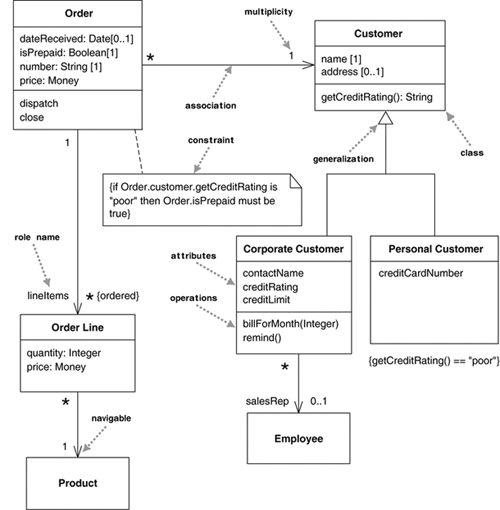
Modeling / UML

Chapter 3. Class Diagrams: The Essentials



Java:

public class OrderLine...

private int quantity;

private Money price;

private Order order;

private Product product

In a language like C#, which has properties, it would correspond to:

public class OrderLine ...

public int Quantity;

public Money Price;

public Order Order;

public Product Product;

Note that an attribute typically corresponds to public properties in a language that supports properties but to private fields in a language that does not.

The navigable association from Order to Customer in Figure 3.1 means that Order is dependent on Customer. A subclass is dependent on its superclass but not vice versa.

A class diagram is a UML diagram that describes the types of objects in the system and the various kinds of static relationships that exist among them. Class diagrams also show the properties and operations of a class and the constraints that apply to the way objects are connected. A class diagram describes the types of objects in the system and the various kinds of static relationships that exist among them. Class diagrams also show the properties and operations of a class and the constraints that apply to the way objects are connected.

**Associations and generalizations:**

**Properties** represent structural features of a class. As a first approximation, you can think of properties as corresponding to fields in a class. Properties are a single concept, but **they appear in two quite distinct notations: attributes and associations.**

The **attribute** notation describes a property as a line of text within the class box itself.

visibility name: type multiplicity = default {property-string}

An example of this is:

- name: String [1] = "Untitled" {readOnly}

This visibility marker indicates whether the attribute is public (+) or private (-).

The name of the attribute—how the class refers to the attribute—roughly corresponds to the name of a field in a programming language.

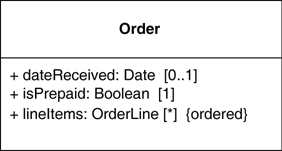
The type of the attribute indicates a restriction on what kind of object may be placed in the attribute. You can think of this as the type of a field in a programming language.

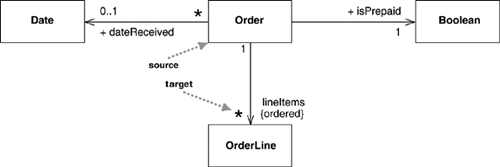
The default value is the value for a newly created object if the attribute isn’t specified during creation.

The {property-string} allows you to indicate additional properties for the attribute. In the example, I used {readOnly} to indicate that clients may not modify the property. If this is missing, you can usually assume that the attribute is modifiable. I’ll describe other property strings as we go.

**Associations:**

Much of the same information that you can show on an attribute appears on an association.





An association is a solid line between two classes, directed from the source class to the target class. Associations can show multiplicities at both ends of the line.

We tend to use attributes for small things, such as dates or Booleans—in general, value types—and associations for more significant classes, such as customers and orders.

**Multiplicity:**

The **multiplicity** of a property is an indication of how many objects may fill the property. The most common multiplicities you will see are

* **1** (An order must have exactly one customer.)
* **0..1** (A corporate customer may or may not have a single sales rep.)
* **\*** (A customer need not place an Order and there is no upper limit to the number of Orders a Customer may place—zero or more orders.)

More generally, multiplicities are defined with a lower bound and an upper bound, such as 2..4

The lower bound may be any positive number or zero; the upper is any positive number or \* (for unlimited).

In attributes, you come across various terms that refer to the multiplicity.

Optional implies a lower bound of 0.

Mandatory implies a lower bound of 1 or possibly more.

Single-valued implies an upper bound of 1.

Multivalued implies an upper bound of more than 1: usually \*.

By default, the elements in a multivalued multiplicity form a set, so if you ask a customer for its orders, they do not come back in any order. If the ordering of the orders in association has meaning, you need to add {ordered} to the association end. If you want to allow duplicates, add {nonunique}. (If you want to explicitly show the default, you can use {unordered} and {unique}.)

If an attribute is multivalued, this implies that the data concerned is a collection. So an Order class would refer to a collection of Order Lines. Because this multiplicity is ordered, that collection must be ordered, (such as a List in Java or an IList in .NET)

Multivalued properties yield a different kind of interface to single-valued properties (in Java):

class Order {

private Set lineItems = new HashSet();

public Set getLineItems() {

return Collections.unmodifiableSet(lineItems);

}

public void addLineItem (OrderItem arg) {

lineItems.add (arg);

}

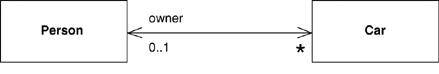
public void removeLineItem (OrderItem arg) {

lineItems.remove(arg);

}

In most cases, you don’t assign to a multivalued property; instead, you update with add and remove methods.

**Bidirectional Associations:**



Or



A bidirectional association is a pair of properties that are linked together as inverses. The Car class has property owner:Person[1], and the Person class has a property cars:Car[\*].

The inverse link between them implies that if you follow both properties, you should get back to a set that contains your starting point. For example, if I begin with a particular MG Midget, find its owner, and then look at its owner’s cars, that set should contain the Midget that I started from.

**Operations:**

The full UML syntax for operations is:

visibility name (parameter-list) : return-type {property-string}

This visibility marker is public (+) or private (-);

The name is a string.

The parameter-list is the list of parameters for the operation.

The return-type is the type of the returned value, if there is one.

The property-string indicates property values that apply to the given operation.

The parameters in the parameter list are notated in a similar way to attributes. The form is:

direction name: type = default value

The name, type, and default value are the same as for attributes.

The direction indicates whether the parameter is input (in), output (out) or both (inout). If no direction is shown, it’s assumed to be in.

An example operation on account might be:

+ balanceOn (date: Date) : Money

UML defines a **query** as an operation that gets a value from a class without changing the system state—in other words, without side effects. You can mark such an operation with the property string {query}. I refer to operations that do change state as **modifiers**, also called commands.

Strictly, the difference between query and modifiers is whether they change the observable state. The observable state is what can be perceived from the outside. An operation that updates a cache would alter the internal state but would have no effect that’s observable from the outside.

Another distinction is between operation and method. An **operation** is something that is invoked on an object—the procedure declaration—whereas a **method** is the body of a procedure. The two are different when you have polymorphism. If you have a supertype with three subtypes, each of which overrides the supertype’s getPrice operation, you have one **operation** and four **methods** that implement it.

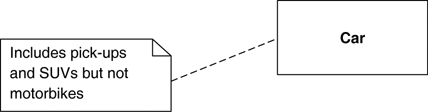
**Generalization:**

A typical example of generalization involves the personal and corporate customers of a business. They have differences but also many similarities. The similarities can be placed in a general Customer class (the supertype), with Personal Customer and Corporate Customer as subtypes.

An important principle of using inheritance effectively is substitutability. I should be able to substitute a Corporate Customer within any code that requires a Customer, and everything should work fine. Essentially, this means that if I write code assuming I have a Customer, I can freely use any subtype of Customer. The Corporate Customer may respond to certain commands differently from another Customer, using polymorphism, but the caller should not need to worry about the difference.

Many other mechanisms can be used to provide substitutable classes. As a result, many people like to differentiate between subtyping, or interface inheritance, and subclassing, or implementation inheritance. A class is a subtype if it is substitutable for its supertype, whether or not it uses inheritance. Subclassing is used as a synonym for regular inheritance.

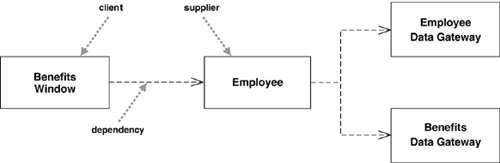
**Notes and Comments:**  
They can appear in any kind of diagram.



**Dependency:**

A dependency exists between two elements if changes to the definition of one element (the supplier) may cause changes to the other (the client). A dependency exists between two elements if changes to the definition of one element (the supplier) may cause changes to the other (the client). With classes, dependencies exist for various reasons: One class sends a message to another; one class has another as part of its data; one class mentions another as a parameter to an operation.

The UML allows you to depict dependencies between all sorts of elements. You use dependencies whenever you want to show how changes in one element might alter other elements.



The Benefits Window class—a user interface, or presentation class—is dependent on the Employee class: a domain object that captures the essential behavior of the system—in this case, business rules. This means that if the employee class changes its interface, the Benefits Window may have to change.

The important thing here is that the dependency is in only one direction and goes from the presentation class to the domain class. This way, we know that we can freely alter the Benefits Window without those changes having any effect on the Employee or other domain objects.

A second notable thing from this diagram is that there is no direct dependency from the Benefits Window to the two Data Gateway classes. If these classes change, the Employee class may have to change. But if the change is only to the **implementation** of the Employee class, not its **interface**, the change stops there.

The basic dependency is not a transitive relationship. An example of a transitive relationship is the “larger beard” relationship. If Jim has a larger beard than Grady, and Grady has a larger beard than Ivar, we can deduce that Jim has a larger beard than Ivar. Some kind of dependencies, such as substitute, are transitive, but in most cases there is a significant difference between direct and indirect dependencies.

A subclass is dependent on its superclass but not vice versa.

**Constraints:**

The UML allows you to use anything to describe constraints. The only rule is that you put them inside braces ({}). Optionally, you can name a constraint by putting the name first, followed by a colon; for example, {disallow incest: husband and wife must not be siblings}.

Quiz:

1 In UML class diagrams, which of the following multiplicities implies a lower bound of 0?

Single-Valued

Optional

Mandatory

Multivalued

2 In UML class diagrams public fields and methods are represented by:

\*

+

#

-

[Flag question: Question 4](https://byu.instructure.com/courses/12738/quizzes/245230/take)

**Question 41 pts**

Deletion of a participant is indicated by an "X" on its lifeline.

Group of answer choices



True



False