9.23 (Optional Case Study) Thinking About Objects: Incorporating Inheritance into the Elevator Simulation

We now examine our elevator simulator design to see how it might benefit from inheritance. To apply inheritance, we first look for commonality among classes in the simulation. We begin by examining the similarities between classes **FloorButton** and **Elevator-Button**. Figure 9.35 shows the attributes and operations of each class. Both classes have their attribute (**pressed**) and operations (**pressButton** and **resetButton**) in common.

FloorButton
- pressed : Boolean = false
+ resetButton( ) : void + pressButton( ) : void

ElevatorButton
- pressed : Boolean = false
+ resetButton( ) : void + pressButton( ) : void

Fig. 9.35   Attributes and operations of classes **FloorButton** and **ElevatorButton**.

We might be tempted to use inheritance in this situation—i.e., to extract the commonality between classes **FloorButton** and **ElevatorButton**, place this commonality into a common superclass **Button**, then derive subclasses **FloorButton** and **ElevatorButton** from class **Button**. However, if the **FloorButton** and **ElevatorButton** objects have the exact same behavior, then we cannot justify using inheritance. In fact, we cannot event justify using two separate classes for these objects! The **FloorButton** signals the **Elevator** to move to the **Floor** of the request. The **ElevatorButton** signals the **Elevator** to move to the opposite **Floor**. As shown in the activity diagram of Fig. 5.30, the **FloorButton** and the **ElevatorButton** signal the **Elevator** to move to a **Floor**. The **Elevator** moves in response to a **FloorButton**'s signal only if the **Elevator** is on the opposite **Floor** of the request and the **Elevator** is idle. The **Elevator** moves in response to a signal from the **ElevatorButton** only if the **Elevator** is idle. However, neither the **FloorButton** nor the **Elevator-Button** orders the **Elevator** to move to the other **Floor**. Rather, the **Elevator** responds to a button signal depending on the **Elevator**'s current state. Each button has only one behavior—to signal the **Elevator** to move. The **FloorButton** and **ElevatorButton** objects are really just different objects of the same class, so we combine (rather than inherit) classes **FloorButton** and **ElevatorButton** into class **Button** and discard class **FloorButton** and **ElevatorButton** from our case study.

Software Engineering Observation 9.1

If several objects have the same attributes and exhibit identical behaviors, they are probably objects of the same class. Before using inheritance in your programs, make sure that each inherited class exhibits has distinct attributes and/or distinct behaviors but still "is a" type of its superclass.

We now turn our attention to classes **ElevatorDoor** and **FloorDoor**. Once again, we may be tempted to use inheritance with these classes, but Fig. 9.36 shows that the attributes and operations of each class are identical to each other—both classes possess attribute **open** and operations **openDoor** and **closeDoor**. Both the **ElevatorDoor** and **FloorDoor** have the same behavior—they simply open and close. A **Person** will react differently depending on which door has opened (i.e., a **Person** enters the **Elevator** when the **FloorDoor** opens and exits the **Elevator** when the **Elevator-Door** opens), but the **Person**'s behavior does not reflect a difference in behavior for the doors. Classes **FloorDoor** and **ElevatorDoor** clearly share the same attributes and behaviors, so we combine **ElevatorDoor** and **FloorDoor** into class **Door** and eliminate classes **FloorDoor** and **ElevatorDoor** from our case study.
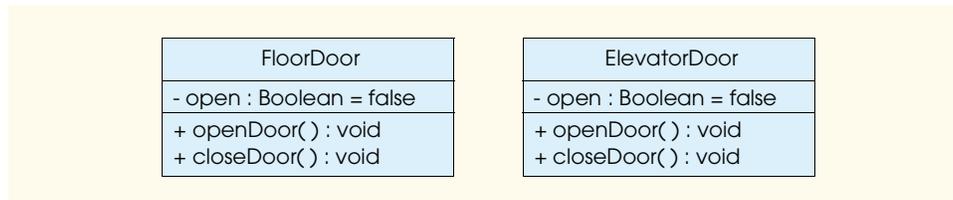
FloorDoor
- open : Boolean = false
+ openDoor( ) : void + closeDoor( ) : void

ElevatorDoor
- open : Boolean = false
+ openDoor( ) : void + closeDoor( ) : void

Fig. 9.36   Attributes and operations of classes **FloorDoor** and **ElevatorDoor**.

Now we begin looking for classes that exhibit similar (but not identical) behaviors. In Section 4.14, we encountered the problem of representing the location of the **Person**—on what **Floor** is the **Person** located when riding in the **Elevator**? Using inheritance, we may now model a solution. Both the **Elevator** and the two **Floor**s are locations at which the **Person** exists in the simulator. In other words, the **Elevator** and the **Floor**s are *types of* locations, but an **Elevator** is certainly not a **Floor**.

We modify classes **Elevator** and **Floor** to inherit from a superclass called **Location**. The UML specifies a relationship called a *generalization* to model inheritance. Figure 9.37 is the class diagram modeling the generalization of superclass **Location** and subclasses **Elevator** and **Floor**. The arrows with empty arrowheads indicate that classes **Elevator** and **Floor** inherit from class **Location**. Class **Location** contains the **private** attribute **locationName**, which contains a **String** value of **"first-Floor"**, **"secondFloor"** or **"elevator"**. We include method **setLocation-Name** so each subclass can set the appropriate **String** value. Note that method **setLocationName** has an access modifier that we have not yet seen—the pound sign (**#**), indicating that method **setLocationName** is **protected**, so only subclasses **Elevator** and **Floor** can use this method to set their **locationName**s. In addition, we include the **public** method **getLocationName**, so any object can return the name of the **Location**.
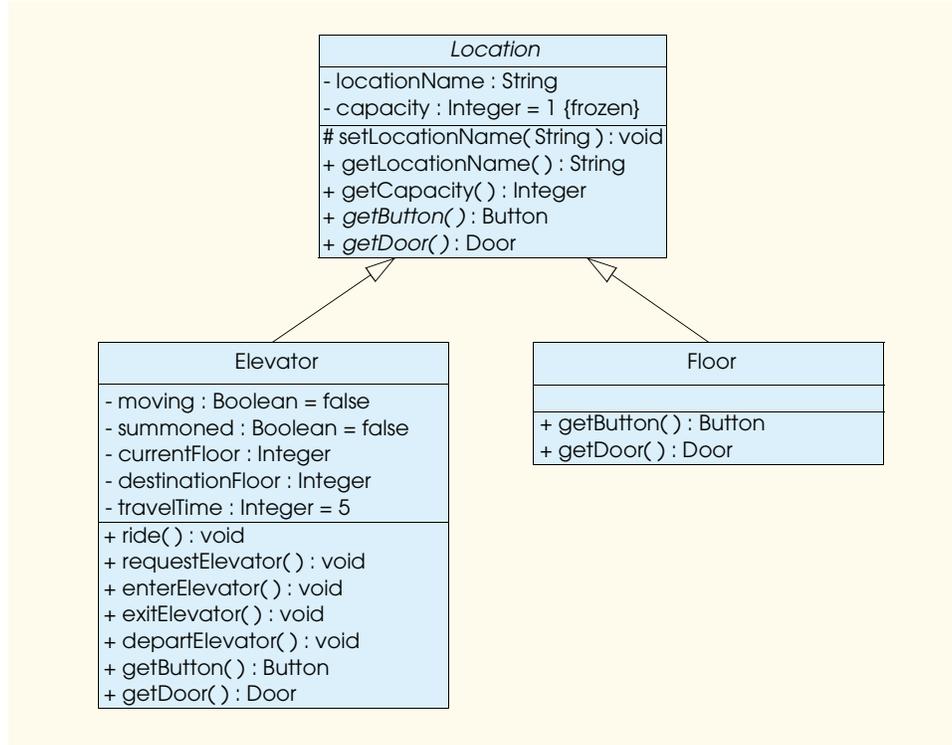
Fig. 9.37   Class diagram modeling generalization of superclass **Location** and subclasses **Elevator** and **Floor**.

We search for more similarities between classes **Floor** and **Elevator**. First of all, according to the class diagram of Fig. 6.21, both classes share integer attribute **capacity** (which equals **1**). Now, class **Location** contains **private** attribute **capacity**, which represents the maximum number of **Person**s that can occupy that location. In our simulation, this attribute will not change in execution, so class **Location** declares attribute **capacity** as a constant by using the term **{frozen}** next to attribute **capacity**. Class **Location** also contains **public** method **getCapacity**, which returns the **capacity** value. Class **Location** does not require method **setCapacity**, because attribute **capacity** cannot be changed. Subclasses **Floor** and **Elevator** inherit attribute **capacity** and method **getCapacity**.

We continue searching for similarities. According to Fig. 3.23, class **Elevator** contains references to its **Button** and its **Door**. Class **Floor** contains references to its **Button** and its **Door** through the **Floor**'s association with class **ElevatorShaft**—class **ElevatorModel** aggregates classes **Floor** and **ElevatorShaft** and can pass an **ElevatorShaft** reference to class **Floor**'s constructor. Using this association, class **Floor** can reference the **ElevatorShaft**'s **Button** and **Door**. Therefore, in our simulation, the **Location** class will contain **public** methods **getButton** and **getDoor** that return **Button** and **Door** references, respectively. Class **Floor** overrides these

methods to return the **Button** and **Door** references of that **Floor**, and class **Elevator** overrides these methods to return the **Button** and **Door** references of the **Elevator**.[1] We declare class **Location** as **abstract** to require subclasses **Floor** and **Elevator** to override methods **getButton** and **getDoor**. The UML requires that we place abstract class names (and abstract methods) in italics, so we place class **Location** and its methods **getButton** and **getDoor** in italics in Fig. 9.37. However, methods **getButton** and **getDoor** are not italicized in subclasses **Floor** and **Elevator**—these methods are concrete because they override the abstract methods and provide implementation. Each concrete methods has a distinct implementation here—i.e., class **Elevator** implements methods **getButton** and **getDoor** differently than does class **Floor**. Note that classes **Elevator** and **Floor** provide methods **getButton** and **getDoor** in their last compartment, because each class has different implementations in the overridden methods.

In this simulation, using inheritance seems appropriate for designing the **Elevator** and **Floor**. Each class represents a **Location** that the **Person** can occupy. However, class **Elevator** contains additional attributes and methods that distinguish it from class **Floor**.

We will now have class **Person** contain a **Location** object reference that represents whether the **Person** is on the first or second **Floor**, or inside the **Elevator**. Figure 9.38 is an updated class diagram of our model that reflects this change, incorporates inheritance and eliminates classes **FloorButton**, **ElevatorButton**, **FloorDoor** and **ElevatorDoor**. We remove the association between **Person** and **Elevator** and the association between **Person** and **Floor** from the class diagram, because the **Person**'s **Location** reference can act as either an **Elevator** or a **Floor** reference. A **Person** sets its **Location** reference to the **Elevator** when that **Person** enters the **Elevator**. A **Person** sets its **Location** reference to a **Floor** when the **Person** walks onto that particular **Floor**. Lastly, we assign class **Elevator** two **Location** references, representing the **Elevator**'s current **Floor** and the destination **Floor**. (We originally used integers to describe these references in Fig. 4.18.)

---

1. Most methods introduced in this chapter—such as method **getRadius** (Fig. 9.29) or **getHeight** (Fig. 9.30)—return primitive-data types, whereas methods **getButton** and **getDoor** of classes **Elevator** and **Floor** return references to user-defined types (a **Button** and **Door** object, respectively).
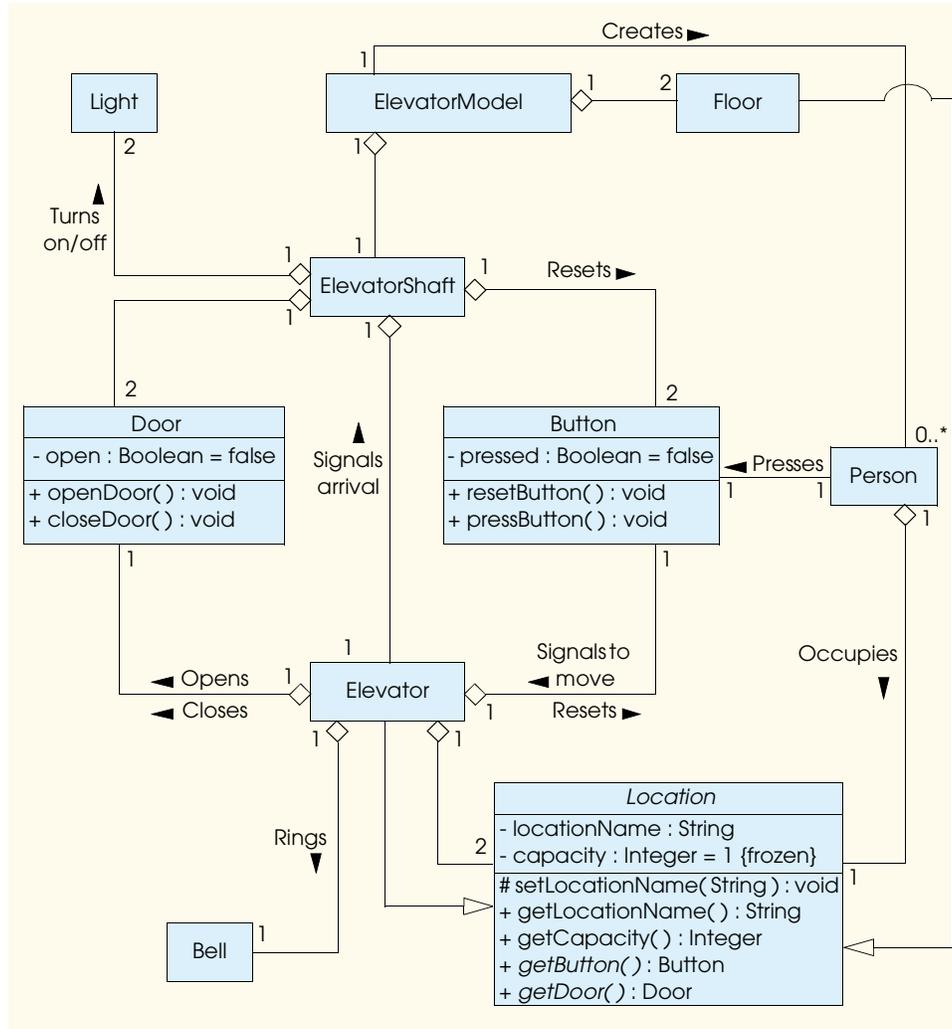
Fig. 9.38   Class diagram of our simulator (incorporating inheritance).

We allow a **Person** occupying a **Location** to interact with several of the objects (accessible from that **Location**) in the simulator. For example, a **Person** can press a **Button** from that **Person**'s specific **Location**. A **Person** can occupy only one **Location** at a time, so that **Person** should be restricted to interacting with only the objects known to that **Location**. Using its **Location** reference, a **Person** cannot perform an illegal action, such as pressing the first **Floor**'s **Button** while riding the **Elevator**. This mimics a real-world situation in that a person cannot press a button on a floor when riding an elevator, and cannot press a button inside the elevator when walking on a floor.

We presented the class attributes and operations with access modifiers in the class diagram of Fig. 8.13. Now, we present a modified class diagram incorporating inheritance in Fig. 9.39. Note that this abbreviated diagram does not show inheritance relationships, but instead shows the attributes and methods after we employ inheritance in our system. This diagram does not include those attributes shown by aggregations in Fig. 9.38 for classes **ElevatorShaft**, **ElevatorModel** and **Elevator**. As we first mentioned in Section 4.14, we sometimes choose to save space in Fig. 9.39 by not showing these additional attributes—we will, however, include them in the Java implementation in the appendices.
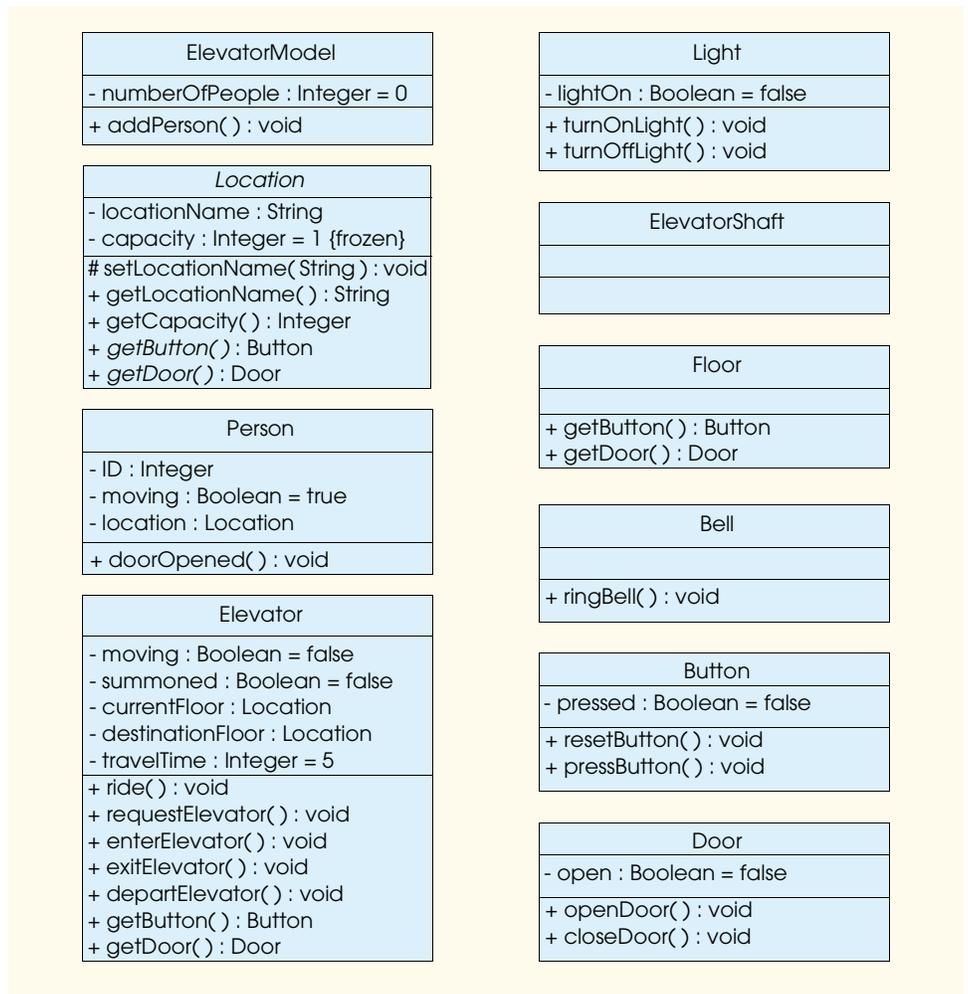
ElevatorModel
- numberOfPeople : Integer = 0
+ addPerson( ) : void

Location
- locationName : String
- capacity : Integer = 1 {frozen}
setLocationName(String) : void
+ getLocationName( ) : String
+ getCapacity( ) : Integer
+ *getButton( )* : Button
+ *getDoor( )* : Door

Person
- ID : Integer
- moving : Boolean = true
- location : Location
+ doorOpened( ) : void

Elevator
- moving : Boolean = false
- summoned : Boolean = false
- currentFloor : Location
- destinationFloor : Location
- travelTime : Integer = 5
+ ride( ) : void
+ requestElevator( ) : void
+ enterElevator( ) : void
+ exitElevator( ) : void
+ departElevator( ) : void
+ getButton( ) : Button
+ getDoor( ) : Door

Light
- lightOn : Boolean = false
+ turnOnLight( ) : void
+ turnOffLight( ) : void

ElevatorShaft

Floor
+ getButton( ) : Button
+ getDoor( ) : Door

Bell
+ ringBell( ) : void

Button
- pressed : Boolean = false
+ resetButton( ) : void
+ pressButton( ) : void

Door
- open : Boolean = false
+ openDoor( ) : void
+ closeDoor( ) : void

Fig. 9.39   Class diagram with attributes and operations (incorporating inheritance).

Software Engineering Observation 9.2

A complete class diagram shows all associations among classes and all attributes and methods for each class. When the number of class attributes, methods and associations is substantial (as in Fig. 9.38 and Fig. 9.39), common practice is to divide this information between two class diagrams: one focusing on associations; the other focusing on attributes and methods. Creating two class diagrams in this manner promotes readability.

Class **Elevator** now has two references to **Location** objects—**currentFloor** and **destinationFloor**—that replace the integer values we used previously in Fig. 4.18 to describe the **Floor**s. **Person** contains a reference to a **Location** object—**location**—that indicates whether **Person** is on a **Floor** or in the **Elevator**.

Implementation: Forward Engineering (Incorporating Inheritance)
"Thinking About Objects" Section 8.17 used the UML to express the Java class structure for our simulation. We continue our implementation while incorporating inheritance, using class **Elevator** as an example.

1. If a class **A** is a subclass of class **B**, then class **A extends** class **B** in the class declaration. For example, class **Elevator** is a subclass of **abstract** superclass **Location**, so the class declaration should read

    ```
    public class Elevator extends Location {

        // class constructor
        public Elevator() {}
    }
    ```

2. If class **B** is an **abstract** class and class **A** is a subclass of class **B**, then class **A** must override the **abstract** methods of class **B** (if class **A** is to be a concrete class). For example, class **Location** contains **abstract** methods **getButton** and **getDoor**, so class **Elevator** must override these methods, because we want to instantiate an **Elevator** object. Figure 9.40 is the Java code implemented for class **Elevator** from Fig. 9.38 and Fig. 9.39. Note that method **getButton** (lines 26–29) returns a reference to the **Elevator**'s **Button** object, and method **getDoor** (lines 32–35) returns a reference to the **Elevator**'s **Door** object—**Elevator** contains associations with both objects, according to the class diagram of Fig. 9.38. Class **Elevator** inherits attributes **capacity** and **locationName**, and non-**abstract** methods **getCapacity**, **setLocationName**, and **getLocationName** from superclass **Location**, so we do not need to implement these attributes and methods in class **Elevator**. Figure 9.39 specifies attributes **moving**, **summoned**, **currentFloor**, **destinationFloor** and **travelTime** and methods **ride**, **requestElevator**, **enterElevator**, **exitElevator** and **departElevator** for class **Elevator**. Lines 6–10 of Fig. 9.40 list these attributes, and lines 19–23 list these methods. The **elevatorButton**, **elevatorDoor** and **bell** references

(lines 11–13) are attributes specified from **Elevator**'s aggregations in Fig. 9.38.

```java
1   // Elevator.java
2   // Generated using class diagrams 9.38 and 9.39
3   public class Elevator extends Location {
4
5      // class attributes
6      private boolean moving;
7      private boolean summoned;
8      private Location currentFloor;
9      private Location destinationFloor;
10     private int travelTime = 5;
11     private Button elevatorButton;
12     private Door elevatorDoor;
13     private Bell bell;
14
15     // class constructor
16     public Elevator() {}
17
18     // class methods
19     public void ride() {}
20     public void requestElevator() {}
21     public void enterElevator() {}
22     public void exitElevator() {}
23     public void departElevator() {}
24
25     // method overriding getButton
26     public Button getButton()
27     {
28        return elevatorButton;
29     }
30
31     // method overriding getDoor
32     public Door getDoor()
33     {
34        return elevatorDoor;
35     }
36  }
```

Fig. 9.40   Class **Elevator** is generated by Fig. 9.38 and Fig. 9.39.

Software Engineering Observation 9.3

Several UML modeling tools convert UML-based designs into Java. These tools can speed the implementation process considerably.[2]

Testing and Debugging Tip 9.1

Using UML modeling tools to generate code automatically helps reduce the amount of programming errors programmers tend to introduce when writing code manually.

---

2. For more information on these tools, refer to the Internet and World-Wide-Web Resources listed at the end of Section 2.9.

We have provided a sound beginning for implementing UML-based designs in Java. In "Thinking About Objects" Section 11.10 we return to interactions, focusing on how objects generate and handle the messages passed in collaborations, and we forward engineer more class diagrams into Java code. We present completely implemented Java code for our simulator in Appendices G, H and I.