Tecniche di Progettazione: Design Patterns

GoF: State
State Pattern Example 1

- Consider a class with two methods, push() and pull(), whose behavior changes depending on the object state.
- 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.
First, let's do this without the State pattern:

```java
/**
 * 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;
    }
/**
 * 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)

```java
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());
    }
}
```
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

- Behavioral Pattern
- Intent
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class
- Example
  - TCP Connection responds differently to clients based on state
    - Established
    - Listening
    - Closed
Applicability

- Use the state pattern in either of the following cases:
  - 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.
Structure
Example Structure

- TCP State is abstract and defines interface
- Each subclass of TCPState represents a state
Participants

- **Context (TCPConnection)**
  - Defines the interface of interest to clients
  - Maintains an instance of a ConcreteState subclass that defines the current state.

- **State (TCPState)**
  - Defines an interface for encapsulating the behavior associated with a particular state of the Context

- **ConcreteState subclasses (TCPEstablished, TCPLlisten, TCPClosed)**
  - Each subclass implements a behavior associated with a state of the Context.
Consequences

- Localizes state-specific behavior and partitions behavior for different states
  - All state specific behavior is stored in one class
  - Alternative is giant case statements 😞
  - Can produce a large number of classes, but is better than the alternative

- Makes state Transitions explicit
  - Current state is stored in one location

- State objects can be shared
  - When state objects have no instance variables
Implementation

- Who defines the state transitions?
  - Can be done by Context
  - Can be by State Objects

- A table-based Alternative
  - State Pattern vs. table-driven approach
    - State pattern models state specific behavior
    - Table-Driven approach focuses on defining state transitions
Implementation

- Creating and destroying State objects
  - Create State objects only when they are needed and destroy them afterwards
    - States changes are infrequent
    - Pay at use
  - Create State objects ahead of time and never destroy them
    - State changes are frequent
    - Pay upfront
State Pattern Example 1 (Continued)

- Using State
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:

```java
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;}
  
  // 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;}

Design patterns, Laura Semini, Università di Pisa, Dipartimento di Informatica.
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);}
State can be an abstract class if there is some common functionality
Other Implementations

- We can let Context to decide the flow of state transition and let the States decide the other actions
  - Example 1 has state transition as the only reaction to the commands, but this is not always the case

- State transition in the state classes is more uniform
  - But more dependencies between classes.

- Another approach to writing a State Machine is to use a table-driven approach
  - Table-driven methods are schemes that allow you to look up information in a table rather than using logic statements (i.e. case, if).
  - Only for state transition
Check list

1. Identify an existing class, or create a new class, that will serve as the “state machine” from the client’s perspective. That class is the “wrapper” class.

2. Create a State base class that replicates the methods of the state machine interface. Each method takes one additional parameter: an instance of the wrapper class. The State base class specifies any useful “default” behavior.

3. Create a State derived class for each domain state. These derived classes only override the methods they need to override.

4. The wrapper class maintains a “current” State object.

5. All client requests to the wrapper class are simply delegated to the current State object, and the wrapper object’s this pointer is passed.

6. The State methods change the “current” state in the wrapper object as appropriate.
State vs Strategy

- Similarities between the State and Strategy patterns!
- E.g. they are both examples of Composition with Delegation.
- The difference is one of intent.
  - A State object encapsulates a state-dependent behavior (and possibly state transitions)
  - A Strategy object encapsulates an algorithm
  - In State a Context object changes state according to well defined state transitions.
Homework

- Complete the colours example
- Modify it accordingly to the implementation strategy
  - Create State objects ahead of time and never destroy them
- And then also
  - Letting Context to decide the flow of state transitions.
  - Using a table to define state transitions