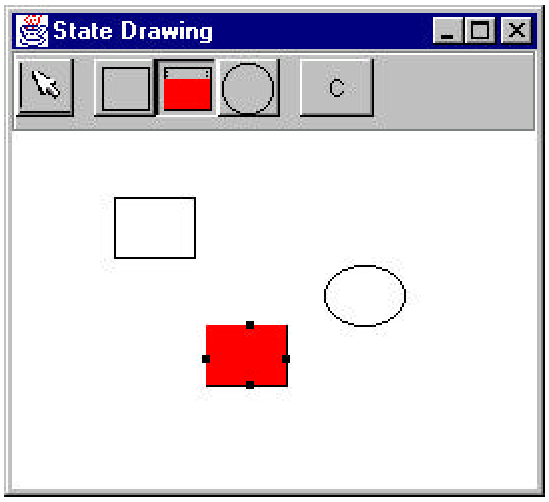
The State Pattern

The State pattern is used when you want to have an enclosing class switch between a number of related contained classes, and pass method calls on to the current contained class. *Design Patterns* suggests that the State pattern switches between internal classes in such a way that the enclosing object appears to change its class. In Java, at least, this is a bit of an exaggeration, but the actual purpose to which the classes are put can change significantly.

Many programmers have had the experience of creating a class which performs slightly different computations or displays different information based on the arguments passed into the class. This frequently leads to some sort of *switch* or *if-else* statements inside the class that determine which behavior to carry out. It is this inelegance that the State pattern seeks to replace.

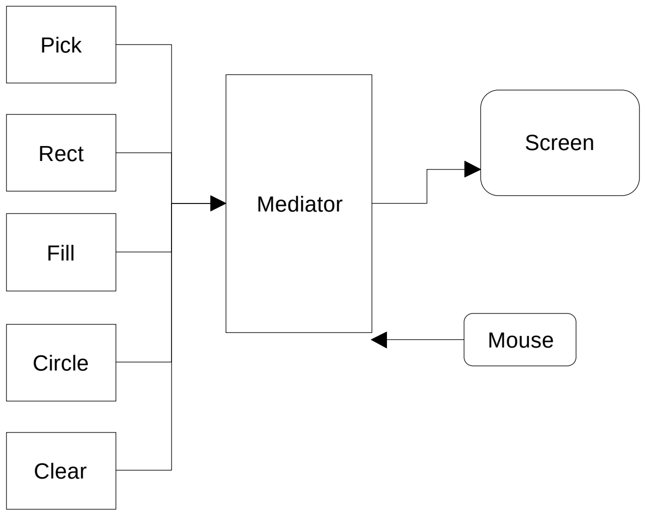
Sample Code

Let's consider the case of a drawing program similar to the one we developed for the Memento class. Our program will have toolbar buttons for Select, Rectangle, Fill, Circle and Clear.



Each one of the tool buttons does something rather different when it is selected and you click or drag your mouse across the screen. Thus, the *state* of the graphical editor affects the behavior the program should exhibit. This suggests some sort of design using the State pattern.

Initially we might design our program like this, with a Mediator managing the actions of 5 command buttons:



However, this initial design puts the entire burden of maintaining the state of the program on the Mediator, and we know that the main purpose of a Mediator is to coordinate activities between various controls, such as the buttons. Keeping the state of the buttons and the desired mouse activity inside the Mediator can make it unduly complicated as well as leading to a set of *if* or *switch* tests which make the program difficult to read and maintain.

Further, this set of large, monolithic conditional statements might have to be repeated for each action the Mediator interprets, such as mouseUp, mouseDrag, rightClick and so forth. This makes the program very hard to read and maintain.

Instead, let’s analyze the expected behavior for each of the buttons:

1. If the Pick button is selected, clicking inside a drawing element should cause it to be highlighted or appear with “handles.” If the mouse is dragged and a drawing element is already selected, the element should move on the screen.
2. If the Rect button is selected, clicking on the screen should cause a new rectangle drawing element to be created.

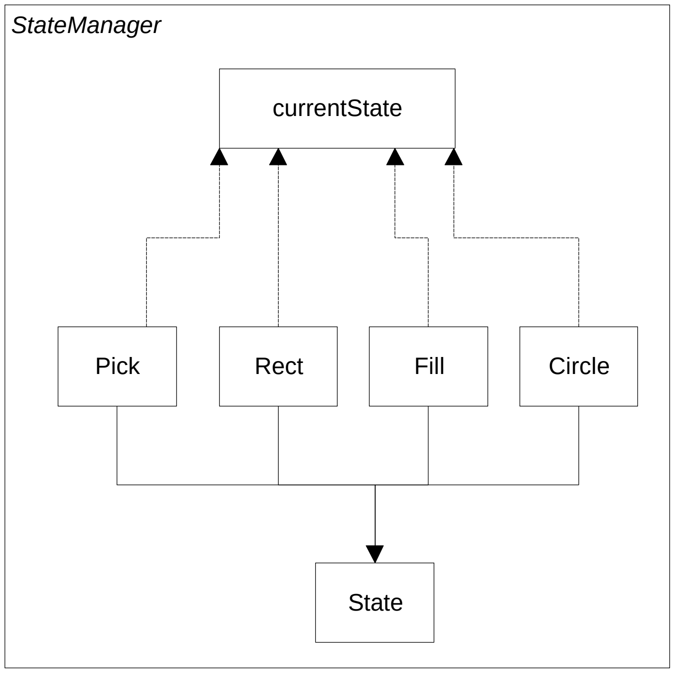
3. If the Fill button is selected and a drawing element is already selected, that element should be filled with the current color. If no drawing is selected, then clicking inside a drawing should fill it with the current color.
4. If the Circle button is selected, clicking on the screen should cause a new circle drawing element to be created.
5. If the Clear button is selected, all the drawing elements are removed.

There are some common threads among several of these actions we should explore. Four of them use the mouse click event to cause actions. One uses the mouse drag event to cause an action. Thus, we really want to create a system that can help us redirect these events based on which button is currently selected.

Let's consider creating a State object that handles mouse activities:

```
public class State {
public void mouseDown(int x, int y){}
public void mouseUp(int x, int y){}
public void mouseDrag(int x, int y){}
}
```

We'll include the mouseUp event in case we need it later. Since none of the cases we've described need all of these events, we'll give our base class empty methods rather than creating an abstract base class. Then we'll create 4 derived State classes for Pick, Rect, Circle and Fill and put instances of all of them inside a StateManager class which sets the current state and executes methods on that state object. In *Design Patterns,* this StateManager class is referred to as a *Context*. This object is illustrated below:



A typical State object simply overrides those event methods that it must handle specially. For example, this is the complete Rectangle state object:

```
public class RectState extends State
{
   private Mediator med;     //save the Mediator
   public RectState(Mediator md)     {
      med = md;
   }
   //------------------------------------
   //create a new Rectangle where mouse clicks
   public void mouseDown(int x, int y)    {
      med.addDrawing(new visRectangle(x, y));
   }
}
```

The RectState object simply tells the Mediator to add a rectangle drawing to the drawing list. Similarly, the Circle state object tells the Mediator to add a circle to the drawin list:

```
public class CircleState extends State
{
   private Mediator med;        //save Mediator
   public CircleState(Mediator md)    {
      med = md;
   }
   //-------------------------------
   //Draw circle where mouse clicks
   public void mouseDown(int x, int y)    {
     med.addDrawing(new visCircle(x, y));
   }
}
```

The only tricky button is the Fill button, because we have defined two actions for it.

1. If an object is already selected, fill it.
2. If the mouse is clicked inside an object, fill that one.

In order to carry out these tasks, we need to add the *select* method to our base State class. This method is called when each tool button is selected:

```
public class State
{
public void mouseDown(int x, int y){}
public void mouseUp(int x, int y){}
public void mouseDrag(int x, int y){}
public void select(Drawing d, Color c){}
}
```

The Drawing argument is either the currently selected Drawing or null if none is selcted, and the color is the current fill color. In this simple program, we have arbitrarily set the fill color to red. So our Fill state class becomes:

```
public class FillState extends State
{
   private Mediator med;   //save Mediator
   private Color color;    //save current color
   public FillState(Mediator md)   {
      med = md;
   }
//-------------------------------
   //Fill drawing if selected
   public void select(Drawing d, Color c)    {
      color = c;
       if(d!= null)
         {
         d.setFill(c);  //fill that drawing
         }
   }
    //-----------------------------------
   //Fill drawing if you click inside one
   public void mouseDown(int x, int y)    {
      Vector drawings = med.getDrawings();
      for(int i=0; i< drawings.size(); i++)
      {
       Drawing d = (Drawing)drawings.elementAt(i);
       if(d.contains(x, y))
          d.setFill(color); //fill drawing
      }
   }
}
```

Switching Between States

Now that we have defined how each state behaves when mouse events are sent to it, we need to discuss how the StateManager switches between states; we simply set the currentState variable to the state is indicated by the button that is selected.

```
import java.awt.*;

public class StateManager
{
 private State currentState;
 RectState rState;         //states are kept here
 ArrowState aState;
 CircleState cState;
 FillState fState;

 public StateManager(Mediator med)
 {
  rState = new RectState(med);   //create instances
  cState = new CircleState(med); //of each state
  aState = new ArrowState(med);
  fState = new FillState(med);
  currentState = aState;
 }
//These methods are called when the tool buttons
//are selected
 public void setRect()  { currentState = rState;  }
 public void setCircle(){ currentState = cState;  }
 public void setFill()  { currentState = fState;  }
 public void setArrow() { currentState = aState;  }
```

Note that in this version of the StateManager, we create an instance of each state during the constructor and copy the correct one into the state variable when the set methods are called. It would also be possible to use a Factory to create these states on demand. This might be advisable if there are a large number of states which each consume a fair number of resources.

The remainder of the state manager code simply calls the methods of whichever state object is current. This is the critical piece -- there is no conditional testing. Instead, the correct state is already in place and its methods are ready to be called.

```
 public void mouseDown(int x, int y)  {
    currentState.mouseDown(x, y);
 }
 public void mouseUp(int x, int y)  {
    currentState.mouseUp(x, y);
 }
 public void mouseDrag(int x, int y)  {
    currentState.mouseDrag(x, y);
 }
 public void select(Drawing d, Color c)  {
    currentState.select(d, c);
 }
}
```

How the Mediator Interacts with the State Manager

We mentioned that it is clearer to separate the state management from the Mediator's button and mouse event management. The Mediator is the critical class, however, since it tells the StateManager when the current program state changes. The beginning part of the Mediator illustrates how this state change takes place:

```
public Mediator() {
   startRect = false;
   dSelected = false;
   drawings = new Vector();
   undoList = new Vector();
   stMgr = new StateManager(this);
}
//-------------------------------------------
public void startRectangle() {
   stMgr.setRect();       //change to rectangle state
   arrowButton.setSelected(false);
   circButton.setSelected(false);
   fillButton.setSelected(false);
 }
//---------------------------------------------
public void startCircle() {
   stMgr.setCircle();     //change to circle state
   rectButton.setSelected(false);
   arrowButton.setSelected(false);
   fillButton.setSelected(false);
}
```

These *startXxx* methods are called from the Execute methods of each button as a Command object.

Consequences of the State Pattern

1. The State pattern localizes state-specific behavior in an individual class for each state, and puts all the behavior for that state in a single object.
2. It eliminates the necessity for a set of long, look-alike conditional statements scattered through the program's code.
3. It makes transition explicit. Rather than having a constant that specifies which state the program is in, and that may not always be checked correctly, this makes the change explicit by copying one of the states to the state variable.
4. State objects can be shared if they have no instance variables. Here only the Fill object has instance variables, and that color could easily be made an argument instead.
5. This approach generates a number of small class objects, but in the process, simplifies and clarifies the program.
6. In Java, all of the States must inherit from a common base class, and they must all have common methods, although some of those methods can be empty. In other languages, the states can be implemented by function pointers with much less type checking, and, of course, greater chance of error.

State Transitions

The transition between states can be specified internally or externally. In our example, the Mediator tells the StateManager when to switch between states. However, it is also possible that each state can decide automatically what each successor state will be. For example, when a rectangle or circle drawing object is created, the program could automatically switch back to the Arrow-object State.

Thought Questions

1. Rewrite the StateManager to use a Factory pattern to produce the states on demand.
2. While visual graphics programs provide obvious examples of State patterns, Java server programs can benefit by this approach. Outline a simple server which uses a state pattern.