UML Tutorials

Using UML Part One – Structural Modeling Diagrams

by Sparx Systems

http://www.sparxsystems.com

Trademarks

Table of Contents

Introduction

The Unified Modeling Language (UML) has become the de-facto standard for building Object-Oriented software. UML 2.1 builds on the already highly successful UML 2.0 standard, which has become an industry standard for modeling, design and construction of software systems as well as more generalized business and scientific processes. UML 2.1 defines thirteen basic diagram types, divided into two general sets: structural modeling diagrams and behavioral modeling diagrams. Part one will deal with structural modeling diagrams.

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."*

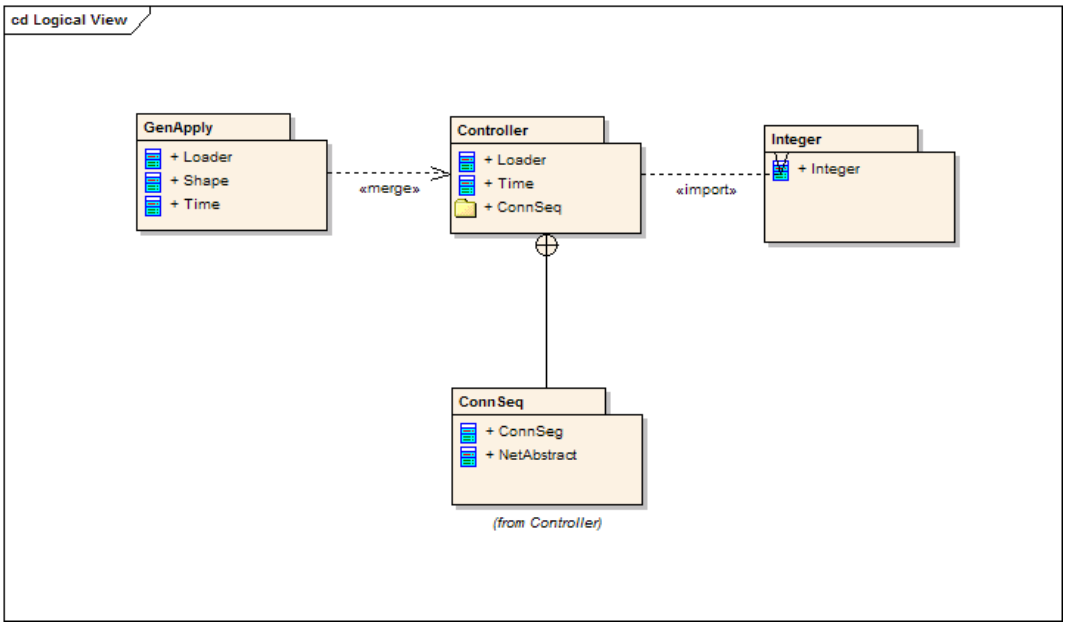
The important point to note here is UML is a "language" for specifying and not a method or procedure. The UML is used to define a software system – to detail the artifacts in the systems, to document and construct; it is the language the blueprint is written in. The UML may be used in a variety of ways to support a software development methodology (such as the Rational Unified Process), but in itself does not specify that methodology or process.

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

Package Diagrams

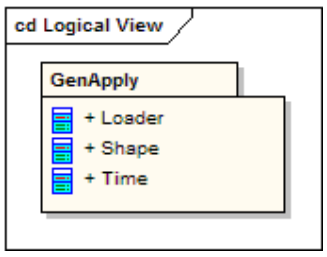
Package Diagrams are used to reflect the organization of packages and their elements. When used to represent class elements, package diagrams are used to provide a visualization of the namespaces. The most common use for package diagrams is to organize use case diagrams and class diagrams, although the 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, this sharing of namespace requires the elements contained in a specific namespace to have unique names.

Packages can be built to represent either physical or logical relationships. When choosing to include classes to specific packages, it is useful to assign the classes with the same inheritance hierarchy to packages, classes that are related via composition and classes that collaborate with also have a strong argument for being included 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 elements' definitions will be expanded to include the element definitions contained in the target. The target elements' definitions will be unaffected, as will the definitions of source code 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, the target package, will use unqualified names when being referred to from the source package. The source package's namespace will gain access to the target's class(s); the target's namespace is not affected.

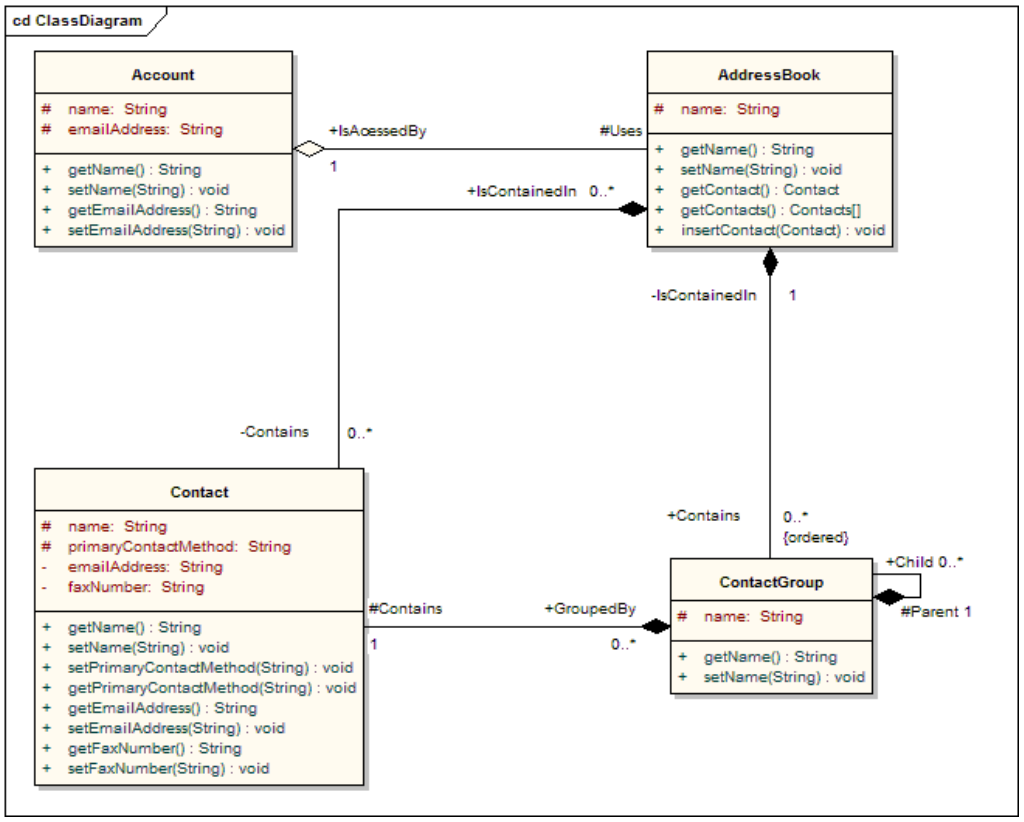
Nesting Connectors

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

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.



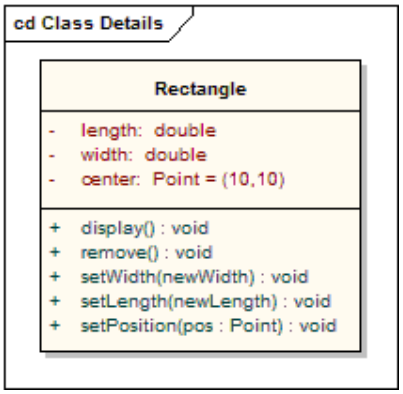
Classes

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 notations

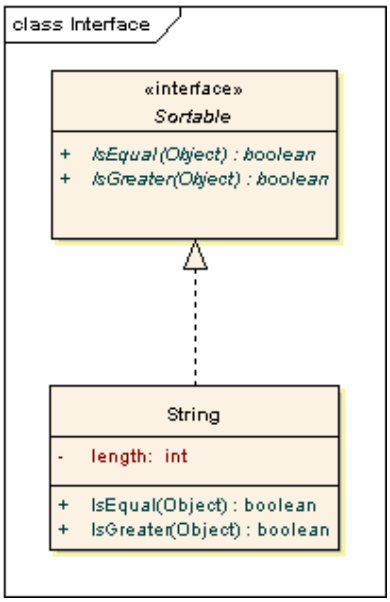
Classes are represented by rectangles which show the name of the class and optionally the name of the operations and attributes. Compartments are used to divide the class name, attributes and operations.

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 the setWidth, setLength and setPosition operations showing their parameters. The notation that precedes the attribute, or operation name, indicates the visibility of the element: if the + symbol is used, the attribute, or operation, has a public level of visibility; if a - symbol is used, the attribute, or operation, is private. In addition the # symbol allows an operation, or attribute, to be defined as protected, while the ~ symbol indicates package visibility.

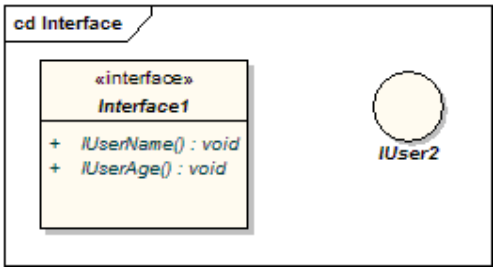


Interfaces

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 – i.e. through the common interface.

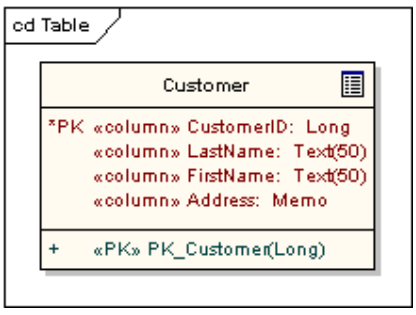


Interfaces may be drawn in a similar style to a class, with operations specified, as shown below. They may also be drawn as a circle with no explicit operations detailed. When drawn as a circle, realization links to the circle form of notation are drawn without target arrows.



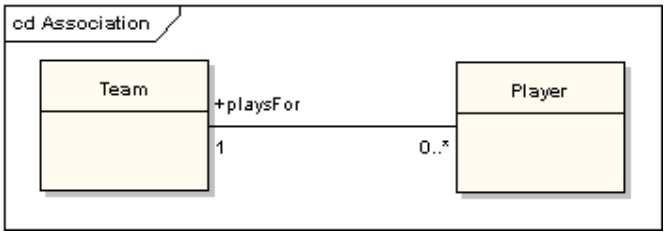
Tables

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».



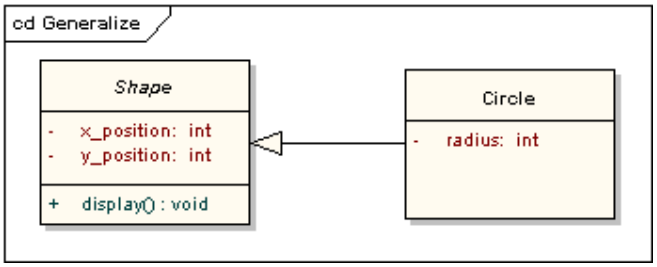
Associations

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.

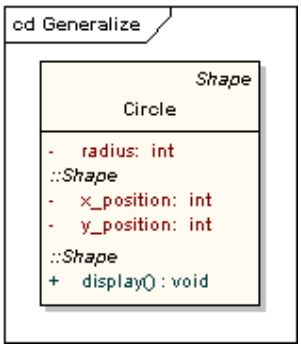


Generalizations

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.

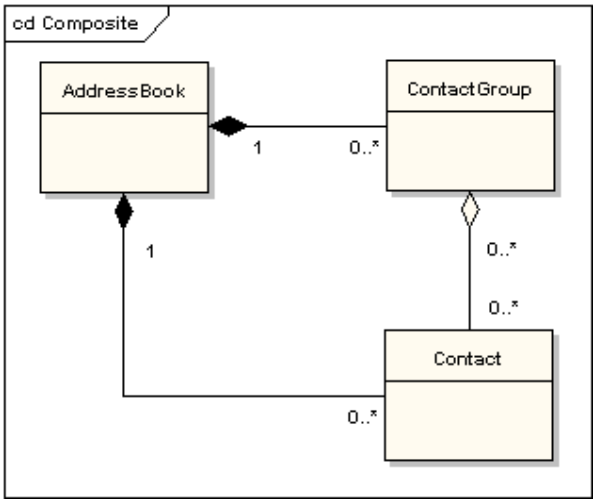


Aggregations

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.

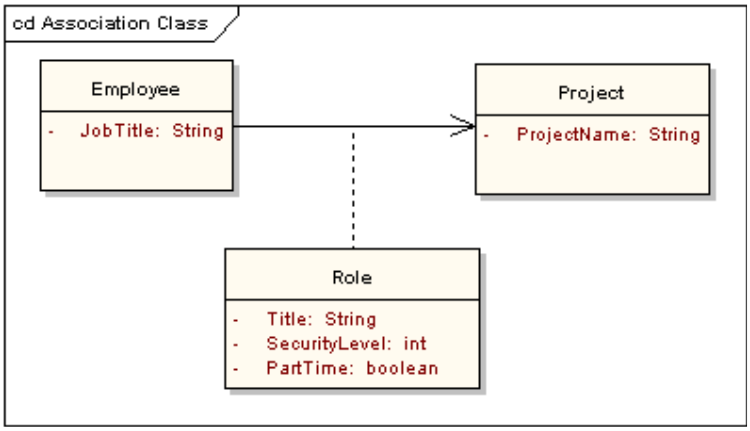
A stronger form of aggregation - a composite aggregation - is shown by a black diamond-shaped arrowhead and is used where components can be included in a maximum of one composition at a time. If the parent of a composite aggregation is deleted, usually all of its parts are deleted with it; however a 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 that the employee takes up on the project is a complex entity in its own right and contains detail that does not belong in the employee or project class. For example, an employee may be working on several projects at the same time and have different job titles and security levels on each.



Dependencies

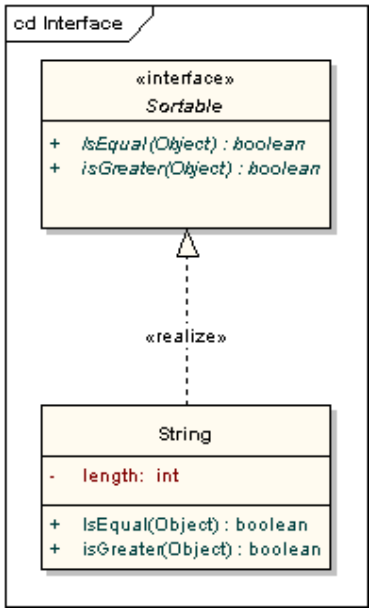
A dependency is used to model a wide range of dependent relationships between model elements. It would normally be used early in the design process where it is known that there is some kind of link between two elements but it is too early to know exactly what the relationship is. Later in the design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

Traces

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

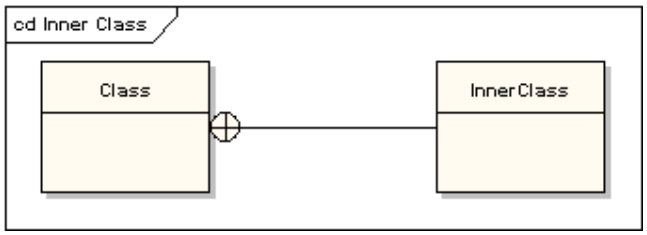
Realizations

The source object implements or realizes the destination. 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 that the source element is nested within the target element. The following diagram shows the definition of an inner class, although in EA it is more usual to show them by their position in the Project View hierarchy.

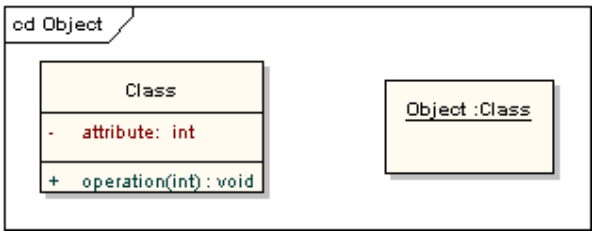


Object Diagrams

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.

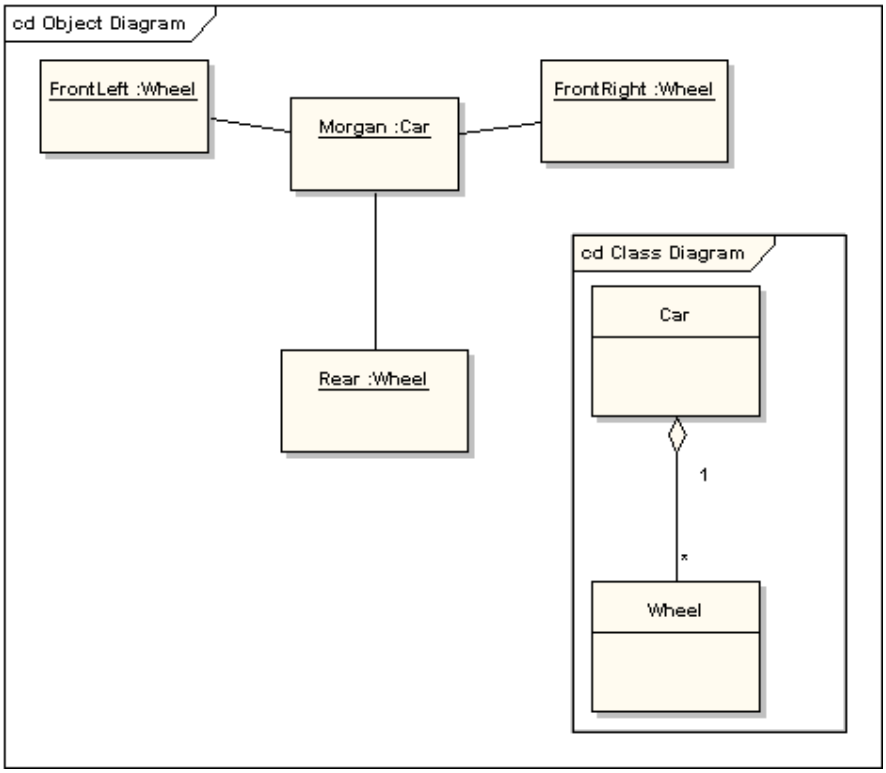


Run Time State

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

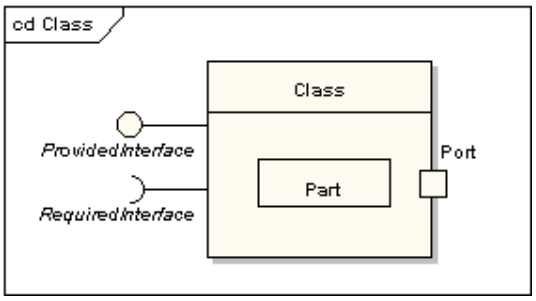
The following diagram shows an object diagram with its defining class diagram inset, and it illustrates the way in which an object diagram may be used to test the multiplicities of assignments in class diagrams. The car class has a 1-to-many multiplicity to the wheel class, but if a 1-to-4 multiplicity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


cd Object Diagram
FrontLeft :Wheel
Morgan :Car
FrontRight :Wheel
Rear :Wheel
cd Class Diagram
Car
1
*
Wheel

Composite Structure

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.

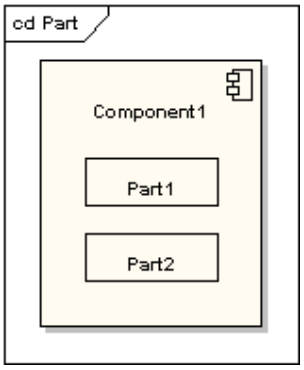
Class elements have been described in great detail in the section in class diagrams. This section describes the way that classes can be displayed as composite elements exposing interfaces and containing ports and parts.



Part

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

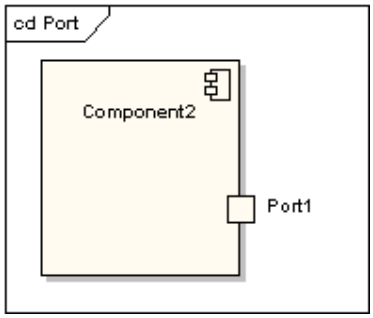
A part is shown as an unadorned rectangle contained within the body of a class or component element.



Port

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

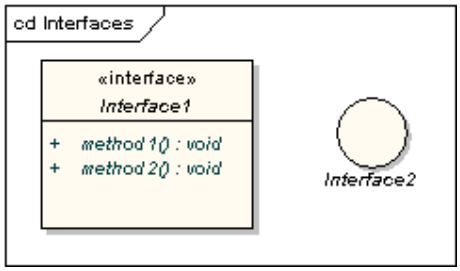
A port is shown as a named rectangle on the boundary edge of its owning classifier.



Interfaces

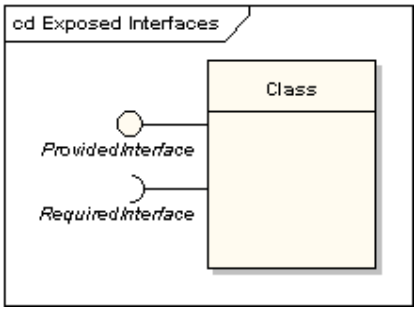
An interface is similar to a class but with a number of restrictions. All interface operations are public and abstract, and do not provide any default implementation. All interface attributes must be constraints. However, while a class may only inherit from a single super-class, it may implement multiple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



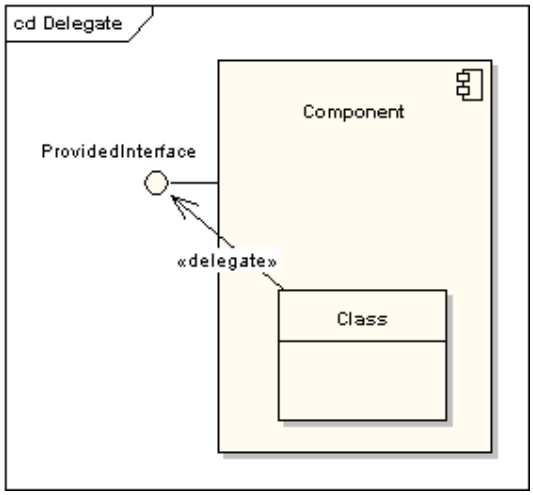
Note that the circle notation does not show the interface operations. When interfaces are shown as being owned by classes, they are referred to as exposed interfaces. An exposed interface can be defined as either provided or required. A provided interface is an affirmation that the containing classifier 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

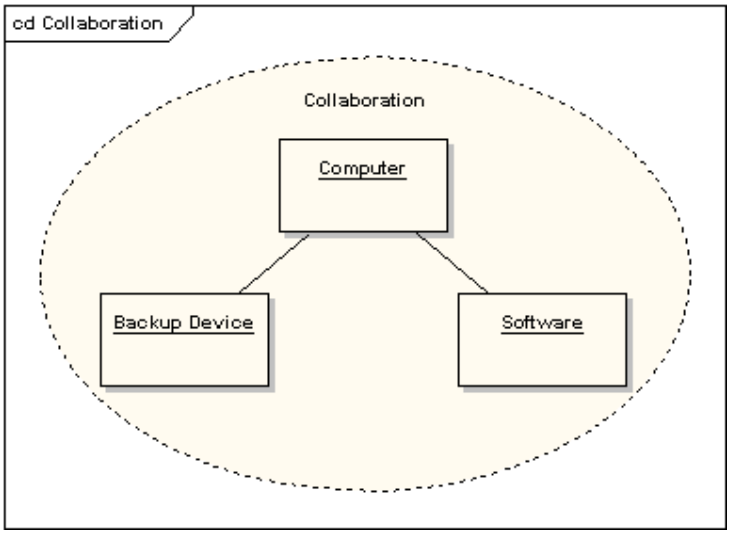
A delegate connector is used for defining the internal workings of a component's external ports and interfaces. A delegate connector is shown as an arrow with a «delegate» keyword. It connects an external contract of a component as shown by its ports to the internal realization of the behavior of the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



Collaboration

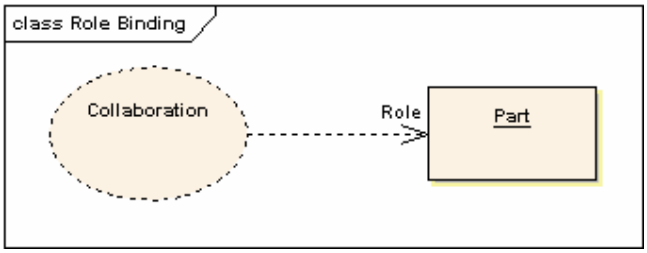
A collaboration defines a set of co-operating roles used collectively to illustrate a specific functionality. A collaboration should only show the roles and attributes required to accomplish its defined task or function. Isolating the primary roles is an exercise in simplifying the structure and clarifying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

A collaboration element is shown as an ellipse.



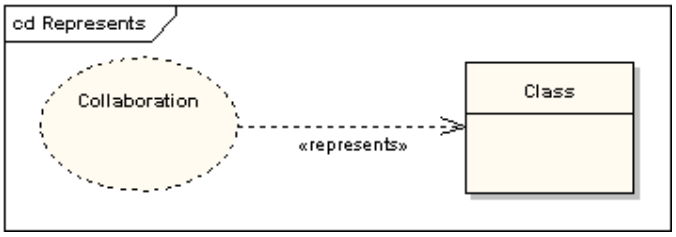
Role Binding

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.



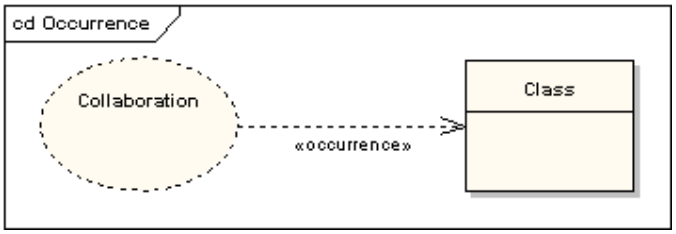
Represents

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».



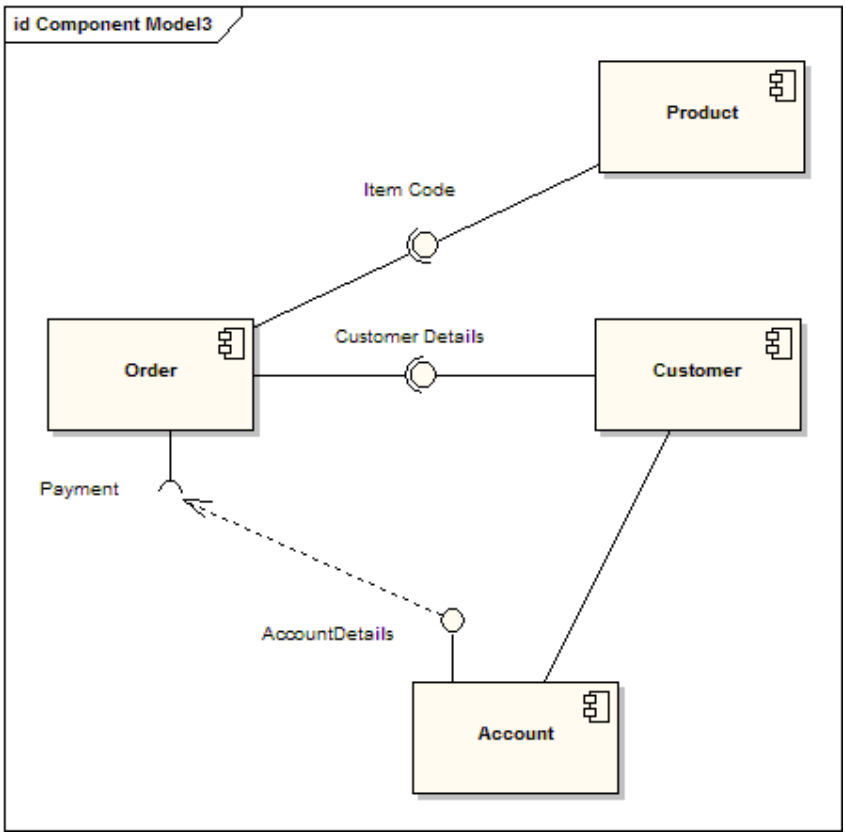
Occurrence

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».


cd Occurrence
Collaboration
«occurrence»
Class

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 the building blocks so a component can eventually encompass a large portion of a system.

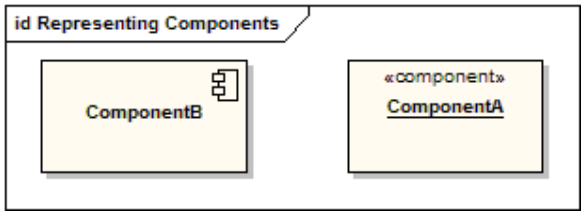


The diagram above demonstrates some components and their inter-relationships. Assembly connectors "link" the provided interfaces supplied by 'Product' and 'Customer' to the required interfaces specified by 'Order'. A dependency relationship maps a customer's associated account details to the required 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 component diagrams offer a more semantically rich grouping mechanism. With component diagrams all the model elements are private, whereas package diagrams only display public items.

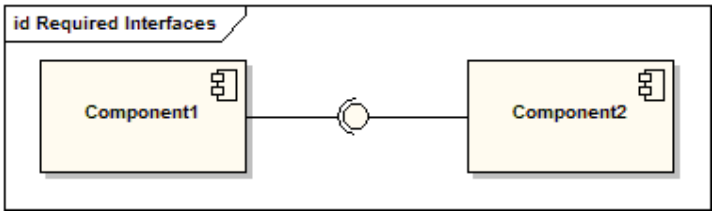
Representing Components

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.



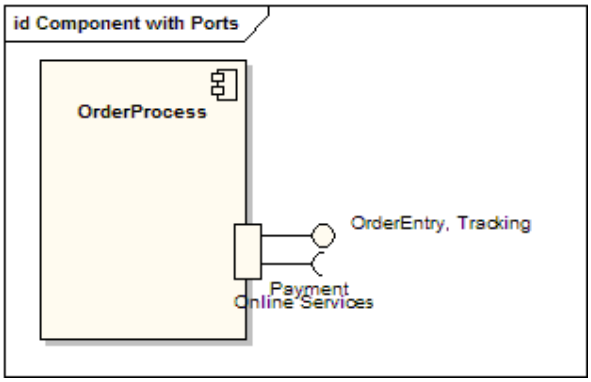
Assembly Connector

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.

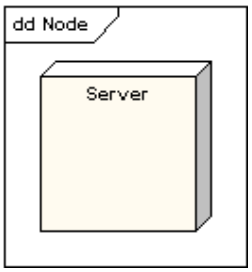


Deployment Diagrams

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.

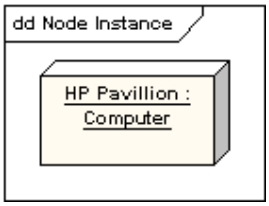
Node

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



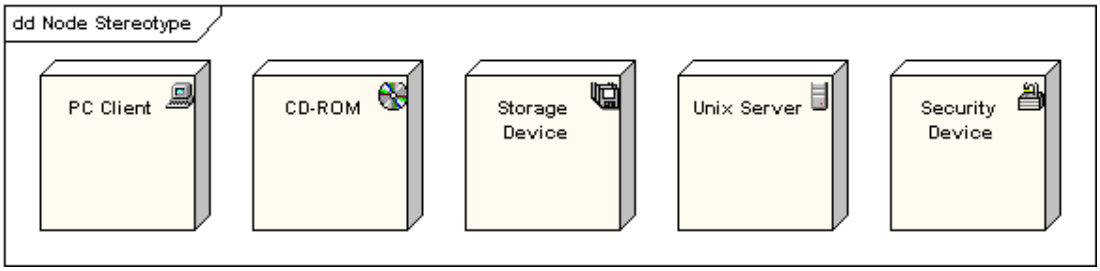
Node Instance

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

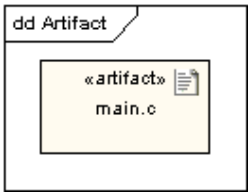
A number of standard stereotypes are provided for nodes, namely «cdrom», «cd-rom», «computer», «disk array», «pc», «pc client», «pc server», «secure», «server», «storage», «unix server», «user pc». These will display an appropriate icon in the top right corner of the node symbol.



Artifact

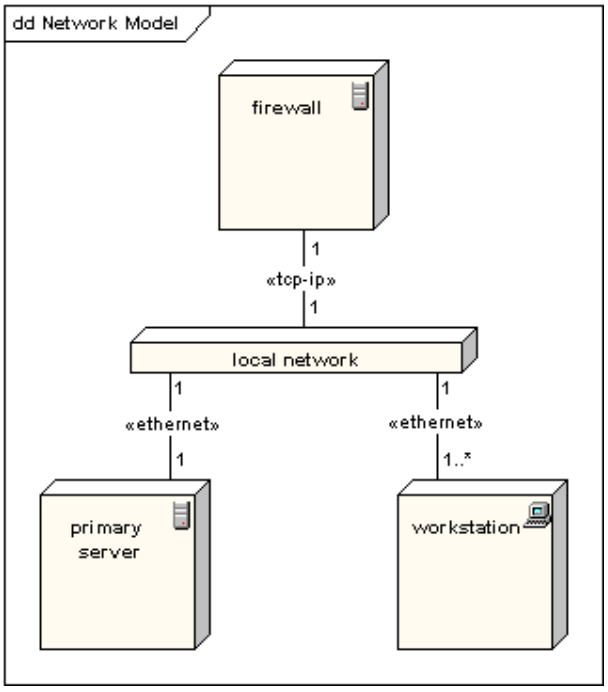
An artifact is a product of the software development process. That may include process models (e.g. use case models, design models, etc), source files, executables, design documents, test reports, prototypes, user manuals, etc.

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



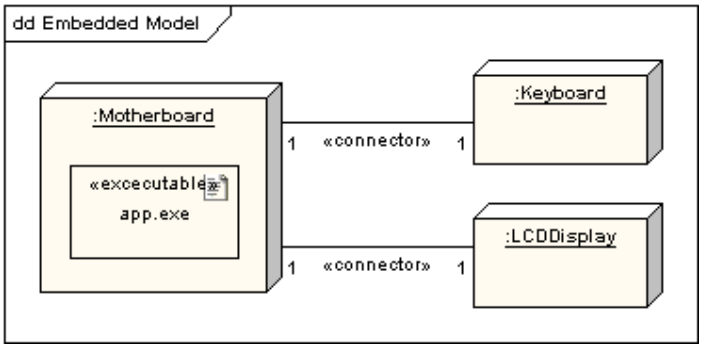
Association

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.


dd Embedded Model
:Motherboard
«executable»
app.exe
:Keyboard
:LCDDisplay
1
«connector»
1
1
«connector»
1

Recommended Reading

For more information, please refer to:

Sparx Systems' Web Site:	www.sparxsystems.com
Object Management Group's Web Site:	www.omg.com
Object Management Group's UML pages:	www.uml.org