Chapter 6. Object Diagrams

An **object diagram** is a snapshot of the objects in a system at a point in time. Because it shows instances rather than classes, an object diagram is often called an instance diagram.

You can use an object diagram to show an example configuration of objects. (See Figure 6.1, which shows a set of classes, and Figure 6.2, which shows an associated set of objects.) This latter use is very useful when the possible connections between objects are complicated.

Figure 6.1. Class diagram of Party composition structure

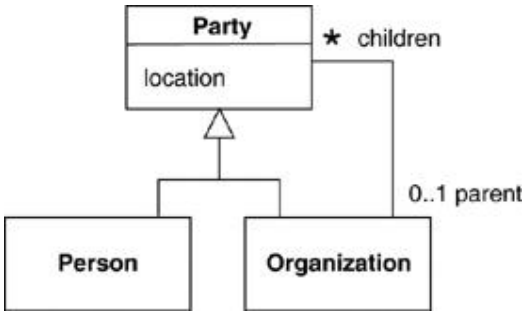
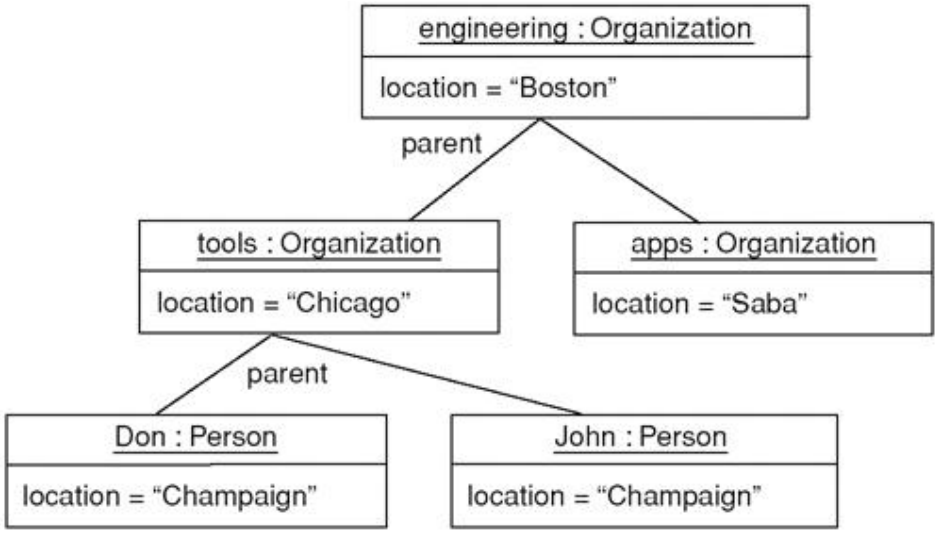


Figure 6.2. Object diagram showing example instances of Party



You can tell that the elements in Figure 6.2 are instances because the names are underlined. Each name takes the form `instance name : class name`. Both parts of the name are optional, so `John`, `:Person`, and `aPerson` are legal names. If you use only the class name, you must include the colon. You can show values for attributes and links, as in Figure 6.2.

Strictly, the elements of an object diagram are instance specifications rather than true instances. The reason is that it's legal to leave mandatory attributes empty or to show instance specifications of abstract classes. You can think of an **instance specification** as a partly defined instance.

Another way of looking at an object diagram is as a communication diagram (page 131) without messages.

When to Use Object Diagrams

Object diagrams are useful for showing examples of objects connected together. In many situations, you can define a structure precisely with a class diagram, but the structure is still difficult to understand. In these situations, a couple of object diagram examples can make all the difference.

Chapter 7. Package Diagrams

Classes represent the basic form of structuring an object-oriented system. Although they are wonderfully useful, you need something more to structure large systems, which may have hundreds of classes.

A **package** is a grouping construct that allows you to take any construct in the UML and group its elements together into higher-level units. Its most common use is to group classes, and that's the way I'm describing it here, but remember that you can use packages for every other bit of the UML as well.

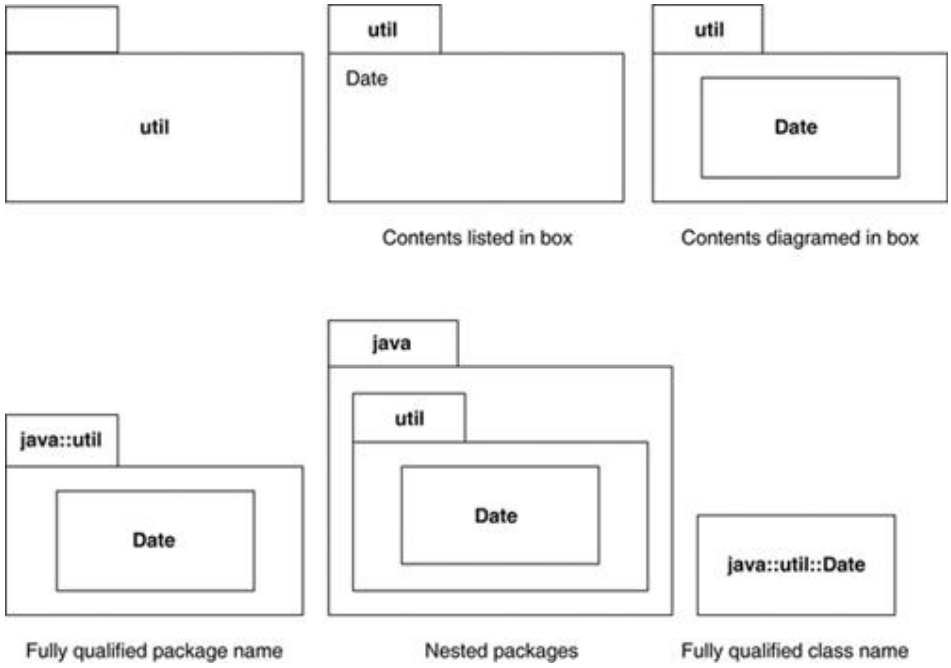
In a UML model, each class is a member of a single package. Packages can also be members of other packages, so you are left with a hierarchic structure in which top-level packages get broken down into subpackages with their own subpackages and so on until the hierarchy bottoms out in classes. A package can contain both subpackages and classes.

In programming terms, packages correspond to such grouping constructs as packages (in Java) and namespaces (in C++ and .NET).

Each package represents a **namespace**, which means that every class must have a unique name within its owning package. If I want to create a class called Date, and a Date class is already in the System package, I can have my Date class as long as I put it in a separate package. To make it clear which is which, I can use a **fully qualified name**, that is, a name that shows the owning package structure. You use double colons to show package names in UML, so the dates might be `System::Date` and `MartinFowler::Util::Date`.

In diagrams, packages are shown with a tabbed folder, as in Figure 7.1. You can simply show the package name or show the contents too. At any point, you can use fully qualified names or simply regular names. Showing the contents with class icons allows you to show all the details of a class, even to the point of showing a class diagram within the package. Simply listing the names makes sense when all you want to do is indicate which classes are in which packages.

Figure 7.1. Ways of showing packages on diagrams



It's quite common to see a class labeled something like `Date (from java.util)` rather than the fully qualified form. This style is a convention that was done a lot by Rational Rose; it isn't part of the standard.

The UML allows classes in a package to be public or private. A public class is part of the interface of the package and can be used by classes in other packages; a private class is hidden. Different programming environments have different rules about visibility between their packaging constructs; you should follow the convention of your programming environment, even if it means bending the UML's rules.

A useful technique here is to reduce the interface of the package by exporting only a small subset of the operations associated with the package's public classes. You can do this by giving all classes private visibility, so that they can be seen only by other classes in the same package, and by adding extra public classes for the public behavior. These extra classes, called *Facades* [Gang of Four], then delegate public operations to their shyer companions in the package.

How do you choose which classes to put in which packages? This is actually quite an involved question that needs a good bit of design skill to answer. Two useful principles are the Common Closure Principle and Common Reuse Principle [Martin]. The Common Closure Principle says that the classes in a package should need changing for similar reasons. The Common Reuse Principle says that classes in a package should all be reused together. Many of the reasons for grouping classes in packages have to do with the dependencies between the packages, which I'll come to next.

Packages and Dependencies

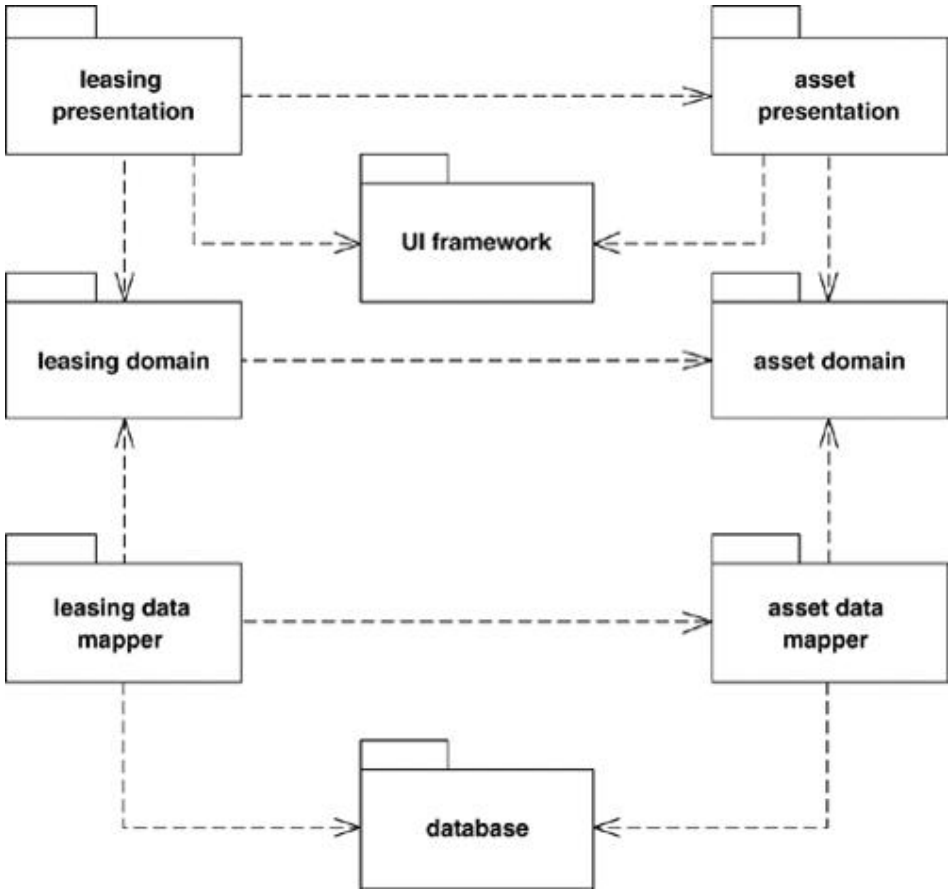
A **package diagram** shows packages and their dependencies. I introduced the concept of dependency on page 47. If you have packages for presentation and domain, you have a dependency from the presentation package to the domain package if any class in the presentation package has a dependency to any class in the domain package. In this way, interpackage dependencies summarize the dependencies between their contents.

The UML has many varieties of dependency, each with particular semantics and stereotype. I find it easier to begin with the unstereotyped dependency and use the more particular dependencies only if I need to, which I hardly ever do.

In a medium to large system, plotting a package diagram can be one of the most valuable things you can do to control the large-scale structure of the system. Ideally, this diagram should be generated from the code base itself, so that you can see what is really there in the system.

A good package structure has a clear flow to the dependencies, a concept that's difficult to define but often easier to recognize. Figure 7.2 shows a sample package diagram for an enterprise application, one that is well-structured and has a clear flow.

Figure 7.2. Package diagram for an enterprise application



Often, you can identify a clear flow because all the dependencies run in a single direction. Although that is a good indicator of a well-structured system, the data mapper packages of Figure 7.2 show an exception to that rule of thumb. The data mapper packages act as an insulating layer between the domain and database packages, an example of the Mapper pattern [Fowler, P of EAA].

Many authors say that there should be no cycles in the dependencies (the Acyclic Dependency Principle [Martin]). I don't treat that as an absolute rule, but I do think that cycles should be localized and that, in particular, you shouldn't have cycles that cross layers.

The more dependencies coming into a package, the more stable the package's interface needs to be, as any change in its interface will ripple into all the packages that are dependent on it (the Stable Dependencies Principle [Martin]). So in Figure 7.2, the asset domain package needs a more stable interface than the leasing data mapper package. Often, you'll find that the more stable packages tend to have a higher proportion of interfaces and abstract classes (the Stable Abstractions Principle [Martin].

The dependency relationships are not transitive (page 48). To see why this is important for dependencies, look at Figure 7.2 again. If a class in the asset domain package changes, we may have a change to classes within the leasing domain package. But this change does not necessarily ripple through to the leasing presentation. (It ripples only if the leasing domain changes its interface.)

Some packages are used in so many places that it would be a mess to draw all the dependency lines to them. In this case, a convention is to use a keyword, such as `«global»`, on the package.

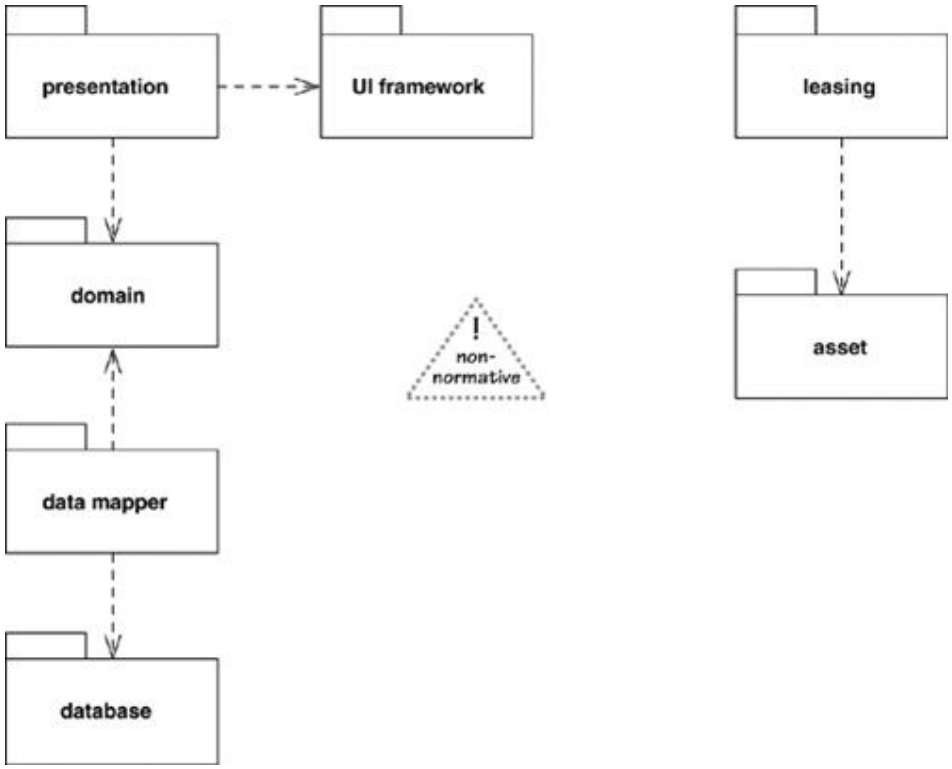
UML packages also define constructs to allow packages to import and merge classes from one package into another, using dependencies with keywords to notate this. However, rules for this kind of thing vary greatly with programming languages. On the whole, I find the general notion of dependencies to be far more useful in practice.

Package Aspects

If you think about Figure 7.2, you'll realize that the diagram has two kinds of structures. One is a structure of layers in the application: presentation, domain, data mapper, and database. The other is a structure of subject areas: leasing and assets.

You can make this more apparent by separating the two aspects, as in Figure 7.3. With this diagram, you can clearly see each aspect. However, these two aspects aren't true packages, because you can't assign classes to a single package. (You would have to pick one from each aspect.) This problem mirrors the problem in the hierarchic namespaces in programming languages. Although diagrams like Figure 7.3 are nonstandard UML, they are often very helpful in explaining the structure of a complex application.

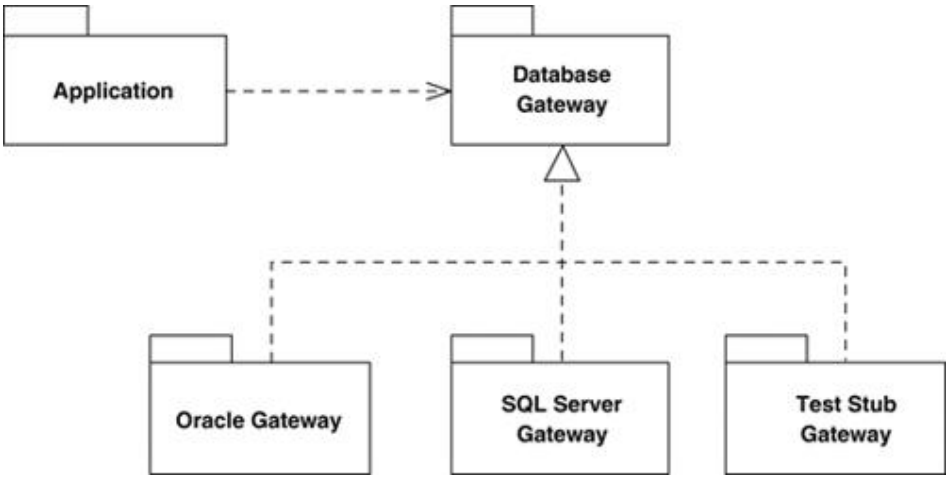
Figure 7.3. Separating Figure 7.2 into two aspects



Implementing Packages

Often, you'll see a case in which one package defines an interface that can be implemented by a number of other packages, such as that of Figure 7.4. In this case, the realization relationship indicates that the database gateway defines an interface and that the other gateway classes provide an implementation. In practice, this would mean that the database gateway package contains interfaces and abstract classes that are fully implemented by the other packages.

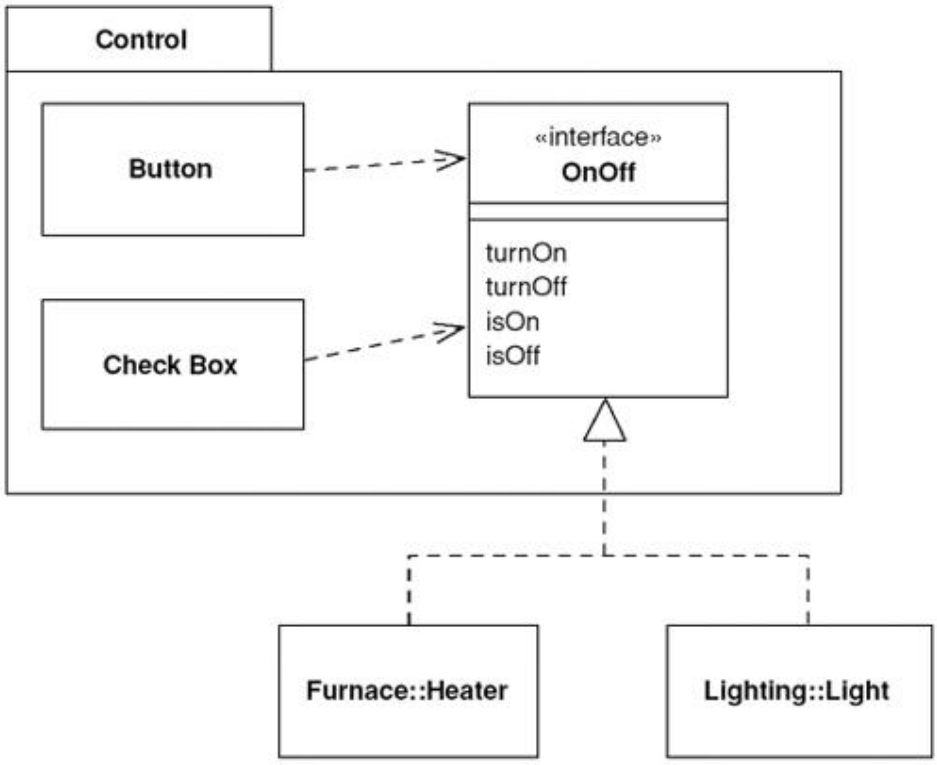
Figure 7.4. A package implemented by other packages



It's quite common for an interface and its implementation to be in separate packages. Indeed, a client package often contains an interface for another package to implement: the same notion of required interface that I discussed on page 70.

Imagine that we want to provide some user interface (UI) controls to turn things on and off. We want this to work with a lot of different things, such as heaters and lights. The UI controls need to invoke methods on the heater, but we don't want the controls to have a dependency to the heater. We can avoid this dependency by defining in the controls package an interface that is then implemented by any class that wants to work with these controls, as in Figure 7.5. This is an example of the pattern Separated Interface [Fowler, P of EAA].

Figure 7.5. Defining a required interface in a client package



When to Use Package Diagrams

I find package diagrams extremely useful on larger-scale systems to get a picture of the dependencies between major elements of a system. These diagrams correspond well to common programming structures. Plotting diagrams of packages and dependencies helps you keep an application's dependencies under control.

Package diagrams represent a compile-time grouping mechanism. For showing how objects are composed at runtime, use a composite structure diagram (page 135).

Where to Find Out More

The best discussion I know of packages and how to use them is [Martin]. Robert Martin has long had an almost pathological obsession with dependencies and writes well about how to pay attention to dependencies so that you can control and minimize them.

Chapter 8. Deployment Diagrams

Deployment diagrams show a system's physical layout, revealing which pieces of software run on what pieces of hardware. Deployment diagrams are really very simple; hence the short chapter.

Figure 8.1 is a simple example of a deployment diagram. The main items on the diagram are nodes connected by communication paths. A **node** is something that can host some software. Nodes come in two forms. A **device** is hardware, it may be a computer or a simpler piece of hardware connected to a system. An **execution environment** is software that itself hosts or contains other software, examples are an operating system or a container process.

Figure 8.1. Example deployment diagram