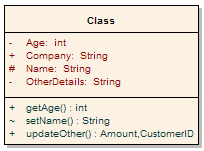
The Class Model   
A Class is a standard UML construct used to detail the pattern from which objects will be produced at run-time. A class is a specification - an object an instance of a class. Classes may be inherited from other classes (that is they inherit all the behavior and state of their parent 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:



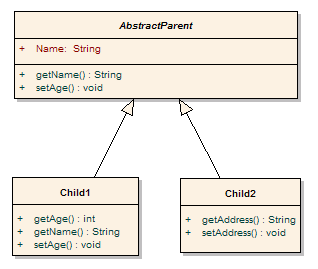
Note that the class has three distinct areas:

|  |  |
| --- | --- |
| 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 |

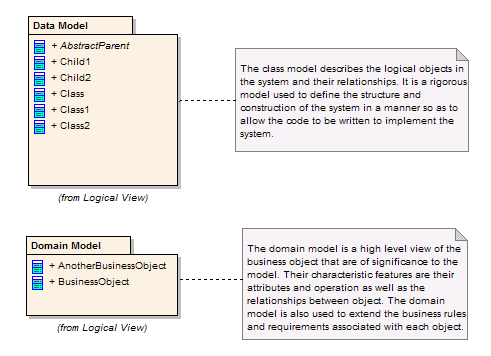
Attributes and methods may be marked as:

|  |  |
| --- | --- |
| - | 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 |

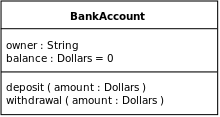
Class inheritance is shown as below: an abstract class in this case, is the parent of two children, each of which inherits the base class features and extends it with their own behavior.



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



The class diagram is the main building block of object oriented modelling. It is used both for general conceptual modelling of the systematics of the application, and for detailed modelling translating the models into programming code. Class diagrams can also be used for data modeling. The classes in a class diagram represent both the main objects, interactions in the application and the classes to be programmed.

[](http://en.wikipedia.org/wiki/File:BankAccount1.svg)

A class with three sections.

In the diagram, classes are represented with boxes which contain three parts:

The top part contains the **name** of the class. It is printed in bold and centered, and the first letter is capitalized.

The middle part contains the **attributes** or **states** of the class. They are left-aligned and the first letter is lowercase.

The bottom part contains the **methods** or **behaviors** the class can execute. They are also left-aligned and the first letter is lowercase.

In the design of a system, a number of classes are identified and grouped together in a class diagram which helps to determine the static relations between those objects. With detailed modelling, the classes of the conceptual design are often split into a number of subclasses.

In order to further describe the behavior of systems, these class diagrams can be complemented by a state diagram or UML state machine.

Members

UML provides mechanisms to represent class members, such as attributes and methods, and additional information about them.

**Visibility**

To specify the visibility of a class member (i.e., any attribute or method), these notations must be placed before the member's name:

|  |  |
| --- | --- |
| + | Public |
| - | Private |
| # | Protected |
| / | Derived (can be combined with one of the others) |
| ~ | Package |

**Scopes**

The UML specifies two types of scope for members: *instance* and *classifier*.

**Classifier members** are commonly recognized as “static” in many programming languages. The scope is the class itself.

Attribute values are equal for all instances

Method invocation does not affect the instance’s state

**Instance members** are scoped to a specific instance.

Attribute values may vary between instances

Method invocation may affect the instance’s state (i.e., change instance’s attributes)

To indicate a classifier scope for a member, its name must be underlined. Otherwise, instance scope is assumed by default.

**Relationships**

A relationship is a general term covering the specific types of logical connections found on class and object diagrams. UML shows the following relationships:

Instance level relationships

**Links**

A *Link* is the basic relationship among objects.

**Association**

[http://upload.wikimedia.org/wikipedia/commons/thumb/4/4d/UML_role_example.gif/400px-UML_role_example.gif](http://en.wikipedia.org/wiki/File:UML_role_example.gif)

Class diagram example of association between two classes

An *association* represents a family of links. A binary association (with two ends) is normally represented as a line. An association can link any number of classes. An association with three links is called a ternary association. An association can be named, and the ends of an association can be adorned with role names, ownership indicators, multiplicity, visibility, and other properties.  
There are four different types of association: bi-directional, uni-directional, Aggregation (includes Composition aggregation) and Reflexive. Bi-directional and uni-directional associations are the most common ones.  
For instance, a flight class is associated with a plane class bi-directionally. Association represents the static relationship shared among the objects of two classes.

**Aggregation**

[http://upload.wikimedia.org/wikipedia/commons/thumb/2/2a/KP-UML-Aggregation-20060420.svg/300px-KP-UML-Aggregation-20060420.svg.png](http://en.wikipedia.org/wiki/File:KP-UML-Aggregation-20060420.svg)

Class diagram showing Aggregation between two classes

*Aggregation* is a variant of the "has a" association relationship; aggregation is more specific than association. It is an association that represents a part-whole or part-of relationship. As a type of association, an aggregation can be named and have the same adornments that an association can. However, an aggregation may not involve more than two classes; it must be a binary association.

*Aggregation* can occur when a class is a collection or container of other classes, but the contained classes do not have a strong *lifecycle dependency* on the container. The contents of the container are not automatically destroyed when the container is.

In UML, it is graphically represented as a *hollow* diamond shape on the containing class with a single line that connects it to the contained class. The aggregate is semantically an extended object that is treated as a unit in many operations, although physically it is made of several lesser objects.

**Composition**

[](http://en.wikipedia.org/wiki/File:AggregationAndComposition.svg)

Class diagram showing Composition between two classes at top and Aggregation between two classes at bottom

*Composition* is a stronger variant of the "has a" association relationship; composition is more specific than aggregation.

*Composition* usually has a strong *lifecycle dependency* between instances of the container class and instances of the contained class(es): if the container is destroyed, normally every instance that it contains is destroyed as well. (Note that, where allowed, a part can be removed from a composite before the composite is deleted, and thus not be deleted as part of the composite.)

The UML graphical representation of a composition relationship is a *filled* diamond shape on the containing class end of the tree of lines that connect contained class(es) to the containing class.

Differences between **composition** and **aggregation**

**Composition** relationship: When attempting to represent real-world whole-part relationships, e.g., an engine is a part of a car.  
**Aggregation** relationship: When representing a software or database relationship, e.g., car model engine ENG01 is part of a car model CM01, as the engine, ENG01 may be also part of a different car model.  
Thus the aggregation relationship is often "catalog" containment to distinguish it from composition's "physical" containment.

**Class level relationships**

**Generalization**

[](http://en.wikipedia.org/wiki/File:KP-UML-Generalization-20060325.svg)

Class diagram showing generalization between one superclass and two subclasses

The Generalization relationship ("is a") indicates that one of the two related classes (the *subclass*) is considered to be a specialized form of the other (the *super type*) and the superclass is considered a '***Generalization'*** of the subclass. In practice, this means that any instance of the subtype is also an instance of the superclass. An exemplary tree of generalizations of this form is found in biological classification: human beings are a subclass of simian, which are a subclass of mammal, and so on. The relationship is most easily understood by the phrase 'an A is a B' (a human is a mammal, a mammal is an animal).

The UML graphical representation of a Generalization is a hollow triangle shape on the superclass end of the line (or tree of lines) that connects it to one or more subtypes.

The generalization relationship is also known as the *inheritance* or *"is a"* relationship.

The *superclass* (base class) in the generalization relationship is also known as the *"parent"*, *superclass*, *base class*, or *base type*.

The *subtype* in the specialization relationship is also known as the *"child"*, *subclass*, *derived class*, *derived type*, *inheriting class*, or *inheriting type*.

Note that this relationship bears no resemblance to the biological parent/child relationship: the use of these terms is extremely common, but can be misleading.

**Generalization-Specialization relationship**

A is a type of B

E. g. "an oak is a type of tree", "an automobile is a type of vehicle"

Generalization can only be shown on class diagrams and on Use case diagrams.

**Realization**

In UML modelling, a realization relationship is a relationship between two model elements, in which one model element (the client) realizes (implements or executes) the behavior that the other model element (the supplier) specifies.

The UML graphical representation of a Realization is a hollow triangle shape on the interface end of the *dashed* line (or tree of lines) that connects it to one or more implementers. A plain arrow head is used on the interface end of the dashed line that connects it to its users. In component diagrams, the ball-and-socket graphic convention is used (implementers expose a ball or lollipop, while users show a socket).

Realizations can only be shown on class or component diagrams.

A realization is a relationship between classes, interfaces, components, and packages that connects a client element with a supplier element. A realization relationship between classes and interfaces and between components and interfaces shows that the class realizes the operations offered by the interface.

**General relationship**

[](http://en.wikipedia.org/wiki/File:Class_Dependency.png)

Class diagram showing dependency between "Car" class and "Wheel" class (An even clearer example would be "Car depends on Wheel", because Car already *aggregates* (and not just *uses*) Wheel)

**Dependency**

Dependency is a weaker form of bond which indicates that one class depends on another because it uses it at some point in time. One class depends on another if the independent class is a parameter variable or local variable of a method of the dependent class. This is different from an association, where an attribute of the dependent class is an instance of the independent class. Sometimes the relationship between two classes is very weak. They are not implemented with member variables at all. Rather they might be implemented as member function arguments.

**Multiplicity**

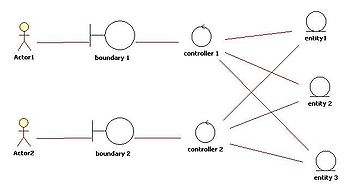
This association relationship indicates that (at least) one of the two related classes make reference to the other. This relationship is usually described as "A has a B" (a mother cat has kittens, kittens have a mother cat).

The UML representation of an association is a line with an optional arrowhead indicating the *role* of the object(s) in the relationship, and an optional notation at each end indicating the *multiplicity* of instances of that entity (the number of objects that participate in the association).

|  |  |
| --- | --- |
| **0..1** | No instances, or one instance |
| **1** | Exactly one instance |
| **0..\*** | Zero or more instances |
| **1..\*** | One or more instances |

**Analysis stereotypes**

In the early stages of a project's technical analysis, class diagrams can be used to produce early conceptual models of the system. Classes at this stage often take the form of boundaries, controls and entities and rarely survive into the design without heavy changes.

[](http://en.wikipedia.org/wiki/File:EntityControlBoundary_Pattern.jpg)

**Entities**

Entity classes model the information handled by the system, and sometimes the behavior associated with the information. They should not be identified as database tables or other data-stores.

They are drawn as circles with a short line attached to the bottom of the circle. Alternatively, they can be drawn as normal classes with the «entity» stereotype notation above the class name.