ClassDraw 1.03

UML Class Diagram Help

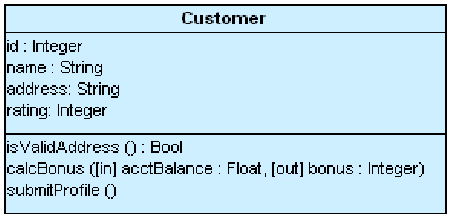
UML stands for *Unified Modeling Language* and is a visual modeling language, defined and maintained by the Object Management Group. It can be used to create models of object-oriented software to help with design, although it can also be used to model other systems that have nothing to do with software, such as business models. UML specifies several diagrams for representing different aspects of a system, of which Class Diagrams are the most well known and widely used among software developers. Class diagrams represent the static elements of a system, rather than processes or snapshots of the system at a given time and describe the classes, the data and functionality they contain and the relationships between them.

Visual modeling techniques are important for large, complex systems, but they also play a very useful role in the development of smaller systems. The use of class diagrams can help to clarify ideas in the initial design stages of an object oriented system when analyzing the requirements and deciding on the fundamental classes and how they will relate to each other. The diagram also serves as a clear visual documentation of the overall design and can be adjusted as the system develops.

The following sections provide an overview of the main features of class diagrams to assist in using ClassDraw. Further resources are widely available which discuss various methodologies for designing a system using UML. You don't have to make use of all the features of class diagrams to benefit from them. It's up to you how much detail you provide to help with your design.

The Class Icon

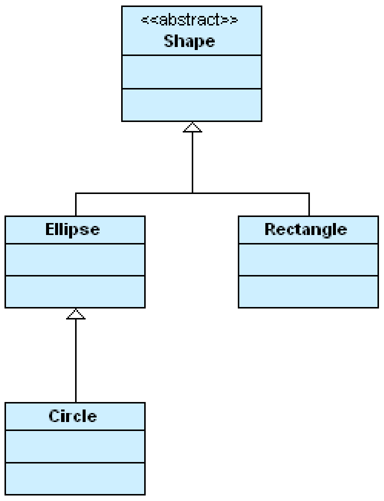
A class is represented visually using an *icon*, which is a rectangle split into three sections. The first section contains the class name, the second contains its *attributes* (member variables) and the third contains its *operations* (member functions). The class icon is intended to provide information that is useful in understanding the role of the class in the context of the rest of the diagram. It does not have to contain every attribute and operation of the class.



In UML notation, an attribute or operation ends with a colon, followed by its type. The colon and type can be omitted if the type is redundant. The type names should be chosen so that they are meaningful to those who will be looking at the diagram. Parameters can also be specified as input or output as shown. Some software developers prefer to specify attributes and operations in the more familiar format of a programming language instead of using the standard UML notation. That's fine, as long as it makes sense to everyone concerned.

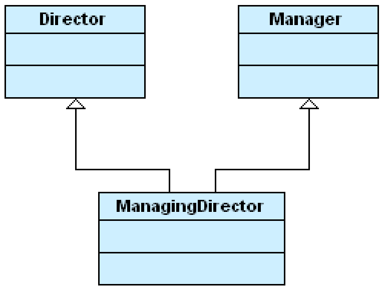
Inheritance Relationships

The inheritance relationship, also known as the *generalization* relationship, is used to indicate that one class is a specialization of another. In the following example, the Ellipse and Rectangle classes are derived from the Shape class, because they both share characteristics that are common to all shapes in the system. The Shape class is the base class in both relationships. Inheritance relationships are shown in a hierarchy with the base classes at the top and the derived classes below them.



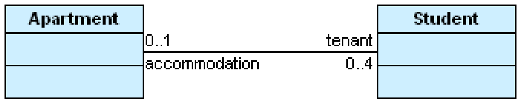
In the above example, Shape is also an *abstract class*. An abstract class is one for which an instance cannot be created, because it only makes sense to create an instance of a class derived from it. Abstract classes are represented by placing the "<<abstract>>" stereotype above the class name or by showing the class name in italics.

It is also possible for a class to inherit from multiple base classes, although some programming languages do not support multiple inheritance. In the example, a managing director is both a manager and a director.

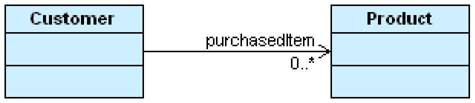


Associations

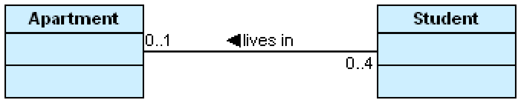
An association is a relationship between two classes represented by a solid line. Associations are bi-directional by default, so both classes know about each other and about the relationship between them. Either end of the line can have a *role name* and *multiplicity*. In the example, Student has the role of "tenant" in relation to Apartment and Apartment has the role of "accommodation" in relation to Student. Also, any instance of Apartment can be associated with up to four students and any student could be associated with 0 or 1 Apartment (a student either has an apartment to live in or does not).



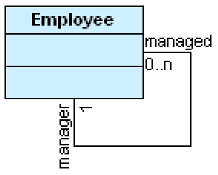
Associations can also be unidirectional, where one class knows about the other class and the relationship but the other class does not. Such associations require an open arrowhead to point to the class that is known and only the known class can have a role name and multiplicity. In the example, the Customer class knows about any number of products purchased but the Product class knows nothing about any customer. The multiplicity "0..*" means zero or more.



An alternative to using role names is to provide a single name for an association centered between the two classes. A direction indicator can also be used to show the direction of the name, but is not necessary if the direction is obvious:

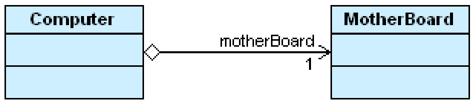


An association can also link a class to itself. Such an association is *reflexive*:



Aggregation

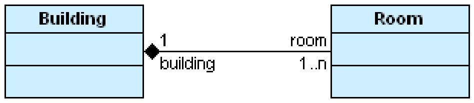
Aggregation is a relationship where one class is part of another class. In basic aggregation, the class that forms part of the whole class can exist independently, so the life of an instance of the part class is not determined by the whole class. Basic aggregation is represented using an empty diamond symbol next to the whole class. In the example, a computer in a warehouse contains a motherboard, but although the motherboard is part of the computer, it can exist as a separate item. In this system, Computer knows about Motherboard but Motherboard doesn't know about Computer, so the aggregation is unidirectional. In a program, this relationship could be implemented as a member variable in the Computer class which is a reference to a Motherboard class.



Association differs from aggregation only in that it does not imply any containment.

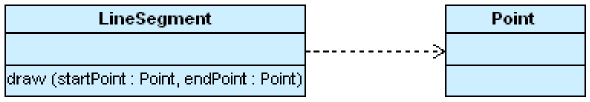
Composition

Composition is a strong type of aggregation where the whole class contains the instance of the part class. The lifetime of the part class depends on the existence of the whole class. Composition relationships are represented using a filled diamond symbol next to the whole class. In the example, a building contains a number of rooms and a room cannot exist without a building. If a Building class instance is destroyed, its Room class instances are destroyed too. In a program, this relationship could be implemented as a member variable in the Building class which is an array of Room classes and the Room class might contain a pointer to a Building class.



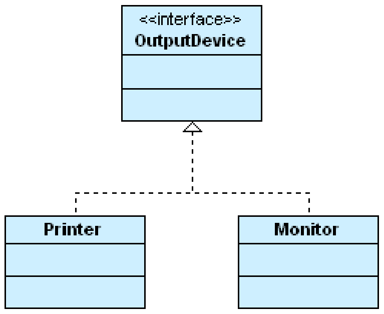
Dependencies

A dependency is a weak relationship between two classes and is represented by a dotted line. In the example, there is a dependency between Point and LineSegment, because LineSegment's draw() operation uses the Point class. It indicates that LineSegment has to know about Point, even if it has no attributes of that type. This example also illustrates how class diagrams are used to focus in on what is important in the context, as you wouldn't normally want to show such detailed dependencies for all your class operations.



Interfaces

An interface is represented in the same way as a class, except that the <<interface>> stereotype appears above the name. You can never create an instance of an interface and it must be related to at least one other class that can implement it. An interface defines a set of things that an implementing class can *do* rather than a set of things that the implementing class *is*. The links between the interface and its implementing classes are dotted, but they appear in other respects like inheritance relationships between classes. Some programming languages such as C++ do not have the concept of an interface built in, but an abstract class with no member variables and only pure virtual functions could be used to represent one. In the example, the OutputDevice interface is implemented by the Printer and Monitor classes.

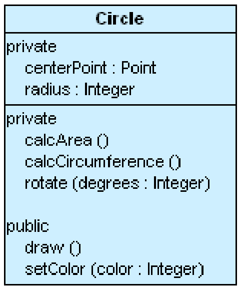


Visibility

Visibility markers indicate if an attribute or operation in a class can be accessed only from within that class (private), from within the class and any derived classes (protected), from within the package that the class is part of (package) or from anywhere (public). Visibility markers are placed at the start of the relevant attribute or operation, for example: +setConnectionStatus().

Visibility	Symbol
private	-
protected	#
package	~
public	+

An alternative to using the above markers is to group attributes and operations by visibility:


Circle
private
centerPoint : Point
radius : Integer
private
calcArea ()
calcCircumference ()
rotate (degrees : Integer)
public
draw ()
setColor (color : Integer)