**What Is the UML?**

The Unified Modeling Language (UML) is a family of graphical notations,

backed by single meta-model, that help in describing and designing software

systems, particularly software systems built using the object-oriented (OO)

style.

Graphical modeling languages have been around in the software industry for

a long time. The fundamental driver behind them all is that programming languages

are not at a high enough level of abstraction to facilitate discussions

about design.

**Ways of Using the UML**

**UML as sketch**: emphasis is on selective communication rather than complete specification.

**UML as blueprint**: intend to be comprehensive, often with the aim of reducing programming to a simple and fairly mechanical activity.

**UML as programming language**: developers draw UML diagrams that are compiled directly to executable code, and the UML becomes the source code.

. Forward Engineering

. Reverse Engineering

. Conceptual Perspective

. Software Perspective

**Notations and Meta-Models**

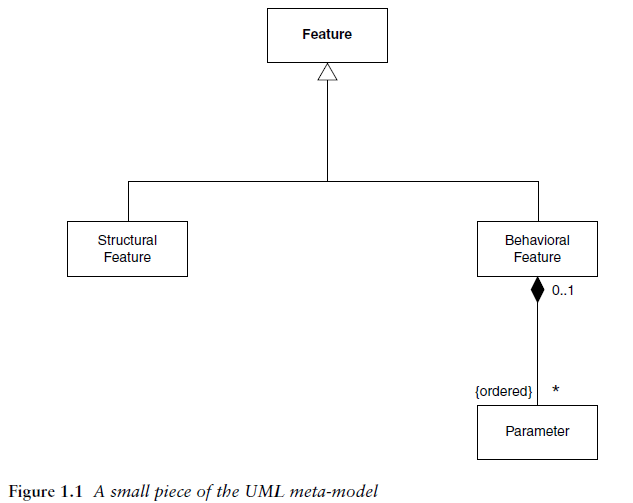
The **notation** is the graphical stuff you see in models;

it is the graphical syntax of themodeling language.

**meta-model:** a diagram, usually a class diagram, that defines the concepts of

the language.

Intuition rather than to formal definition.

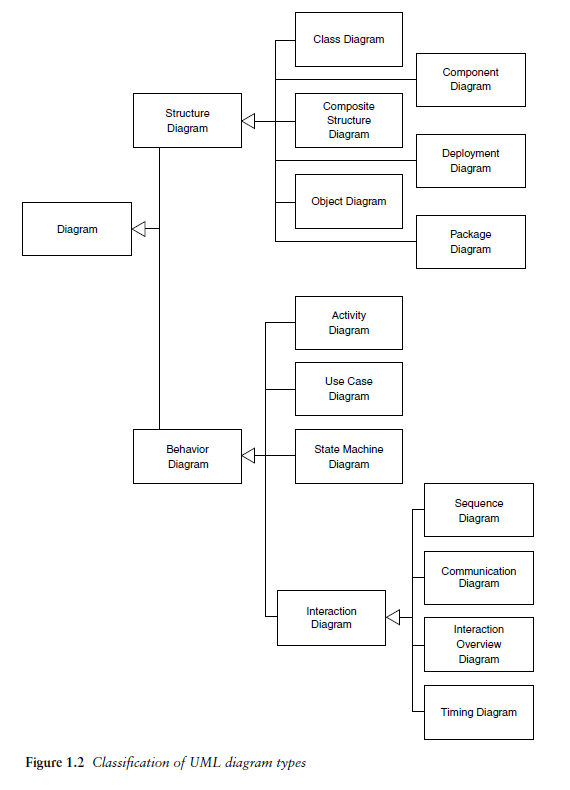


**TIP** - As you get deeper into the more detailed usage of the UML, you realize that

you need much more than the graphical notation. This is why UML tools are so

complex.

**UML Diagrams**



**What Is Legal UML?**

**prescriptive rules**

**descriptive rules**

**TIP** - You cannot look at a UML diagram and say *exactly* what

the equivalent code would look like. However, you can get a *rough idea* of what

the code would look like.

**UML Is Not Enough**

You shouldn’t hesitate to use a non-UML diagram if no UML diagram

suits your purpose.

. Screen flow diagram

. Decision table

**Class Diagrams**

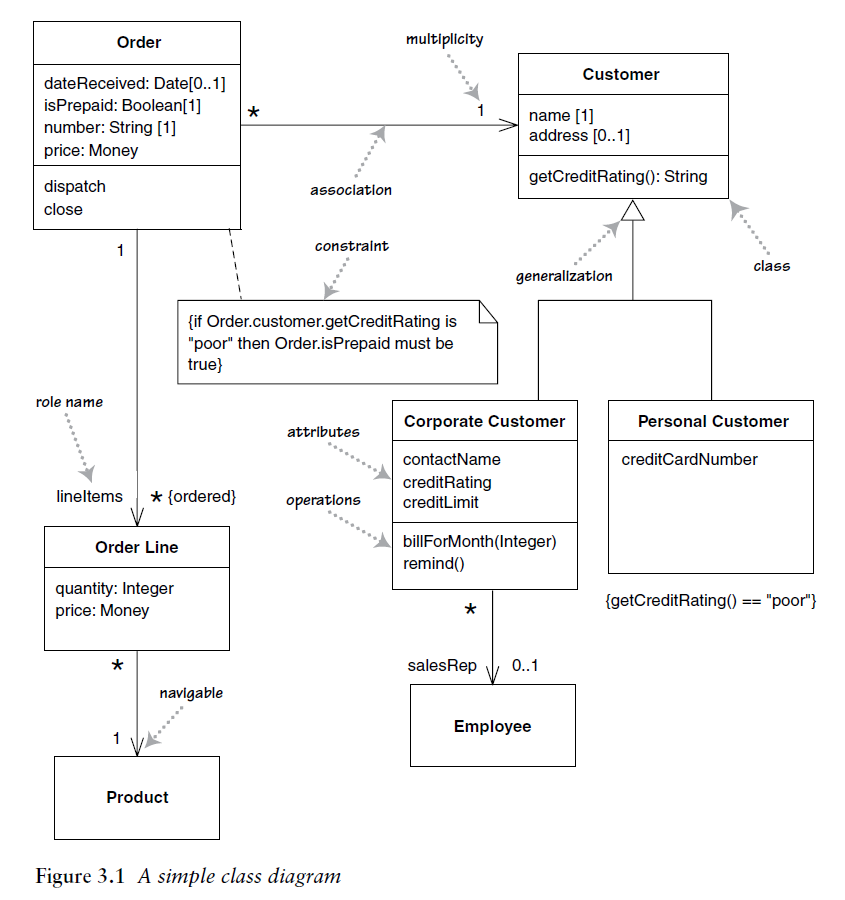
A **class diagram** describes the types of objects in the system and the various

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 connectd. The UML uses the term **feature** as a general term

that covers properties and operations of a class.



**TIP** - **Properties** are a single concept, but they appear in two quite distinct notations:

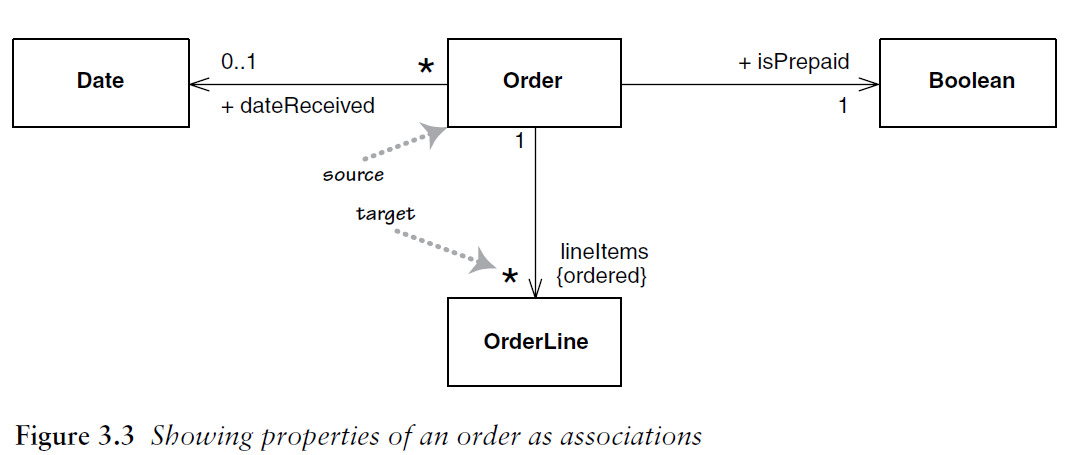
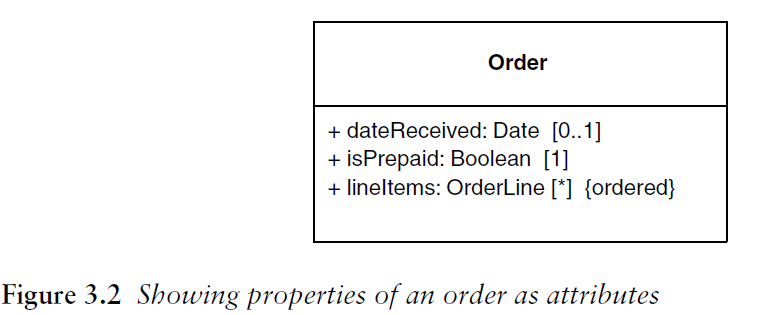
attributes and associations. Although they look quite different on a diagram,

they are really the same thing.

**Associations**

An **association** is a solid line between two classes, directed from the source

class to the target class. The name of the property goes at the target end of the association, together with its multiplicity. The target end of the association links to the class that is the type of the property.



In general, I 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.)

**TIP** - If the lower and upper bounds are the same, you can use one number;

hence, 1 is equivalent to 1..1. Because it’s a common case, \* is short for 0..\*.

**TIP** - The default multiplicity of an attribute is [1].

**TIP** - If the ordering of the items 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**}.) You may also see collection-oriented names, such as {**bag**} for unordered, nonunique.

**TIP** - You should be very afraid of classes that are nothing but a collection of fields

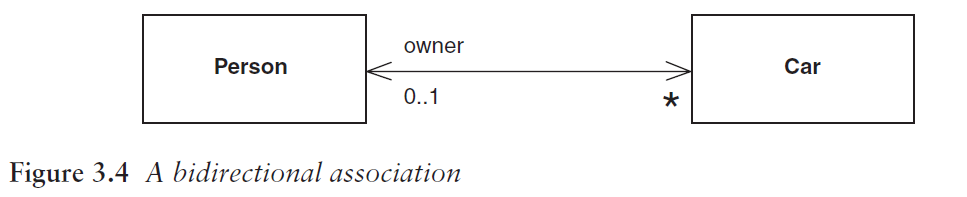
and their accessors. Object-oriented design is about providing objects that are

able to do rich behavior, so they shouldn’t be simply providing data to other

objects. If you are making repeated calls for data by using accessors, that’s a

sign that some **behavior should be moved to the object that has the data**.

**Bidirectional Associations**



A bidirectional association is a pair of properties that are linked together as

inverses. The Car class has property owner:Person [*0*..1] , and the Person class has a property cars:Car[\*]. (Note how I named the cars property in the plural form of the property’s type, a common but non-normative convention.)

The inverse link between them implies that if you follow both properties,

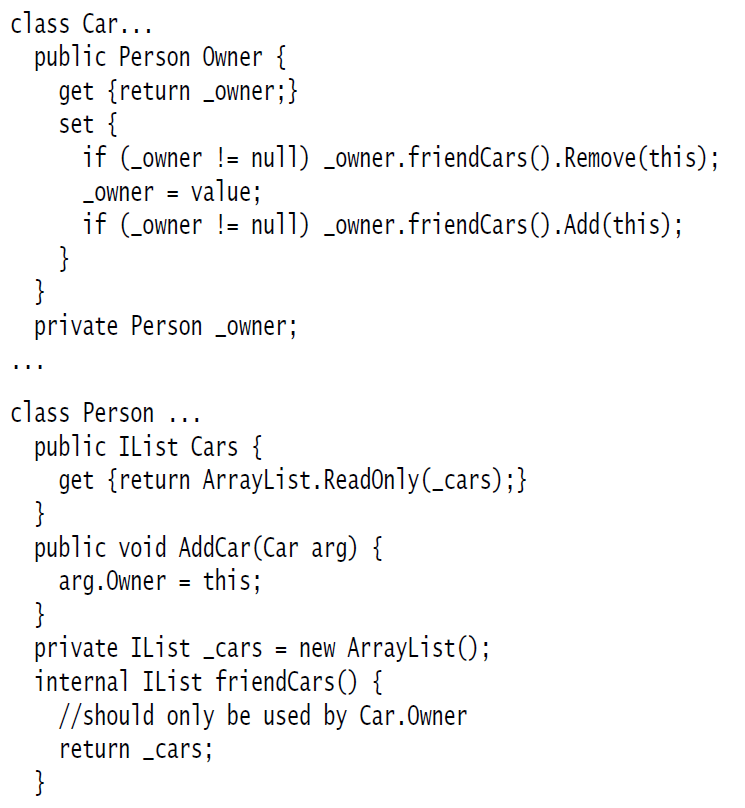
you should get back to a set that contains your starting point. For example, if I

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.

The primary thing is to let one side of the association—a single-valued side,

if possible—control the relationship. For this to work, the slave end (Person)

needs to leak the encapsulation of its data to the master end. This adds to the slave class an awkward method, which shouldn’t really be there, unless the language has fine-grained access control.



**Operations**

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

E.g. -> + balanceOn (date: Date) : Money

**TIP** - 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**

With a software perspective, the obvious interpretation is **inheritance**.

With perspectives of modeling, idea is that everything we say about a Supertype—associations,

attributes, operations—is true also for a Subtype.

**Subtyping vs. Subclassing**

Subtyping: is about interface inheritance

Subclassing: is about implementation inheritance

**Notes and Comments**