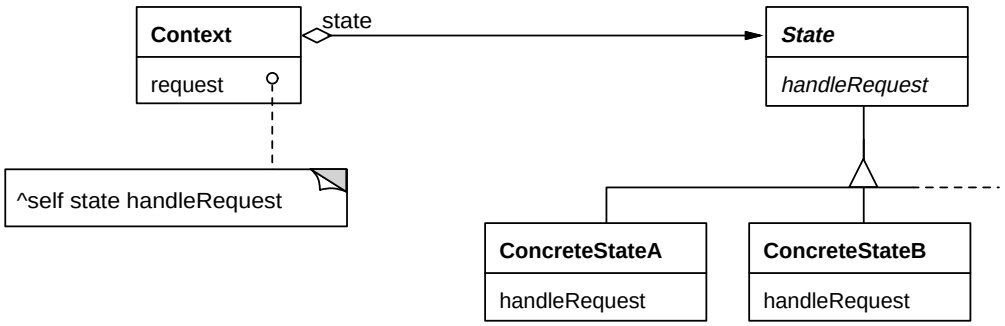
STATE (DP 305) Object Behavioral

Intent

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

Structure



Discussion

In many circumstances, an object's behavior will change over time in response to requests that are made of that object. In effect, the "history" of the object's life determines how it behaves in the present. We have all seen code that looks like the following:

```
Order>>submit
   self state == #created
      ifTrue:[self error: 'Must be validated first'.
   self state == #validated
      ifTrue:[OrderProcessor submitOrder: self.
              self state: #submitted].
   self state == #submitted
      ifTrue:[self error: 'Can''t submit this order twice'].
   self state == #deleted
      ifTrue:[self error: 'Order has been deleted.'].
```

Code like this serves as a warning to the reader, saying "This object has some significant history. You ignore it at your peril." Many design methods represent this kind of changing behavior of an object through state transition diagrams that represent the different conditions or "States" of an object, the events that cause transitions between those states, and the actions that occur when a transition is made. However, while the design diagram may nicely sum up this kind of information, that clarity is usually lost in the translation to code that looks like the previous example.

The State pattern brings this kind of clarity back to the code. With the State pattern, each of the states is represented as a separate object. The Client initializes itself with its initial-state object. To transition to a new state, it replaces its state object with an appropriate new one. When the Client needs to make decisions or provide behavior *based on* its state, it does so by delegating to its state object. In this way, the Client's implementation remains relatively free of state dependencies because it delegates them to its collaborating state objects.

The key to the State pattern lies in separating the state-dependent behavior of the client into a separate set of State objects. The State objects implement that part of the system's behavior that is dependent on the current state of the client, rather than the client itself. States are defined in a hierarchy where each subclass represents an individual state. The client is implemented in a Context class that collaborates with the State hierarchy. When the Context needs to handle a request that is state-dependent, it delegates the request to its current State. Each State subclass implements its handling of the request differently.

Not a Change of Class

The Intent says that the State pattern allows an object to appear to change class when its internal state changes. Some developers may prefer to look at the problem that way, but the truth is simply that the object has a lot of behavior that depends on its internal state, so much so that it acts very differently when its state changes. These differences may be so profound that the object acts like a totally different specialization of its type, but not all examples of the State pattern are quite so dramatic.

The basic issue is not so much that the object appears to change class, but that the object's external behavior is highly dependent on its internal state. Most objects have internal state that can change and affect their behavior. The difference with the State pattern is that a Context has a limited number of well-defined states and a consistent set of rules that control the transitions between these states.

The Code Transformation

Applying the State pattern causes a fairly consistent code transformation. Without the pattern, behavior that depends on the states often uses case statements to vary the behavior accordingly:

```
Object subclass: #Context
   instanceVariableNames: 'state'
   classVariableNames: ''
   poolDictionaries: ''
```

```
Context>>request
    "Handle the request according to the Context's state."
    self state == #stateA ifTrue: [^self stateARequest].
    self state == #stateB ifTrue: [^self stateBRequest].
    ...
    ^self error: 'unknown state'
```

The *Smalltalk Companion* contains many examples of the evils of case statements and why they should be avoided. In this context, the case statement has two basic problems:

1. The same basic case structure must be duplicated for each public request that a client might make of the `Context`. Each such `request` method will have to test for `#stateA`, `#stateB`, and so on.
2. New states cannot be added without modifying all of the `request` methods.

The State pattern transforms the code to remove the state dependent code from the `Context`. Various ways of handling each request are implemented polymorphically. The `Context` still knows its `state`, but rather than running it through a case statement, the `Context` delegates to its `state` as a real object.

```
Object subclass: #Context
    instanceVariableNames: 'state'
    classVariableNames: ''
    poolDictionaries: ''

Context>>request
    "Handle the request according to the Context's
    currentState."
    self state handle
```

The `state` object comes from a hierarchy of possible states. The abstract class defines the interface that all states have.

```
Object subclass: #State
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''

State>>handle
    self subclassResponsibility
```

Subclasses of `State` implement different states that the Context can be in. Each subclass implements `handle` to provide the behavior appropriate to handle the request in that state.

```
Object subclass: #ConcreteStateA
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

```
State>>handle
   "Handle the request one way."

Object subclass: #ConcreteStateB
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''

State>>handle
   "Handle the request another way."
   ...
```

Other subclasses implement other states. Each subclass encapsulates the behavior for that state so that it doesn't clutter the `Context` or the other states. Adding new states in the future is easy: The main step is to add a new `State` subclass that implements the `handle` messages appropriately.

Changing the Class for Real

If the point of the State pattern is to make the object seem like it has changed its class, why not go ahead a actually coerce the instance to a different class? In languages without reflection, like C++, there is a tendency (almost a necessity) to use somewhat complicated means to do what in Smalltalk is possible with messages that change the identity or metaclass of an object. In all dialects we can use the method `become:` to change the identify of an object. In VisualWorks we can use the method `changeClassToThatOf:` to change its metaclass.

At first glance, the State pattern appears to be another instance of this unnecessary complication. Why not just change the class? Simply, there are two basic problems with using `changeClassToThatOf:` or another coercion method:

1. All of the instance variables that any of the state classes may want to use must be defined in the superclass. The `changeClassToThatOf:` method will not work if a subclass defines any additional instance variables. A variant on this would be to use `become:` instead of `changeClassToThatOf:`. However, that would necessitate copying the values of all of the instance variables in the class over to the new instance before the `become:` is sent. Note that `changeClassToThatOf:` does not even exist in Visual Smalltalk nor IBM Smalltalk, so the `become:` approach is the only choice in these environments.

2. The bigger, and more central problem, is that we are now precluded from making subclasses of our "Context" object as would have been possible with using the State pattern. If we wanted to make a subclass, say to add an instance variable holding the longest run so far, we would be forced to make additional subclasses of all of the subclasses of the State/Context class. Preventing this class explosion is the biggest

single reason to use composition as in the State pattern rather than reflective facilities.

So, we have seen that the use of the language-specific features does not add flexibility in this case—it instead makes our design less flexible and harder to reuse. This is an important point to keep in mind when designing an OO program—the "neatest" or "slickest" solution may not necessarily be the best. Often the most straightforward solution is better than those that use special language features.

Applicability

Design Patterns lists a couple of considerations for the appropriateness of the State pattern. Here are some more:

- The Context must have a limited, well-defined list of possible states. It can have other internal state variables as well, just like any object. But the state that defines its current status can only have a few valid values so that each of those values can be implemented as a State subclass. If the state values can have a virtually unlimited number of valid combinations, then the State class needs a virtually unlimited number of subclasses, so the State pattern is not appropriate.
- The Context must have well-defined transitions between states. In any possible state, it must be clear when the Context should change to another state and, in each case, what that other state should be. If the transition between states is inconsistent and subjective, the State pattern is not appropriate.

Implementation

One of the Implementation issues deserves further clarification:

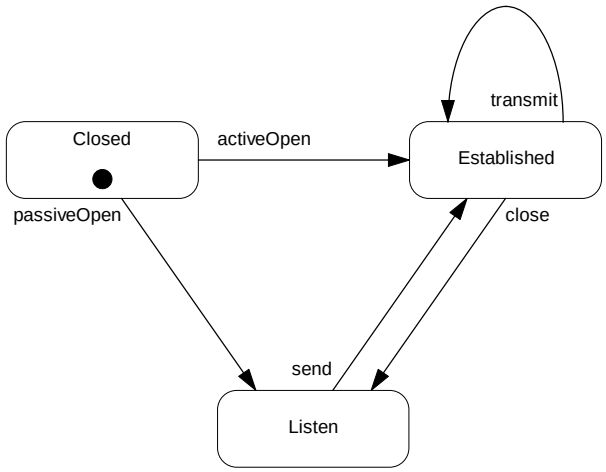
Who defines the state transitions? As *Design Patterns* explains, the State pattern does not specify which participant—the Context or the States—defines the criteria for the transitions from one state to the next. One advantage of the State pattern is that it removes the state dependent behavior from the Context and encapsulates it in the State objects. Similarly, the pattern should extract the state transition behavior from the Context and distribute it among the State objects. This way, each State knows when it should transition to another State and what other State it should transition to in each case. This requires that each State know what its Context is and have some way to tell its Context to switch to a new State.

Sample Code

Without the State Pattern

Let's look at the example from *Design Patterns*. We have a class named `TCPConnection` whose instances represent network connections (TCP stands for Transmission Control Protocol, a standard for network communication). A connection

may be in one of several states (e.g., Established, Listening, Closed, etc.), and its behavior depends on its current state. For example, when asked to send data, a connection object reacts differently depending on whether it's Established or Closed. These states form a finite state machine that is an implicit part of the connection. The following state transition diagram shows these states and the transitions between them:



The `TCPConnection` begins its life in the `Closed` state. An `activeOpen` or a `passiveOpen` command will send it to the `Established` or `Listen` states respectively. Once in the `Established` state, a `close` command will send it to the `Listen` state, while a `transmit` command will perform an action, but keep it in the `Established` state. In the `Listen` state, a `send` command will transfer it to the `Established` state.

One approach to implementing such state-dependent behavior might be to code a conditional of some sort which, based on the current state, invokes different code or methods within the `TCPConnection` object.

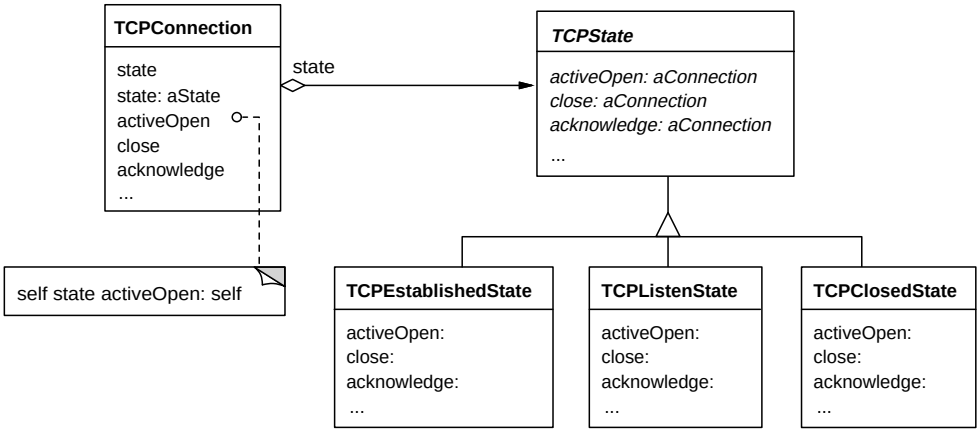
```
Object subclass: #TCPConnection
   instanceVariableNames: 'state'
   classVariableNames: ''
   poolDictionaries: ''

TCPConnection>>activeOpen
   "Open the connection."
   self state == #established ifTrue: [^self].
   self state == #listen ifTrue: [^self].
   self state == #closed
      ifTrue: [^self changeStateTo: #established].
   ^self error: 'unknown state'
```

```
TCPConnection>>send
   "Prepare the connection to send data."
   self state == #established ifTrue: [^self].
   self state == #listen ifTrue:
      [^self changeStateTo: #established].
   self state == #closed ifTrue:
      [^self error: 'closed connection'].
   ^self error: 'unknown state'
```

`TCPConnection` implements other requests—`passiveOpen`, `close`, and `transmit`—in a similar manner. Of course this means if we need to add new states later in the system's life, we'll have to revisit the conditional, adding code to account for these new states.

With the State Pattern

The State pattern offers a cleaner and more easily extensible solution. The structure looks like this:



This structure has the finite state machine shown earlier implicit in its design.

Let's assume that you already have the case statement code shown above and you want to convert it to the State pattern structure shown above. The first step is to represent the connection's state with a separate object. Thus `TCPConnection`'s state variable is not a simple enumeration object, such as a `Symbol`, but is now a real `TCPState` object with its own behavior.

```
Object subclass: #TCPConnection
   instanceVariableNames: 'state'
   classVariableNames: ''
   poolDictionaries: ''
```

```
Object subclass: #TCPState
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
```

The second step is to implement `TCPConnection` to delegate to `TCPState`. In this example, we only show the implementation of `activeOpen` and `send`. A real framework would also implement `passiveOpen`, `close`, and `transmit`.

```
TCPConnection>>activeOpen
   "Open the connection."
   self state activeOpen: aTCPConnection

TCPConnection>>send
   "Prepare the connection to send data."
   self state sendThrough: aTCPConnection
```

Step three: Implement `TCPState` to accept the delegation.

```
TCPState>>activeOpen: aTCPConnection
   "Open the connection."
   self subclassResponsibility

TCPState>>sendThrough: aTCPConnection
   "Prepare to send data through the connection."
   self subclassResponsibility
```

Step four: Since `TCPState` is actually an abstract class, implement a concrete subclass for each possible state.

```
TCPState subclass: #TCPEstablishedState
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''

Object subclass: #TCPListenState
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''

Object subclass: #TCPClosedState
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
```

Step five: Implement each of the superclasses appropriately for each of the subclasses.

```
TCPEstablishedState>>activeOpen: aTCPConnection
   "Do nothing."
   ^self
```

```
TCPEstablishedState>>sendThrough: aTCPConnection
   "Do nothing."
   ^self

TCPListenState>>activeOpen: aTCPConnection
   "Do nothing."
   ^self

TCPListenState>>sendThrough: aTCPConnection
   "Prepare to send data through aTCPConnection."
   aTCPConnection
      establishConnection;
      send: aString

TCPClosedState>>activeOpen: aTCPConnection
   "Open aTCPConnection."
   ^aTCPConnection establishConnection

TCPClosedState>>sendThrough: aTCPConnection
   "This is an error!"
   self error: 'closed connection'
```

Step six: Implement the messages that the TCPStates send to TCPConnection. Also, set a TCPConnection's default state to closed.

```
TCPConnection>>establishConnection
   "Establish a connection. Do all of the work necessary
   to properly establish a valid connection and verify
   its success."
   self state: TCPEstablishedState new

TCPConnection>>initialize
   "Set the connection's default state."
   state := TCPClosedState new.
   ^self
```

Here's an example of how you would subsequently use TCPConnection:

```
| connection |
connection := TCPConnection new.
connection
   activeOpen;
   transmit;
   close
```

Known Smalltalk Uses

SMTP/Sockets

"A Very Simple VSE SMTP Client" (ParcPlace, 1996) shows a textbook example of the State pattern's use to implement an explicit FSM in a communications protocol. This article shows how to implement an SMTP mail system in Visual Smalltalk Enterprise. It represents the SMTP states as State classes that cooperate with a `SocketClient` Context class to send messages to an SMTP server via TCP/IP sockets.

ControlMode

VisualWorks includes an example of the State pattern that is similar to `DrawingController` (DP 313). In VisualWorks, the class `ModalController` is used by the UI Builder framework to handle the positioning and resizing of the different UI components. `ModalController` "redirects mouse-related events to an internally held `ControlMode` object, which may then take some action."[1] A `ModalController` contains (through the intermediary of a `ValueHolder`) an instance of a `ControlMode` subclass which "implements some or all of the basic control sequence of a `ModalController`"[2].

Both the `ModalController`/`ControlMode` combination and the HotDraw example from *Design Patterns* are unusual in that there is no explicit FSM to provide the transitions. Instead the user will, by his actions, determine what the next state will be by picking from the list of possible states. This is an example where the States cannot control the state transitions, so the Context must control them itself.

AlgeBrain

AlgeBrain is an intelligent tutoring system for algebra (being built by Sherman along with colleagues Kevin Singley and Peter Fairweather). It uses State objects to represent the current state of solving an algebraic equation. An `AlgebraTutor` is configured at any point in time with an instance of one of several concrete `AlgebraState` subclasses—different `AlgebraState` objects know what user actions are acceptable or appropriate given the current state of the problem. Thus, when a user performs an action, the tutor delegates its response to its current State object; each of these determines if the user's action is correct, and can provide focused, context-dependent error messages. Each State object also knows what it is expecting in the way of user actions, and so can be used to provide help regarding what to do next to solve the current problem.

[1] VisualWorks `ModalController` class comment.
[2] VisualWorks `ControlMode` class comment.

Relational Database Interfacing

Brown (1996) demonstrates the use of the State pattern in an order processing system. The State objects provide for differing behaviors for saving and deleting objects from a relational database.

Related Patterns

State vs. Strategy

State is often confused with its close cousin Strategy, and it is sometimes difficult to determine which pattern to use, or if an implementation of one is actually the other. A good rule of thumb to follow to distinguish the two is that if the Context will contain only one (of several possible) "State" or "Strategy" objects during its lifetime, you are probably using the Strategy pattern. If, on the other hand, the Context changes during the normal course of an application so that, over time, it may contain many different "state" objects , then you may be referring to a State implementation, particularly if there are well-defined orders of transition between the different states. A subtler distinction can sometimes be found in the setting of an object's attributes. An object is usually put into a state by an external client, while it will choose a strategy on its own.

Another distinction is that a Context seems to hide the Strategy it's using, but the State it's using is quite apparent to its Client. For example, when a Client tells a `StorageDevice` to store some text, the device may use one of several different techniques to compress the text before storing it, including no compression at all. However, the Client doesn't care how the device compresses the text, whether it does so at all, or whether it compresses the text the same way every time. It just wants the device to store and retrieve the text on command. Because of the way the compression is private to the device and hidden from the Client, the different compression objects are Strategies. On the other hand, once a Client opens a `TCPConnection`, it certainly expects the connection to behave like it's open. Once the Client closes the connection, it expected the connected to act like it's closed. Because the connection's status is readily apparent to the Client, the status objects are States.