Design Patterns: State, Bridge, Visitor

State

We've been talking about bad uses of case statements in programs. What is one example?

Another way in which case statements are sometimes used is to implement finite-state machines.

Here's an example.

An example: Horner's Rule

A finite-state machine can be used to convert an ASCII string of characters representing a real number to its actual numerical value.

Input symbols are written in **bold** above the transitions.

Symbol classes (such as *digit*) are written in *italic*.

Actions performed by the program are in **courier** below the transitions.

The variable *c* denotes the input character.

The letters shown in the FSM stand for the following:

- **c** - current character
- **s** - sign of the number
- **v** - value of the number
- **p** - power

Outline for Week 14

I. State pattern
II. Bridge
III. Visitor
Note that this FSM assumes that the string contains a valid floating point number that

- starts with an optional + or −,
- has at least one digit, an optional decimal point,
- and any number (including 0) of digits before and after the decimal point.

A value of 0 is returned if an invalid string is encountered.

**Table form of FSM:**

<table>
<thead>
<tr>
<th>State/Input</th>
<th>+ or −</th>
<th>.</th>
<th>digit</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>INTEGER</td>
<td>DECIMAL</td>
<td>INTEGER</td>
<td>ERROR</td>
</tr>
<tr>
<td>INTEGER</td>
<td>ERROR</td>
<td>DECIMAL</td>
<td>INTEGER</td>
<td>END</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>ERROR</td>
<td>ERROR</td>
<td>DECIMAL</td>
<td>END</td>
</tr>
</tbody>
</table>

Using switch statements, this FSM can be coded as follows:

```java
public class Parser {
    static double toDouble(String s) {
        double sign = 1; // sign of number (either 1 or −1)
        double value = 0; // current value of the number
        double power = 0.1; // current power of 10 for
                             // digits after decimal point
        int i = 0;
        final int START = 0;
        final int INTEGER = 1;
        final int DECIMAL = 2;
        final int ERROR = 3;
        int state = START;
        char ch; // current character in string

        while (state != ERROR && i < s.length()) {
            ch = s.charAt(i++);
            switch (state) {
                case START: if (ch == '.')
                    state = DECIMAL;
                else if (ch == '−') {
                    sign = -1.0;
                    state = INTEGER;
                } else if (ch == '⁺')
```
state = INTEGER;
else if (Character.isDigit(ch)) {
    value = ch - '0';
    state = INTEGER;
}
else
    state = ERROR;
break;

case INTEGER: if (ch == '.')
    state = DECIMAL;
else if (Character.isDigit(ch))
    value = 10.0 * value + (ch - '0');
else {
    value = 0.0;
    state = ERROR;
}
break;

case DECIMAL: if (Character.isDigit(ch)) {
    value += power * (ch - '0');
power /= 10.0;
}
else {
    value = 0.0;
    state = ERROR;
}
break;

default: System.out.println("Invalid state: " + state);
}
return sign * value;

public static void main(String[] args) {
    if (args.length == 1)
        System.out.println(toDouble(args[0]));
}

This FSM can be represented more elegantly by the State pattern.

How can we code State in a more o-o fashion? **Hint**: We can make State an interface! Each state will implement this interface.

**Horner’s Rule**: To use the State pattern for Horner’s rule, the first step is to define a State interface. Consider the table form of the FSM.

- The rows of the table represent the different states.
• The columns of the table represent the different behaviors of each state.

Therefore, what methods should be defined in the State interface? Well, what do we need to test for each state?

Submit your State interface here.

```java
public interface State {
    void onPoint();
    void onOther();
}
```

How should the States be defined?

```java
class ______ implements _____ { ... }
class ______ implements _____ { ... }
class ______ implements _____ { ... }
```

In Java, we can define a class within another class. This is called an inner class.

Thus, our States can be defined as inner classes of Parser.

Here is the code for the Parser class, minus its inner classes:

```java
public class Parser {
    private final State start = new Start();
    private final State integer = new Integr();
    private final State decimal = new Decimal();
    private State state = start;
    double sign = 1; // sign of number (either 1 or -1)
    double value = 0; // current value of the number
    double power = 0.1; // current power of 10 for digits after decimal point
    char ch; //current character in string

    double toDouble(String s) {
        int i = 0;
        while (i < s.length()) {
```
ch = s.charAt(i++);
if (ch == '.') state.onPoint();
else if (ch == '+') state.onPlus();
else if (ch == '-') state.onMinus();
else if (Character.isDigit(ch)) state.onDigit();
else state.onOther();
}
return sign * value;
}

public static void main(String[] args) {
    System.out.println(new Parser().toDouble("-914.334"));
}

Exercise: Depending on your row, choose one method to implement in all three classes. Submit your code here.

    void onMinus();
    void onPlus();
    void onDigit();
    void onOther();

If an illegal character is found, throw a NumberFormatException.

State

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

Problem: A monolithic object’s behavior is a function of its state, and it must change its behavior at run-time depending on that state. Or, an application is characterized by large and numerous case statements that vector flow of control based on the state of the application.

Solution: Define a set of State objects. Defer the implementation of the different methods to the State objects. The client changes its state from time to time.

Implementation: • Define a “context” class to present a single interface to the outside world.
    • Define a State abstract base class.
• Represent the different “states” of the state machine as derived classes of the State base class.
• Define state-specific behavior in the appropriate State derived classes.
• Maintain a pointer to the current “state” in the “context” class.
• To change the state of the state machine, change the current “state” pointer.

State vs. Strategy
Recall the Strategy pattern from online Week 6 (Lecture 9).

In this case, you need to choose an algorithm for a task depending on some “parameter” of the situation.

For example, consider quadrature (numerical integration) again. Each time you calculate the area of a region, you need to know what the function is that you are calculating the region underneath.

Or consider how to open, read/write from, and close different kinds of files.

The definition of Strategy from HFDP is,

The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Let’s provide a definition of Strategy in the format that we have used before.

Strategy

Intent: To decouple algorithms from the objects that use them.

Problem: We need to write code that will work with objects that behave differently. We don’t want the host class to know how the objects behave, but just to invoke the different behaviors when appropriate.
**Solution:** Create different classes, each of which defines the behavior for one type of object. Then, when the host class wants to invoke a certain behavior, it merely calls a method of the appropriate behavior class.

**Implementation:** The behavior classes are usually subclasses of a class that defines an abstract behavior. Then the host class invokes a method of the abstract behavior class, and what is really called is a method of the concrete behavior class.

Strategy allows clients to change algorithms at run time by using a different strategy object. This basically lets them appear to change class at run time.

[HFDP, “§”10.10] Hmmm … that’s interesting. Strategy lets objects appear to change class. Isn’t that what State does?

How can we tell the difference between Strategy and State?

**Bridge**

[HFDP, §14.1] Now, let’s look at the class diagram for the State pattern (for example, at http://www.dofactory.com/Patterns/PatternState.aspx).

Note that the Context object holds a reference to an abstract class. This is the abstract class from which the concrete states are derived.

It turns out that the same structure is valuable for implementing drivers. What’s a driver?

Drivers are generally used by various applications. Consider a printer driver, for example. There is a driver for a particular printer for each OS. Many different applications can use that printer, providing that the right driver is present.

This is an example of the bridge pattern. We have various abstractions (the applications) and various implementations (the drivers). If the applications aren’t in a hierarchy, the class diagram for Bridge is exactly the same as the class diagram for State.

But often, the abstractions are in a hierarchy. HFDP gives the example of televisions and remotes.
and so do remotes. If they’re “universal remotes,” they can be used by lots of different kinds of TVs.

This leads to the UML diagram on p. 613 of HFDP, or view it online.

What are the advantages of Bridge?

- OK, we just told you the right way to handle this problem. What would be the wrong way? Can you imagine using a lot more subclasses?

Here’s another example, from StackOverflow: Suppose you have shapes, each of which can be various colors.

Would you subclass Shape with Rectangle and Circle …

```
Shape
  \-- Rectangle
  \-- Circle

and then subclass again for color:

Shape
  \-- Rectangle
    \-- BlueRectangle
    \-- RedRectangle
  \-- Circle
    \-- BlueCircle
    \-- RedCircle
```

The Bridge pattern says to represent this as

```
Shape
  \-- Rectangle
  \-- Circle
    \-- Red
    \-- Blue
```
This is another way of preferring composition over inheritance. (What was another example we saw of that?)

**Bridge**

**Intent:** To decouple interfaces from the implementations that use them.

**Problem:** We need to construct an interface that may change (or be added to), and several implementations of that interface. We don’t want to have to have each implementation implement each interface.

**Solution:** Use a hierarchy of interfaces, and a hierarchy of implementations. The implementations code to a single abstract interface. The other interfaces are written in terms of the abstract interface.

**Implementation:** Use two class hierarchies, where the interface hierarchy has an abstract implementation.

Now finish this example of the Bridge pattern.

**Visitor**

[HFD, §14.9] Remember our discussion of overloading vs. overriding, from Lecture 16 and §2.8 of Skrien? At that time, we said,

In summary, the compiler decides which overloaded method to call by looking at the declared type of

- the object being sent the message and
- the declared types of the arguments to the method call.

The method is chosen at runtime by dynamic method invocation using the actual value of the object being sent the message.

The actual classes of the arguments to the method call do not play a role.
This is very different from a language like CLOS, which uses the actual types of the arguments to decide which method to execute.

Suppose we did want the classes of the arguments to be used to determine which method to call.

My favorite example is “double-dispatching” in arithmetic expressions.

- If you add an integer and a floating-point number, what type should the result be?
- Assuming you have a Fraction class, if you add an integer and a fraction, what type should the result be?
- If you add a floating-point number and a complex number, what type should the result be?

Should either the floating-point or complex number be able to be the receiver? Should either be able to be the argument?

So, the method called should depend both on the class of the receiver and the class of the argument. How do we achieve this effect?

Let’s say that we implement the Sum method in all numeric classes—Integer, Floating Point, Fraction, and Complex.

So, if we’re performing an addition, we invoke the Sum method of the ___________ class.

Now, this Sum method knows that what it does actually depends on the class of its argument. How does it achieve this effect?

This method, e.g., in the Complex class, is called something like, SumFromFloatingPoint.

- What does it do?
- What does it return?
OK, suppose that we have the four numeric classes mentioned above. How many Sum… methods do we need altogether?

Here is how Visitor is structured.

- Define an interface or abstract class Visitor.
- Visitor contains a `visit()` method, which is implemented in each subclass of Visitor. (In our example, these are the `sumFrom` methods.)
- These methods are invoked from subclasses of the Element hierarchy. Each one of these classes has an `accept()` method, which takes an object of the Visitor hierarchy as a parameter.
- Each descendant of the Element class implements `accept()` by calling the `visit()` method on the Visitor object it was passed, with `this` as the only parameter.
- To perform an operation, the client creates a Visitor object, and calls `accept()` on the Element object, passing the Visitor object.

The Visitor pattern can be used to avoid tight coupling, as Bob Martin explains.

Can you think of another example of Visitor? What’s the example given in the text for Visitor? Submit here.