CHAPTER 11

Class Diagrams

MAJOR TOPICS

OBJECTIVES

At the completion of this chapter, you will be able to:

- Use class diagrams to illustrate system design.
- Describe the difference between reference objects and value objects.
- Identify when it is appropriate to use associations, aggregation, and composition.
- Use generalizations to illustrate inheritance.

PRE-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What do class diagrams illustrate?

   ....

   ....

2. What three things does a UML class define?

   ....

   ....

INTRODUCTION

In the previous chapter, you were introduced to the UML class notation. This class notation is used to represent a class name, variables, and the methods that comprise a class's interface. Figure 11-1 illustrates the UML class notation for the Asset class.

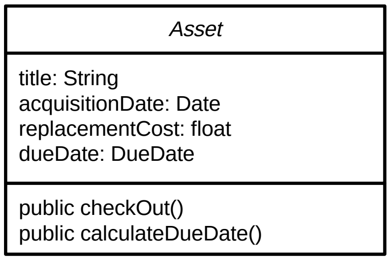

Figure 11-1: UML class notation

In an earlier chapter, you were introduced to collaboration diagrams. Collaboration diagrams illustrate the interaction between classes. Class diagrams are used to illustrate the static relationships between the classes. Ultimately, the program code you will write is a static set of relationships between the classes you define. The interactions between them happen only at runtime. In this way, class diagrams are the foundation of the design process. They form a snapshot of the system that will be constructed.

The UML classes, as illustrated in the preceding figure, are the building blocks of UML class diagrams. They define a class' name, its attributes, and its public interface. These elements are translated directly into program code. The static relationships also translate into program code. Classes participate in four principal relationships:

- Association
- Aggregation
- Composition
- Generalization

In this chapter, you will learn to define these static relationships between classes and illustrate your design using class diagrams.

ASSOCIATION

The simplest relationship is an association. An association illustrates a "uses a" relationship between instances of a class. An association provides a path for communication between objects. In its most basic form, an association is represented by a line between two classes. The examples in this chapter build upon the collaboration diagram developed in Chapter 13. Figure 11-2 illustrates an association between the CheckOutController class and the Patron class.

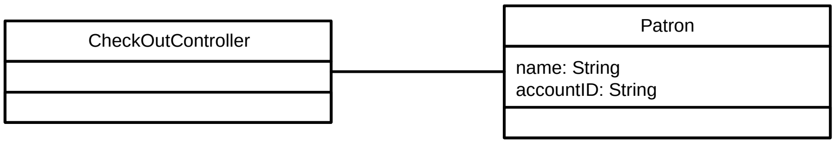

Figure 11-2: Association example

The collaboration diagrams you developed earlier illustrated a relationship between the CheckOutController and a Patron. This relationship is realized using a class diagram. The association between them is a path for communication. The CheckOutController class defines a pointer to a Patron object. The pointer will be used to communicate with the patron.

Navigability

Note from the earlier analysis that a Patron does not need to communicate back to the CheckOutController; the association between these classes is unidirectional. This relationship is known as navigability and is represented using an arrowhead. Figure 11-3 illustrates the one-way association relationship between the CheckOutController class and the Patron class.

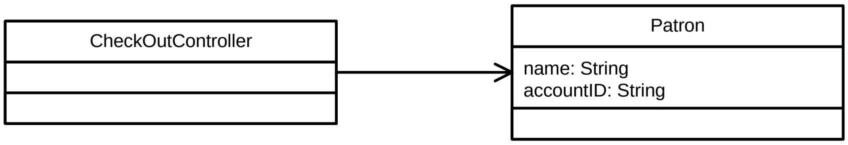

Figure 11-3: Navigability example

This figure clearly shows that the CheckOutController class must communicate with a Patron but a Patron need not communicate with a CheckOutController. In the program code, the CheckOutController will have a pointer to a Patron, but a Patron will not have a pointer to a CheckOutController.

Multiplicity

Multiplicity is used to denote the number of instances of a class involved in a relationship. During the check-out process, the CheckOutController handles only one patron at a time. Therefore the CheckOutController maintains an association with a single Patron object. The multiplicity of a relationship is denoted by labeling its ends. Figure 11-4 illustrates the multiplicity of the association relationship between the CheckOutController class and the Patron class.

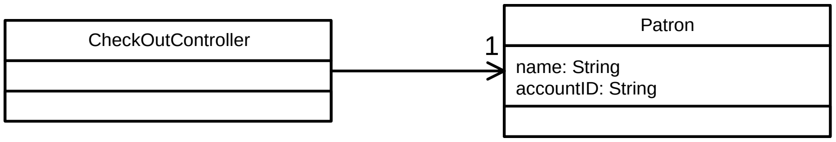

Figure 11-4: Multiplicity example

Multiplicity can be represented by a single cardinal number, such as the example in the preceding figure. It can also be represented using a range of numbers or an asterisk.

AGGREGATION

Aggregation is used to show that one class is part of another class. Figure 11-5 illustrates the aggregation relationship between the AssetList class and the Asset class. An asset list contains assets; this is an aggregation relationship. This relationship is denoted using an open diamond and a line between the classes. The open diamond on the AssetList side of the relationship tells you that the AssetList "has an" Asset. The multiplicity indicator on the Asset side of the relationship tells you that an asset list can have any number of assets, from zero to infinity.

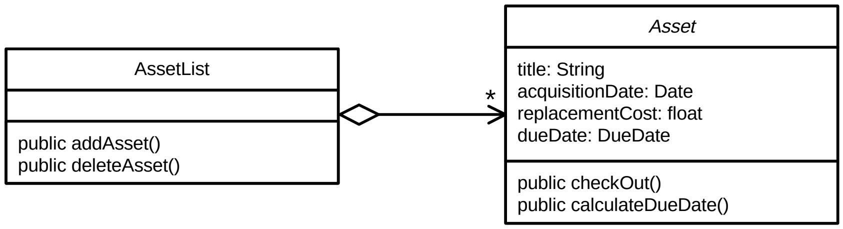

Figure 11-5: Aggregation example

It can be difficult to determine when aggregation is appropriate. Examine the Asset class in the preceding figure. The Asset class has four attributes: title, acquisitionDate, replacementCost, and dueDate. When this class diagram is translated into program code, each of these attributes will translate into a class member variable. Each of these attributes participates in a "has a" relationship. Just as an AssetList "has an" Asset, so too an Asset "has a" String, a Date, and so on.

The difference between the two relationships is subtle. Instances of Asset are treated as reference objects by the AssetList class. The attributes of the Asset class are treated as value objects.

Reference objects

An instance of Patron is an example of a reference object. Reference objects are said to have identity. A patron named James Smith will be represented within the system by a Patron object. Only one James Smith object will exist, and the classes that interact with the James Smith object will pass object references to each other. In this way, any class with a reference to the James Smith object can make changes to the object and the changes will be reflected in all classes.

Value objects

Attributes are value objects. Value objects are a part of one, and only one, class. The Asset class has an attribute dueDate. Many of the assets checked out on the same day will have the same due date, but these objects do not have identity. If one asset is checked in the following day and checked out by another patron, its dueDate will be changed to a new dueDate. Even though multiple assets have the same original due date, only the dueDate for the Asset being checked out will change because each maintains its own dueDate value object. The dueDate object will exist only as long as the Asset object that owns it exists. If the Asset object is destroyed, the dueDate will be destroyed along with it. Also, queries about the due date of an Asset object will return a copy of the dueDate object, not a reference.

In some circumstances, value objects must participate in complicated relationships with other classes. It may be beneficial to the design to illustrate these relationships within a class diagram. In these cases, a second type of "has a" relationship, called composition, is used.

COMPOSITION

Composition is a second type of "has a" relationship between a class and a value object. It is represented by a closed diamond and a line between classes. Figure 11-6 illustrates the composition relationships between the Patron class and the AssetList class, and between the CheckOutController class and the AssetList class. The CheckOutController maintains a list of assets being checked out. Therefore, it must contain an AssetList. A Patron also contains an AssetList. The Patron always maintains a list of the assets he has checked out. These lists are not the same. The CheckOutController's list contains only those assets currently being checked out. The Patron may already have assets checked out and so his list is maintained separately. The AssetList class is included in the class diagram because of the relationship it maintains with the Asset class.

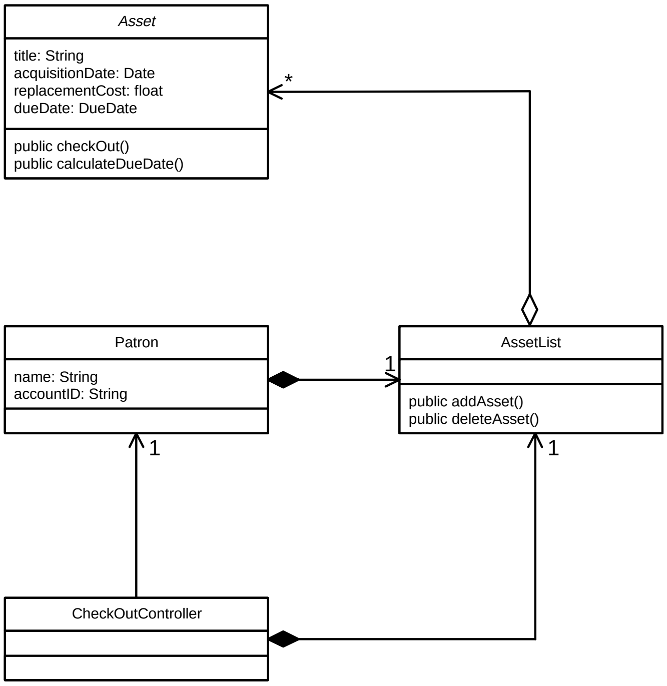

Figure 11-6: Composition example

As noted previously, classes involved in a composition relationship are similar to class attributes. Indeed, an alternative notation for the composition relationship is to include the component within the attributes section of its owner. Figure 11-7 illustrates the alternative notation for the composition relationship.

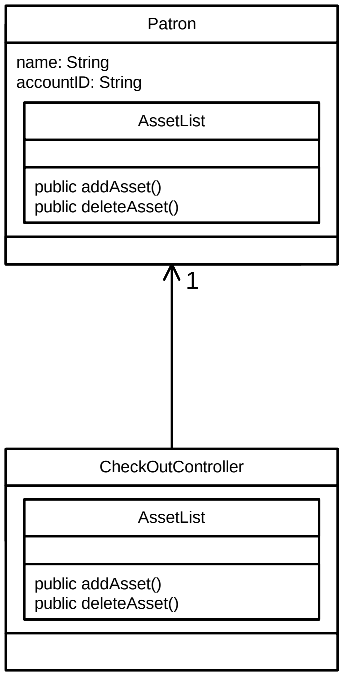

Figure 11-7: Alternative notation for composition

GENERALIZATION

Generalization is the "is a" relationship. It is used to denote inheritance within a class diagram. A generalization is represented by a line and an open arrowhead. Figure 11-8 illustrates the generalization relationship between the Book and AudioCassette classes and the Asset class. Notice that the name of the Asset class is in italic print. This indicates that the Asset class is an abstract class. The AssetList class maintains an aggregation relationship with the Asset class. Because the Book and AudioCassette classes inherit from the Asset class, either class can be substituted in this relationship.

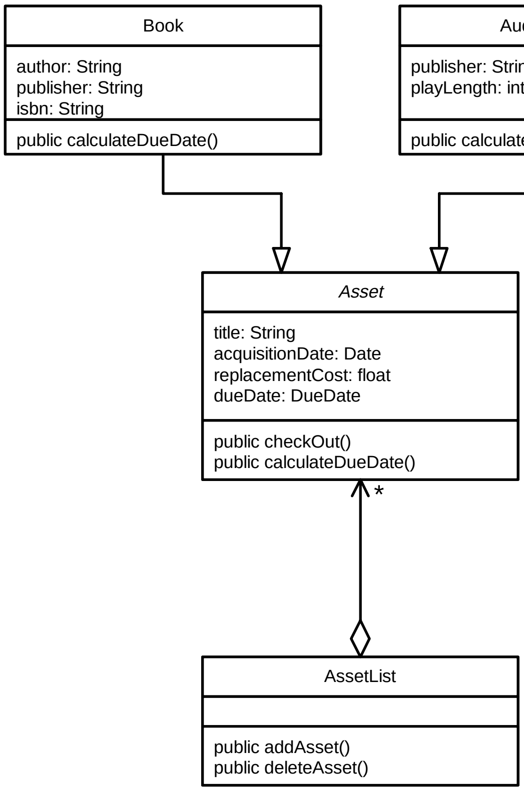

Figure 11-8: Generalization example

CHECK OUT ASSET CLASS DIAGRAM

Figure 11-9 is the complete class diagram for the Check Out Asset use case. Compare this diagram to Figure 10-6, the Check Out Asset collaboration diagram. Consider how the classes might communicate through this static structure to realize the functionality of the Check Out Asset use case.

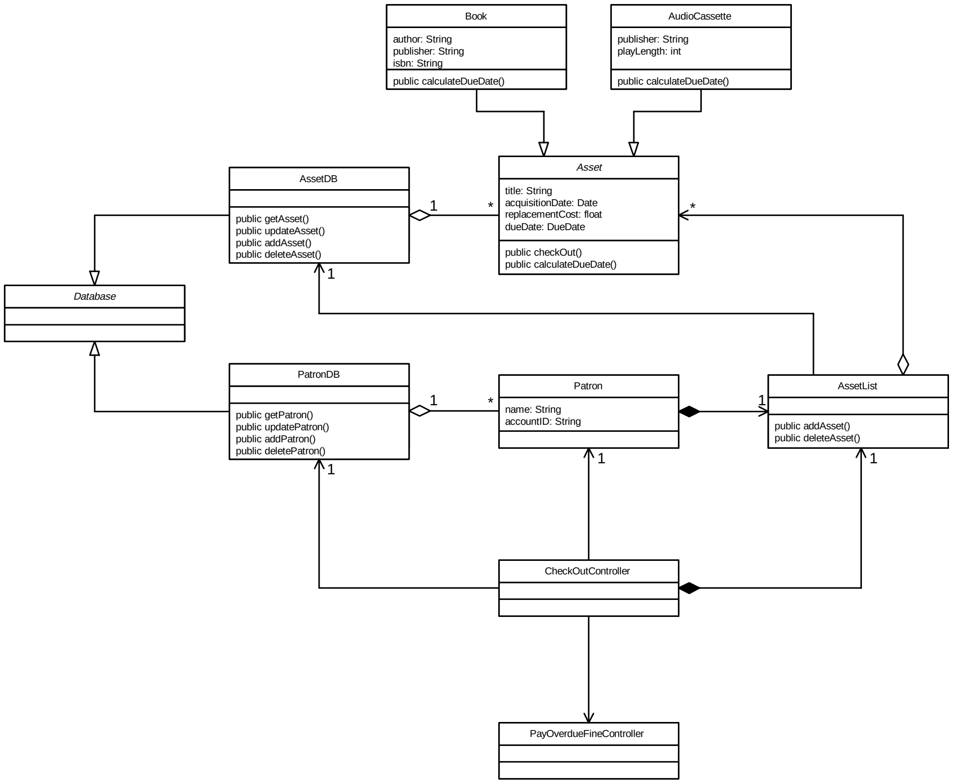

Figure 11-9: Check Out Asset class diagram

In the next few chapters, you will learn how to create two new types of diagrams: sequence diagrams and statechart diagrams. You will use sequence diagrams to illustrate the flow of messages between design classes, and you will use statechart diagrams to illustrate the design of individual classes.

Exercise 11-1: Creating class diagrams

1. Examine the collaboration diagrams you developed in Exercise 10-1. Consider how each might be realized as a static class diagram.
2. Develop a class diagram for each collaboration diagram.

SUMMARY

Class diagrams illustrate the static relationships between classes in a system. Classes participate in four principal relationships: association, aggregation, composition, and generalization. Association is the "uses a" relationship. Aggregation and composition are "has a" relationships. Aggregations are relationships between a class and a reference object. Compositions are relationships between a class and a value object. Generalization is the "is a" relationship, which is used to denote inheritance.

POST-TEST QUESTIONS

The answers to these questions are in Appendix A at the end of this manual.

1. What are the four principal relationships classes participate in?

   ....

   ....

2. Which of the four principal relationships illustrates a “uses a” relationship, which illustrates a “has a” relationship, and which illustrates an “is a” relationship between instances of a class.

   ....

   ....