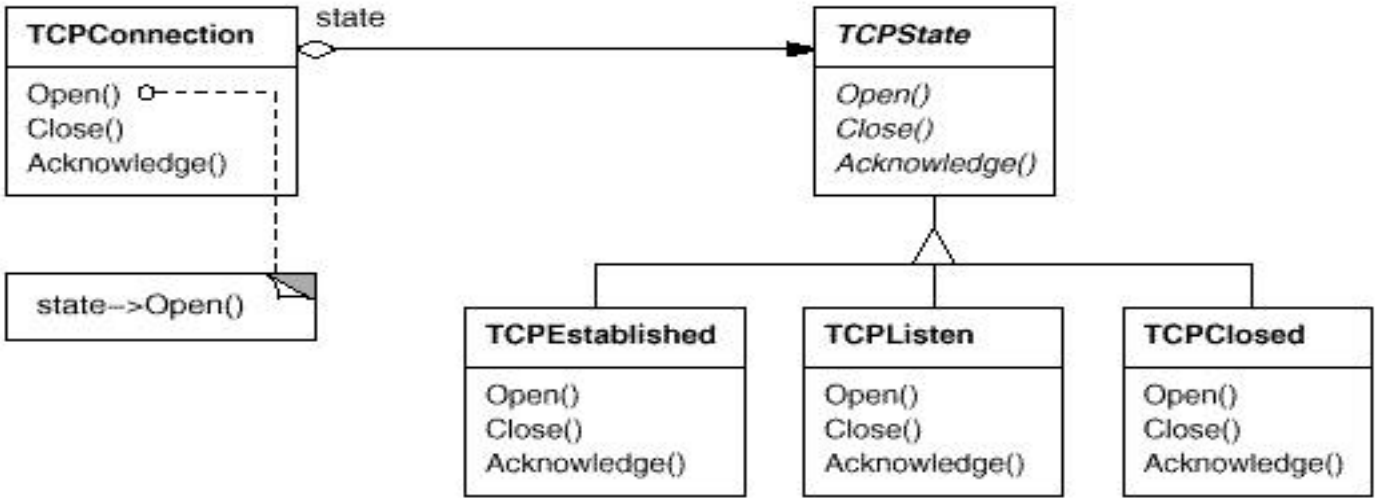
The
State and Strategy
Patterns

The State Pattern

- Intent
  - ⇨ Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- Motivation



The State Pattern

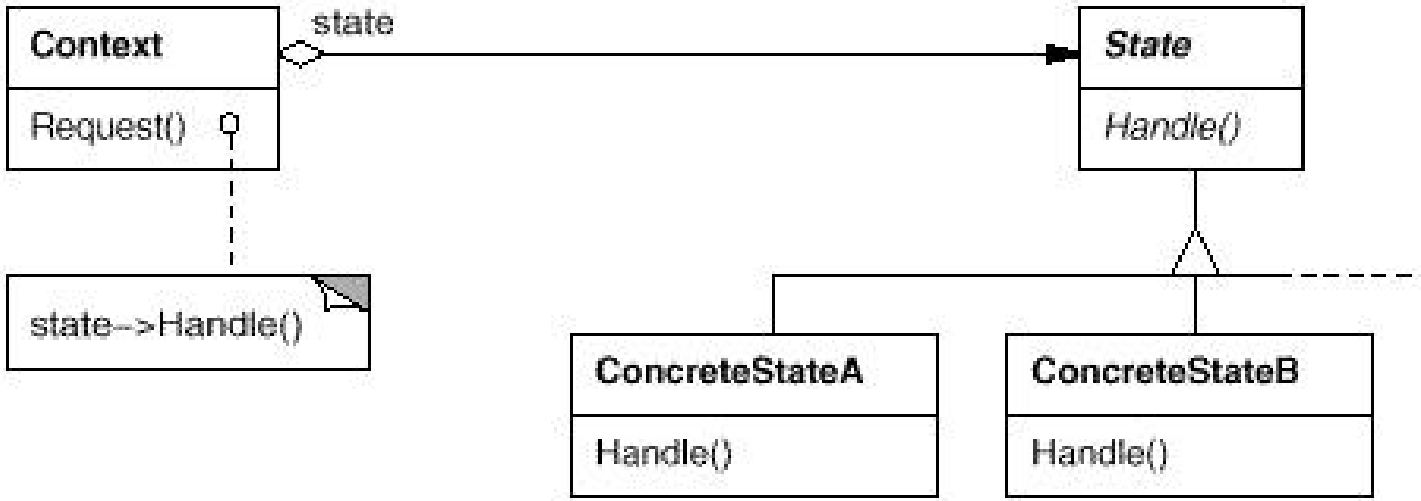
- Applicability

  Use the State pattern whenever:

  - ⇨ An object's behavior depends on its state, and it must change its behavior at run-time depending on that state
  - ⇨ Operations have large, multipart conditional statements that depend on the object's state. The State pattern puts each branch of the conditional in a separate class.

The State Pattern

- Structure

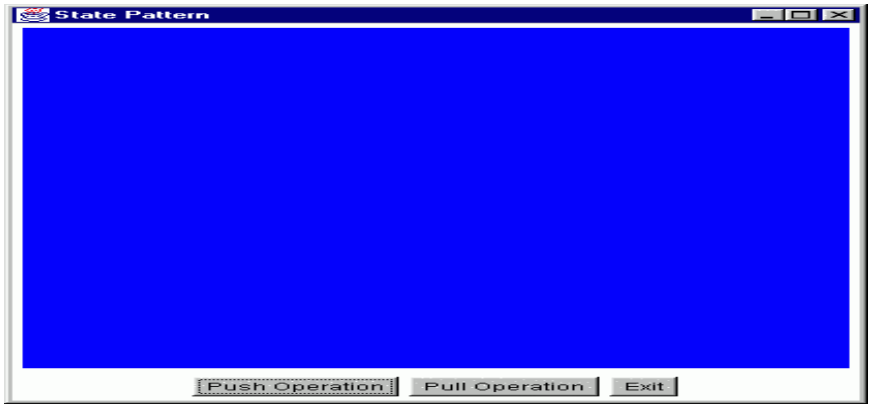


The State Pattern

- Consequences
  - Benefits
    - Puts all behavior associated with a state into one object
    - Allows state transition logic to be be incorporated into a state object rather than in a monolithic if or switch statement
    - Helps avoid inconsistent states since state changes occur using just the one state object and not several objects or attributes
  - Liabilities
    - Increased number of objects

State Pattern Example 1

- Consider a class that has two methods, push() and pull(), whose behavior changes depending on the state of the object
- To send the push and pull requests to the object, we'll use the following GUI with "Push" and "Pull" buttons:



- The state of the object will be indicated by the color of the canvas in the top part of the GUI
- The states are: black, red, blue and green

State Pattern Example 1 (Continued)

- First, let's do this without the State pattern:

```
/**
 * Class ContextNoSP has behavior dependent on its state.
 * The push() and pull() methods do different things
 *   depending on the state of the object.
 * This class does NOT use the State pattern.
 */
public class ContextNoSP  {

  // The state!
  private Color state = null;

  // Creates a new ContextNoSP with the specified state (color).
  public ContextNoSP(Color color) {state = color;}
  // Creates a new ContextNoSP with the default state
  public ContextNoSP() {this(Color.red);}
```

State Pattern Example 1 (Continued)

```
// Returns the state.
public Color getState() {return state;}

// Sets the state.
public void setState(Color state) {this.state = state;}

/**
 * The push() method performs different actions depending
 *   on the state of the object.  Actually, right now
 *   the only action is to make a state transition.
 */
public void push() {
  if (state == Color.red) state = Color.blue;
  else if (state == Color.green) state = Color.black;
  else if (state == Color.black) state = Color.red;
  else if (state == Color.blue) state = Color.green;
}
```

State Pattern Example 1 (Continued)

```
    /**
     * The pull() method performs different actions depending
     *   on the state of the object.  Actually, right now
     *   the only action is to make a state transition.
     */
    public void pull() {
      if (state == Color.red) state = Color.green;
      else if (state == Color.green) state = Color.blue;
      else if (state == Color.black) state = Color.green;
      else if (state == Color.blue) state = Color.red;
    }

}
```

State Pattern Example 1 (Continued)

- Here's part of the GUI test program:

```
/**
 * Test program for the ContextNoSP class
 * which does NOT use the State pattern.
 */
public class TestNoSP extends Frame
  implements ActionListener  {

  // GUI attributes.
  private Button pushButton = new Button("Push Operation");
  private Button pullButton = new Button("Pull Operation");
  private Button exitButton = new Button("Exit");
  private Canvas canvas = new Canvas();

  // The Context.
  private ContextNoSP context = null;
```

State Pattern Example 1 (Continued)

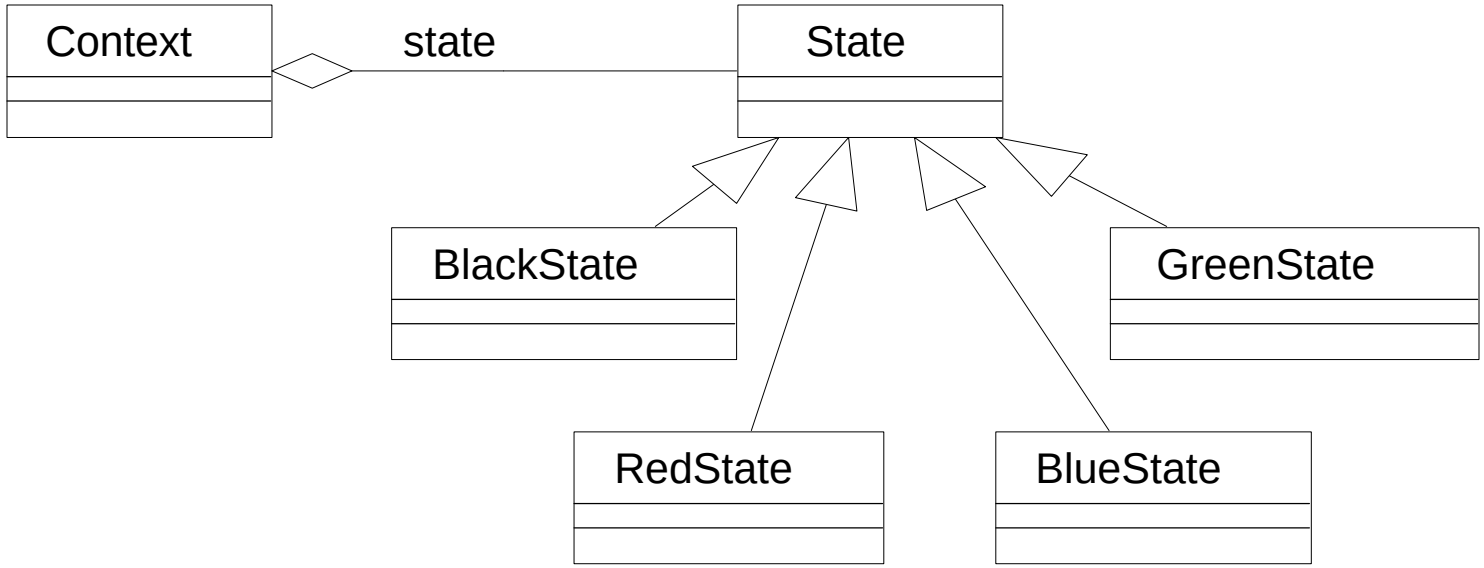
```
public TestNoSP() {
  super("No State Pattern");
  context = new ContextNoSP();
  setupWindow();
}

private void setupWindow() { // Setup GUI }

// Handle GUI actions.
public void actionPerformed(ActionEvent event) {
  Object src = event.getSource();
  if (src == pushButton) {
    context.push();
    canvas.setBackground(context.getState());
  }
```

State Pattern Example 1 (Continued)

```
        else if (src == pullButton) {
          context.pull();
          canvas.setBackground(context.getState());

        }
        else if (src == exitButton) {
          System.exit(0);
      }
    }

    // Main method.
    public static void main(String[] argv) {
      TestNoSP gui = new TestNoSP();
      gui.setVisible(true);
    }

}
```

State Pattern Example 1 (Continued)

- Now let's use the State pattern!
- Here's the class diagram:



State Pattern Example 1 (Continued)

- First, we'll define the abstract State class:

```
/**
 * Abstract class which defines the interface for the
 *   behavior of a particular state of the Context.
 */
public abstract class State {
  public abstract void handlePush(Context c);
  public abstract void handlePull(Context c);
  public abstract Color getColor();
}
```

- Next, we'll write concrete State classes for all the different states: RedState, BlackState, BlueState and GreenState

State Pattern Example 1 (Continued)

- For example, here's the BlackState class:

```
public class BlackState extends State {
  // Next state for the Black state:
  //   On a push(), go to "red"
  //   On a pull(), go to "green"

  public void handlePush(Context c) {
    c.setState(new RedState());
  }

  public void handlePull(Context c) {
    c.setState(new GreenState());
  }

  public Color getColor() {return (Color.black);}
}
```

State Pattern Example 1 (Continued)

- And, here's the new Context class that uses the State pattern and the State classes:

```
/**
 * Class Context has behavior dependent on its state.
 * This class uses the State pattern.
 * Now when we get a pull() or push() request, we
 *   delegate the behavior to our contained state object!
 */
public class Context {

  // The contained state.
  private State state = null;  // State attribute

  // Creates a new Context with the specified state.
  public Context(State state) {this.state = state;}
```

State Pattern Example 1 (Continued)

```
// Creates a new Context with the default state.
public Context() {this(new RedState());}

// Returns the state.
public State getState() {return state;}

// Sets the state.
public void setState(State state) {this.state = state;}
```

State Pattern Example 1 (Continued)

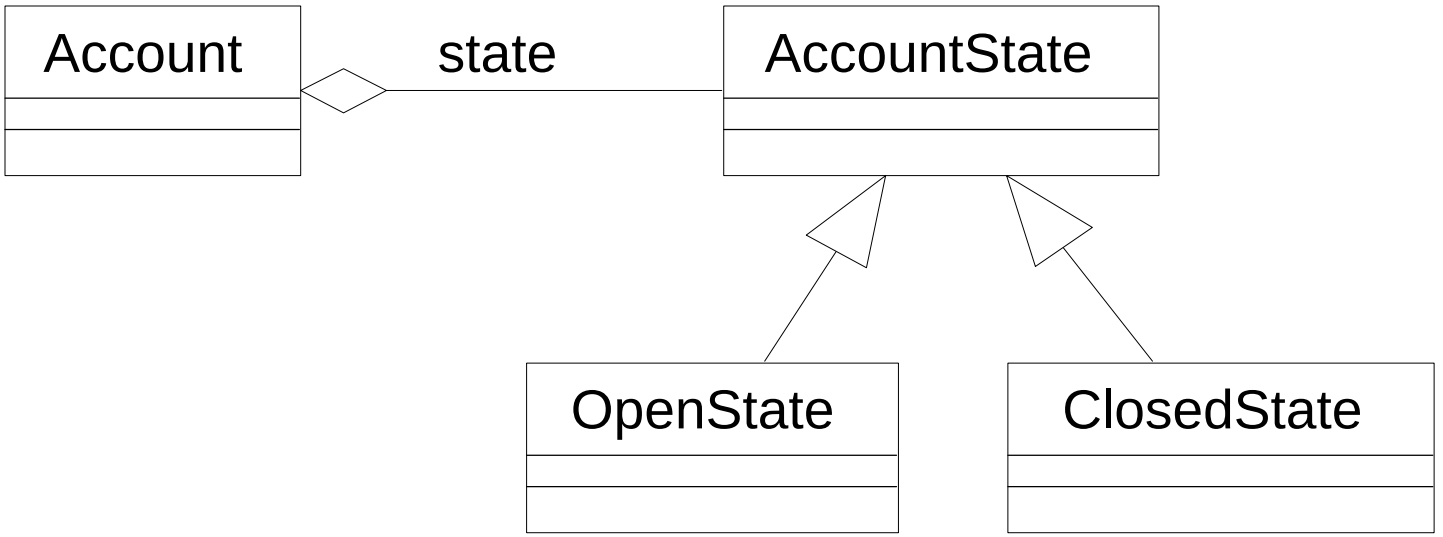
```
    /**
     * The push() method performs different actions depending
     *   on the state of the object.  Using the State pattern,
     *   we delegate this behavior to our contained state object.
     */
    public void push() {state.handlePush(this);}

    /**
     * The pull() method performs different actions depending
     *  on the state of the object.  Using the State pattern,
     *  we delegate this behavior to our contained state object.
     */
    public void pull() {state.handlePull(this);}

}
```

The State Pattern

- Implementation Issues
  - Who defines the state transitions?
    - The Context class => ok for simple situations
    - The ConcreteState classes => generally more flexible, but causes implementation dependencies between the ConcreteState classes
    - Example 1 has the ConcreteState classes define the state transitions
  - When are the ConcreteState objects created?
    - Create ConcreteState objects as needed
    - Create all ConcreteState objects once and have the Context object keep references to them
    - Example 1 creates them as needed
  - Can't we just use a state-transition table for all this?
    - Harder to understand
    - Difficult to add other actions and behavior

State Pattern Example 2

- Situation: A bank account can change from an open account to a closed account and back to an open account again. The behavior of the two types of accounts is different.
- Solution: Use the State pattern!



State Pattern Example 3 - SPOP

- This example comes from Roger Whitney, San Diego State University
- Consider a simplified version of the Post Office Protocol used to download e-mail from a mail server
- Simple POP (SPOP) supports the following command:
  - USER username
    - The USER command with a username must be the first command issued
  - PASS password
    - The PASS command with a password or the QUIT command must come after USER. If the username and password are valid, then the user can use other commands.
  - LIST <message number>
    - The LIST command returns the size of all messages in the mail box. If the optional message number is specified, then it returns the size of that message.

State Pattern Example 3 - SPOP (Continued)

- RETR <message number>
  - The RETR command retrieves all message in the mail box. If the optional message number is specified, then it retrieves that message.
- QUIT
  - The QUIT command updates the mail box to reflect transactions taken, then logs the user out.

State Pattern Example 3 - SPOP (Continued)

- Here's a version of an SPop class without using the State pattern:

```
public class SPop {
  static final int QUIT = 1;
  static final int HAVE_USER_NAME = 2;
  static final int START = 3;
  static final int AUTHORIZED = 4;
  private int state = START;
  String userName;
  String password;
```

State Pattern Example 3 - SPOP (Continued)

```
public void user(String userName) {
  switch (state) {
    case START: {
      this.userName = userName;
      state = HAVE_USER_NAME;
      break;
    }
    default: { // Invalid command
      sendErrorMessageOrWhatEver();
      endLastSessionWithoutUpdate();
      userName = null;
      password = null;
      state = START;
    }
  }
}
```

State Pattern Example 3 - SPOP (Continued)

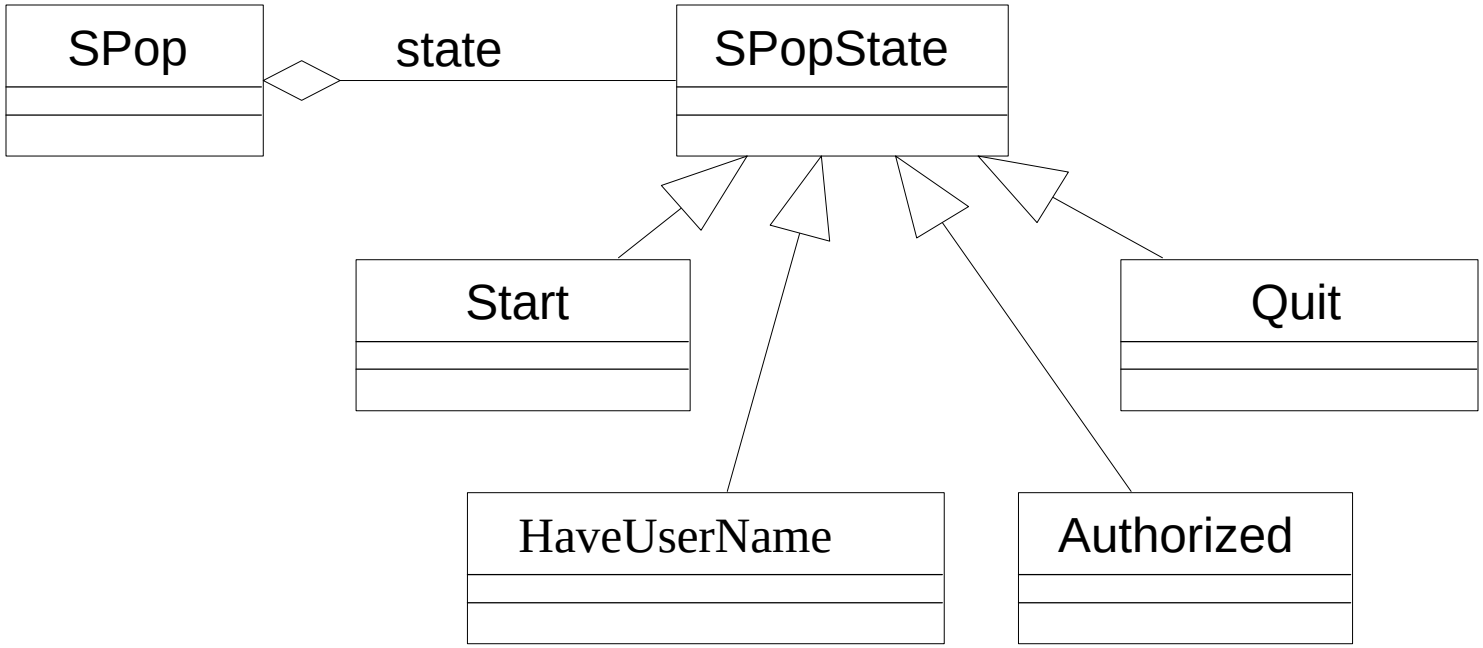
```
public void pass(String password) {
  switch (state) {
    case HAVE_USER_NAME: {
      this.password = password;
      if (validateUser())
        state = AUTHORIZED;
      else {
        sendErrorMessageOrWhatEver();
        userName = null;
        password = null;
        state = START;
      }
    }
```

State Pattern Example 3 - SPOP (Continued)

```
        default: {  // Invalid command
          sendErrorMessageOrWhatEver();
          endLastSessionWithoutUpdate();
          state = START;
        }
      }
    }
    ...
}
```

State Pattern Example 3 - SPOP (Continued)

- Now let's use the State pattern!
- Here's the class diagram:



State Pattern Example 3 - SPOP (Continued)

- First, we'll define the SPopState class. Notice that this class is a concrete class that defines default actions.

```
public class SPopState {

  public SPopState user(String userName) {default action here}

  public SPopState pass(String password) {default action here}

  public SPopState list(int messageNumber) {default action here}

  public SPopState retr(int messageNumber) {default action here}

  public SPopState quit() {default action here}

}
```

State Pattern Example 3 - SPOP (Continued)

- Here's the Start class:

```
public class Start extends SPopState {

  public SPopState user(String userName) {
    return new HaveUserName(userName);
  }

}
```

State Pattern Example 3 - SPOP (Continued)

- Here's the HaveUserName class:

```
public class HaveUserName extends SPopState {

  String userName;

  public HaveUserName(String userName) {
    this.userName = userName;
  }

  public SPopState pass(String password) {
    if (validateUser(userName, password)
      return new Authorized(userName);
    else
      return new Start();
  }
}
```

State Pattern Example 3 - SPOP (Continued)

- Finally, here is the SPop class that uses these state classes:

```
public class SPop {
  private SPopState state = new Start();

  public void user(String userName) {
    state = state.user(userName);
  }

  public void pass(String password) {
    state = state.pass(password);
  }

  public void list(int messageNumber) {
    state = state.list(messageNumber);
  }
  ...
}
```

State Pattern Example 3 - SPOP (Continued)

- Note, that in this example, the state classes specify the next state
- We could have the SPop class itself determine the state transition (the state classes now return true of false):

```
public class SPop {
  private SPopState state = new Start();
  public void user(String userName) {
    state.user(userName);
    state = new HaveUserName(userName);
  }
  public void pass(String password) {
    if (state.pass(password))
      state = new Authorized();
    else
      state = new Start();
  }
}
```

State Pattern Example 3 - SPOP (Continued)

- Multiple instances of SPop could share state objects if the state objects have no required instance variables or the state objects store their instance variables elsewhere
- Such sharing of objects is an example of the Flyweight Pattern
- How can the state object store its state elsewhere?
  - ⇨ Have the Context store this data and pass it to the state object (a push model)
  - ⇨ Have the Context store this data and have the state object retrieve it when needed (a pull model)

State Pattern Example 3 - SPOP (Continued)

- Here's an example of the Context storing the state and passing it to the state objects:

```
public class SPop {
  private SPopState state = new Start();
  String userName;
  String password;

  public void user(String newName) {
    this.userName = newName;
    state.user(newName);
  }

  public void pass(String password) {
    state.pass(userName, password);
  }
  ...
}
```

State Pattern Example 3 - SPOP (Continued)

- Here the Context stores the state and the state objects retrieve it:

```
public class SPop {
  private SPopState state = new Start();
  String userName;
  String password;

  public String getUserName() {return userName;}

  public String getPassword() {return password;}

  public void user(String newName) {
    this.userName = newName ;
    state.user(this);
  }
  ...
}
```

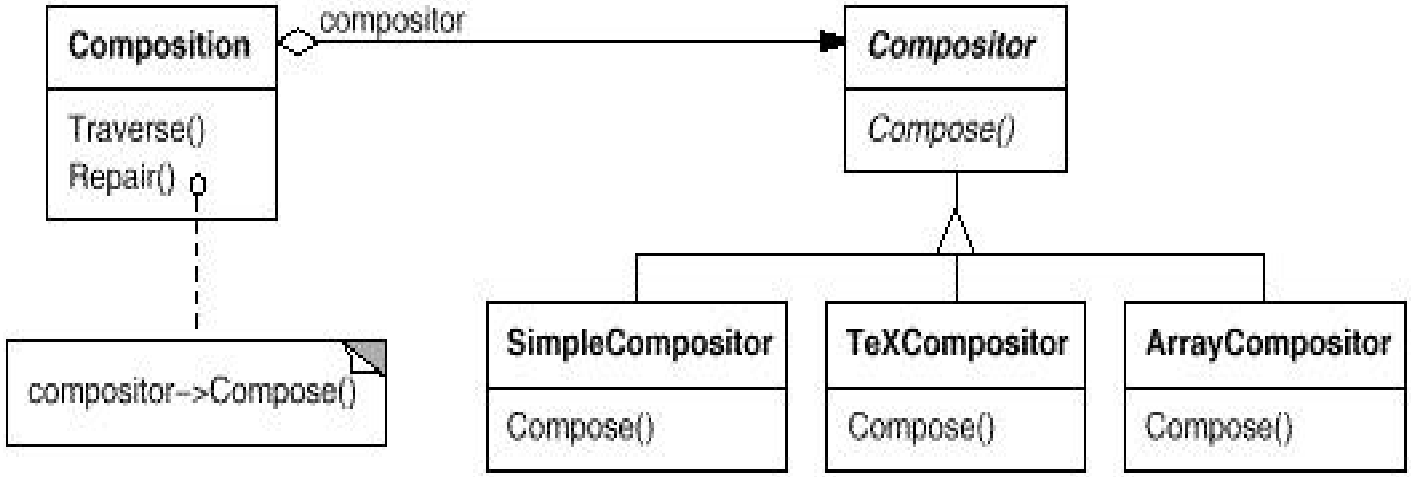
State Pattern Example 3 - SPOP (Continued)

- And here is how the HaveUserName state object retrieves the state in its user() method:

```
public class HaveUserName extends SPopState {

  public SPopState user(SPop mailServer) {
    String userName = mailServer.getUserName();
    ...
  }
  ...
}
```

The Strategy Pattern

- Intent
  - ⇨ Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Motivation



The Strategy Pattern

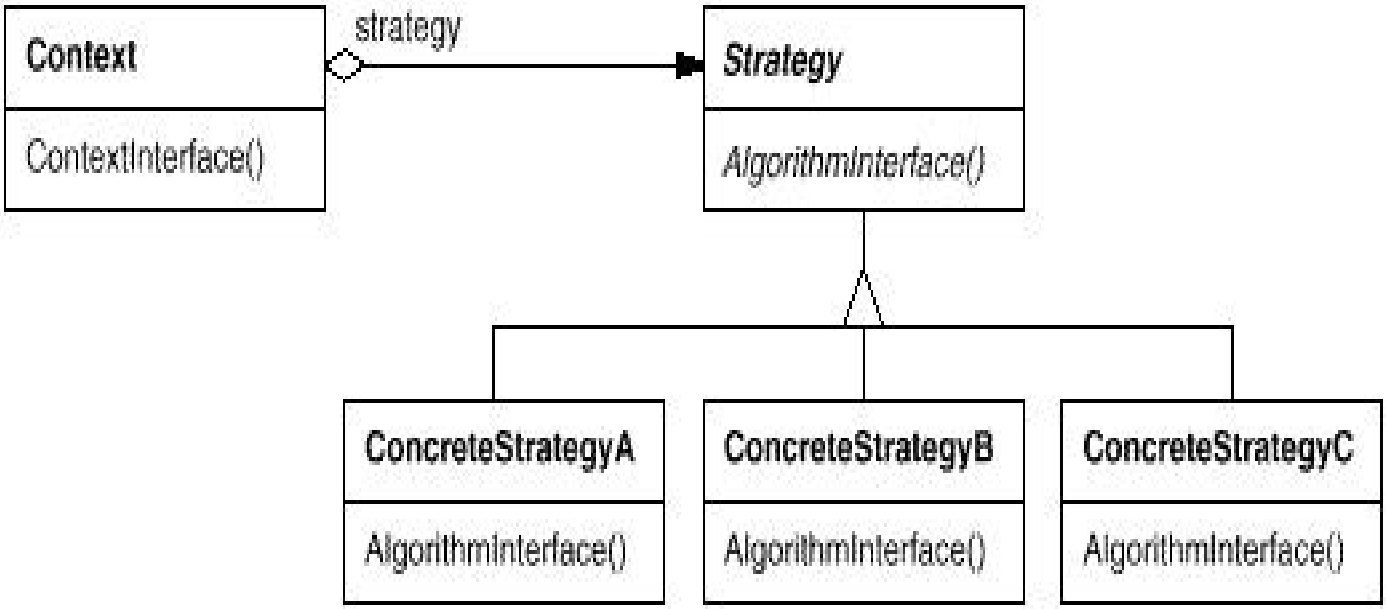
- Applicability

  Use the Strategy pattern whenever:

  - Many related classes differ only in their behavior
  - You need different variants of an algorithm
  - An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
  - A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

The Strategy Pattern

- Structure

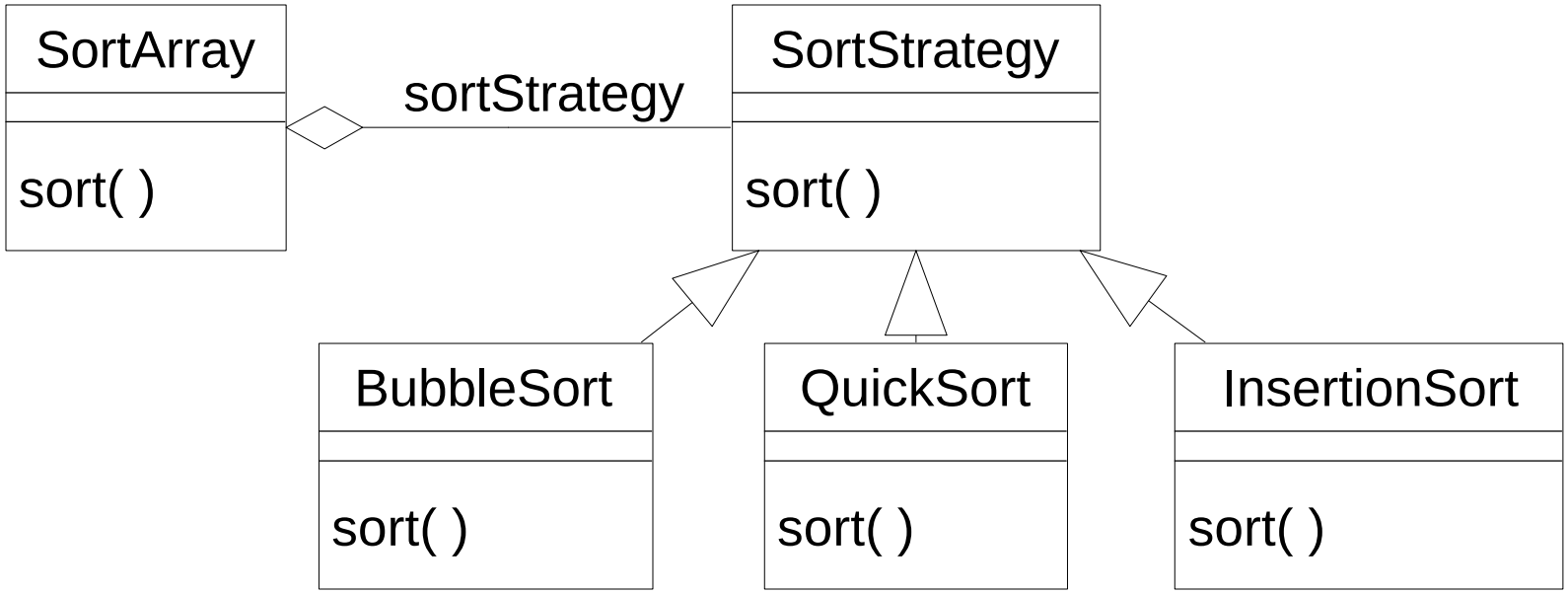


The Strategy Pattern

- Consequences
  - Benefits
    - Provides an alternative to subclassing the Context class to get a variety of algorithms or behaviors
    - Eliminates large conditional statements
    - Provides a choice of implementations for the same behavior
  - Liabilities
    - Increases the number of objects
    - All algorithms must use the same Strategy interface

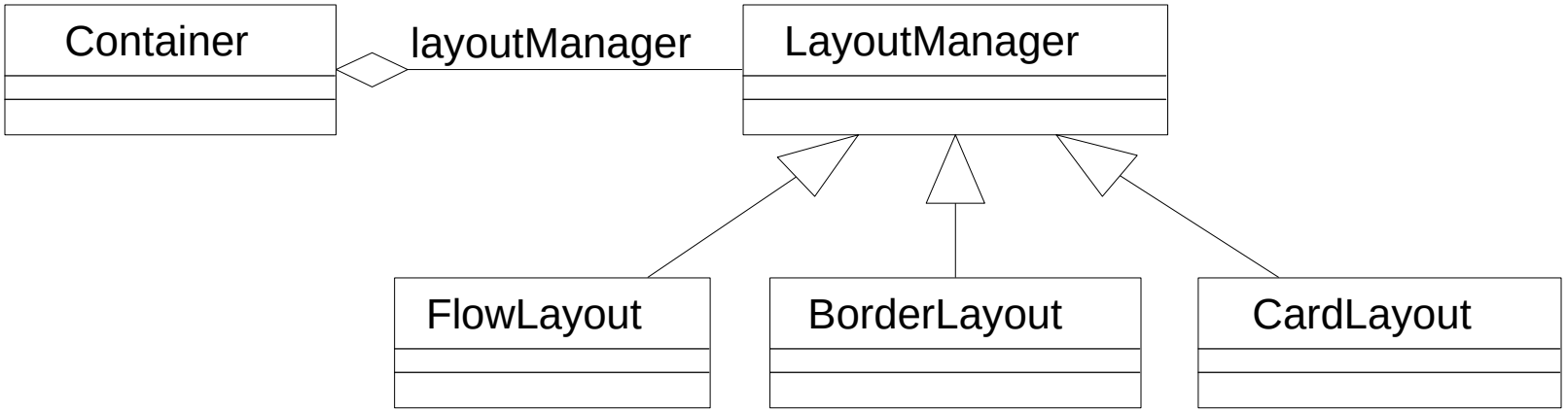
Strategy Pattern Example 1

- Situation: A class wants to decide at run-time what algorithm it should use to sort an array. Many different sort algorithms are already available.
- Solution: Encapsulate the different sort algorithms using the Strategy pattern!



Strategy Pattern Example 2

- Situation: A GUI container object wants to decide at run-time what strategy it should use to layout the GUI components it contains. Many different layout strategies are already available.
- Solution: Encapsulate the different layout strategies using the Strategy pattern!
- Hey! This is what the Java AWT does with its LayoutManagers!



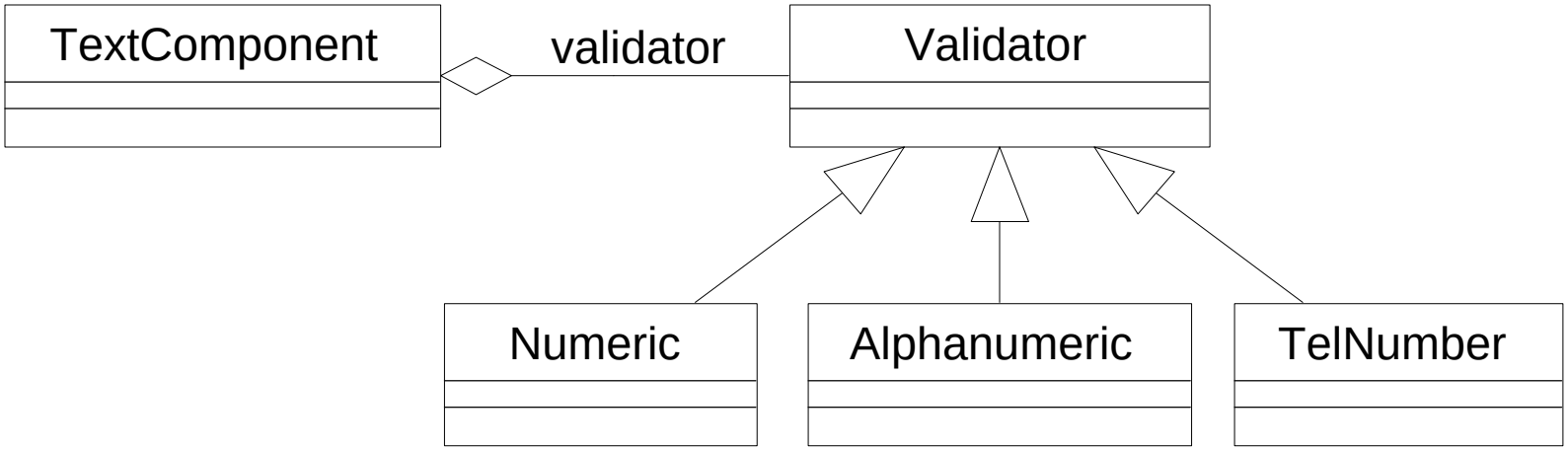
Strategy Pattern Example 2 (Continued)

- Some client code:

```
Frame f = new Frame();
f.setLayout(new FlowLayout());
f.add(new Button("Press"));
```

Strategy Pattern Example 3

- Situation: A GUI text component object wants to decide at run-time what strategy it should use to validate user input. Many different validation strategies are possible: numeric fields, alphanumeric fields, telephone-number fields, etc.
- Solution: Encapsulate the different input validation strategies using the Strategy pattern!



Strategy Pattern Example 3 (Continued)

- This is the technique used by the Java Swing GUI text components. Every text component has a reference to a document model which provides the required user input validation strategy.

The Null Object Pattern

- Sometimes the Context may not want to use the strategy provided by its contained Strategy object. That is, the Context wants a "do-nothing" strategy.
- One way to do this is to have the Context assign a null reference to its contained Strategy object. In this case, the Context must always check for this null value:

```
if (strategy != null)
   strategy.doOperation();
```

The Null Object Pattern

- Another way to accomplish this is to actually have a “do-nothing” strategy class which implements all the required operations of a Strategy object, but these operations do nothing. Now clients do not have to distinguish between strategy objects which actually do something useful and those that do nothing.
- Using a “do-nothing” object for this purpose is known as the *Null Object Pattern*

The Strategy Pattern

- Note the similarities between the State and Strategy patterns! The difference is one of intent.
  - ⇨ A State object encapsulates a state-dependent behavior (and possibly state transitions)
  - ⇨ A Strategy object encapsulates an algorithm
- And they are both examples of Composition with Delegation!