Chapter 8

Frameworks

CHAPTER TOPICS

In Chapter 6, you saw how the inheritance mechanism can be used to derive a new class that extends and customizes a given class. In this chapter we will go beyond simple inheritance and turn to larger clusters of classes, called *frameworks*, that collectively form the basis for customization. We will study how to use frameworks to derive new classes or even entire applications. Then we will turn to the design of a sample framework and show how that framework forms the basis of the Violet UML editor.

8.1 Frameworks

A framework is a set of classes and interface types that structures the essential mechanisms of a particular domain.

A *framework* is a set of cooperating classes that implements the mechanisms that are essential for a particular problem domain. A programmer can create new functionality in the problem domain by extending framework classes. For example, Swing is a framework for the problem domain of graphical user interface programming. A programmer can implement new GUI programs by forming subclasses of `JFrame`, `JComponent`, and so on.

Unlike a design pattern, a framework is not a general design rule. It consists of classes that provide functionality in a particular domain. Typically, a framework uses multiple patterns.

An application framework is a framework for creating applications of a particular type.

An *application framework* consists of a set of classes that implements services common to a certain type of application. To build an actual application, the programmer subclasses some of the framework classes and implements additional functionality that is specific to the application that the programmer is building. Thus, the first characteristic of an application framework is:

- An application framework supplies a set of classes that an application programmer augments to build an application, often by forming subclasses of framework classes.

Inversion of control in a framework signifies that the framework classes, and not the application classes, are responsible for the control flow in the application.

The programmer has little or no influence on the order in which the methods of the programmer-supplied classes are called. The majority of activity occurs in the framework, and eventually some objects of the programmer-defined classes are constructed. Then the framework calls their methods in the order that it deems appropriate. This phenomenon is often called *inversion of control*.

- In an application framework, the framework classes, and not the application-specific classes, control the flow of execution.

It is the role of the framework to determine which methods to call at what time. Its designers have expert knowledge about control flow. It is the job of the application programmer to override those methods to fulfill the application-specific tasks.

TIP Designing a single class is an order of magnitude harder than designing a single method because you must anticipate what other programmers will do with it. Similarly, designing a framework is much harder than designing a class library or a single application because you must anticipate what other programmers want to achieve. A good rule of thumb for validating the design of a framework is to use it to build at least three different applications.

8.2 Applets as a Simple Framework

An applet is a Java program that runs inside a browser.

The applet package is a simple framework that demonstrates subclassing from framework classes and inversion of control.

Java applets are Java programs that run inside a Web browser (see Figure 1).

The `java.applet` package is a simple application framework: It contains superclasses to make applets, and the application programmer adds classes and overrides methods to make an actual applet. The `main` method is not supplied by the programmer of a specific applet. The sequencing of the operations that the programmer supplies is under the control of the framework.

To design an applet, you must write a class that extends the `Applet` class. You must override some or all of the following methods:

- `init`: Called exactly once, when the applet is first loaded. Purpose: Initialize data structures and add user interface elements.
- `start`: Called when the applet is first loaded and every time the user restores the browser window containing the applet. Purpose: Start or restart animations or other computationally intensive tasks.

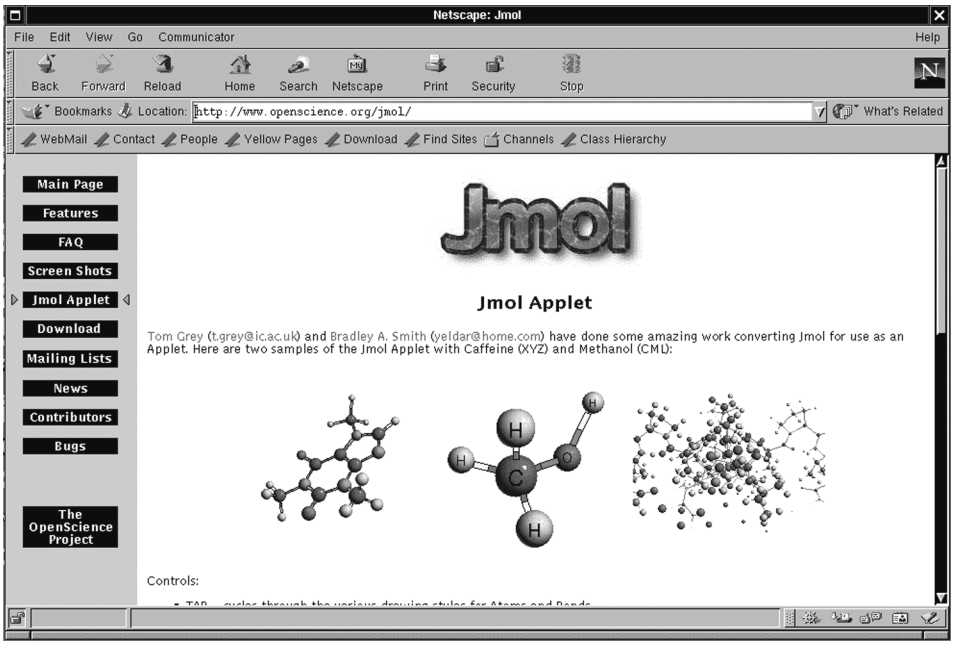


Figure 1

An Applet

- `stop`: Called when the user leaves the browser window containing the applet, and when the browser terminates. Purpose: Stop computationally intensive tasks when the applet is not being viewed.
- `destroy`: Called when the browser terminates. Purpose: Relinquish any resources that were acquired during `init` or other processing.
- `paint`: Called when the applet window needs repainting. Purpose: Redraw the window contents to reflect the current state of the applet data structures.

The sample applet at the end of this section is quite typical. The applet shows a scrolling banner (see Figure 2). A Web designer can customize the applet by specifying different messages, fonts, and delay timings. Here is a typical HTML file:

```
<applet code="BannerApplet.class" width="300" height="100">
<param name="message" value="Hello, World!"/>
<param name="fontname" value="Serif"/>
<param name="fontsize" value="64"/>
<param name="delay" value="10"/>
</applet>
```

The `init` method reads these parameters with the `getParameter` method. It then initializes a `Font` object and a timer. The timer moves the starting position of the string and calls `repaint` whenever the timer delay interval has lapsed.

The `start` method starts the timer and the `stop` method stops it. Thus, the message does not scroll when the applet is not visible. You can verify this by minimizing the browser window and restoring it again. The scrolling picks up where it left off when you minimized the window.

Finally, the `paint` method draws the string.

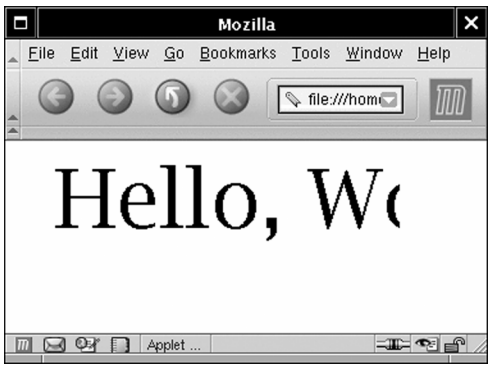


Figure 2

The Scrolling Banner Applet

You can see the typical characteristics of the framework in this example.

- The applet programmer uses inheritance to extend the `Applet` framework class to a specific program.
- The `Applet` class deals with the behavior that is common to all applets: interaction with the browser, parsing `param` tags, determining when the applet is visible, and so on. The applet programmer only fills in customized behavior for a particular program.
- Inversion of control means that the applet programmer is not concerned with the overall flow of control, but only fills in handlers for initialization, starting, stopping, and painting. When these methods are called is beyond the control of the applet programmer.

Ch8/applet/BannerApplet.java

```
1  import java.applet.*;
2  import java.awt.*;
3  import java.awt.event.*;
4  import java.awt.font.*;
5  import java.awt.geom.*;
6  import javax.swing.*;
7
8  public class BannerApplet extends Applet
9  {
10    public void init()
11    {
12       message = getParameter("message");
13       String fontname = getParameter("fontname");
14       int fontsize = Integer.parseInt(getParameter("fontsize"));
15       delay = Integer.parseInt(getParameter("delay"));
16       font = new Font(fontname, Font.PLAIN, fontsize);
17       Graphics2D g2 = (Graphics2D) getGraphics();
18       FontRenderContext context = g2.getFontRenderContext();
19       bounds = font.getStringBounds(message, context);
20
21       timer = new Timer(delay, new
22          ActionListener()
23          {
24             public void actionPerformed(ActionEvent event)
25             {
26                start--;
27                if (start + bounds.getWidth() < 0)
28                   start = getWidth();
29                repaint();
30             }
31          });
32    }
33
34    public void start()
35    {
36       timer.start();
37    }
38
```

```
39    public void stop()
40    {
41       timer.stop();
42    }
43
44    public void paint(Graphics g)
45    {
46       g.setFont(font);
47       g.drawString(message, start, (int) -bounds.getY());
48    }
49
50    private Timer timer;
51    private int start;
52    private int delay;
53    private String message;
54    private Font font;
55    private Rectangle2D bounds;
56 }
```

8.3 The Collections Framework

The collections library is both a repository of common data structures and a framework for new collection classes.

As you know, the Java library contains useful data structures such as linked lists and hash tables. Most programmers are simply interested in the collection library as a provider of common data structures. However, the designers of these collection classes supplied more than just a set of useful classes. They provided a framework that makes it easy to add more collection classes in such a way that the new classes can interact with existing collections. We will demonstrate this capability by adding the queue class of Chapter 3 to the framework. We will then critically examine the collections framework.

8.3.1 An Overview of the Collections Framework

A collection is a data structure that contains objects, which are called the *elements* of the collection. The *collections framework* specifies a number of interface types for collections. They include

- `Collection`: the most general collection interface type
- `Set`: an unordered collection that does not permit duplicate elements
- `SortedSet`: a set whose elements are visited in sorted order
- `List`: an ordered collection

The framework also supplies concrete classes that implement these interface types. Among the most important classes are

- `HashSet`: a set implementation that uses hashing to locate the set elements
- `TreeSet`: a sorted set implementation that stores the elements in a balanced binary tree
- `LinkedList` and `ArrayList`: two implementations of the `List` interface

These interface types and classes are shown in Figure 3.

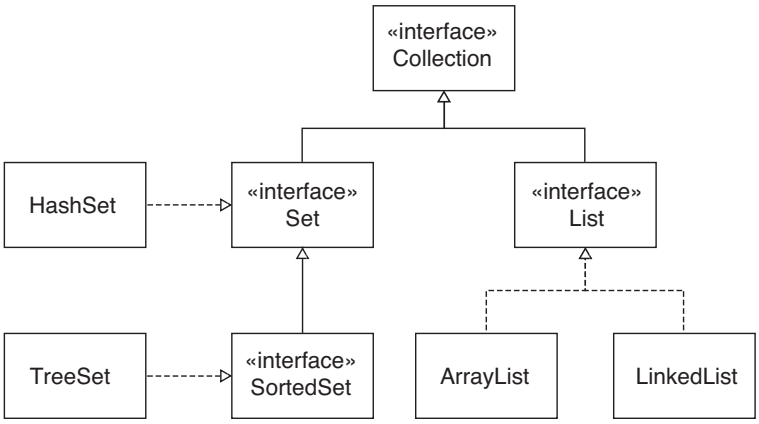


Figure 3

Collection Interface Types and Implementing Classes

All collection classes and interfaces are generic types; the type parameter denotes the type of the collected elements.

NOTE The collections framework also defines a `Map` interface type and implementations `HashMap` and `TreeMap`. A map associates one set of objects, called the *keys*, with another set of objects, called the *values*. An example of such an association is the map of applet parameters that associates parameter names with parameter values. However, the `Map` type is not a subtype of the `Collection` type. Programmers generally prefer to use methods that locate map values from their keys. If a map was implemented as a collection, programmers would need to work with a sequence of key/value pairs.

For simplicity, we will not consider maps in our discussion of the collections framework.

8.3.2 The `Collection` and `Iterator` Interface Types

The two fundamental interface types of the collections framework are `Collection` and `Iterator`. A collection is any class that can hold elements in some way. Individual collection classes may have different disciplines for storing and locating elements. For example, a linked list keeps elements in the order in which they were inserted, whereas a sorted set keeps them in ascending sort order. An iterator is a mechanism for visiting the elements of the collection. We discussed iterators already in Chapters 1 and 3. Recall that the `Iterator<E>` interface type has three methods:

```
boolean hasNext()
E next()
void remove()
```

The `Collection<E>` interface extends the `Iterable<E>` interface type. That interface type has a single method

```
Iterator<E> iterator()
```

NOTE *Any* class that implements the `Iterable<E>` interface type can be used in the "for each" loop. Therefore, you use the "for each" loop with all collections.

The `Collection<E>` interface type has the following methods:

```
boolean add(E obj)
boolean addAll(Collection<? extends E> c)
void clear()
boolean contains(Object obj)
boolean containsAll(Collection<?> c)
boolean equals(Object obj)
int hashCode()
boolean isEmpty()
Iterator<E> iterator()
boolean remove(Object obj)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
int size()
Object[] toArray()
E[] toArray(E[] a)
```

That is a hefty interface type. It would be quite burdensome to supply all of these methods for every collection class. For that reason, the framework supplies a class `AbstractCollection` that implements almost all of these methods. For example, here is the implementation of the `toArray` method in the `AbstractCollection<E>` class.

```
public Object[] toArray()
{
   Object[] result = new Object[size()];
   Iterator e = iterator();
   for (int i = 0; e.hasNext(); i++)
      result[i] = e.next();
   return result;
}
```

This is again the TEMPLATE METHOD pattern at work: The `toArray` method is synthesized from the primitive operations `size` and `iterator`.

NOTE Because it is impossible to construct an array from a generic type parameter, this method returns an `Object[]` array, not an array of type `E[]`.

The `AbstractCollection` class leaves only two methods undefined. They are

```
int size()
Iterator<E> iterator()
```

Any concrete collection class must minimally supply implementations of these two methods. However, most concrete collection classes also override the `add` and `remove` methods.

NOTE The `AbstractCollection` class defines the `add` method as a dummy operation that throws an `UnsupportedOperationException`. That default is reasonable for immutable collections.

8.3.3 Adding a New Collection to the Framework

In this section, you will see how to fit the queue class of Chapter 3 into the collections framework.

We will enhance the queue class of Chapter 3 and define a generic class `BoundedQueue` that extends the `AbstractCollection` class (see Figure 4).

We have to make a slight change to the `add` method. The collections framework requires that the `add` method return `true` if adding the element modifies the collection. The queue class always returns `true`, but a set class would return `false` if the element to be added was already present in the set.

Finally, we need to supply an iterator that visits the queue elements. You will find the code at the end of this section.

A class that is added to the collections hierarchy can benefit from the mechanisms that the framework provides.

What is the benefit of adding the queue class to the collections framework? The Java library contains a number of mechanisms that work for arbitrary collections. For example, all collections have an `addAll` method that does a bulk addition of all elements of one collection to another. You can pass a `BoundedQueue` object to this method. Moreover, the `Collections` class that you encountered in Chapter 4 has static methods for a number of common algorithms, such as finding the minimum and maximum element in any collection. Thus, a large number of methods can be applied to `BoundedQueue` objects when the class becomes a part of the framework.

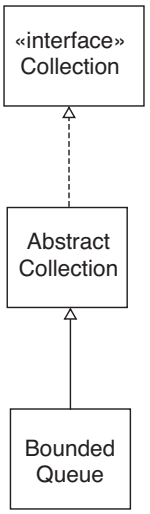


Figure 4

Adding the `BoundedQueue` Class to the Collections Framework

NOTE As of version 5.0, the standard library has a `Queue` interface type. That interface type has been designed primarily for threadsafe queues. For simplicity, our `BoundedQueue` class doesn't implement it.

NOTE Because it is not possible to construct arrays with a generic type, the `BoundedQueue` stores its value in an `Object[]` array. Casts are used when accessing elements of type `E`. The compiler flags these casts as unsafe because it cannot verify their correctness. You can do better—see Exercise 8.7.

Ch8/queue/BoundedQueue.java

```
1  import java.util.*;
2
3  /**
4     A first-in, first-out bounded collection of objects.
5  */
6  public class BoundedQueue<E> extends AbstractCollection<E>
7  {
8     /**
9        Constructs an empty queue.
10       @param capacity the maximum capacity of the queue
11       @precondition capacity > 0
12    */
13    public BoundedQueue(int capacity)
14    {
15       elements = new Object[capacity];
16       count = 0;
17       head = 0;
18       tail = 0;
19    }
20
21    public Iterator<E> iterator()
22    {
23       return new
24          Iterator<E>()
25          {
26             public boolean hasNext()
27             {
28                return visited < count;
29             }
30
31             public E next()
32             {
33                int index = (head + visited) % elements.length;
34                E r = (E) elements[index];
35                visited++;
36                return r;
37             }
38
39             public void remove()
40             {
```

```
41                throw new UnsupportedOperationException();
42             }
43
44             private int visited = 0;
45          };
46    }
47
48    /**
49       Removes object at head.
50       @return the object that has been removed from the queue
51       @precondition size() > 0
52    */
53    public E remove()
54    {
55       E r = (E) elements[head];
56       head = (head + 1) % elements.length;
57       count--;
58       return r;
59    }
60
61    /**
62       Appends an object at tail.
63       @param anObject the object to be appended
64       @return true since this operation modifies the queue.
65       (This is a requirement of the collections framework.)
66       @precondition !isFull()
67    */
68    public boolean add(E anObject)
69    {
70       elements[tail] = anObject;
71       tail = (tail + 1) % elements.length;
72       count++;
73       return true;
74    }
75
76    public int size()
77    {
78       return count;
79    }
80
81    /**
82       Checks whether this queue is full.
83       @return true if the queue is full
84    */
85    public boolean isFull()
86    {
87       return count == elements.length;
88    }
89
90    /**
91       Gets object at head.
92       @return the object that is at the head of the queue
93       @precondition size() > 0
94    */
95    public E peek()
96    {
```

```
97        return (E) elements[head];
98     }
99
100    private Object[] elements;
101    private int head;
102    private int tail;
103    private int count;
104 }
```

Ch8/queue/QueueTester.java

```
1  import java.util.*;
2
3  public class QueueTester
4  {
5     public static void main(String[] args)
6     {
7        BoundedQueue<String> q = new BoundedQueue<String>(10);
8
9        q.add("Belgium");
10       q.add("Italy");
11       q.add("France");
12       q.remove();
13       q.add("Thailand");
14
15       ArrayList<String> a = new ArrayList<String>();
16       a.addAll(q);
17       System.out.println("Result of bulk add: " + a);
18       System.out.println("Minimum: " + Collections.min(q));
19    }
20 }
```

8.3.4 The Set Interface Type

As you have seen, the `Collection` interface type defines methods that are common to all collections of objects. That interface type has two important subtypes, `Set` and `List`. Let's discuss the `Set` interface first. Its definition is

```
public interface Set<E> extends Collection<E> { }
```

Perhaps surprisingly, the `Set` interface type adds *no methods* to the `Collection` interface type. Why have another interface type when there are no new methods?

Conceptually, a set is a collection that eliminates duplicates. That is, inserting an element that is already present has no effect on the set. Furthermore, sets are *unordered* collections. Two sets should be considered equal if they contain the same elements, but not necessarily in the same order.

That is, the `add` and `equals` methods of a set have conceptual restrictions when compared to the same methods of the `Collection` interface type. Some algorithms may require sets, not arbitrary collections. By supplying a separate interface type, a method can require a `Set` parameter and thus refuse collections that aren't sets.

8.3.5 The `List` Interface Type

The Java collections framework defines a "list" as an ordered collection in which each element can be accessed by an integer index. The `List<E>` interface type adds the following methods to the `Collection<E>` interface type:

```
void add(int index, E obj)
boolean addAll(int index, Collection<? extends E> c)
E get(int index)
int indexOf(E obj)
int lastIndexOf(Object obj)
ListIterator<E> listIterator()
ListIterator<E> listIterator(int index)
E remove(int index)
E set(int index, E element)
List<E> subList(int fromIndex, int toIndex)
```

As you can see, most of these methods are concerned with the index positions.

The `ListIterator<E>` interface type is a subtype of `Iterator<E>`. Here are the added methods:

```
int nextIndex()
int previousIndex()
boolean hasPrevious()
E previous()
void add(E obj)
void set(E obj)
```

Recall from Chapter 1 that an iterator is conceptually located between two elements. The `nextIndex` and `previousIndex` methods yield the index positions of the neighbor elements. These methods are conceptually tied to the fact that the list iterator visits an indexed collection.

The other methods are unrelated to indexing. They simply allow backwards movement and element replacement.

Of course, the best-known class that implements the `List` interface type is the `ArrayList` class. More surprisingly, the `LinkedList` class also implements the `List` interface type. That flies in the face of everything that is taught in a data structures class. Accessing elements in a linked list by their index is slow: To visit the element with a given index, you must first visit all of its predecessors.

This is indeed a weakness in the design of the collections framework. It would have been an easy matter to supply two interface types: `OrderedCollection` for linked lists and `IndexedCollection` for arrays.

The library programmers belatedly noticed this problem when they implemented the `binarySearch` method in the `Collections` class. The binary search algorithm locates an element in a *sorted* collection. You start with the middle element. If that element is larger than the element you are looking for, you search the first half. Otherwise, you search the second half. Either way, every step cuts the number of elements to consider in half. The algorithm takes $O(\log_2(n))$ steps if the collection has n elements, *provided* you can access an individual element in constant time. Otherwise, the algorithm is completely pointless and it would be faster to use a sequential search that simply looks at all elements.

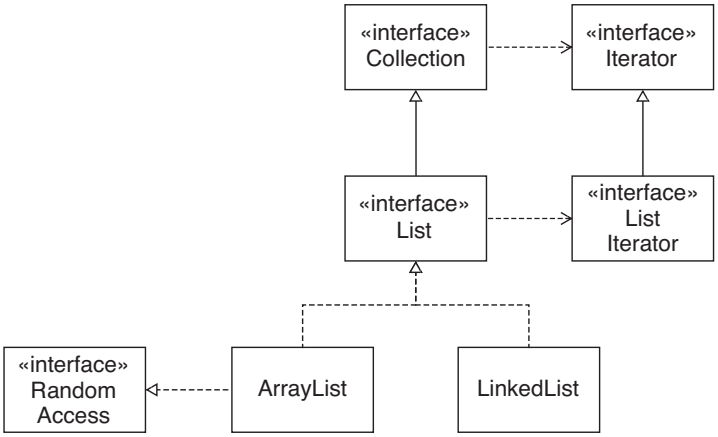


Figure 5

The List Classes

To fix this problem, version 1.4 of the library added an interface type `RandomAccess` that has no methods. It is simply a tagging interface type, to be used with an `instanceof` test. For example, a search method can test whether a `List` supports fast element access or not:

```
if (list instanceof RandomAccess)
   // Use binary search
else
   // Use linear search
```

The `ArrayList` class implements this interface type, but the `LinkedList` class does not.

As so often in software design, it is better to be familiar with the foundations of computer science and apply them correctly than to try to patch up one's design errors later.

Figure 5 shows the `List` interface type and the classes that implement it.

8.3.6 Optional Operations

If you look at the API documentation of the collections framework, you will find many methods that are tagged as "optional operations". Among them is the `add` method of the `Collection` interface type. The `AbstractCollection` class defines the `add` method so that an `UnsupportedOperationException` is thrown when it is called. The optional operations are controversial, but there is a good reason why the library designers make use of them. The need for optional operations arises from certain *views*. A view is an object of a class that implements one of the interface types in the collections framework, and that permits restricted access to a data structure.

The collections framework defines a number of methods that yield views. Here is a typical example. An array is a built-in Java type with no methods. The `asList` method of the `Arrays` class turns an array into a collection that implements the `List` interface type:

```
String[] strings = { "Kenya", "Thailand", "Portugal" };
List<String> view = Arrays.asList(strings);
```

You can apply the `List` methods to the `view` object and access the array elements. The `view` object *does not copy* the elements in the array. The `get` and `set` methods of the view object are defined to access the original array. You can think of the view as a shallow copy of the array.

What is the use? A `List` has a richer interface than an array. You can now take advantage of operations supplied by the collections framework, such as bulk add:

```
anotherCollection.addAll(view);
```

The `addAll` method asks the view for an iterator, and that iterator enumerates all elements of the original array.

However, there are some operations that you cannot carry out. You cannot call the `add` or `remove` methods on the view. After all, it is not possible to change the size of the underlying array. For that reason, these methods are "optional". The `asList` view simply defines them to throw an `UnsupportedOperationException`.

Would it have been possible to define a separate interface type that omits the `add` and `remove` methods? The problem is that you soon have an inflation of interface types. Some views are read-only, other views (such as the one returned by the `asList` method) allow modifications, as long as the size of the collection stays the same. These are called "modifiable" in the API documentation. Having three versions of every interface type (read only, modifiable, and resizable) adds quite a bit of complexity. The drawback of the "optional" operations is that the compiler cannot check for errors.

NOTE The `Collections` utility class has convenient static methods that give unmodifiable views of collections, lists, sets, and so on. These views are useful if you want to give a client of a class the ability to view a collection but not to modify it. For example, the `Mailbox` class of Chapter 2 can give out an unmodifiable list of messages like this:

```
public class Mailbox
{
   public List<Message> getMessages()
   {
      return Collections.unmodifiableList(messages);
   }
   . . .
   private ArrayList<Message> messages;
}
```

The `Collections.unmodifiableList` method returns an object of a class that implements the `List` interface type. Its accessor methods are defined to retrieve the elements of the underlying list, and its mutator methods fail by throwing an `UnsupportedOperationException`.

8.4 A Graph Editor Framework

8.4.1 The Problem Domain

The problem domain for our graph editor framework is the interactive editing of graphs that consist of nodes and edges.

In this section we will introduce a simple application framework in which the programmer has to add a number of classes to complete an application. The problem domain that we address is the *interactive editing of graphs*. A *graph* is made up of *nodes* and *edges* that have certain shapes.

An application that is based on the graph editor framework defines specific behavior for the nodes and edges.

Consider a class diagram. The nodes are rectangles, and the edges are either arrows or lines with diamonds. A different example is an electronic circuit diagram, where nodes are transistors, diodes, resistors, and capacitors. Edges are simply wires. There are numerous other examples, such as chemical formulas, flowcharts, organization charts, and logic circuits.

Traditionally, a programmer who wants to implement, say, a class diagram editor, starts from scratch and creates an application that can edit only class diagrams. If the programmer is lucky, code for a similar program, say a flowchart editor, is available for inspection. However, it may well be difficult to separate the code that is common to all diagrams from the flowchart-specific tasks, and much of the code may need to be recreated for the class diagram editor.

The graph editor framework encapsulates those aspects that are common to all graph editing applications.

In contrast, the graph editor framework encapsulates those aspects that are common to all graph editors, in particular the user interface and the handling of commands and mouse events. The framework provides a way for specific diagram types to express their special demands that go beyond the common services.

8.4.2 The User Interface

Many of the tasks, such as selecting, moving, and connecting elements, are similar for all editors. Let's be specific and describe the user interface that our very primitive editor will have. The screen is divided into two parts, shown in Figure 6.

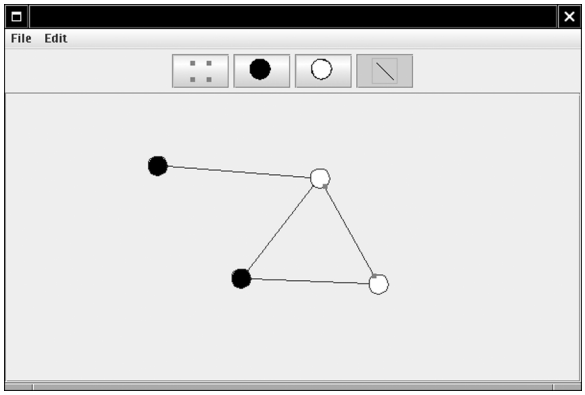
On the top is a *toolbar*, a collection of buttons. There is one button for each node type and one for each edge type. We will see later how a specific application supplies the icons for the buttons. The leftmost button is the *grabber* tool that is used for selecting nodes or edges. Exactly one of the tool buttons is active at any time.

There are also menu options for loading and saving a diagram, and for deleting selected nodes and edges.

In the middle is the diagram drawing area. The mouse is used for drawing. The program user can click the mouse on a node, an edge, or in empty space. The user can also use the mouse to connect nodes or to drag a node to a new position. The mouse actions depend on where the user clicks or drags, and what the currently selected tool is.

Figure 6

An Instance of the Graph Editor Framework



- When the current tool is a node, clicking on an empty space inserts a new node. Its type is that of the currently selected node in the toolbar.
- When the current tool is the grabber, clicking inside a node or on an edge selects that node or edge.
- When the current tool is the grabber, starting a drag operation inside an existing node moves the node as well as the edges that are connected to it.
- When the current tool is an edge, starting a drag operation inside an existing node and dragging the cursor inside another existing node inserts a new edge. Its type is that of the currently selected edge in the toolbar.

Of course, programs written with this framework are rather limited in their functionality. There is no provision to supply text labels for edges and nodes. There is no support for common commands such as cut/copy/paste or undo/redo. These features can be handled by an extended version of this framework. This example is kept as simple as possible to show the main concept: the separation of framework code and application-specific code.

8.4.3 Division of Responsibility

The framework programmer is responsible for generic mechanisms, whereas the application programmer needs to supply code that is specific to a particular application.

When designing a framework, one must divide responsibilities between the framework and specific instances of the framework. For example, it is clear that the code to draw a transistor-shaped node is not part of the general framework—only of the electronic circuit instance.

Drawing the shapes of nodes and edges is the responsibility of the application programmer. The same holds for *hit testing:* finding out whether a node or edge is hit by a mouse click. This can be tricky for odd shapes and cannot be the responsibility of the framework.

On the other hand, drawing the toolbar and managing the mouse clicks is the job of the framework. An application programmer need not be concerned with these aspects of a graph editor at all.

A concrete graph class must enumerate all node and edge types for the given graph.

This brings up a very interesting problem. The framework must have some idea of the node and edge types in the application so that each type of node or edge can be painted as an icon in a button. Just as importantly, it must be possible to add new nodes and edges of the types that are specified in the buttons. The application programmer must tell the framework about the node and edge types that can occur in a particular kind of graph.

There are several ways of achieving this task. For example, a concrete graph could produce a list of class names or `Class` objects to describe the node and edge classes.

However, we follow a slightly different approach. In our graph editor framework, a concrete graph must give the framework *prototype objects*. For example, the application instance in Figure 6 was created by defining a node class, `CircleNode`, an edge class, `LineEdge`, and a `SimpleGraph` class that specifies two node prototypes and an edge prototype.

```
public class SimpleGraph extends Graph
{
   public Node[] getNodePrototypes()
   {
      Node[] nodeTypes =
         {
            new CircleNode(Color.BLACK),
            new CircleNode(Color.WHITE)
         };
      return nodeTypes;
   }

   public Edge[] getEdgePrototypes()
   {
      Edge[] edgeTypes =
         {
            new LineEdge()
         };
      return edgeTypes;
   }
}
```

When the toolbar is constructed, it queries the graph for the node and edge prototypes and adds a button for each of them. The nodes and edges draw themselves in the `paintIcon` method of the button icon object.

When a user inserts a new node or edge, the object corresponding to the selected tool button is *cloned* and then added to the graph:

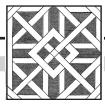
```
Node prototype = node of currently selected toolbar button;
Node newNode = (Node) prototype.clone();
Point2D mousePoint = current mouse position;
graph.add(newNode, mousePoint);
```

Why use prototype objects and not classes? Note that the two circle nodes are instances of the same class, one with a black fill color and the other with a white fill color. Thus, cloning prototype objects is a bit more economical than instantiating classes.

The PROTOTYPE pattern teaches how a system can instantiate classes that are not known when the system is built.

This mechanism is an example of the PROTOTYPE pattern. The prototype pattern gives a solution to the problem of dealing with an open-ended collection of node and edge types whose exact nature was not known when the framework code was designed.

PATTERN



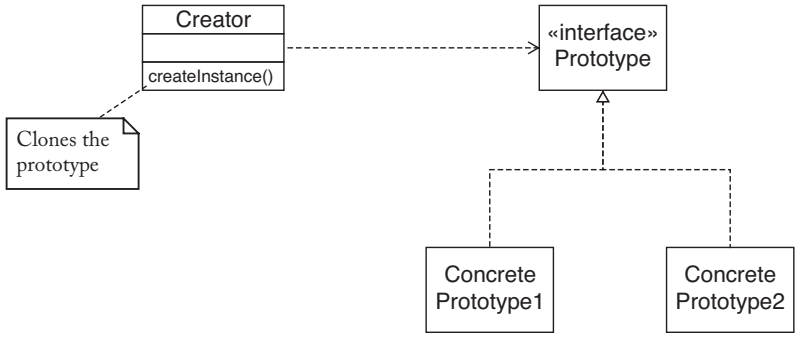
PROTOTYPE

Context

1. A system needs to create several kinds of objects whose classes are not known when the system is built.
2. You do not want to require a separate class for each kind of object.
3. You want to avoid a separate hierarchy of classes whose responsibility it is to create the objects.

Solution

1. Define a prototype interface that is common to all created objects.
2. Supply a prototype object for each kind of object that the system creates.
3. Clone the prototype object whenever a new object of the given kind is required.



For example, in the case of the node and edge types, we have

Name in Design Pattern	Actual Name
`Prototype`	`Node`
`ConcretePrototype1`	`CircleNode`
`Creator`	The `GraphPanel` class that handles the mouse operation for adding new nodes to the graph

8.4.4 Framework Classes

The Node and Edge interface types describe the behavior that is common to all nodes and edges.

The framework defines the interface types `Node` and `Edge`. The methods of these interface types define the shapes of the nodes and edges.

Both `Node` and `Edge` have a `draw` method that is used when painting the graph and a `contains` method that is used to test whether the mouse point falls on a node or an edge.

Both interface types have a `getBounds` method that returns the rectangle enclosing the node or edge shape. That method is needed to compute the total size of the graph as the union of the bounding rectangles of its parts. The scroll pane that holds the graph panel needs to know the graph size in order to draw the scroll bars.

The `Edge` interface type has methods that yield the nodes at the start and end of the edge.

The `getConnectionPoint` method in the `Node` interface type computes an optimal attachment point on the boundary of a node (see Figure 7). Since the node boundary may have an arbitrary shape, this computation must be carried out by each concrete node class.

The `getConnectionPoints` method of the `Edge` interface type yields the two end points of the edge. This method is needed to draw the "grabbers" that mark the currently selected edge.

The `clone` method is declared in both interface types because we require all implementing classes to supply a public implementation of the `clone` method. That method is required to clone prototypes when inserting new nodes or edges into the graph. (Recall that the `clone` method of the `Object` class has `protected` visibility.)

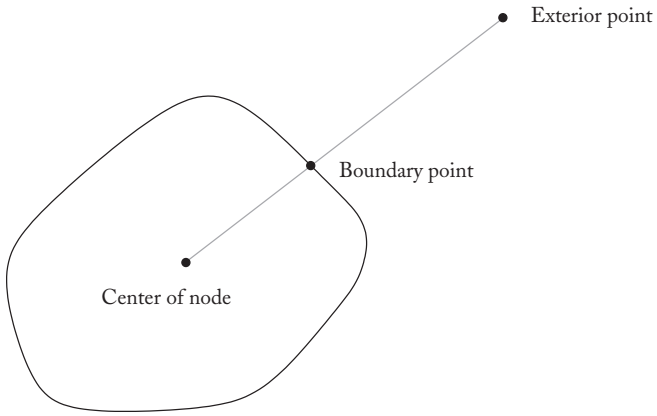


Figure 7

Node Connection Points

Ch8/graphed/Node.java

```
1  import java.awt.*;
2  import java.awt.geom.*;
3  import java.io.*;
4
5  /**
6     A node in a graph.
7  */
8  public interface Node extends Serializable, Cloneable
9  {
10    /**
11       Draws the node.
12       @param g2 the graphics context
13    */
14    void draw(Graphics2D g2);
15
16    /**
17       Translates the node by a given amount.
18       @param dx the amount to translate in the x-direction
19       @param dy the amount to translate in the y-direction
20    */
21    void translate(double dx, double dy);
22
23    /**
24       Tests whether the node contains a point.
25       @param aPoint the point to test
26       @return true if this node contains aPoint
27    */
28    boolean contains(Point2D aPoint);
29
30    /**
31       Gets the best connection point to connect this node
32       with another node. This should be a point on the boundary
33       of the shape of this node.
34       @param aPoint an exterior point that is to be joined
35       with this node
36       @return the recommended connection point
37    */
38    Point2D getConnectionPoint(Point2D aPoint);
39
40    /**
41       Gets the bounding rectangle of the shape of this node.
42       @return the bounding rectangle
43    */
44    Rectangle2D getBounds();
45
46    Object clone();
47 }
```

Ch8/graphed/Edge.java

```
1  import java.awt.*;
2  import java.awt.geom.*;
3  import java.io.*;
4
```

```
/**
   An edge in a graph.
*/
public interface Edge extends Serializable, Cloneable
{
   /**
      Draws the edge.
      @param g2 the graphics context
   */
   void draw(Graphics2D g2);

   /**
      Tests whether the edge contains a point.
      @param aPoint the point to test
      @return true if this edge contains aPoint
   */
   boolean contains(Point2D aPoint);

   /**
      Connects this edge to two nodes.
      @param aStart the starting node
      @param anEnd the ending node
   */
   void connect(Node aStart, Node anEnd);

   /**
      Gets the starting node.
      @return the starting node
   */
   Node getStart();

   /**
      Gets the ending node.
      @return the ending node
   */
   Node getEnd();

   /**
      Gets the points at which this edge is connected to
      its nodes.
      @return a line joining the two connection points
   */
   Line2D getConnectionPoints();

   /**
      Gets the smallest rectangle that bounds this edge.
      The bounding rectangle contains all labels.
      @return the bounding rectangle
   */
   Rectangle2D getBounds(Graphics2D g2);

   Object clone();
}
```

The programmer using this framework must define specific node and edge classes that realize these interface types:

```
class Transistor implements Node { . . . }
class Wire implements Edge { . . . }
```

For the convenience of the programmer, the framework also supplies an abstract class `AbstractEdge` that provides reasonable implementations of some, but not all, of the methods in the `Edge` interface type. Whenever these default implementations are appropriate, a programmer can extend that class rather than having to implement all methods of the interface type. There is no corresponding `AbstractNode` class since all of the methods of the `Node` interface type require knowledge of the node shape.

Ch8/graphed/AbstractEdge.java

```
1  import java.awt.*;
2  import java.awt.geom.*;
3
4  /**
5     A class that supplies convenience implementations for
6     a number of methods in the Edge interface type.
7  */
8  public abstract class AbstractEdge implements Edge
9  {
10    public Object clone()
11    {
12       try
13       {
14          return super.clone();
15       }
16       catch (CloneNotSupportedException exception)
17       {
18          return null;
19       }
20    }
21
22    public void connect(Node s, Node e)
23    {
24       start = s;
25       end = e;
26    }
27
28    public Node getStart()
29    {
30       return start;
31    }
32
33    public Node getEnd()
34    {
35       return end;
36    }
37
```

```
38    public Rectangle2D getBounds(Graphics2D g2)
39    {
40       Line2D conn = getConnectionPoints();
41       Rectangle2D r = new Rectangle2D.Double();
42       r.setFrameFromDiagonal(conn.getX1(), conn.getY1(),
43             conn.getX2(), conn.getY2());
44       return r;
45    }
46
47    public Line2D getConnectionPoints()
48    {
49       Rectangle2D startBounds = start.getBounds();
50       Rectangle2D endBounds = end.getBounds();
51       Point2D startCenter = new Point2D.Double(
52             startBounds.getCenterX(), startBounds.getCenterY());
53       Point2D endCenter = new Point2D.Double(
54             endBounds.getCenterX(), endBounds.getCenterY());
55       return new Line2D.Double(
56             start.getConnectionPoint(endCenter),
57             end.getConnectionPoint(startCenter));
58    }
59
60    private Node start;
61    private Node end;
62 }
```

The Graph class supplies methods for adding, finding, and removing nodes and edges.

The Graph class collects the nodes and edges. It has methods for adding, removing, finding, and drawing nodes and edges. Note that this class supplies quite a bit of useful functionality. This is, of course, characteristic of frameworks. In order to supply a significant value to application programmers, the framework classes must be able to supply a substantial amount of work.

Nevertheless, the Graph class is abstract. Subclasses of Graph must override the abstract methods

```
public abstract Node[] getNodePrototypes()
public abstract Edge[] getEdgePrototypes()
```

These methods are called when a graph is added to a frame. They populate the toolbar with the tools that are necessary to edit the graph. For example, the getNodePrototypes method of the SimpleGraph class specifies two circle node prototypes.

Ch8/graphed/Graph.java

```
1  import java.awt.*;
2  import java.awt.geom.*;
3  import java.io.*;
4  import java.util.*;
5  import java.util.List;
6
7  /**
8     A graph consisting of selectable nodes and edges.
```

```
 9 */
10 public abstract class Graph implements Serializable
11 {
12    /**
13       Constructs a graph with no nodes or edges.
14    */
15    public Graph()
16    {
17       nodes = new ArrayList<Node>();
18       edges = new ArrayList<Edge>();
19    }
20
21    /**
22       Adds an edge to the graph that joins the nodes containing
23       the given points. If the points aren't both inside nodes,
24       then no edge is added.
25       @param e the edge to add
26       @param p1 a point in the starting node
27       @param p2 a point in the ending node
28    */
29    public boolean connect(Edge e, Point2D p1, Point2D p2)
30    {
31       Node n1 = findNode(p1);
32       Node n2 = findNode(p2);
33       if (n1 != null && n2 != null)
34       {
35          e.connect(n1, n2);
36          edges.add(e);
37          return true;
38       }
39       return false;
40    }
41
42    /**
43       Adds a node to the graph so that the top left corner of
44       the bounding rectangle is at the given point.
45       @param n the node to add
46       @param p the desired location
47    */
48    public boolean add(Node n, Point2D p)
49    {
50       Rectangle2D bounds = n.getBounds();
51       n.translate(p.getX() - bounds.getX(),
52             p.getY() - bounds.getY());
53       nodes.add(n);
54       return true;
55    }
56
57    /**
58       Finds a node containing the given point.
59       @param p a point
60       @return a node containing p or null if no nodes contain p
61    */
62    public Node findNode(Point2D p)
63    {
```

```
64          for (int i = nodes.size() - 1; i >= 0; i--)
65          {
66             Node n = nodes.get(i);
67             if (n.contains(p)) return n;
68          }
69          return null;
70       }
71
72       /**
73          Finds an edge containing the given point.
74          @param p a point
75          @return an edge containing p or null if no edges contain p
76       */
77       public Edge findEdge(Point2D p)
78       {
79          for (int i = edges.size() - 1; i >= 0; i--)
80          {
81             Edge e = edges.get(i);
82             if (e.contains(p)) return e;
83          }
84          return null;
85       }
86
87       /**
88          Draws the graph.
89          @param g2 the graphics context
90       */
91       public void draw(Graphics2D g2)
92       {
93          for (Node n : nodes)
94             n.draw(g2);
95
96          for (Edge e : edges)
97             e.draw(g2);
98       }
99
100      /**
101         Removes a node and all edges that start or end with that node.
102         @param n the node to remove
103      */
104      public void removeNode(Node n)
105      {
106         for (int i = edges.size() - 1; i >= 0; i--)
107         {
108            Edge e = edges.get(i);
109            if (e.getStart() == n || e.getEnd() == n)
110               edges.remove(e);
111         }
112         nodes.remove(n);
113      }
114
115      /**
116         Removes an edge from the graph.
117         @param e the edge to remove
118      */
119      public void removeEdge(Edge e)
```

```
120     {
121        edges.remove(e);
122     }
123
124     /**
125        Gets the smallest rectangle enclosing the graph.
126        @param g2 the graphics context
127        @return the bounding rectangle
128     */
129     public Rectangle2D getBounds(Graphics2D g2)
130     {
131        Rectangle2D r = null;
132        for (Node n : nodes)
133        {
134           Rectangle2D b = n.getBounds();
135           if (r == null) r = b;
136           else r.add(b);
137        }
138        for (Edge e : edges)
139           r.add(e.getBounds(g2));
140        return r == null ? new Rectangle2D.Double() : r;
141     }
142
143     /**
144        Gets the node types of a particular graph type.
145        @return an array of node prototypes
146     */
147     public abstract Node[] getNodePrototypes();
148
149     /**
150        Gets the edge types of a particular graph type.
151        @return an array of edge prototypes
152     */
153     public abstract Edge[] getEdgePrototypes();
154
155     /**
156        Gets the nodes of this graph.
157        @return an unmodifiable list of the nodes
158     */
159     public List<Node> getNodes()
160     {
161        return Collections.unmodifiableList(nodes);
162     }
163
164     /**
165        Gets the edges of this graph.
166        @return an unmodifiable list of the edges
167     */
168     public List<Edge> getEdges()
169     {
170        return Collections.unmodifiableList(edges);
171     }
172
173     private ArrayList<Node> nodes;
174     private ArrayList<Edge> edges;
175  }
```

The GraphFrame, ToolBar, and GraphPanel framework classes are responsible for the user interface. Application programmers need not subclass these classes.

The graph editor uses the following classes for editing the graph:

- `GraphFrame`: a frame that manages the toolbar, the menu bar, and the graph panel.
- `ToolBar`: a panel that holds toggle buttons for the node and edge icons.
- `GraphPanel`: a panel that shows the graph and handles the mouse clicks and drags for the editing commands.

We do not list these classes here. The implementations are straightforward but a bit long. The graph frame attaches the toolbar and graph panel, sets up the menu, and loads and saves graphs using object serialization, as discussed in Chapter 7. The toolbar sets up a row of buttons with icon objects that paint the nodes and edges, and which are scaled down to fit inside the buttons. The mouse handling of the graph panel is similar to that of the scene editor in Chapter 6.

Interestingly enough, the `Node` and `Edge` interface types are rich enough that the framework classes do not need to know anything about particular node and edge shapes. The mechanics of mouse movement, rubber banding, and screen update are completely solved at this level and are of no concern to the programmer using the framework. Because all drawing and mouse operations are taken care of in the framework classes, the programmer building a graphical editor on top of the framework can simply focus on implementing the node and edge types.

8.4.5 Turning the Framework into an Application

The classes for the simple graph editor are summarized in Figure 8. The top four classes are application-specific. All other classes belong to the framework.

Let's summarize the responsibilities of the programmer creating a specific diagram editor:

- For each node and edge type, define a class that implements the `Node` or `Edge` interface type and supply all required methods, such as drawing and containment testing. For convenience, you may want to subclass the `AbstractEdge` class.
- Define a subclass of the `Graph` class whose `getNodePrototypes` and `getEdgePrototypes` methods supply prototype objects for nodes and edges.
- Supply a class with a `main` method such as the `SimpleGraphEditor` class below.

To build a graph editor application, subclass the Graph class and provide classes that implement the Node and Edge interface types.

Note that the programmer who turns the framework into an application supplies only application-specific classes and does not implement the user interface or control flow. This is characteristic of using a framework.

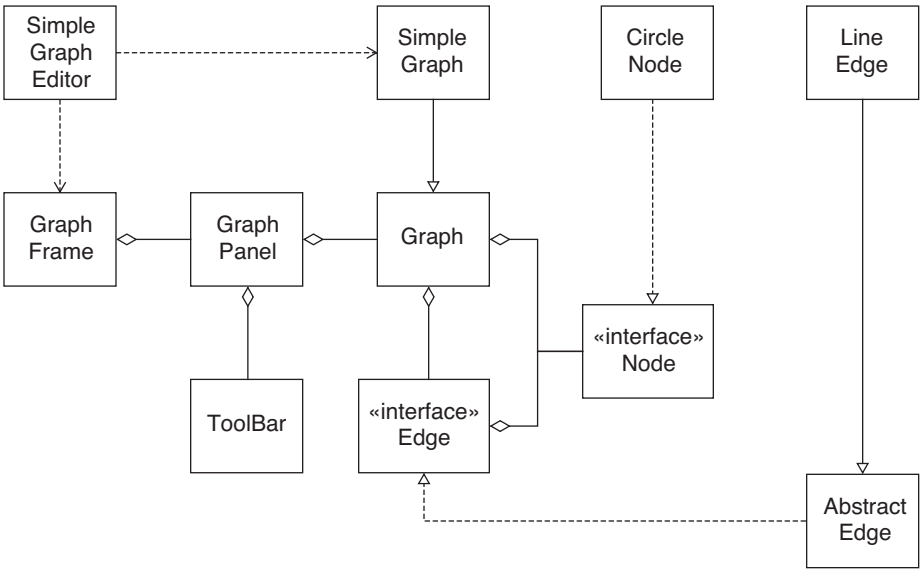


Figure 8

Application and Framework Classes

Ch8/graphed/SimpleGraph.java

```
1  import java.awt.*;
2  import java.util.*;
3
4  /**
5     A simple graph with round nodes and straight edges.
6  */
7  public class SimpleGraph extends Graph
8  {
9     public Node[] getNodePrototypes()
10    {
11       Node[] nodeTypes =
12          {
13             new CircleNode(Color.BLACK),
14             new CircleNode(Color.WHITE)
15          };
16       return nodeTypes;
17    }
18
19    public Edge[] getEdgePrototypes()
20    {
21       Edge[] edgeTypes =
22          {
23             new LineEdge()
24          };
25       return edgeTypes;
26    }
27 }
```

Ch8/graphed/SimpleGraphEditor.java

```
1  import javax.swing.*;
2
3  /**
4     A program for editing UML diagrams.
5  */
6  public class SimpleGraphEditor
7  {
8     public static void main(String[] args)
9     {
10       JFrame frame = new GraphFrame(new SimpleGraph());
11       frame.setVisible(true);
12    }
13 }
```

Ch8/graphed/CircleNode.java

```
1  import java.awt.*;
2  import java.awt.geom.*;
3
4  /**
5     A circular node that is filled with a color.
6  */
7  public class CircleNode implements Node
8  {
9     /**
10       Construct a circle node with a given size and color.
11       @param aColor the fill color
12    */
13    public CircleNode(Color aColor)
14    {
15       size = DEFAULT_SIZE;
16       x = 0;
17       y = 0;
18       color = aColor;
19    }
20
21    public Object clone()
22    {
23       try
24       {
25          return super.clone();
26       }
27       catch (CloneNotSupportedException exception)
28       {
29          return null;
30       }
31    }
32
33    public void draw(Graphics2D g2)
34    {
35       Ellipse2D circle = new Ellipse2D.Double(
36             x, y, size, size);
37       Color oldColor = g2.getColor();
38       g2.setColor(color);
```

```
39          g2.fill(circle);
40          g2.setColor(oldColor);
41          g2.draw(circle);
42       }
43
44       public void translate(double dx, double dy)
45       {
46          x += dx;
47          y += dy;
48       }
49
50       public boolean contains(Point2D p)
51       {
52          Ellipse2D circle = new Ellipse2D.Double(
53                x, y, size, size);
54          return circle.contains(p);
55       }
56
57       public Rectangle2D getBounds()
58       {
59          return new Rectangle2D.Double(
60                x, y, size, size);
61       }
62
63       public Point2D getConnectionPoint(Point2D other)
64       {
65          double centerX = x + size / 2;
66          double centerY = y + size / 2;
67          double dx = other.getX() - centerX;
68          double dy = other.getY() - centerY;
69          double distance = Math.sqrt(dx * dx + dy * dy);
70          if (distance == 0) return other;
71          else return new Point2D.Double(
72                centerX + dx * (size / 2) / distance,
73                centerY + dy * (size / 2) / distance);
74       }
75
76       private double x;
77       private double y;
78       private double size;
79       private Color color;
80       private static final int DEFAULT_SIZE = 20;
81    }
```

Ch8/graphed/LineEdge.java

```
1  import java.awt.*;
2  import java.awt.geom.*;
3
4  /**
5     An edge that is shaped like a straight line.
6  */
7  public class LineEdge extends AbstractEdge
8  {
```

```
 9     public void draw(Graphics2D g2)
10     {
11        g2.draw(getConnectionPoints());
12     }
13
14     public boolean contains(Point2D aPoint)
15     {
16        final double MAX_DIST = 2;
17        return getConnectionPoints().ptSegDist(aPoint)
18              < MAX_DIST;
19     }
20  }
```

8.4.6 Generic Framework Code

The generic framework code does not need to know about specific node and edge types.

In the last section you saw how to customize the framework to a specific editor application. In this section we will investigate how the framework code is able to function without knowing anything about the types of nodes and edges.

The framework code is too long to analyze here in its entirety, and some technical details, particularly of the mouse tracking, are not terribly interesting. Let's consider two operations: adding a new node and adding a new edge.

First let's look at adding a new node. When the mouse is clicked outside an existing node, then a new node of the current type is added. This is where the `clone` operation comes in. The `getSelectedTool` method of the `ToolBar` class returns an object of the desired node type. Of course, you cannot simply insert that object into the diagram. If you did, all nodes of the same type would end up in identical positions. Instead you invoke `clone` and add the cloned node to the graph. The `mousePressed` method of the mouse listener in the `GraphPanel` class carries out these actions.

```
public void mousePressed(MouseEvent event)
{
   Point2D mousePoint = event.getPoint();
   Object tool = toolBar.getSelectedTool();
   . . .
   if (tool instanceof Node)
   {
      Node prototype = (Node) tool;
      Node newNode = (Node) prototype.clone();
      graph.add(newNode, mousePoint);
   }
   . . .
   repaint();
}
```

Figure 9 shows the sequence diagram. Note how the code is completely independent of the actual node type in a particular application.

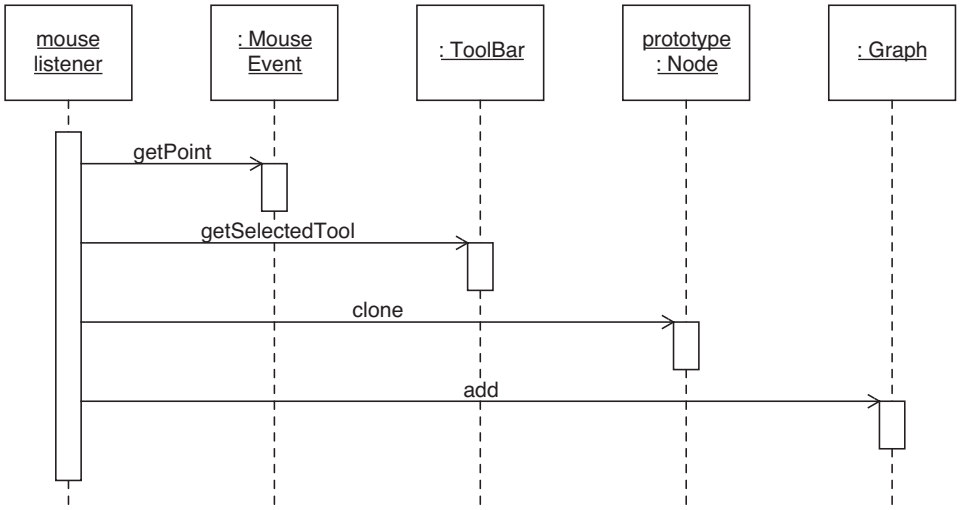


Figure 9

Inserting a New Node

Next, consider a more involved action, adding a new edge. When the mouse is clicked, we must first determine whether the click is inside an existing node. This operation is carried out in the `findNode` method of the `Graph` class, by calling the `contains` method of the `Node` interface:

```
public Node findNode(Point2D p)
{
   for (Node n : nodes)
      if (n.contains(p)) return n;
   return null;
}
```

If the mouse is clicked inside an existing node and the current tool is an edge, we remember the mouse position in the `rubberBandStart` field of the `GraphPanel` class.

```
public void mousePressed(MouseEvent event)
{
   . . .
   Node n = graph.findNode(mousePoint);
   if (tool instanceof Edge)
   {
      if (n != null) rubberBandStart = mousePoint;
   }
   . . .
}
```

In the `mouseDragged` method, there are two possibilities. If the current tool is not an edge, then the purpose of the dragging is to move the selected node elsewhere. We don't

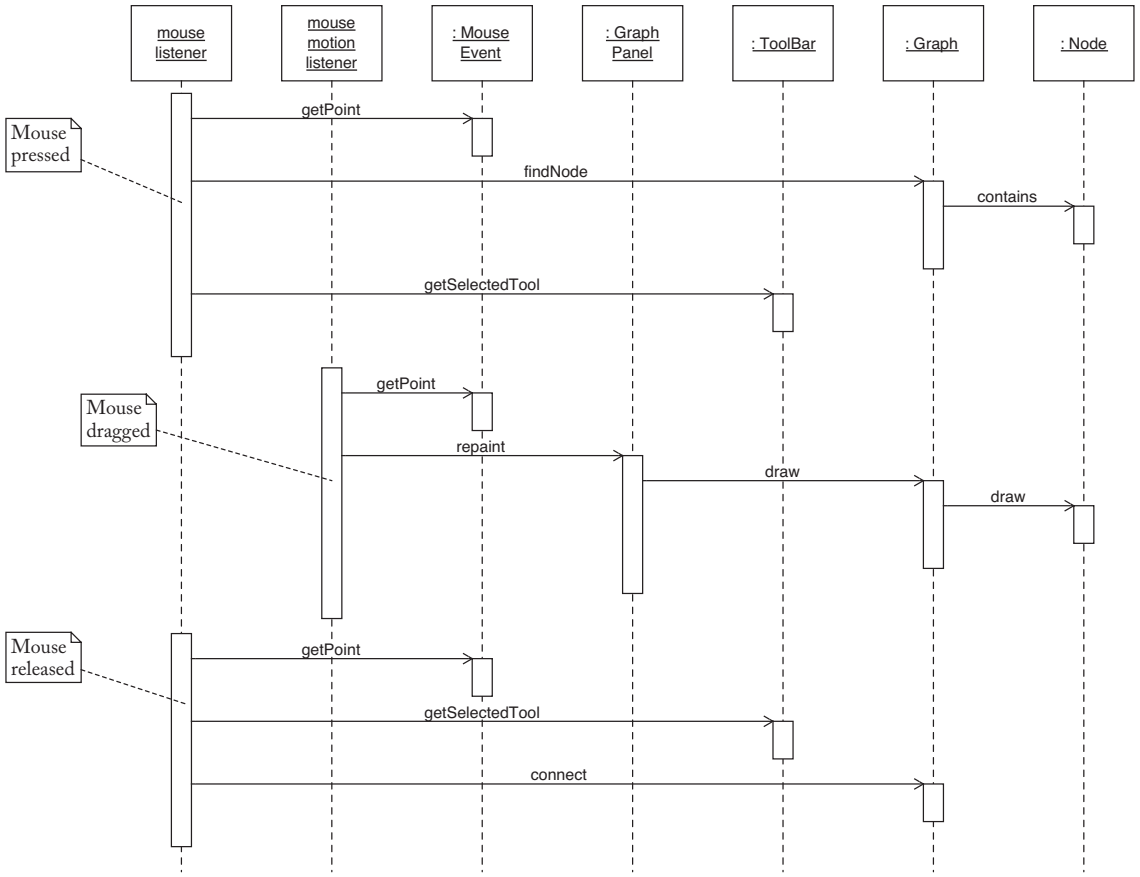


Figure 10

Inserting a New Edge

care about that case right now. However, if we are currently inserting an edge, then we want to draw a "rubber band", a line that follows the mouse pointer.

```
public void mouseDragged(MouseEvent event)
{
   Point2D mousePoint = event.getPoint();
   . . .
   lastMousePoint = mousePoint;
   repaint();
}
```

The `repaint` method invokes the `paintComponent` method of the `GraphPanel`. It draws the graph and, if `rubberBandStart` is not `null`, the rubber banded line.

```
public void paintComponent(Graphics g)
{
```

```
      Graphics2D g2 = (Graphics2D) g;
      graph.draw(g2);
      if (rubberBandStart != null)
         g2.draw(new Line2D.Double(rubberBandStart, lastMousePoint));
      . . .
   }
```

When the mouse button goes up, we are ready to add the edge.

```
   public void mouseReleased(MouseEvent event)
   {
      Object tool = toolBar.getSelectedTool();
      if (rubberBandStart != null)
      {
         Point2D mousePoint = event.getPoint();
         Edge prototype = (Edge) tool;
         Edge newEdge = (Edge) prototype.clone();
         graph.connect(newEdge, rubberBandStart, mousePoint);
         rubberBandStart = null;
         repaint();
      }
   }
```

Figure 10 shows the sequence diagram.

These scenarios are representative of the ability of the framework code to operate without an exact knowledge of the node and edge types.

8.5 Enhancing the Graph Editor Framework

8.5.1 Editing Node and Edge Properties

In this section, we will discuss an important enhancement of the graph editor framework: the ability to edit properties of nodes and edges. We add a menu option Edit → Properties that pops up a dialog box to edit the properties of the selected node or edge (see Figure 11).

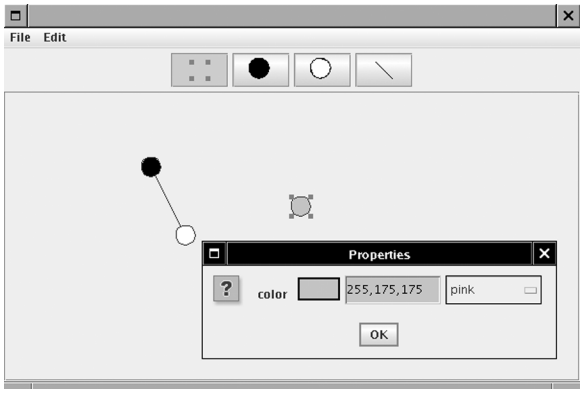


Figure 11

Editing a Node Property

Clearly, such a facility is necessary to enable users to select colors, line styles, text labels, and so on. The challenge for the framework designer is to find a mechanism that allows arbitrary node and edge classes to expose their properties, and then to provide a generic user interface for editing them.

To enable a graph editor application to edit the properties of nodes or edges, an application programmer simply implements them as JavaBeans properties. The graph editor framework contains the code for editing the properties.

Fortunately, this problem has been solved elsewhere. Recall from Chapter 7 that GUI builders are able to edit arbitrary properties of JavaBeans components. We will therefore require the implementors of nodes and edges to expose editable properties using the JavaBeans convention: with `get` and `set` methods. To edit the properties, we supply a property sheet dialog box that is similar to the property editor in a GUI builder.

For example, the `CircleNode` class can expose a `Color` property simply by providing two methods

```
public void setColor(Color newValue)
public Color getColor()
```

No further work is necessary. The graph editor can now edit node colors.

Let's consider a more complex change: to support both solid and dotted lines. We will define an enumerated type `LineStyle` with two instances:

```
LineStyle.SOLID
LineStyle.DOTTED
```

(See Chapter 7 for a discussion of the implementation of enumerated types in Java.)

The `LineStyle` enumeration has a convenience method

```
Stroke getStroke()
```

That method yields a solid or dotted stroke object. The `LineEdge` method uses that object in its `draw` method:

```
public void draw(Graphics2D g2)
{
   Stroke oldStroke = g2.getStroke();
   g2.setStroke(lineStyle.getStroke());
   g2.draw(getConnectionPoints());
   g2.setStroke(oldStroke);
}
```

The effect is either a solid or dotted line that joins the connection points.

Of course, we need to add getters and setters for the line style to the `LineEdge` class.

Altogether, the following changes are required to add colored nodes and dotted lines to the simple graph editor:

- Add `setColor` and `getColor` methods to `CircleNode`.
- Supply a `LineStyle` enumeration.
- Enhance the `LineEdge` class to draw both solid and dotted lines, and add `getLineStyle` and `setLineStyle` methods.

It is a simple matter to support additional graph properties, such as line shapes, arrow shapes, text labels, and so on.

8.5.2 Another Graph Editor Instance: A Simple UML Class Editor

Figure 12 shows a simple UML class diagram editor that has been built on top of the graph editor framework.

The editor is essentially the same as the Violet UML editor. However, it supports only class diagrams, and it lacks some convenience features such as keyboard shortcuts, image export, and snap-to-grid.

To build a simple UML editor, add class node and class relationship edge classes to the graph editor framework.

Of course, the node and edge classes of this editor are more complex. They format and draw text, compute edges with multiple segments, and add arrow tips and diamonds. It is instructive to enumerate the classes that carry out this new functionality. None of these classes are difficult to implement, although there is an undeniable tedium to some of the layout computations.

- The `RectangularNode` class describes a node that is shaped like a rectangle. It is the superclass of `ClassNode`.
- The `SegmentedLineEdge` class implements an edge that consists of multiple line segments. It is the superclass of `ClassRelationshipEdge`.
- `ArrowHead` and `BentStyle` classes are enumerations for arrow heads and edge shapes, similar to the `LineStyle` class of the preceding section.
- `MultiLineString` formats a string that may extend over multiple lines. A `ClassNode` uses multiline strings for the class name, the attributes, and the methods.
- Finally, the `ClassDiagramGraph` class adds the `ClassNode` and various edge prototypes to the toolbar.

The basic framework is not affected at all by these changes. The implementor of the UML editor need not be concerned about frames, toolbars, or event handling. Even the editing of properties is automatically provided because the framework supplies a dialog

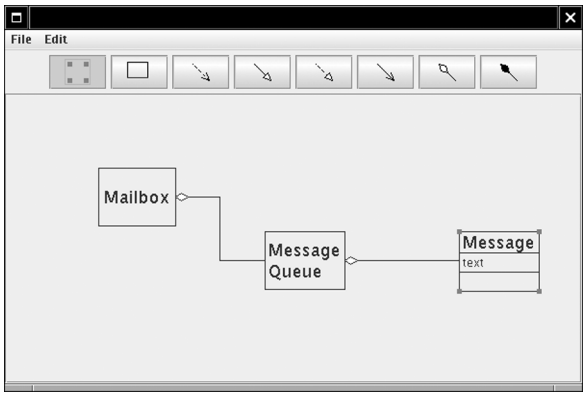


Figure 12

A Simple UML Class Diagram Editor

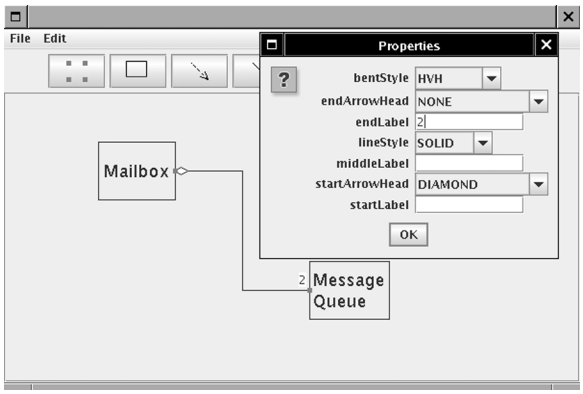


Figure 13

The Edge Property Editor

box that manipulates the JavaBeans properties (see Figure 13). Thus, the framework allows the implementor of any particular graph type to focus on the intricacies of the nodes and edges of just that graph.

8.5.3 Evolving the Framework

The Violet UML editor uses an enhanced version of the graph editor framework. The simple graph editor can take advantage of the enhancements with no changes in application code.

The Violet UML editor uses an enhanced version of the graph editor framework that adds a number of useful features such as graphics export, a grid for easier alignment, and simultaneous display of multiple graphs. The companion code for this book does not include the Violet code because some of it is rather lengthy. You can find the source code at `http://horstmann.com/violet`.

Remarkably, you can still integrate the simple graph editor with its circle nodes and line edges into the enhanced framework (see Figure 14).

This demonstrates another advantage of using a framework. By decoupling the framework and the application code, the application designers can take advantage of the framework evolution, without having to change the application-specific code.

8.5.4 A Note on Framework Design

In this chapter, you have learned how to put existing application frameworks to use. In order to use a framework, you have to understand the requirements that the designer of the framework set forth for application programmers. For example, to turn the graph editor framework into an application, you have to supply subclasses of `Graph`, `Node`, and `Edge`. Other frameworks have similar requirements.

Designing your own framework is a far bigger challenge than using a framework. You need to have a thorough understanding of the problem domain that the framework addresses. You need to design an architecture that enables application programmers to

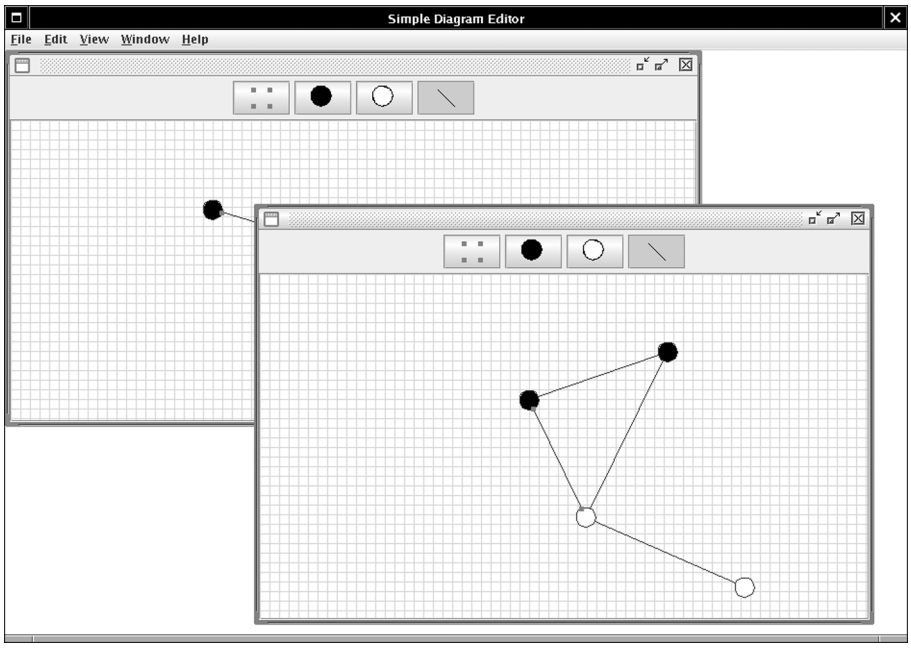


Figure 14

The Simple Graph Editor Takes Advantage of the Enhanced Framework

add application-specific code, without changing the framework code. The design of the framework should shield application programmers from internal mechanisms and allow them to focus on application-specific tasks. On the other hand, you need to provide "hooks" that allow application programmers to modify the generic framework behavior when applications require nonstandard mechanisms. It is notoriously difficult to anticipate the needs of application programmers. In fact, it is commonly said that a framework can only claim to have withstood the test of time if it is the basis of at least three different applications. Rules for the effective design of application frameworks are an area of active research at this time.

Exercises

Exercise 8.1. The `java.io` package contains pluggable streams, such as `PushbackInputStream` and `ZipInputStream`. Explain why the stream classes form a framework. Describe how a programmer can add new stream classes to the framework, and what benefits those classes automatically have.

Exercise 8.2. Search the Web for application frameworks until you have found frameworks for three distinct problem domains. Summarize your findings.

Exercise 8.3. Turn the scene editor of Chapter 6 into an applet.

Exercise 8.4. Write an applet that can display a bar chart. The applet should obtain the chart values from a set of `param` tags.

Exercise 8.5. Explain the phenomenon of "inversion of control", using the graph editor framework as an example.

Exercise 8.6. Re-implement the `BoundedQueue` class as a subtype of the `Queue` interface type in the standard library.

Exercise 8.7. Prove the following class invariant for the `BoundedQueue<E>` class:

- All values in the `elements` array belong to a subtype of `E`.

Why does this invariant show that the class implementation is safe, despite the compiler warnings? Why can't the compiler determine that the implementation is safe?

Exercise 8.8. Suppose the designers of the collections framework had decided to offer separate interface types for ordered collections (such as linked lists) and indexed collections (such as array lists). Explain the changes that must be made to the framework.

Exercise 8.9. Suppose the designers of the collections framework had, instead of allowing "unsupported operations", supported three kinds of data structures: read-only, modifiable, and resizable. Explain the changes that must be made to the framework. How do the basic interface types change? Which classes need to be added? Which methods need to be added to the `Arrays` and `Collections` classes?

Exercise 8.10. The `RandomAccess` interface type has no methods. The `Set` interface type adds no methods to its superinterface. What are the similarities and differences between the functionality that they are designed to provide?

Exercise 8.11. The standard C++ library defines a collections framework (known as STL) that is quite different from the Java framework. Explain the major differences.

Exercise 8.12. Contrast the algorithms available in the Java collections framework with those of the standard C++ library.

Exercise 8.13. Enhance the `SimpleGraphEditor` to support both circular and rectangular nodes.

Exercise 8.14. Enhance the `SimpleGraphEditor` to support lines with arrow tips.

Exercise 8.15. Enhance the `SimpleGraphEditor` to support text annotations of lines. *Hint:* Make a label property.

Exercise 8.16. Enhance the `SimpleGraphEditor` to support multiple arrow shapes: v-shaped arrow tips, triangles, and diamonds.

Exercise 8.17. Add cut/copy/paste operations to the graph editor framework.

Exercise 8.18. Design a sorting algorithm animation framework. An algorithm animation shows an algorithm in slow motion. For example, if you animate the merge sort algorithm, you can see how the algorithm sorts and merges intervals of increasing size. Your framework should allow a programmer to plug in various sorting algorithms.

Exercise 8.19. Design a framework for simulating the processing of customers at a bank or supermarket. Such a simulation is based on the notion of *events*. Each event has a time stamp. Events are placed in an event queue. Whenever one event has finished processing, the event with the earliest time stamp is removed from the event queue. That time stamp becomes the current system time. The event is processed, and the cycle repeats. There are different kinds of events. Arrival events cause customers to arrive at the bank. A stream of them needs to be generated to ensure the continued arrival of customers, with somewhat random times between arrivals. This is typically done by seeding the event queue with one arrival event, and having the processing method schedule the next arrival event. Whenever a teller is done processing a customer, the teller obtains the next waiting customer and schedules a "done processing" event, some random time away from the current time. In the framework, supply an abstract event class and the event processing mechanism. Then supply two applications that use the framework: a bank with a number of tellers and a single queue of waiting customers, and a supermarket with a number of cashiers and one queue per cashier.