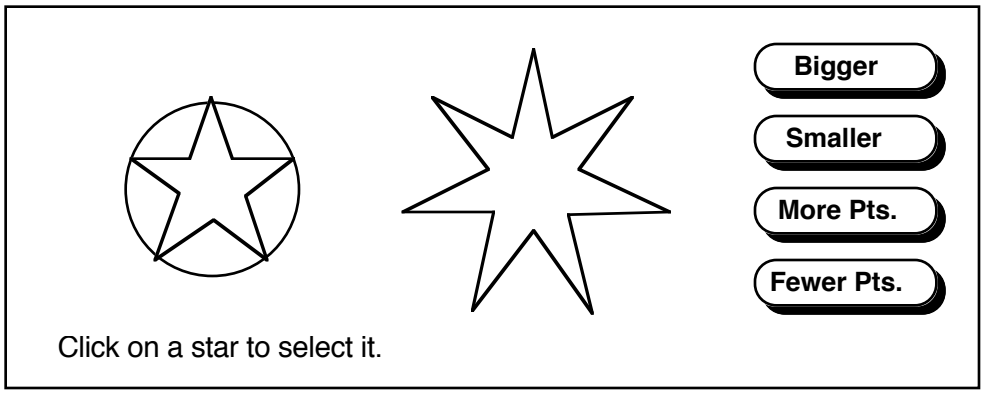
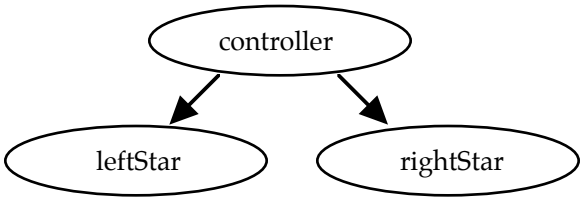
# Chapter 5 Objects containing Objects

## 5.1 Further organization of the stars program

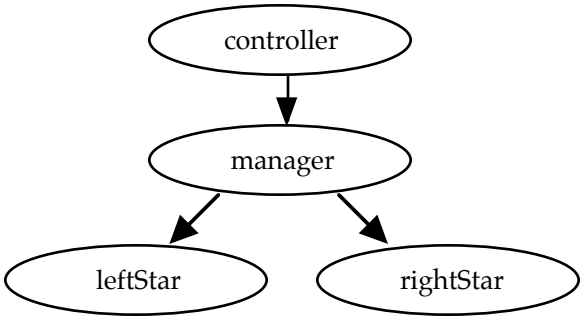



Ignoring the buttons for the moment, the program in the previous chapter has a structure that might be expressed by the following diagram:




Events ( bigger, fewerPoints etc.) are decoded and requests are dispatched to each star.

Handling the two stars increases the size and complexity of the controller. The controller could be made smaller, if a new class which we will call *StarManager* (and a new object – named *manager*) is created as an intermediate between the controller and the two stars. The idea would be to group the stars in the object named *manager* and have the controller deal only with the stars as a group. The program would then be structured as:




Events ( bigger, fewerPoints etc.) are decoded and requests are dispatched to the manager.

Requests are dispatched from the manager to each star.

Each star responds

The controller would be simplified by the replacement of separate messages to each star with a single message to both. This is in keeping with the principal of making the controller as simple as possible.

## The class *StarProg*

The class *StarProg* shown here is a modification (and simplification) of the class *StarProg* in section 4.2. The instructions that have been removed from *StarProg* appear in the class *StarManager* on the next page. The class *Star* for this program is the same as that in section 4.2.

```
public class StarProg extends EventPanel
   {
   private StarManager manager;
   private AButton bigger, smaller, more, fewer;

   public StarProg()
      {
      manager = new StarManager (150, 100);
      bigger = new AButton("Bigger",    320,  20);
      smaller = new AButton("Smaller",  320,  50);
      more = new AButton("More Pts.",   320,  80);
      fewer = new AButton("Fewer Pts.", 320, 110);
      }

   public void paint(Graphics g)
      {
      manager.paint(g);
      bigger.paint(g);
      smaller.paint(g);
      more.paint(g);
      fewer.paint(g);
      }

   public void mousePressed(MouseEvent e)
      {
      int x = e.getX(), y = e.getY();
      if (x < 320)
         manager.mark(x, y);
      else if (bigger.contains(x,y))
         manager.bigger();
      else if (smaller.contains(x,y))
         manager.smaller();
      else if (more.contains(x,y))
         manager.morePoints();
      else if (fewer.contains(x,y))
         manager.fewerPoints();
      repaint();
      }
   }
```

### The class *StarManager*

```
public class StarManager
    {
    private Star leftStar, rightStar;

    public StarManager(int centerHorizontal, int centerVertical)
        {
        leftStar = new Star(centerHorizontal - 75, centerVertical);
        rightStar = new Star(centerHorizontal + 75, centerVertical);
        }

    public void paint(Graphics g)
        {
        leftStar.paint(g); rightStar.paint(g);
        }

    public void mark(int x, int y)
        {
        if (leftStar.contains(x, y))
            leftStar.mark();
        else
            leftStar.unmark();
        if (rightStar.contains(x, y))
            rightStar.mark();
        else
            rightStar.unmark();
        }

    public void bigger()
        {
        leftStar.bigger(); rightStar.bigger();
        }

    public void smaller()
        {
        leftStar.smaller(); rightStar.smaller();
        }

    public void morePoints()
        {
        leftStar.morePoints(); rightStar.morePoints();
        }

    public void fewerPoints()
        {
        leftStar.fewerPoints(); rightStar.fewerPoints();
        }
    }
```

## Exercises — 5.1

1. Make the modification suggested above to the programs suggested in exercises of chapter 4.

## 5.2 Feedback from Methods I — Avoiding Unnecessary Drawing

We have written only one method, the contains method of the class *AButton*, which has returned a result. It is appropriate to have methods return results in a much more general context.

Painting the stars in the program above takes some time and causes a visible redrawing of the graphics on the screen. When the user presses a key that causes no change it would be nice if the program refrained from repainting the screen. This could be done if the methods of the *StarManager* class returned a true or false value indicating whether or not anything was done. In this case, the *keyPressed* method would be rewritten as:

```
public void keyPressed(KeyEvent e)
    {
    char key = e.getKeyChar();
    boolean changesMade = false;
    if (key == 'b')
        changesMade = manager.bigger();
    else if (key == 's')
        changesMade = manager.smaller();
    else if (key == 'm')
        changesMade = manager.morePoints();
    else if (key == 'f')
        changesMade = manager.fewerPoints();
    if (changesMade)
        repaint();
    }
```

Note that the result returned by a method is saved by placing a variable of the appropriate type to the left of an assignment and the call of the method to the right.

Each of the methods of *StarManager* would be required to return the appropriate result. If the methods of *Star* did so, the methods of *StarManager* could do so in the following way.

```
public boolean bigger()
    {
    boolean changesMade = false;
    if (leftStar.bigger())
        changesMade = true;
    if (rightStar.bigger())
        changesMade = true;
    return changesMade;
    )
```

The methods of Star must also return results. *Bigger* might look like:

```
public boolean bigger()
    {
    if (marked)
        {
        radius += 10;
        return true;
        }
    return false;
    }
```

Note that the keyword *return* abruptly ends a method when it is encountered. Thus the method above returns true when the radius is changed (it doesn't get to the line *return false;*).

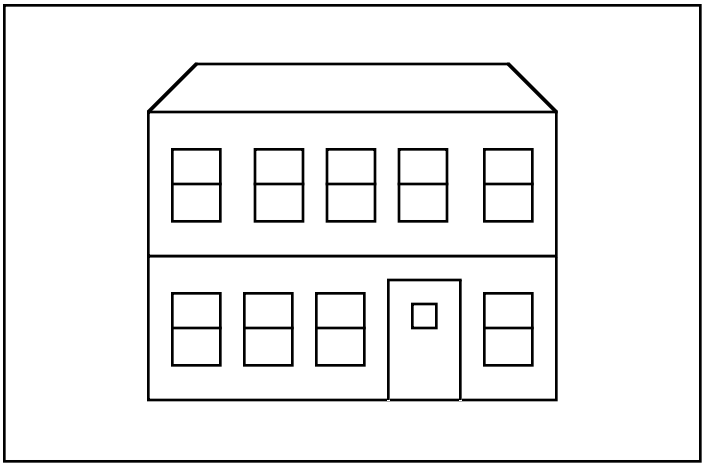
The method *bigger* above could be constructed like the one in *StarManager* and shown below, but with only one thing to do this longer form is unnecessary.

```
public boolean bigger()
    {
    boolean changesMade = false;
    if (marked)
        {
        radius += 10;
        changesMade = true;
        }
    return changesMade;
    }
```

## Exercises – 5.2

1. Use feedback from methods to create versions of the programs in exercises of chapter 4 that avoid unnecessary drawing.

## 5.3 A deeply nested example - a house

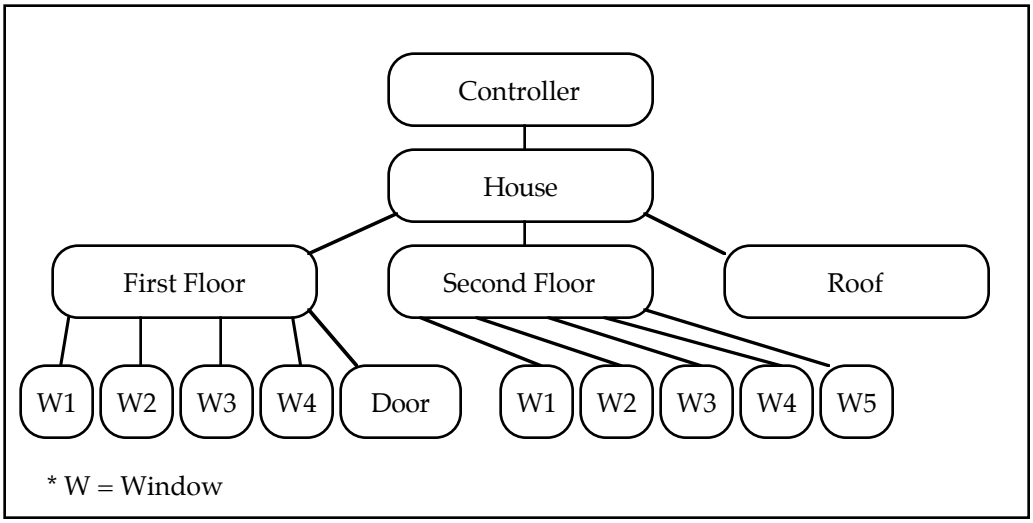


The structure of the house – the house has floors, the floors have windows, etc. – leads naturally to the following program structure.

### Program Structure

The program will consist of:

- a controller containing a house.
- a house containing a first floor, a second floor and a roof.
- a first floor containing four windows and a door.
- four windows and a door.
- a second floor containing five windows.
- five windows.
- a roof.




Each box in the diagram above represents an object. There are seven classes (the nine windows all being of the same class). A program with this structure is found on the next two pages.

## Controller

```
public class HouseProgram
                  extends EventPanel
   {
   private House house;

   public HouseProgram()
       {
       house = new House(50, 30);
       }

   public void paint(Graphics g)
       {
       house.paint(g);
       }
   }
```

## House

```
public class House
    {
    private int left, top;
    private Floor1 first;
    private Floor2 second;
    private Roof roof;

    public House(int h, int v)
       {
       left = h; top = v;
       first = new Floor1(left, top +110);
       second = new Floor2(left, top +30);
       roof = new Roof(left, top);
       }

    public void paint(Graphics g)
       {
       first.paint(g);
       second.paint(g);
       roof.paint(g);
       }
    }
```

## Floor 1

```
public class Floor1
   {
   private int left, top;
   private AWindow w1, w2, w3, w4;
   private Door door;

   public Floor1(int h, int v)
       {
       left = h; top = v;
       w1 = new AWindow(left+10, top+20);
       w2 = new AWindow(left+50, top+20);
       w3 = new AWindow(left+90, top+20);
       door = new Door(left+130, top+10);
       w4 = new AWindow(left+180, top+20);
       }

   public void paint(Graphics g)
       {
       g.drawRect(left, top, 220, 80);
       w1.paint(g);
       w2.paint(g);
       w3.paint(g);
       w4.paint(g);
       door.paint(g);
       }
   }
```

## Floor 2

```
public class Floor2
   {
   private int left, top;
   private AWindow w1, w2, w3, w4, w5;

   public Floor2(int h, int v)
       {
       left = h; top = v;
       w1 = new AWindow(left+10, top+20);
       w2 = new AWindow(left+60, top+20);
       w3 = new AWindow(left+100, top+20);
       w4 = new AWindow(left+140, top+20);
       w5 = new AWindow(left+180, top+20);
       }

   public void paint(Graphics g)
       {
       g.drawRect(left, top, 220, 80);
       w1.paint(g);
       w2.paint(g);
       w3.paint(g);
       w4.paint(g);
       w5.paint(g);
       }
   }
```

### Roof

```
public class Roof
    {
    private Polygon outline;
    private int left, top;

    public Roof(int h, int v)
        {
        left = h; top = v;
        outline = new Polygon();
        outline.addPoint(left+30, top);
        outline.addPoint(left+190, top);
        outline.addPoint(left+220, top+30);
        outline.addPoint(left, top+30);
        }

    public void paint(Graphics g)
        {
        g.drawPolygon(outline);
        }
    }
```

### AWindow

```
public class AWindow
    {
    private int left, top, half;

    public AWindow(int h, int v)
        {
        left = h; top = v;
        half = top + 20;
        }

    public void paint(Graphics g)
        {
        g.drawRect(left, top, 30, 20);
        g.drawRect(left, half, 30, 20);
        }
    }
```

### Door

```
public class Door
    {
    private int left, top, winLeft, winTop;

    public Door(int h, int v)
        {
        left = h; top = v;
        winLeft = left + 10;
        winTop = top + 10;
        }

    public void paint(Graphics g)
        {
        g.drawRect(left, top, 40, 70);
        g.drawRect(winLeft, winTop, 20, 20);
        }
    }
```

## Exercises — 5.3

1. Allow the user to light the windows by clicking on them. Completion of this exercise is important to an understanding of objects and communication between them. Do this in the same way that the stars were marked in the program of section 5.1. It's nicest if clicking on the windows toggles the lights. You might also include the window in the door.

   Set up communications from the controller to the windows, via the house and floors. The house and the floors should each have a method to pass information down the line. These methods will be sent information — in this case the coordinates of a mouse click — and then forward the information to the next object down the line.
2. Make a house (or similar building) with three floors and two different types of windows.
3. Allow the user to change the colors of the roof and the exterior walls. Allow these items to be selected as the stars were and use color buttons to select colors.
4. Alter the program above so that two houses will be displayed, side by side. Do this by inserting a Houses class between the controller and the House class.
5. Make another deeply nested program such as a Snowman, Faces, University, or FamilyTree program.

## 5.4 Feedback from Methods II — Retrieving Information from Nested Objects

In the house program it would seem appropriate to print the number of windows or the number of window panes or something similar. In the controller, one might have:

```
public class HouseProgram extends ...

    public void paint(Graphics g)
        {
        house.paint(g);
        g.drawString("There are 9 windows.", 20, 280);
        }
```

It would be better to ask the house how many windows there are. Then a change to the house would not require a change to the controller.

```
public class HouseProgram extends ...

    public void paint(Graphics g)
        {
        house.paint(g);
        int count = house.getWindowCount();
        g.drawString("There are " + count + " windows.", 20, 280);
                        or
        house.paint(g);
        g.drawString("There are " + house.getWindowCount() + " windows.", 20, 280);
        }
```

It is certainly possible to write a method *getWindowCount* in the *House* class to return 9, but it would be better if putting another window on one of the floors would require changes in the class for that floor only. With this in mind, one would write:

```
public class House

    public int getWindowCount()
        {
        int count1 = first.getWindowCount();
        int count2 = second.getWindowCount();
        return count1 + count2;
                        or
        return first.getWindowCount() + second.getWindowCount();
        }
```

The method above requires that each floor has a *getWindowCount* method that returns the number of windows on that floor. Thus we finish this example with the methods:

```
public class Floor1

    public int getWindowCount()
        {
        return 4; // return 5 if you consider the glass in the door to be a window
        }

public class Floor2

    public int getWindowCount()
        {
        return 5;
        }
```

## Exercises — 5.4

1. Report the number of window panes in a building. To do this, have the windows and doors report the number of panes in each.
2. Put wreaths or pumpkins or other decorations in some of the windows of the house. Report the number of decorations.
3. Report the number of lighted windows (in the program suggested in 5.3 exercise 2).
4. Report the number of houses, windows, lighted windows… in a program that draws several houses (such as the program suggested in 5.3 exercises 4 and 5).