How Can the UML Model Applications?

Your business models have established why you are building this system (i.e., how it will support the business). Your use case models have depicted how the actors will use the system. And the architecture models set the initial organization of your system. Armed with this understanding, you are well prepared to model your application. These prior models limit the “shape” of the application. The business and use case models serve to limit the design space (which is a good thing because it takes the whole universe of possible solutions and constrains that solution space). The architecture models set the overall structure of your solution. In this way, developing these models helps to control the cost and schedule of the project.

Review of Class Diagrams Basics

After these boundaries are set, the most common diagram utilized to depict the static structure of your application is the class diagram. Class diagrams were introduced earlier in Chapter 4, “Architectural Modeling.” There, you saw class diagrams used for architectural modeling. In case you skipped past that section, let’s review some of the class diagram basics as they pertain to application modeling and then explore some additional modeling elements you might find in these diagrams.

Classes

The class diagram shows the important classes in the application and their relationships to the other classes. Classes primarily represent the “things” in the system. Through the development of the upstream models (business, requirements, architecture), you will have already discovered many important classes. Finding these classes is much harder to do well without these other models, but it can be done. If you don’t have the upstream models, just look at the problem statement for your application. Pick out the real “things” in the problem statement, such as account, aircraft, customer, product, transmitter, report, and so forth (see Figure 5-1). These will be the domain classes that should be in your class diagrams. As the models develop, non-real things will show up as classes, too, such as controller classes that coordinate processing, among others.
But what are classes with regard to implementation of your software? Classes describe a group of similar objects. Classes are templates used to create those objects in your application.

Watch Out—Breaking the Rules

The terms “class” and “object” are often used interchangeably when discussing UML models. Although this is technically incorrect, as long as you keep in mind that the class is the specification and the object(s) are the implementation, you should be fine. Don’t worry, we won’t tell the “Grand High Poobahs of Modeling” what you’ve done.

Just as a cookie cutter will cut out many identical cookies, if you have a class “Planet,” you can create multiple planet objects in your application by using the Planet class. Now, having a bunch of similar planets might not be very interesting. But just as you can take many of the same cookies and vary them by adding sprinkles or icing or food coloring, you can specialize your classes through polymorphism (more on this later in this chapter).

You will need to capture important information about your classes. You do this by using attributes (similar to the descriptions in Chapter 4 around architecture, but now specific for the application’s design itself). Attributes capture the essential characteristics of the class that are needed by the application—not all its characteristics, just those that are applicable to the problem at hand. For example, if your application simulates the position of the planets in the solar system, your Planet class might capture the planet’s distance from the sun and orbit (see Figure 5-2) but would not capture the core composition of the planet.
Watch Out—Dear Occupant

The attributes you capture in your classes determine the nature of the abstraction that the class is trying to capture. This can be critical to the success of your project. I was involved in a post-mortem on a project that had “serious difficulties.” One problem centered on the class “Address.” This team’s concept of “address” was that an address was the location of your home. Although this is not an incorrect abstraction of address, it was not the best for this application. The application was not managing the location of homes. This was a mailing application. In this context, the better abstraction for address was that it is where you send mail (which might be your home, a hotel if you are traveling, a hospital if you are ill, a vacation home, etc.).

Lessons Learned

1. Make sure your classes and their attributes are capturing the right abstraction for your application.

Another way of considering attributes is in the context of data. Object-oriented analysis and design encapsulates data and processing together. Your attributes are the data that is encapsulated inside the class. The data (attributes) inside the class can be available to other classes depending on the attribute’s visibility. Attributes can have public, protected, private, or package visibility (see Table
Table 5-1 Kinds of Visibility

<table>
<thead>
<tr>
<th>Visibility</th>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private</td>
<td>-</td>
<td>Only the class itself can access these attributes.</td>
</tr>
<tr>
<td>Protected</td>
<td>#</td>
<td>Any child class of this class can access these attributes.</td>
</tr>
<tr>
<td>Public</td>
<td>+</td>
<td>Any class can access these attributes.</td>
</tr>
<tr>
<td>Package</td>
<td>~</td>
<td>Any class in the same package can access these attributes.</td>
</tr>
</tbody>
</table>

In theory, a class’s attributes should all be private to ensure strong encapsulation (this protects the attributes from being accessed or altered by other objects, except as allowed by the owning object). However, you might find situations, say, during detailed design or implementation, where the level of encapsulation needs to be “softened.” In these cases, you can use the other levels of visibility to allow greater accessibility to a class’s attributes. However, attributes should remain private unless there is a strong need otherwise.

So what can classes do? That is established by the class’s operations.

Operations

Operations are the services that the class provides. Operations are shown in the third compartment of the class symbol. Typically, a class will contain the operations that provide access to its attributes along with the other processing functions of the class (see Figure 5-3). This is how external objects get access to the private attributes of the owning object. The class itself might not perform all of the operations within it, but the class is responsible for ensuring that something performs the operations. In this case, the class is acting as a controller class, controlling all or part of the application.
Depending on the maturity of the diagram you have, just the operation name might be shown. As an operation develops, its full signature will develop, showing its name, parameters, default values, return type, and so forth (see Figure 5-4). How much detail is actually specified will depend on how your organization uses modeling: that is, how much detail you require before handing the design to the programmers for implementation.
discussing UML models (discussed earlier in this chapter), you will hear people interchange the terms “operation” and “method.” This, too, is technically incorrect. Although an operation specifies a service provided by its class, a method is the implementation of that operation. Various possible methods (implementations) can satisfy an operation. For example, if you have a Sort operation, the method could implement it with a bubble sort, a hash sort, or some other sorting algorithm.

**Associations**

As discussed in Chapter 4, associations show the relationships between the objects in your system. They typically show the communication pathways between the objects, which can be bidirectional or unidirectional, as discussed earlier. Directionality (a.k.a. navigability) usually should be specified later in the design phase when you have more knowledge of how the processing will actually be traversing the associations. In this way, you will know how the implementation of the associations should be optimized.

**Other Association Adornments**

Some of the adornments that appear on association ends (e.g., multiplicity, diamonds) were also discussed in Chapter 4. You will often run into a few others.

Rolenames can appear on the end of an association. These rolenames describe the class that is attached to that same end of the association as the rolename. The function of the rolename is to indicate how the class will behave in that specific relationship with the class it is associated with. This is similar to the various roles we all play in life. When I am serving as an engineer, I perform certain behaviors. When I am acting as an investor, I perform different behaviors (although there can be some common behaviors). An engineer might read, analyze, design, build, etc. An investor might read, analyze, buy, sell, manage an account, etc. (see Figure 5-7). In this way, rolenames partition the behaviors of their classes.
Although rolenames partition the behavior(s) a class presents in an association with another class, qualifiers partition the set of objects that can participate in an association. Let's look at some examples using the classes Payroll and Person.

Figure 5-8 says that the Payroll department pays a Person. Figure 5-9 shows how a rolename can be added for clarity—Payroll is paying a Person who is acting as an employee (not a contractor or vendor). Yet, in both of these diagrams, there is little specificity.

When we add a qualifier (employee number), as in Figure 5-10, we now know that only that specific Person object participates in the association (i.e., only that Person with that specific employee number is being paid). If you augment this with multiplicity, you add more information to the model. In Figure 5-11, we see that because of the 0..1 multiplicity, Payroll pays either nobody (indicating an employee number that is not assigned to an employee) or one specific employee.
Figure 5-10 Association with rolename and qualifier.

Figure 5-11 Association with rolename, qualifier, and multiplicity.

Change the qualifier and multiplicity, and you get new semantics, as in Figure 5-12, where Payroll pays many full-time employees (the part-time employees are excluded by the qualifier “full-time”).

Figure 5-12 Association with rolename and qualifier selecting a set of objects.

More on Class Diagrams

Aggregation and Composition

In Chapter 4, two related association types were mentioned—aggregation and composition. (In UML 2.0, a composition association is also called a composite aggregation.) Figure 5-13 shows both associations.

Aggregation is shown with an empty (or hollow) diamond, whereas composition is noted with a filled-in diamond. In both cases, the diamond appears on the “Whole” end of the association. These associations are read: “The Part is part of the Whole” and “The Whole has a Part” (the number of specific parts is determined by the multiplicity on the part end of the association).
These associations carry the special meaning that one element of the association “is part of” the other. The difference between aggregation and composition is how tight the relationship is between the participating elements. Aggregation is a looser form of association, whereas composition indicates a much tighter form of containment. In a composition, the Part(s) can only be part of one Whole. Also, with composition, when the Whole is destroyed, so are all its parts. (In fact, it is the responsibility of the Whole to ensure that its parts are destroyed.) In Figure 5-14, you can see such relationships as they pertain to a flashlight.

![Figure 5-13 Aggregation and composition of a flashlight](image)

A flashlight has a switch. This is a composition because the switch is part of that one flashlight—when you throw away (destroy) the flashlight, the switch goes with it. On the other hand, you could remove the battery from the flashlight and use it in another one, which is why this is shown as an aggregation. One caveat—these “part of” relationships aren’t just for physical elements. For example, a Person can have a Belief, or a BrandName can have Market-Value.