1.1 Modeling with Objects

Fifteen years ago I went to the local concert hall and asked the ticket agent for two tickets for the March 21 concert. "Where do you want to sit?" he asked.

"That depends on what's available," I answered.

The agent grabbed a printed map of the concert hall. It was clearly dated "March 21," noted the name of the performer, and showed a map of the auditorium's seats. Seats that had already been sold were marked with a red X. The seats were also color-coded: the most expensive seats were green, the moderately priced seats were black, and the least expensive seats were blue.

Fifteen years ago, the ticket agent showed me the map and stabbed his finger on a pair of seats. "These are the best seats left, but the choice is yours."

I quickly scanned the map and noticed that a pair of less expensive seats with almost the same sightlines was not far away. I chose the cheaper seats, and the agent promptly marked them with a red X.

Fast-forward fifteen years. Today I order tickets from the comfort of my home over the Web. I visit the concert hall's Web site and find the performance I want. I click the "purchase tickets online" button and am shown a color-coded map of the theatre. I click on the seats I want, enter my credit card information, and am assured that the tickets will be mailed to me promptly.

1.1.1 Using Models

A **model** is a simplified description of something. It helps us, for example, make decisions, predict future events, maintain up-to-date information, simulate a process, and so forth. Originally, the local concert hall modeled ticket sales with a simple paper map of the auditorium. Later, a Web-based computerized model performed the same functions—and probably many more.

KEY IDEA

A model is a simplified description of something.

To be useful, a model must be able to answer one or more questions. The paper-based model of ticket sales could be used to answer questions such as:

- What is the date of the concert?
- Who is playing?
- How many tickets have been sold to date?
- How many tickets are still unsold?
- Is the ticket for seat 22H still available?
- What is the price of the ticket for seat 22H?

- ➤ Which row has unsold tickets for 10 consecutive seats and is closest to the stage?
- ➤ What is the total value of all the tickets sold to date?

Models often change over time. For instance, the ticket sales model was updated with two new red X's when I bought my tickets. Without being updated, the model quickly diverges from the thing it represents and loses its value because the answers it provides are wrong.

We often speak of models or elements of a model as if they were real. When the ticket agent pointed to the map and said, "These are the best seats left," we both knew that what he was pointing at were not seats, but only images that *represented* actual seats. The model provided a correspondence. Anyone could use that model to find those two seats in the concert hall.

We often build models without even being aware of it. For example, you might make a mental list of the errands you want to run before having supper ready for your roommate at 6 o'clock, as shown in Figure 1-1: stopping at the library to pick up a book (10 minutes), checking e-mail on a public terminal at the library (5 minutes), and buying a few groceries (10 minutes). Checking your watch (it's 4:15) and factoring in 45 minutes for the bike ride home and 30 minutes to prepare supper, you estimate that you can do it all, with a little time to spare. It takes longer than expected, however, to find the book, and there's a line at the library checkout counter. The library errand took 20 minutes instead of 10. Now it's 4:35, and you must make some choices based on your updated model: have supper a little late, skip the e-mail, hope that you can cook supper in 25 minutes instead of 30, and so forth. You have been modeling your time usage for the next two hours.

(figure 1-1)

Sample schedule

4:15	Pick up library book.
4:25	Check e-mail.
4:30	Buy groceries.
4:40	Bike home.
5:25	Cook supper.
6:00	Supper.

KEY IDEA

Models focus on relevant features.

Models form an **abstraction**. Abstractions focus only on the relevant information and organize the remaining details into useful higher-level "chunks" of information. People can only manage about seven pieces of information at a time, so we must carefully choose the information we manage. By using abstraction to eliminate or hide some details and group similar details together into a chunk, we can manage more complex ideas. Abstraction is the key to dealing with complexity.

For example, the ticket sales model gives ticket buyers and agents information about which tickets are available, where the corresponding seats are located, and their price. These were all relevant to my decision of which tickets to purchase. The map did not provide information about the seat's fabric color, and I really didn't care, because that was irrelevant to my decision. Furthermore, the color-coding of the concert hall map conveniently chunked information, which helped me make a decision quickly. It was easier to see all the least expensive seats in blue rather than consulting a long list of seat numbers.

Beyond information, models also provide operations that can be performed on them. In the concert hall model operations include "sell a ticket" and "add a new concert." In the informal time management model for errands, operations include "insert a new errand," "drop an errand from the list," and "recompute the estimated start time for each errand."

1.1.2 Using Software Objects to Create Models

KEY IDEA

Object-oriented programs use software objects to model the problem at hand.

The concert hall's computer program and the paper map it replaced model the ticket-selling problem using different technologies. One uses pre-printed sheets of paper marked with a simple X. The other involves a computer with a detailed set of instructions, called a **program**. If the program is written in an **object-oriented programming language**, such as Java, the computer program uses cooperating **software objects**. A software object usually corresponds to an identifiable entity in the problem. The concert hall program probably has an object modeling the concert hall's physical layout, a collection of objects that each models a seat in the concert hall, and another collection of objects that each models an upcoming concert. A program maintaining student enrollments in courses would likely model each student with an object, and use other objects to model each course.

KEY IDEA

Computer science has a specialized vocabulary to allow precise communication. You must learn this vocabulary.

Each of the software objects can perform tasks such as:

- Maintain information about part of the problem the program models.
- Answer questions about that part of the problem based on the information it maintains.
- Change its information to reflect changes in the real-world entity it models.

KEY IDEA

Server objects provide services—queries and commands—to client objects.

The information kept by the object is called its **attributes**. Objects respond to **queries** for information and to **commands** to change their attributes. Queries and commands are collectively referred to as **services**. An object provides these services to other objects, called **clients**. The object providing the service is called, appropriately, the **server**. We will explore these concepts in the coming pages.

Queries and Attributes

Queries are the questions to which an object can respond. A query is always answered by the object to which it is directed. It might be true or false ("Is the ticket for seat 22H still for sale?"), a number ("How many tickets have been sold?"), a string of characters

("What is the name of the band that is playing?"), or even another object such as a date object ("What is the date of the concert?"). Queries are said to `return` answers to their clients. An object can't respond to just any query, only to those it was designed and programmed to support.

KEY IDEA

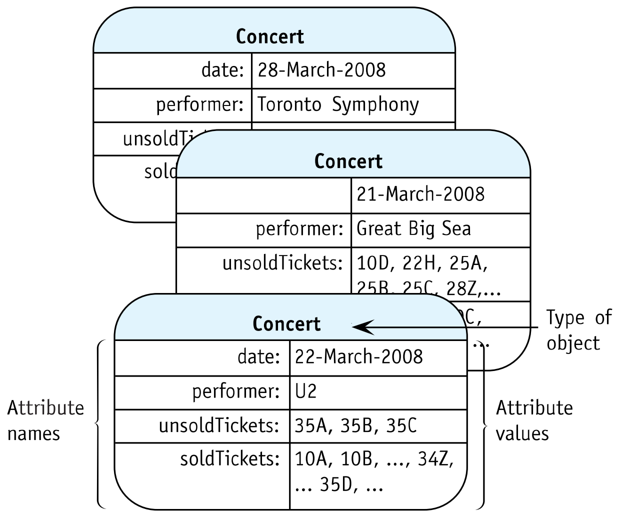
Objects have attributes. Answers to queries are based on the values of the attributes.

The answers provided by queries are always based on the object's attributes. If an object must answer the query, "What is the date of the concert?" then it must have an attribute with information about the date. Similarly, if it must answer the question, "How many tickets have been sold to date?" it must have an attribute that has that information directly, or it must have a way to calculate that information, perhaps by counting the number of tickets that have been sold. Information about which tickets have been sold would be kept in an attribute.

The concert hall's program must model ticket sales for many concerts, each represented by its own concert object. If we look at several concert objects, we'll notice they all have the same set of attributes, although the values of those attributes may be different. One way to show the differing attribute values is with an **object diagram**, as shown in Figure 1-2. Each rounded rectangle represents a different concert object. The type of object is shown at the top. Below that is a table with attribute names on the left and attribute values on the right. For example, the attribute "date" has a value of "21-March-2008" for one concert. That same concert has the value "Great Big Sea" for the "performer" attribute.

(figure 1-2)

Object diagram showing three concert objects with their attributes



One analogy for objects is that an object is like a form, such as an income tax form. The government prints millions of copies of the form asking for a person's name, address, taxpayer identification number, earned income, and so forth. Each piece of information is provided in a little box, appropriately labeled on the form. Each copy of the form starts like all the others. When filled out, however, each form has unique values in those boxes. It could be that two people have exactly the same income and birthday, with the result that some forms have the same values in the same boxes—but that's only a coincidence.

Just as each copy of that tax form asks for the same information, every concert object has the same set of attributes. Each copy of the form is filled out with information for a specific taxpayer; likewise, each concert object's attributes have values for a specific concert. In general, there may be many objects of a given type. All have the same set of attributes, but probably have different values for those attributes.

KEY IDEA

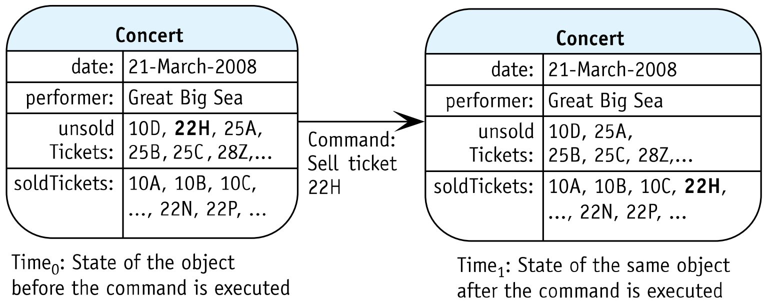
Every object of a given type has the same set of attributes, but usually has different values for the attributes.

Commands

When a ticket is sold for seat 22H for the March 21 concert, the appropriate concert object must record that fact. This record keeping is done with a command. The object is "commanded" to change its attributes to reflect the new reality. This change can be visualized with a **state change diagram**, as shown in Figure 1-3. A state change diagram shows the state of the object before the command and the state of the object after the command. The **state** is the set of attributes and their values at a given point in time. As time passes, it is normal for the state of an object to change.

KEY IDEA

Commands change the state of the object.



Concert
date: 21-March-2008
performer: Great Big Sea
unsold Tickets: 10D, 22H, 25A, 25B, 25C, 28Z,...
soldTickets: 10A, 10B, 10C, ..., 22N, 22P, ...
Time$_0$: State of the object before the command is executed
Command: Sell ticket 22H
Concert
date: 21-March-2008
performer: Great Big Sea
unsold Tickets: 10D, 25A, 25B, 25C, 28Z,...
soldTickets: 10A, 10B, 10C, 22H, ..., 22N, 22P, ...
Time$_1$: State of the same object after the command is executed

(figure 1-3)

State change diagram showing the change in state after a command to sell seat 22H is given to a concert object

Classes

When we write a Java program, we don't write objects, we write classes. A **class** is a definition for a group of objects that have the same attributes and services. A programmer writing the concert hall program would write a concert class to specify that all concert objects have attributes storing the concert's date, performers, and so on. The class also specifies services that all concert objects have, such as "sell a ticket," and "how many tickets have been sold?"

KEY IDEA

A Java programmer writes a class by specifying the attributes and services the classes' objects will possess.

Once a concert class is defined, the programmer can use it to create as many concert objects as she needs. Each object is an **instance**, or one particular example, of a class. When an object is first brought into existence, we sometimes say it has been **instantiated**.

The distinction between class and object is important. It's the same as the distinction between a factory and the cars made in the factory, or the distinction between a cookie cutter and the cookies it shapes. The pattern used to sew a dress is different from the dress produced from it, just as a blueprint is different from the house it specifies. In each case, one thing (the class, factory, or cookie cutter) specifies what something else (objects, cars, or cookies) will be like. Furthermore, classes, factories, and cookie cutters can all be used to make many instances of the things they specify. One factory makes many cars; one class can make many objects. Finally, just as most of us are not interested in cookie cutters for their own sakes, but in the cookies made from them, our primary interest in classes is to get what we really want: software objects that help model some problem for us.

Class Diagrams

KEY IDEA

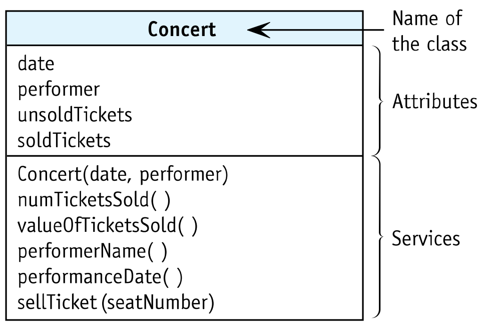
A class diagram summarizes all of the objects belonging to that class.

Just as architects and dress designers communicate parts of their designs visually through blueprints and patterns, software professionals use diagrams to design, document, and communicate their programs. We've already seen an object diagram in Figure 1-2 and a state change diagram (consisting of two object diagrams) in Figure 1-3. Another kind of diagram is the **class diagram**. Class diagrams show the attributes and services common to all objects belonging to the class. The class diagram for the concert class summarizes all the possible concert objects by showing the attributes and services each object has in common with all other concert objects.

A class diagram is a rectangle divided into three areas (see Figure 1-4). The top area contains the name of the class. Attributes are named in the middle area, and services are in the bottom area.

(figure 1-4)

Class diagram for the Concert class showing four attributes and six services



1.1.3 Modeling Robots

Every computer program has a model of a problem. Sometimes the problem is tangible, such as tracking concert ticket sales or the time required to run errands before supper. At other times, the problem may be more abstract: the future earnings of a