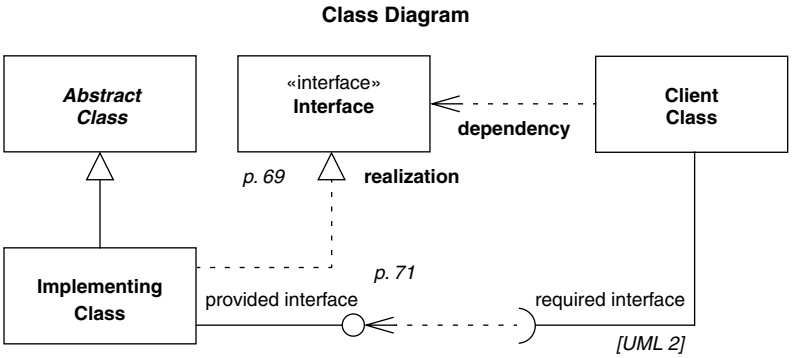
Covers through Version 2.0 OMG UML Standard

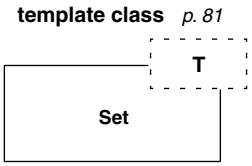
UML Distilled Third Edition

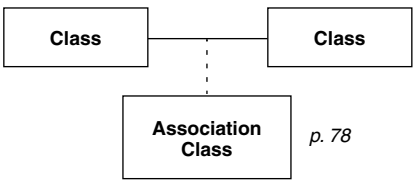
A Brief Guide to the Standard Object Modeling Language

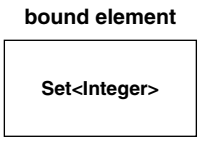
MARTIN FOWLER

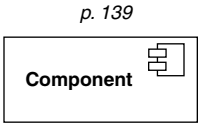
Forewords by Cris Kobryn, Grady Booch, Ivar Jacobson, and Jim Rumbaugh

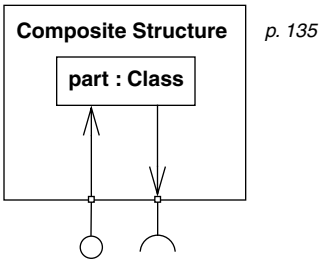

Class Diagram
Abstract Class
«interface» Interface
Client Class
dependency
p. 69
realization
Implementing Class
p. 71
provided interface
required interface
[UML 2]

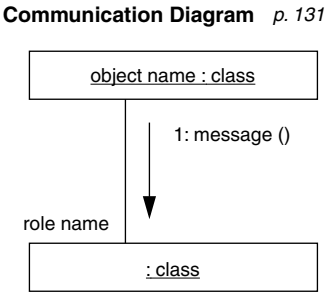

template class p. 81
T
Set

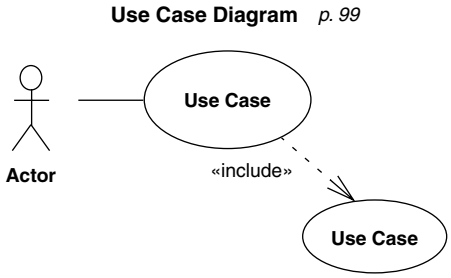

Class
Class
Association Class
p. 78


bound element
Set<Integer>


p. 139
Component


Composite Structure
p. 135
part : Class


Communication Diagram p. 131
object name : class
1: message ()
role name
: class


Use Case Diagram p. 99
Use Case
Actor
«include»
Use Case

help illustrate the discussion. I prefer the diagrams as sketches that highlight the most important parts of the system. Obviously, the writer of the document needs to decide what is important and what isn't, but the writer is much better equipped than the reader to do that.

A package diagram makes a good logical road map of the system. This diagram helps me understand the logical pieces of the system and see the dependencies and keep them under control. A deployment diagram (see Chapter 8), which shows the high-level physical picture, may also prove useful at this stage.

Within each package, I like to see a class diagram. I don't show every operation on every class. I show only the important features that help me understand what is in there. This class diagram acts as a graphical table of contents.

The class diagram should be supported by a handful of interaction diagrams that show the most important interactions in the system. Again, selectivity is important here; remember that, in this kind of document, comprehensiveness is the enemy of comprehensibility.

If a class has complex life-cycle behavior, I draw a state machine diagram (see Chapter 10) to describe it. I do this only if the behavior is sufficiently complex, which I find doesn't happen often.

I'll often include some important code, written in a literate program style. If a particularly complex algorithm is involved, I'll consider using an activity diagram (see Chapter 11) but only if it gives me more understanding than the code alone.

If I find concepts that are coming up repeatedly, I use patterns (page 27) to capture the basic ideas.

One of the most important things to document is the design alternatives you didn't take and why you didn't do them. That's often the most forgotten but most useful piece of external documentation you can provide.

Understanding Legacy Code

The UML can help you figure out a gnarly bunch of unfamiliar code in a couple of ways. Building a sketch of key facts can act as a graphical note-taking mechanism that helps you capture important information as you learn about it. Sketches of key classes in a package and their key interactions can help clarify what's going on.

With modern tools, you can generate detailed diagrams for key parts of a system. Don't use these tools to generate big paper reports; instead, use them to drill into key areas as you are exploring the code itself. A particularly nice capability is that of generating a sequence diagram to see how multiple objects collaborate in handling a complex method.

Choosing a Development Process

I'm strongly in favor of iterative development processes. As I've said in this book before: You should use iterative development only on projects that you want to succeed.

Perhaps that's a bit glib, but as I get older, I get more aggressive about using iterative development. Done well, it is an essential technique, one you can use to expose risk early and to obtain better control over development. It is not the same as having no management, although to be fair, I should point out that some have used it that way. It does need to be well planned. But it is a solid approach, and every OO development book encourages using it—for good reason.

You should not be surprised to hear that as one the authors of the Manifesto for Agile Software Development, I'm very much a fan of agile approaches. I've also had a lot of positive experiences with Extreme Programming, and certainly you should consider its practices very seriously.

Where to Find Out More

Books on software process have always been common, and the rise of agile software development has led to many new books. Overall, my favorite book on process in general is [McConnell]. He gives a broad and practical coverage of many of the issues involved in software development and a long list of useful practices.

From the agile community, [Cockburn, agile] and [Highsmith] provide a good overview. For a lot of good advice about applying the UML in an agile way, see [Ambler].

One of the most popular agile methods is Extreme Programming (XP), which you can delve into via such Web sites as **http://xprogramming.com** and **http://www.extremeprogramming.org.** XP has spawned many books, which is why I now refer to it as the formerly lightweight methodology. The usual starting point is [Beck].

Although it's written for XP, [Beck and Fowler] gives more details on planning an iterative project. Much of this is also covered by the other XP books, but if you're interested only in the planning aspect, this would be a good choice.

For more information on the Rational Unified Process, my favorite introduction is [Kruchten].

This page intentionally left blank

Chapter 3

Class Diagrams: The Essentials

If someone were to come up to you in a dark alley and say, "Psst, wanna see a UML diagram?" that diagram would probably be a class diagram. The majority of UML diagrams I see are class diagrams.

The class diagram is not only widely used but also subject to the greatest range of modeling concepts. Although the basic elements are needed by everyone, the advanced concepts are used less often. Therefore, I've broken my discussion of class diagrams into two parts: the essentials (this chapter) and the advanced (Chapter 5).

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. The UML uses the term **feature** as a general term that covers properties and operations of a class.

Figure 3.1 shows a simple class model that would not surprise anyone who has worked with order processing. The boxes in the diagram are classes, which are divided into three compartments: the name of the class (in bold), its attributes, and its operations. Figure 3.1 also shows two kinds of relationships between classes: associations and generalizations.

Properties

Properties represent structural features of a class. As a first approximation, you can think of properties as corresponding to fields in a class. The reality is rather involved, as we shall see, but that's a reasonable place to start.

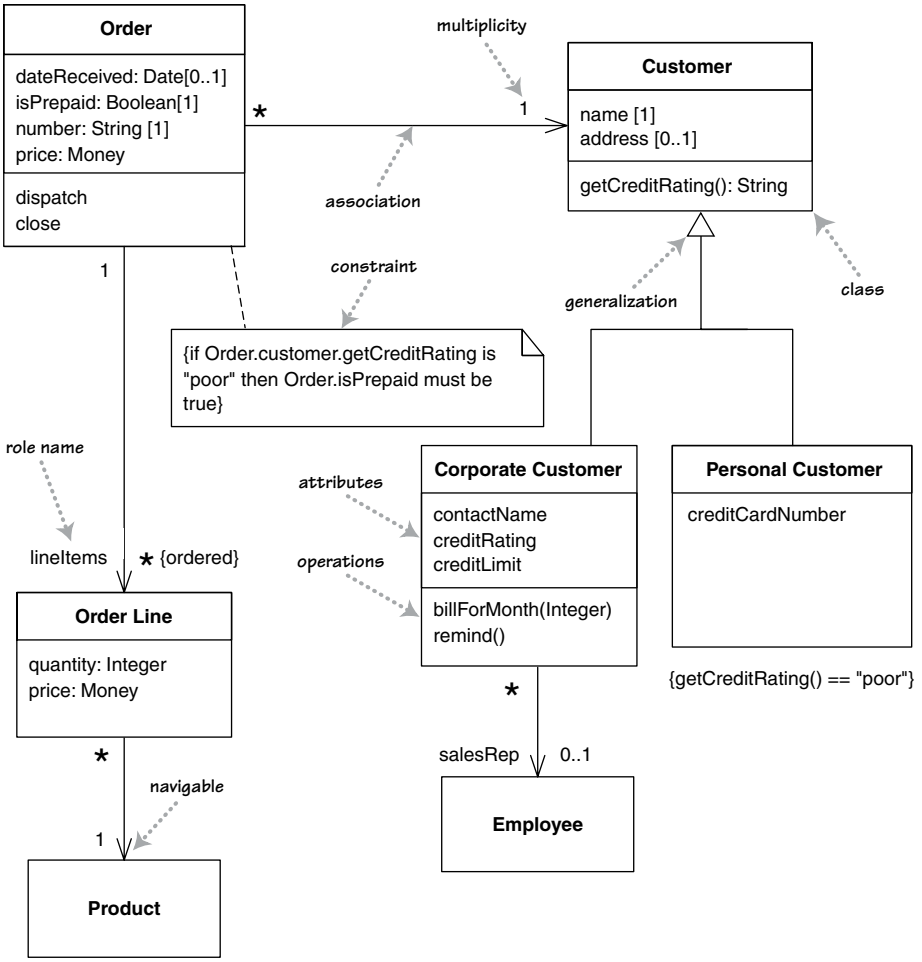


Figure 3.1 *A simple class diagram*

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.

Attributes

The **attribute** notation describes a property as a line of text within the class box itself. The full form of an attribute is:

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

An example of this is:

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

Only the `name` is necessary.

- This `visibility` marker indicates whether the attribute is public (+) or private (-); I'll discuss other visibilities on page 83.
- 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.
- I'll explain `multiplicity` on page 38.
- 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

The other way to notate a property is as an association. Much of the same information that you can show on an attribute appears on an association. Figures 3.2 and 3.3 show the same properties represented in the two different notations.

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

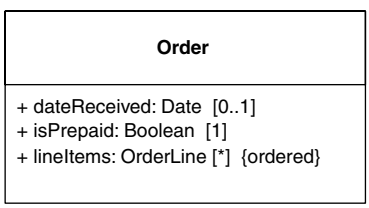


Figure 3.2 *Showing properties of an order as attributes*