Chapter 10

Composing Objects

The design of `RYGTrafficLight` (7.4) can be improved. The core functionality of traffic light objects is illuminating the correct lamp at the correct time, and `RYGTrafficLight` does this well. Some clients, however, may desire a different display. In most modern traffic lights the lamps are arranged vertically instead of horizontally. Perhaps instead of colored lights, printed or spoken commands are to be issued, such as "Stop," "Prepare to stop," and "Proceed." The sequence of commands would correspond to lamp colors, so the core logic of `RYGTrafficLight` would still be appropriate, but the output to the user would be much different.

Traffic light objects have two distinct responsibilities:

- The **core logic** manages the state of the traffic light, properly sequencing the signals.
- The **presentation** displays the state of the light in a manner appropriate to the client.

We will devise separate classes to address these responsibilities. In software design, the part of the system responsible for the core logic is called the *model*. The presentation part is commonly referred to as the *view*.

10.1 Modeling Traffic Light Logic

Since the core logic can be presented in a variety of ways, the names we use for variables and methods will be more generic than those used in `RYGTrafficLight` (7.4). `TrafficLightModel` (10.1) is a new version of our traffic light class that incorporates class constants:

```
public class TrafficLightModel {
    //  Constants used to describe the states of the traffic light
    public static final int OFF     = 0;
    public static final int STOP    = 1;
    public static final int CAUTION = 2;
    public static final int GO      = 3;

    //  The light's current state, one of the constants listed above
    private int state;
```



```
    //  Creates a light with a given initial color
    public TrafficLightModel(int initialState) {
        setState(initialState);  //  Ensures a legal state
    }

    //  Creates a light that is initially in the "stop" state
    public TrafficLightModel() {
        this(STOP);  //  Default light is STOP for safety's skae
    }

    //  Returns the current color of the light
    public int getState() {
        return state;
    }

    //  Determines if a given integer maps to a legal state
    public boolean isLegalState(int potentialState) {
        return potentialState == OFF || potentialState == STOP
               || potentialState == CAUTION || potentialState == GO;
    }

    //  Sets the light's state.  The state is unchanged if
    //  the integer value passed does not map to a legal state.
    public void setState(int newState) {
        if (isLegalState(newState)) {
            state = newState;
        } else {    //  For safety's sake any other int value makes STOP
            state = STOP;
        }
    }

    //  Changes the light's state to the next legal state in its normal cycle.
    public void change() {
        if (getState() == STOP) {
            setState(GO);
        } else if (getState() == CAUTION) {
            setState(STOP);
        } else if (getState() == GO) {
            setState(CAUTION);
        }  //  else the signal remains the same; i.e., OFF lights stay OFF
    }
}
```

Listing 10.1: `TrafficLightModel`—final version of the traffic light class

The differences from `RYGTrafficLight` (7.4) include:

- The class's name is `TrafficLightModel`. The name implies that a `TrafficLightModel` object will model the behavior of a traffic light, but it will not contribute to the view.

- Colors have been replaced with descriptive state names: `STOP`, `CAUTION`, `GO`. These states can be interpreted by the view component as colored lights, voice commands, or whatever else as needed.
- A new state has been added: `OFF`. If added to `RYGTrafficLight` (7.4), `OFF` would correspond to the color black, meaning no lamps lit. This state is useful to make flashing traffic lights, like `STOP`, `OFF`, `STOP`, `OFF`, ..., which could be visualized as a flashing red light.
- The drawing functionality has been removed since `TrafficLightModel` objects provide only the model and not the view.
- Clients can see that state of the object via the `getState()` method. This is essential since the view needs to be able to see the state of the model in order to render the traffic light correctly.
- In a limited way clients can modify the model's state without using the `change()` method. The `setState()` method allows a client to set the `state` instance variable to any valid state value, but it does not allow the client to set the state to an arbitrary value, like 82.
- The Boolean method `isLegalState()` determines which integers map to legal states and which do not.
- The `change()` method makes use of two other methods—`getState()` and `setState()`—to examine and update the `state` instance variable. We could have written `change()` similar to our earlier traffic light code:

```
public void change() {
    if (state == STOP) {
        state = GO;
    } else if (state == CAUTION) {
        state = STOP;
    } else if (state == GO) {
        state = CAUTION;
    }  //  else the signal remains the same; i.e., OFF lights stay OFF
}
```

This code is simpler to understand (which is good), but the new version that uses `getState()` and `setState()` offers an important advantage. Any access to the `state` instance variable must go through `getState()` or `setState()`. For `setState()` the advantage is clear since `setState()` uses `isLegalState()` to ensure an invalid integer is not assigned to the `state` variable. Consider for a moment a more sophisticated scenario. Suppose we wish to analyze how our traffic light models are used. We will keep track of an object's state by logging each time the `state` variable is examined or modified. We can use an integer instance variable named `readCount` that is incremented each time `state` is examined (read). Similarly, we can use an instance variable named `writeCount` that is incremented each time `state` is modified (written). If we limit access to the `state` variable to the `getState()` and `setState()` methods, the code to keep track of those statistics can be isolated to those two methods. If we instead allow methods such as `change()` to directly examine or modify the `state` variable, we would have to duplicate our logging code everywhere we allow such direct access. These presents several disadvantages:

– We have to type in more code. When directly assigning `state` we must remember to also increment `writeCount`:

```
state = GO;
writeCount++;
```

It is simpler and cleaner instead to use


```
setState(GO);
```

and let `setState()` take care of the accounting. The shortest, simplest code that accomplishes the task is usually desirable. More complex code is harder to understand and write correctly.

– We must ensure that the logging code is being used consistently in all places that it should appear. Is it really the same code in all the places that it must appear, or are there errors in one or more of the places it appears? It is easy to mistakenly write within a method

```
state = GO;
readCount++;
```

Here, `writeCount` should have been incremented instead of `readCount`. Errors like this one can be difficult sometimes to track down.

– We must ensure that we do not forget to write the logging code in all the places it should appear. One occurrence in a method of a statement like

```
state = GO;
```

without the associated logging code will lead to errors in our analysis. If we instead use `setState()`, the accounting is automatically handled for us.

While the code presented here is not as sophisticated as this logging scenario, we could mistakenly write code in one of our methods like

```
state = newValue;
```

thinking that `newValue` is in the range 0–3. If it is not, the traffic light model would enter an undefined state. If we use `setState()` instead, it is impossible for the traffic light model to be in an undefined state.

The `TrafficLightModel` class represents objects that that are more complex than any objects we have devised up to this point. Neither constructor assigns the `state` instance variable directly. One constructor defers to the other one via the `this` call (see § 7.1); the other constructor uses the `setState()` method to assign `state`. Since `setState()` contains the logic to make sure the integer passed in is legitimate, it makes sense not to duplicate the effort of checking the legitimacy in two different places within the class. Within the `setState()` method, another method, `isLegalState()`, is used to actually check to see if the proposed integer is valid.

The various methods within `TrafficLightModel` interact with the class constants and instance variable to perform their tasks. Figure 10.1 illustrates the relationships of the pieces of a `TrafficLightModel` object.

Biological cells are units that have complex interworkings. Cells have parts such as nuclei, ribosomes, and mitochondria that much all work togther using complex processes to allow the cell to live and perform its function within the larger framework of a tissue. The tissue is a building block for an organ, and a number of organs are assembled into organisms.

If like manner, a software object such a `TrafficLightModel` instance is itself reasonably complex (although not nearly as complex as a living cell!). The component relationship diagram in Figure 10.1 almost gives the appearance of the biochemical pathways within a living cell. Like cells, programming objects themselves can be quite complex and can be used to build up more complex software structures called *components* which are used to build software systems.

By itself a `TrafficLightModel` object is not too useful. A `TrafficLightModel` object must be paired with a view object to be a fully functional traffic light.

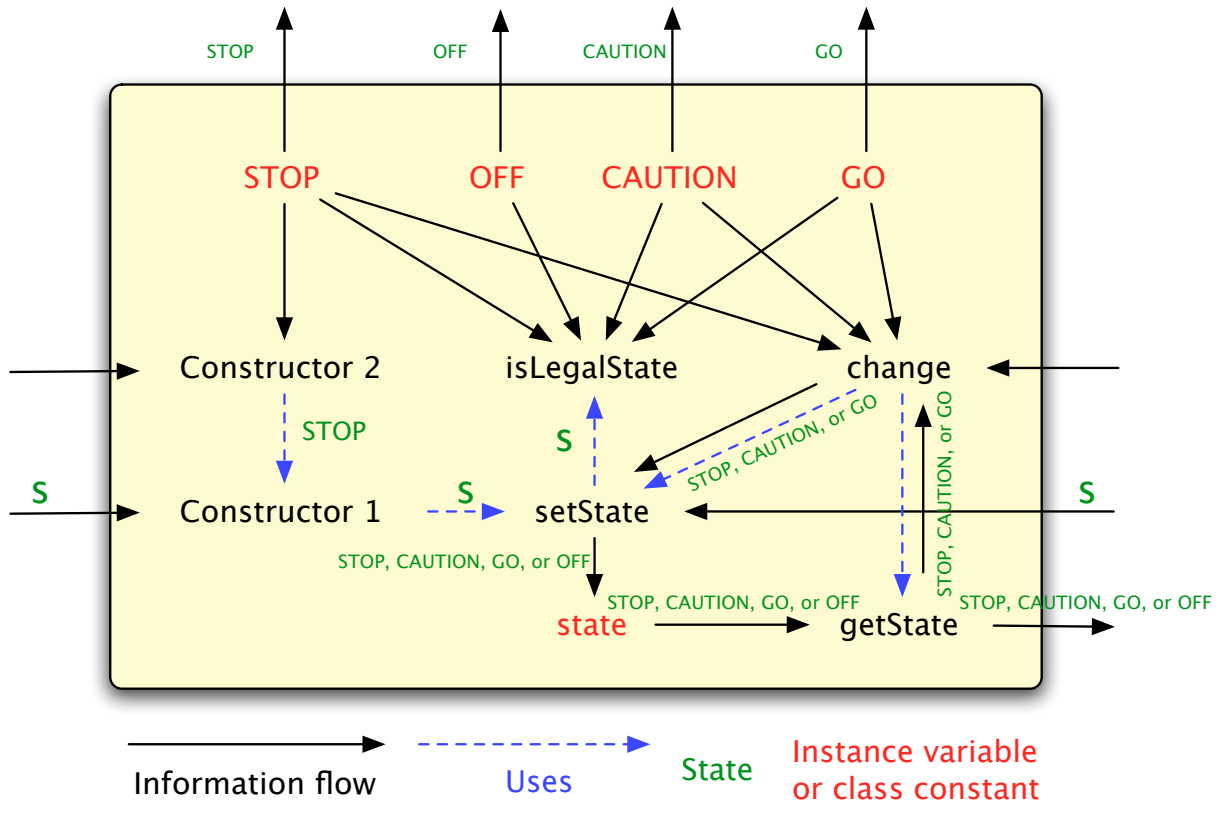

Figure 10.1: Component relationships within traffic light model objects. The solid arrows indicate information flow. Class constants can be read (all arrows flow out) but cannot be modified (no arrows flow in). The `state` variable can be read only by `getState()` (only arrow out) and modified only by the `setState()` method (only arrow in). Arrows that enter or leave the traffic light model object indicate information provided by or available to client code. The dashed arrows show when a method employs the services of another method; for example, `setState()` uses `isLegalState` in order to complete its task. Similarly, `change()` calls `getState()` and can call `setState()`.

10.2 Textual Visualization of a Traffic Light

`TextTrafficLight` (10.2) uses `TrafficLightModel` (10.1) to display a text-based visualization of the state of a traffic light:

```
public class TextTrafficLight {
    //  Provides core functionality
    private TrafficLightModel light;

    //  Create a text traffic light from a traffic light model
    public TextTrafficLight(TrafficLightModel lt) {
        light = lt;
    }

    //  Returns the current state
    public int getState() {
        return light.getState();
    }
```



```
    //  Sets the state
    public void setState(int newState) {
        light.setState(newState);
    }

    //  Change can change the model's state, if the model allows it
    public void change() {
        light.change();
    }

    //  Renders each lamp
    public String drawLamps() {
        if (light.getState() == TrafficLightModel.STOP) {
            return "(R) ( ) ( )";  //  Red stop light
        } else if (light.getState() == TrafficLightModel.CAUTION) {
            return "( ) (Y) ( )";  //  Yellow caution light
        } else if (light.getState() == TrafficLightModel.GO) {
            return "( ) ( ) (G)";  //  Green go light
        } else {
            return "( ) ( ) ( )";  //  Otherwise off
        }
    }

    //  Renders the lamps within a frame
    public String show() {
        return "[" + drawLamps() + "]";
    }
}
```

Listing 10.2: `TextTrafficLight`—a text-based traffic light visualization

The following interactive session demonstrates how a client can make and use a `TextTrafficLight` object:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> t = new TextTrafficLight(new TrafficLightModel(TrafficLightModel.STOP));
> System.out.println(t.show());
[(R) ( ) ( )]
> t.change(); System.out.println(t.show());
[( ) ( ) (G)]
> t.change(); System.out.println(t.show());
[( ) (Y) ( )]
> t.change(); System.out.println(t.show());
[(R) ( ) ( )]
> t.change(); System.out.println(t.show());
[( ) ( ) (G)]
> t.change(); System.out.println(t.show());
[( ) (Y) ( )]
> t.change(); System.out.println(t.show());
```



```
[(R) ( ) ( )]
```

In `TextTrafficLight` (10.2):

- The lone instance variable, `light`, is an object reference to a `TrafficLightModel` object. We say a `TextTrafficLight` object is *composed of* a `TrafficLightModel` object. A `TextTrafficLight` object uses a `TrafficLightModel` and expects it to manage the traffic light's state.
- The `getState()`, `setState()`, and `change()` methods do nothing interesting themselves except call the equivalent methods of the model. This technique is called *delegation*. We say a `TextTrafficLight` object delegates the work of its method to its contained `TrafficLightModel` object. This concept of composition with delegation is common in OO programming.

What advantage does this composition provide? We have separated the implementation of a traffic light's behavior (what colors it can assume and the sequence of colors in a cycle) from its appearance. This decoupling is good, since given a `TrafficLightModel` we can create a new visualization without worrying about the details of how the light operates. The visualization works with the model through the model's public interface and does not know anything about the model's implementation.

To see the payoff of this decoupling, we will create a new visualization of our traffic light. In our `TextTrafficLight` objects the lamps are arranged horizontally, left to right, red to green. Most traffic lights used today in the US arrange their lamps vertically, top to bottom, red to green. In `VerticalTextLight` (10.3) we model such a vertical traffic light:

```
public class VerticalTextLight {
    //  Provides core functionality
    private TrafficLightModel light;

    //  Create a text traffic light from a traffic light model
    public VerticalTextLight(TrafficLightModel lt) {
        light = lt;
    }

    //  Returns the current state
    public int getState() {
        return light.getState();
    }

    //  Sets the state
    public void setState(int newState) {
        light.setState(newState);
    }

    //  Change changes the model's state
    public void change() {
        light.change();
    }

    //  Renders each lamp
    public String drawLamps() {
```



```
        if (light.getState() == TrafficLightModel.STOP) {
            return "|(R)|\n|( )|\n|( )|\n";  //  Red stop light
        } else if (light.getState() == TrafficLightModel.CAUTION) {
            return "|( )|\n|(Y)|\n|( )|\n";  //  Yellow caution light
        } else if (light.getState() == TrafficLightModel.GO) {
            return "|( )|\n|( )|\n|(G)|\n";  //  Green go light
        } else {
            return "|( )|\n|( )|\n|( )|\n";  //  Off otherwise
        }
    }

    //  Renders the lamps within a frame
    public String show() {
        return "+---+\n" + drawLamps() + "+---+";
    }
}
```

Listing 10.3: `VerticalTextLight`—an alternate text-based traffic light visualization

The `"\n"` sequence is an *escape sequence* that represents the newline character. This forces the output to appear on the next line. The following interactive session illustrates:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> t = new VerticalTextLight(new TrafficLightModel(TrafficLightModel.STOP));
> System.out.println(t.show());
+---+
|(R)|
|( )|
|( )|
+---+
> t.change(); System.out.println(t.show());
+---+
|( )|
|( )|
|(G)|
+---+
> t.change(); System.out.println(t.show());
+---+
|( )|
|(Y)|
|( )|
+---+
> t.change(); System.out.println(t.show());
+---+
|(R)|
|( )|
|( )|
```



```
+---+
```

An even more impressive visualization can be achieved using *real* graphics. We will defer the better example until § 12.3, after the concept of inheritance is introduced.

`VerticalTextLight` (10.3) and `TextTrafficLight` (10.2) both leverage the `TrafficLightModel` (10.1) code, the only difference is the presentation. The design is more *modular* than, say, `RYGTrafficLight` (7.4), that combines the logic and the presentation into one class. By comparison, the `RYGTrafficLight` class is *non-modular*. `TextTrafficLight` and `VerticalTextLight` objects are visual shells into which we "plug in" our `TrafficLightModel` object to handle the logic.

Modular design, as we have just demonstrated, can lead to *code reuse*. Code reuse is the ability to use existing code in a variety of applications. A *component* is a reusable software unit—a class—that can be used as is in many different software contexts. In our case, `TrafficLightModel` is a component that can be plugged into different traffic light visualization objects. Component-based software development seeks to streamline programming by making it easier to build complex systems by assembling components and writing as little original code as possible. Component-based software development thus consists largely of component construction and combining components to build the overall application.

10.3 Intersection Example

Traffic lights are used to control traffic flow though intersections. We can model an intersection as an object. Our intersection object is composed of traffic light objects, and these lights must be coordinated to ensure the safe flow of traffic through the intersection.

An intersection object is provided (via a constructor) four traffic light objects in the order north, south, east, and west. The intersection object's `change()` method first does a simple consistency check and then, if the check was successful, changes the color of the lights.

`Intersection` (10.4) is the blueprint for such an intersection object.

```
public class Intersection {
    //  Four traffic lights for each of the four directions
    private TextTrafficLight northLight;
    private TextTrafficLight southLight;
    private TextTrafficLight eastLight;
    private TextTrafficLight westLight;

    //  The constructor requires the client to provide the traffic
    //  light objects
    public Intersection(TextTrafficLight north, TextTrafficLight south,
                        TextTrafficLight east, TextTrafficLight west) {
        northLight = north;
        southLight = south;
        eastLight = east;
        westLight = west;
        //  Check to see if the configuration of lights is valid
        if (!isValidConfiguration()) {
            System.out.println("Illegal configuration");
        }
```



```
    }

    //  Checks to see that there are no conflicts with the current
    //  light colors.   The first two parameters and the last two
    //  parameters are opposing lights.   Callers must ensure the
    //  proper ordering of the parameters.
    public boolean isValidConfiguration() {
        //  Sanity check:
        //  Opposite lights should be the same color for this
        //  controller to work properly; adjacent lights should not be
        //  the same color.  (Other controllers might specify
        //  different rules.)
        if (northLight.getState() != southLight.getState()
                || eastLight.getState() != westLight.getState()
                || northLight.getState() == eastLight.getState()) {
            //  Some conflict exists; for safety's sake, make them all
            //  red
            northLight.setState(TrafficLightModel.STOP);
            southLight.setState(TrafficLightModel.STOP);
            eastLight.setState(TrafficLightModel.STOP);
            westLight.setState(TrafficLightModel.STOP);
            return false;  //  Error condition, return false for failure
        } else {
            return true;
        }
    }

    //  Properly synchronizes the changing of all the
    //  lights in the intersection, then it (re)displays
    //  the intersection.
    public void change() {
        if (northLight.getState() == TrafficLightModel.CAUTION
                ||  eastLight.getState() == TrafficLightModel.CAUTION) {
            //  If any light is yellow, all need to change
            northLight.change();
            southLight.change();
            eastLight.change();
            westLight.change();
        } else if (northLight.getState() != TrafficLightModel.STOP) {
            //  If north/south lights are NOT red, change them
            northLight.change();
            southLight.change();
        } else if (eastLight.getState() != TrafficLightModel.STOP) {
            //  If east/west lights are NOT red, change them
            eastLight.change();
            westLight.change();
        }
    }

    //  Draw the contents of the intersection in two dimensions; e.g.,
```



```
    //                      (North)
    //              (West)          (East)
    //                      (South)
    public void show() {
        System.out.println("                  " + northLight.show());
        System.out.println();
        System.out.println(westLight.show() + "                      " +
                           eastLight.show());
        System.out.println();
        System.out.println("                  " + southLight.show());
    }
}
```

Listing 10.4: `Intersection`—simulates a four-way intersection of simple traffic lights

The following interactive session exercises an `Intersection` object:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> i = new Intersection(new TextTrafficLight
                         (new TrafficLightModel(TrafficLightModel.STOP)),
                       new TextTrafficLight
                         (new TrafficLightModel(TrafficLightModel.STOP)),
                       new TextTrafficLight
                         (new TrafficLightModel(TrafficLightModel.GO)),
                       new TextTrafficLight
                         (new TrafficLightModel(TrafficLightModel.GO)));
> i.show();
                 [(R) ( ) ( )]

[( ) ( ) (G)]                       [( ) ( ) (G)]

                 [(R) ( ) ( )]

> i.change();  i.show();
                 [(R) ( ) ( )]

[( ) (Y) ( )]                       [( ) (Y) ( )]

                 [(R) ( ) ( )]
> i.change();  i.show();
                 [( ) ( ) (G)]

[(R) ( ) ( )]                       [(R) ( ) ( )]

                 [( ) ( ) (G)]
> i.change();  i.show();
                 [( ) (Y) ( )]
```



```
[(R) ( ) ( )]                    [(R) ( ) ( )]

                 [( ) (Y) ( )]
> i.change();  i.show();
                 [(R) ( ) ( )]

[( ) ( ) (G)]                    [( ) ( ) (G)]

                 [(R) ( ) ( )]
> i.change();  i.show();
                 [( ) ( ) (R)]

[( ) (Y) ( )]                    [( ) (Y) ( )]

                 [( ) ( ) (R)]
```

The sequence of signals allows for safe and efficient traffic flow through the modeled intersection.

In `Intersection` (10.4) we see:

- Each `Intersection` object contains four `TextTrafficLight` objects, one for each direction:

  ```
  private TextTrafficLight northLight;
  private TextTrafficLight southLight;
  private TextTrafficLight eastLight;
  private TextTrafficLight westLight;
  ```

  These statements declare the instance variable names, but do not create the objects; recall that objects are created with the `new` operator.

  These `TextTrafficLight` objects are components being reused by the `Intersection` class.

- The constructor is responsible for initializing each traffic light object.

  ```
  public Intersection(TextTrafficLight north, TextTrafficLight south,
                      TextTrafficLight east, TextTrafficLight west) {
      northLight = north;
      . . .
  ```

  It is the client's responsibility to create each light in the proper state. We did so in the rather lengthy initialization statement at the beginning on the interactive session above. The constructor does check to see if the initial state of the system of lights is acceptable by calling `isValidConfiguration()`. In the case of an illegal initial configuration of lights (for example, all the lights are green, or north and west both green), all the lights are made red. This leads to a defunct but safe intersection.

- The `change()` method coordinates the changing of each individual light.

- The `show()` method draws the lights (actually asks the individual lights to draw themselves) in locations corresponding to their location.

`IntersectionMain` (10.5) builds a program around `Intersection` (10.4):

```
import java.util.Scanner;

public class IntersectionMain {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        Intersection intersection
            = new Intersection(new TextTrafficLight
                                   (new TrafficLightModel(TrafficLightModel.STOP)),
                               new TextTrafficLight
                                   (new TrafficLightModel(TrafficLightModel.STOP)),
                               new TextTrafficLight
                                   (new TrafficLightModel(TrafficLightModel.GO)),
                               new TextTrafficLight
                                   (new TrafficLightModel(TrafficLightModel.GO)));
        while (true) {   //  infinite loop
            scan.nextLine();   //  Wait for return key press
            intersection.change();
            intersection.show();
        }
    }
}
```

Listing 10.5: `IntersectionMain`—simulates a four-way intersection of simple traffic lights

The `scan.nextLine()` waits for the user to enter a string and press Return. If the user simply presses Return, an empty string (no characters) is returned by `nextLine()`, and the program's execution continues. The statement beginning

```
while (true) {
```

forms an infinite loop; code within its body executes over and over again continuously. Loops are covered in more detail in Chapter 17. This infinite loop ensures that the cycle continues until the user terminates the program.

10.4 Summary

- Variables within objects can be references to other objects.
- Object composition allows complex objects to be built up from simpler objects.
- Delegation is when a complex object implements an operation by calling the corresponding operation of its contained object. The corresponding operation in the contained object has some conceptual similarity to the container's operation.
- Modular design separates functionality into focused, individual classes that can be used in a variety of software contexts.

- Code reuse is the ability to use existing code (classes) in a variety of applications.
-

10.5 Exercises

1. What is meant by *object composition*? How is it accomplished?
2. What is meant by *composition with delegation*? How is it accomplished?
3. What is meant by *modular design*? How is it accomplished?
4. What advantages are provided by modular design?
5. What is *code reuse*, and why is it important?
6. How does `Intersection` (10.4) behave when the initial configuration is north = green, east = green, south = red, and west = red?
7. What issues are there with `VerticalTextLight` (10.3) objects work with the existing `Intersection` (10.4) code?
8. Not only can a model be plugged into multiple views. A new model can be made and plugged into an existing view. To verify this, develop a new model that makes a flashing red light. That is, calling the `change()` method will cycle the colors red, black, red, black, ... Test your new class with the existing `TextTrafficLight` (10.2) and `VerticalTextLight` (10.3).

