaction Overview Diagram

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 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.

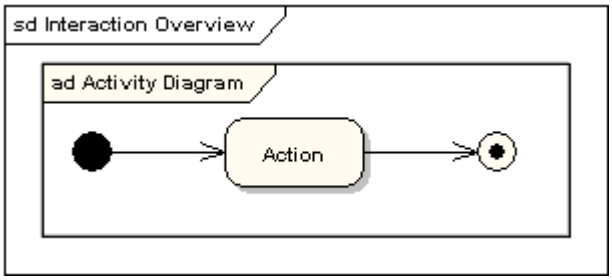
Interaction Occurrence

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



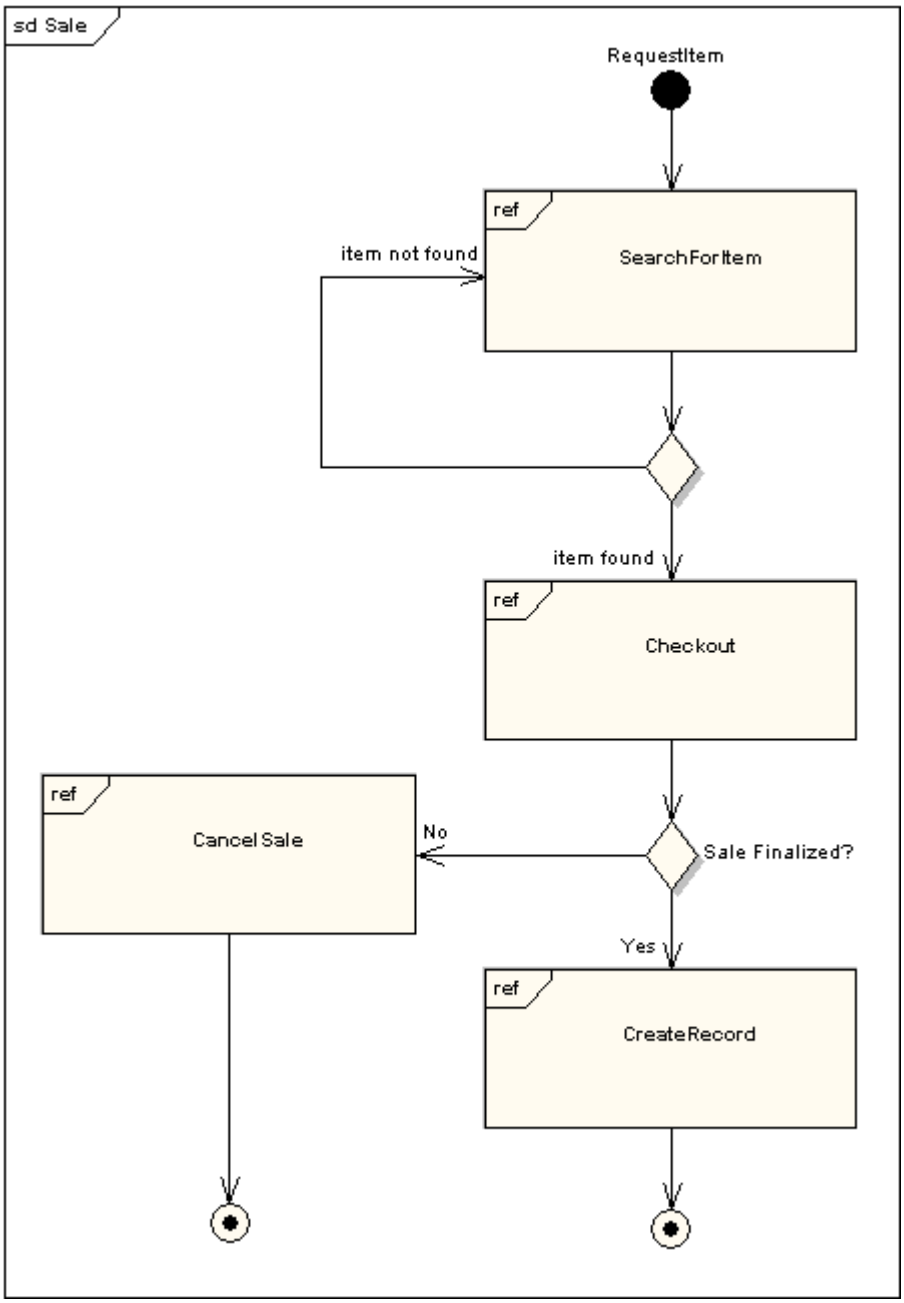
Interaction Element

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.



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 is 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.