Methods for States

A Pattern for Realizing Object Lifecycles

Kevlin Henney
kevlin@curbralan.com
kevlin@acm.org

March 2003

Abstract

Substance doesn't change. Method contains no permanence. Substance relates to the form of the atom. Method relates to what the atom does. In technical composition a similar distinction exists between physical description and functional description. A complex assembly is best described first in terms of its substances: its subassemblies and parts. Then, next, it is described in terms of its methods: its functions as they occur in sequence. If you confuse physical and functional description, substance and method, you get all tangled up and so does the reader.

Robert M Pirsig, *Zen and the Art of Motorcycle Maintenance*

The intent of the METHODS FOR STATES pattern is to encapsulate modal behavior of an object within a single class. In stateful objects with strongly modal lifecycles, the behavior of a given method can be history sensitive, differing significantly according to the current state of the object. Simple lifecycle models can be implemented in terms of flags and conditional statements in each method; an approach whose code comprehensibility scales poorly. More sophisticated modal behavior can be realized through object delegation, drawing on a community of dependent classes to express the behavior. However, the larger the community, the more pronounced coupling and comprehensibility problems become.

Using METHODS FOR STATES a class is able to express all of the different behaviors as ordinary methods. It can do so without either the control coupling and reduced readability of large conditional statements or a large supporting cast of ancillary classes. Indirection, based on referring to methods as objects, is used to both represent the state and dispatch from a public method request to the correct underlying method for the state.

A structured but lightweight pattern form is used in the paper: the *problem* is summarized; a worked *example* with code presented in C++ and Ruby follows, exploring some possible solutions but not the pattern's proposed solution; the *forces* that circumscribe the problem are listed; the pattern *solution* is described; a *resolution* of the example is presented; the pattern *consequences* are then detailed; an *appendix* lists problem–solution thumbnails for related patterns; *acknowledgments* and *references* are listed at the end.

Problem

Some types of stateful object can be said to have strongly modal lifecycles. In such an object the behavior of some of its methods appears to alter significantly over the course the object's life. It is possible to distinguish different *modes* of operation — sometimes known as *macrostates* or, more ambiguously, *states* — that collectively describe such behaviors. A method's behavior depends on the object's internal state, including its current mode, and a change in mode comes about as a response to an event, such as a method call.

Assuming that each mode does not have associated state of its own, what is the most effective configuration for expressing the mode and its dependent behaviors?

Example

Consider a simple digital clock:

- It displays hours and minutes in normal 24-hour time.
- Two buttons can be used to adjust the current time: one button to change the mode and another to increment the hour or minute, depending on the mode.
- A heartbeat event is generated internally once a second to allow the clock to update. To keep the example simple, no latency is assumed.

The clock has three specific modes:

- Displaying the time, in which the increment button is ignored and the update event advances the stored time.
- Setting the hours, in which the clock ignores the update event and the increment button increments the displayed hour value.
- Setting the minutes, in which the clock ignores the update event and the increment button increments the displayed minute value.

This is not exactly a complex state machine, but there is enough here to get your teeth into. What are the options available for implementing such a state model?

Flags for States in C++

Let's start what might be termed the FLAGS FOR STATES pattern (see *Appendix*), expressed using a flag with a `switch` accompaniment in C++. The following code shows the essential public interface and a simple representation:

```
class clock
{
public:
    void change_mode();
    void increment();
    void tick();
    ...
private:
    enum mode
    {
        displaying_time, setting_hours, setting_minutes
    };
    mode behavior;
    int hour, minute, second;
};
```

The clock has a simple lifecycle model: the only significant behavioral changes come from the change-mode button. The change in mode is independent of data values or any intermediate behavior. Therefore, only the `change_mode` function affects `behavior`:

```
void clock::change_mode()
{
    static const mode next[] =
    {
        setting_hours, setting_minutes, displaying_time
    };
    behavior = next[behavior];
}
```

This particular implementation takes advantage of an `enum`'s implicit conversion to an integer, using it to lookup the next state in a table. More verbosely, a `switch` statement could have been used to achieve the same effect. More tersely, the next state can be calculated by incrementing the current one, wrapping around from last to first as necessary, because the state transition model forms a simple cycle.

The realization of `increment` is less open to alternatives:

```
void clock::increment()
{
    switch(behavior)
    {
    case displaying_time:
        break;
    case setting_hours:
        hour = (hour + 1) % 24;
        break;
    case setting_minutes:
        minute = (minute + 1) % 60;
        break;
    }
}
```

The empty `case` for `displaying_time` is included for completeness, demonstrating statewide coverage for the event explicitly. The event response for `tick` is complementary:

```
void clock::tick()
{
    switch(behavior)
    {
    case displaying_time:
        if(++second == 60)
        {
            second = 0;
            if(++minute == 60)
            {
                minute = 0;
                hour   = (hour + 1) % 24;
            }
        }
        break;
```

```
    case setting_hours:
        break;
    case setting_minutes:
        break;
    }
}
```

There is a temptation to tow an orthodox object-oriented hard-line and come down against flags and `switch`es, assuming them to be an indicator of deficient design. However, this line is a narrow and not always convincing one. For the scope and scale of the given problem almost any other solution not based on explicit selection will be longer and more intricate.

However, if we imagine a slightly broader version of the problem, the context shifts and the flag and `switch` solution becomes increasingly — indeed, in terms of lines of code, exponentially — inappropriate. Consider adding an alarm feature to the clock, and perhaps a date feature. Taking it further, a digital watch typically offers all these features and more: stopwatch, multiple time zones, phone book, etc. The FLAGS FOR STATES design will collapse under its own weight, acquiring the flexibility of a block of concrete, the cohesiveness of loose sand, and the plot comprehensibility of a telephone directory. Such consequences would justify the pursuit of alternative approaches.

Objects for States in Ruby

The OBJECTS FOR STATES pattern† (see *Appendix*) offers itself up as a likely candidate. The organizing principle behind the pattern is the introduction of an object to represent the behavior of the main object — the clock in this case — in each mode. The behavioral object offers a method for each event the main object can respond to — in this case `change_mode`, `increment`, and `tick`.

Here is a sketch of this from the main object's perspective in Ruby:

```
class Clock
    def initialize(hour, minute, second)
        @now      = TimeOfDay.new(hour, minute, second)
        @behavior = DisplayingTime.new
    end
    def change_mode
        @behavior = @behavior.change_mode(@now)
    end
    def increment
        @behavior = @behavior.increment(@now)
    end
    def tick
        @behavior = @behavior.tick(@now)
    end
end
```

All the responsibility for behavior is forwarded to the behavior object. For each mode of behavior there is a class that implements the same method interface, namely `change_mode`, `increment`, and `tick`. Explicit `switch`-like selection has been replaced with runtime polymorphism, and each delegated method selects the next behavior. When there is a state

† OBJECTS FOR STATES is also known as the STATE pattern, but this name is misleading and not particularly descriptive. It is misleading because it is often mistaken as definitive: *the* state pattern. It is not particularly descriptive because there is no suggestion of the solution structure in the name, just a hand-waiving reference to the problem. The OBJECTS FOR STATES name is listed [Gamma+1995] as a synonym for STATE. It more accurately captures the pattern's intent and structure, and should be preferred.

transition the delegated method will return the mode object for the new state; when there is no transition, the delegated method simply returns `self`. The behavior for each mode has been localized and encapsulated rather than scattered across different `case`s in many functions. The reason for passing the `@now` instance variable becomes apparent when you consider that the behavioral objects need to work with their context, so they must also affect the state of their associated `Clock` object's data. The type of the current time is a simple data structure with fields for the current hour, minute, and second:

```
TimeOfDay = Struct.new(:hour, :minute, :second)
```

Objects are handled by reference rather than by copy, so the behavioral objects modifying the passed data object affect the state of the main clock object. Keeping to a fairly conventional OO style:

```
class DisplayingTime
    def change_mode(time_of_day)
        SettingHours.new
    end
    def increment(time_of_day)
        self
    end
    def tick(time_of_day)
        if (time_of_day.second += 1) == 60
            time_of_day.second = 0
            if (time_of_day.minute += 1) == 60
                time_of_day.minute = 0
                time_of_day.hour   = (time_of_day.hour + 1) % 24
            end
        end
        self
    end
end

class SettingHours
    def change_mode(time_of_day)
        SettingMinutes.new
    end
    def increment(time_of_day)
        time_of_day.hour = (time_of_day.hour + 1) % 24
        self
    end
    def tick(time_of_day)
        self
    end
end

class SettingMinutes
    def change_mode(time_of_day)
        DisplayingTime.new
    end
    def increment(time_of_day)
        time_of_day.minute = (time_of_day.minute + 1) % 60
        self
    end
    def tick(time_of_day)
        self
    end
end
```

The behavioral object is sometimes also known as the *state object*, but this name is confusing: as you can see from the code, the state object is often stateless. In this case, because the state object is stateless and immutable, a single instance for each concrete class will suffice, avoiding the need for dynamic object creation. The appropriate solution in this case is simply to use a class variable. There is a temptation to use a SINGLETON [Gamma+1995], but this temptation should be resisted. In this case, as in many others, it overcomplicates the design. The use of a single, shared instance for each mode is the business of the `Clock` class, not the mode's, and an ordinary class variable is both simple and sufficient.

Another temptation that would offer little in return would be to introduce a common superclass for the `DisplayingTime`, `SettingHours`, and `SettingMinutes` classes. The classes currently share the same implicit interface, but no effort is made to factor out any common behavior. In some designs this may make sense, but there is little to be gained by adding an extra class to the current example. The only duplicated code is the empty `tick` method in both the `SettingHours` and the `SettingMinutes` classes. Such light and incidental duplication of nothingness does not really warrant extracting a superclass.

The implementation of the state model using objects and polymorphism is elegant, albeit longer than the explicit-conditional approach. Its broader benefits are not as easily discernible as they would be in a more extensive state model. A relatively large community of specific classes and methods seems to have sprung up to solve a comparatively simple problem.

Forces

History-sensitive method behavior implies that additional object state is required to track the current mode. This state needs to be clear and easy to manage. Handling the additional state should also not weigh down the implementation of each method with additional complexity.

Simple lifecycle models can be implemented in terms of flags and conditional statements in each method; an approach whose code comprehensibility scales poorly. More sophisticated modal behavior can be realized through object delegation, drawing on a community of dependent classes to express the behavior, typically using OBJECTS FOR STATES [Gamma+1995]. However, the larger the community, the more pronounced coupling and comprehensibility problems become. If only a single method is history sensitive, elaborating a whole class hierarchy certainly seems like overkill.

Particular method behaviors may be shared across different methods in different states: for example, methods that do nothing or methods that trigger a particular event, such as an exception. Avoiding duplicate code is generally considered a good — indeed, fundamental — practice. Common behavior can be factored out into private methods and called from the relevant `case` when using FLAGS FOR STATES. Alternatively, a class hierarchy, based typically on nesting of states, allows common method implementations to be pulled up the hierarchy. However, this does not accommodate behaviors that are common but not related so simply, crosscutting the hierarchy.

Some developers have made the mistake of assuming that OBJECTS FOR STATES is the "one true way" to realize state models in code. This is partly because of its inclusion in *Design Patterns* [Gamma+1995] as the only object lifecycle pattern and partly because of its branding as *the* STATE pattern. The pattern is powerful and certainly not trivial; applying it

uniformly to code as a cure-all for all state models is sure to complicate the source and confuse the reader.‡

The benefit of a flag-based approach is that all the behavior is defined in a single class rather than across many. Access to the context of the main object is also simple: ordinary methods have such direct access, which is generally not the case for separate objects. However, the effect on each individual method is to obfuscate rather than clarify intent. Each history-sensitive method suffers from strong control coupling to the flag variable.

What is needed of a solution is the ability to express and select each different behavior simply, without interference from explicit conditional statements or separation across multiple classes.

Solution

Represent an object's mode as a simple data structure containing references to methods. Each history-sensitive public method of the object forwards a call, along with any arguments received, to a corresponding entry in the data structure. Each different behavior for the object is implemented as its own private method. Each mode is associated with its own data structure instance, which holds references to the relevant private methods.

The method references may be true direct method references, such as member function pointers in C++, or they may take the form of the symbolic method names that are resolved using reflection.

The data structure holding the method references can be a record-like data structure with named fields, such as a C++ `struct`. Alternatively, a dictionary object can be used to look up the private method reference corresponding to each history-sensitive public method. In effect, this configuration emulates the normal method lookup table (*vtable*) mechanism, with a little added customization, evolution, and intelligence. Where only a single public method is history sensitive, no intermediate data structure is needed to represent the mode: a single method reference will suffice. Global, module, or class-wide variables can be used to hold the single instance of the data structure (or method reference) required for each mode.

Resolution

Returning to the clock example, the forces occurring in both the C++ and Ruby designs can be resolved conveniently and idiomatically with METHODS FOR STATES.

Methods for States in C++

In C++ the design can be modified so that each public member function forwards its call to the relevant member function pointer in a `struct` indicative of the current mode:

```
class clock
{
public:
    void change_mode()
    {
        (this->*(behavior->change_mode))();
    }
```

‡ Misapplication of OBJECTS FOR STATES only presents itself as a force as a consequence of common programmer design pattern knowledge: a few years ago, before *Design Patterns* became widely read as an OO design book, it would not be considered a force; it is possible that the same may be true in the future for different reasons.

```
    void increment()
    {
        (this->*(behavior->increment))();
    }
    void tick()
    {
        (this->*(behavior->tick))();
    }
private:
    typedef void (clock::*function)();
    struct mode
    {
        const function change_mode, increment, tick;
    };
    static const mode displaying_time
    static const mode setting_hours;
    static const mode setting_minutes;
    const mode *behavior;
    int hour, minute, second;
    ...
};
```

A palette of suitable behavior is provided by private member functions:

```
class clock
{
    ...
private:
    ...
    template<const mode *next_mode>
    void change_to()
    {
        behavior = next_mode;
    }
    void next_hour()
    {
        hour = (hour + 1) % 24;
    }
    void next_minute()
    {
        minute = (minute + 1) % 60;
    }
    void update_time()
    {
        if(++second == 60)
        {
            second = 0;
            if(++minute == 60)
            {
                minute = 0;
                hour   = (hour + 1) % 24;
            }
        }
    }
    void do_nothing()
    {
    }
};
```

And the rest is down to initialization, letting the data do the work:

```
const clock::mode clock::displaying_time =
{
    &clock::change_to<&setting_hours>, &clock::do_nothing, &clock::update_time
};
const clock::mode clock::setting_hours =
{
    &clock::change_to<&setting_minutes>, &clock::next_hour, &clock::do_nothing
};
const clock::mode clock::setting_minutes =
{
    &clock::change_to<&displaying_time>, &clock::next_minute, &clock::do_nothing
};
```

Methods for States in Ruby

To implement the clock example in Ruby, class variables — prefixed with `@@` — and `Struct` objects are used:

```
Mode = Struct.new(:increment, :tick)

class Clock
    @@displaying_time = Mode.new(:do_nothing, :update_time)
    @@setting_hours   = Mode.new(:next_hour, :do_nothing)
    @@setting_minutes = Mode.new(:next_minute, :do_nothing)
end
```

Method calls are forwarded by accessing the appropriate method name from the corresponding `Struct` attribute, and resolving it against the current object:

```
class Clock
    def initialize(hour, minute, second)
        @hour, @minute, @second = hour, minute, second
        @behavior = @@displaying_time
    end
    def change_mode
        send(@behavior.change_mode)
    end
    def increment
        send(@behavior.increment)
    end
    def tick
        send(@behavior.tick)
    end
end
```

A lookup table simplifies the `change_mode` method, so that only a single implementation is required:

```
class Clock
    @@mode_changes =
    {
        @@displaying_time => @@setting_hours,
```

```
        @@setting_hours   => @@setting_minutes,
        @@setting_minutes => @@displaying_time
    }
    def next_mode
        @behavior = @@mode_changes[@behavior]
    end
    def next_hour
        @hour = (@hour + 1) % 24
    end
    def next_minute
        @minute = (@minute + 1) % 60
    end
    def update_time
        if (@second += 1) == 60
            @second = 0
            if (@minute += 1) == 60
                @minute = 0
                @hour   = (@hour + 1) % 24
            end
        end
    end
    def do_nothing
    end
end
```

Because the underlying implementation of `change_mode` is the same no matter what the mode, i.e. `next_mode`, it need not participate in the dynamic lookup. It has therefore been excluded for the method referencing structure.

Consequences

Using METHODS FOR STATES allows a class to express all of its different behaviors in ordinary methods on itself. It achieves its behavior without either the control coupling and reduced readability of large conditional statements or a large supporting cast of ancillary classes.

Indirection, based on referencing methods as objects in their own right, is used both to represent the state and to dispatch from a public method request to the correct underlying method for the current mode. The overall control flow and effect is that of DOUBLE DISPATCH (see *Appendix*). In terms of performance, METHODS FOR STATES requires an additional two levels of indirection to resolve a method call.

The behavior of the class's objects is fully encapsulated within the class, the same unit of code to which the behavior is coupled, rather than fragmented across multiple small classes. The only additional data type required is for the data structure holding the method references, which may be defined as a nested or module-level type, or may exist as an associative collection, or may be nothing more than a single method reference. There is no proliferation of classes, nested or otherwise. On the other hand, if there are many distinct behaviors, the main class may end up far longer than was intended or is manageable — a shopping list of different options.

Each distinct behavior is assigned its own method. This allocation of responsibility is clearer and more cohesive than asking the reader or class author to wade through potentially large rambling selection statements. The independence of public methods from their runtime implementation also allows more fluidity in the modes and their transitions, and improves opportunities for sharing of common behavior between different modes:

- `switch` statements are notoriously tedious and error prone for such extension — especially in the large — leading to both bugs and duplicate code.
- Representing modes in a class hierarchy supports simple addition of new modes, but allows convenient sharing of common implementation only between a macrostate and its superstate. This relationship is mirrored in the class hierarchy, so method sharing across unrelated states is inconvenient.

With METHODS FOR STATES, common behavior is capitalized on more readily than in other designs, whether in the form of doing nothing or changing mode in a consistent fashion. The more that event responses between different modes overlap, the more suitable this pattern becomes.

The potential independence of public methods from the underlying private methods means that method names cannot always be relied upon to indicate the context of calling. The initialization of each mode's data structure instance must be read to understand what methods are used in what modes and to what end.

Other than the one-off initialization of each mode's corresponding lookup data structure, no additional object creation is needed to support METHODS FOR STATES. The data structure instances can be initialized at the earliest opportunity — program startup or class load time — and remain unchanged. Because they are immutable the sharing is intrinsically thread-safe.

No object context needs to be passed around because ordinary methods can already see the internal state of the object they represent the behavior of. This directness makes the development and comprehension of each specific method behavior simpler: no additional arguments are required for the underlying methods; no tricks with indirection or data visibility are required to deal with whatever restrictions the language places on an object's access to the internals of another. Although the use of closure-based objects — such as Java's inner classes — simplify the context-access issues from one dependent object on another, such an approach requires a great deal more object creation to work; a one off at-startup initialization will not do the trick.

METHODS FOR STATES is most suited to languages that support simple method references and resolution. For example, in C++ a member function pointer is resolved against an object pointer using the `->*` operator. Similarly, in C plain function pointers support the simple implementation of this pattern in a non-object-oriented context. In Ruby a method's symbolic name can be invoked on an object using the `send` method that is common to all objects. In contrast, Java's reflection mechanism requires more gymnastics — and correspondingly more obscurity and less convenience — to call a method on an object given its name as a string. C# could be said to suffer a similar problem, but it also has features that support a simple enough workaround: `delegate`s and `static` methods. Instead of expressing the behaviors as ordinary private methods, they can be expressed as private `static` methods that take an additional argument to an instance of the class, effectively emulating the `this` reference implicit in ordinary methods. For `static` methods, `delegate` variables behave much as function pointers do in C, allowing a cleaner and more efficient implementation of METHODS FOR STATES than is possible with reflection.

METHODS FOR STATES can often provide a simpler and more manageable alternative to OBJECTS FOR STATES (see *Appendix*), a pattern with similar intent but markedly different structure and options. Incorporated into existing state pattern languages [Dyson+1998, Yacoub+2000], METHODS FOR STATES would expand the vocabulary, options, and range of designs available to a programmer beyond the narrower set offered by an OBJECTS FOR STATES view of state machines. Where per-mode state is needed, OBJECTS FOR STATES offers a more cohesive design than trying to shore up METHODS FOR STATES with extra optional variables. In languages, such as Java, that require awkward code to realize METHODS FOR

STATES, or that lead to the creation of further objects and classes, an OBJECTS FOR STATES solution is often preferable.

METHODS FOR STATES is not a viable substitute in situations that better suit COLLECTIONS FOR STATES (see *Appendix*), but it can be used as a complement. COLLECTIONS FOR STATES groups multiple objects together according to their mode, representing the concept of the mode extrinsically. Objects in the same mode are held in the same collection, reducing the need for intrinsic mode representation. However, where COLLECTIONS FOR STATES is being applied as a speed optimization, e.g. acting collectively on modal objects that already have an internal lifecycle model, retaining METHODS OF STATES internally may still make sense.

Appendix

The following table lists thumbnails for patterns external to this paper that are related in some way to METHODS FOR STATES. OBJECTS FOR STATES provides a common alternative to METHODS FOR STATES, and vice-versa. Similarly FLAGS FOR STATES can be used in limited scenarios where either METHODS FOR STATES or OBJECTS FOR STATES might be regarded as overkill. COLLECTIONS FOR STATES is a complementary pattern: it may be applied in its own right or in conjunction with METHODS FOR STATES, OBJECTS FOR STATES, or FLAGS FOR STATES, but not as a substitutable alternative. DOUBLE DISPATCH describes the most generalized form of the dispatch used in the heart of METHODS FOR STATES.

Name	*Problem*	*Solution*
COLLECTIONS FOR STATES [Henney1999]	A number of objects are managed and held in a collection, and operated on according to their common state. What is a suitable expression of the state with respect to each object?	Represent each state of interest with a separate collection that refers to all objects in that state. State transitions become transfers between collections.
DOUBLE DISPATCH [Beck1997]	How can you select a method based on the type of the target and the type or value of one other variable without hardwiring the selection as a conditional statement?	Delegate the selection of the actual method via a helper object that then calls back on the main object. The type of the helper object determines which method is selected. The helper object is normally the other variable in the interaction.
FLAGS FOR STATES	How can an object significantly change its behavior for only a couple of methods based on only one or two alternative internal states?	Represent the behavioral state of the object explicitly using a flag. In each of the history-sensitive methods, use a conditional to check the flag and act accordingly.
OBJECTS FOR STATES [Gamma+1995, Dyson+1998]	How can an object significantly change its behavior, depending on its internal state, without hardwired multi-part conditional code?	Separate the behavior from the main class, which holds the context, into a separate class hierarchy where each class represents the behavior in a particular state. Method calls on the context are forwarded to the mode object.

Acknowledgements

This paper is derived, in part, from a previously published article [Henney2002].

I would like to thank Pascal Costanza for his excellent shepherding and his patience, Jon Jagger for his additional comments on the preconference version of the paper, and to the participants in the writer's workshop at VikingPLoP 2002: Mikio Aoyama, Walter Cazzola, Lars Grunske, Juha Parssinen, Michael Pont, and Kristian Elof Sørensen.

References

[Beck1997] Kent Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, 1997.

[Dyson+1998] Paul Dyson and Bruce Anderson, "State Patterns", *Pattern Languages of Program Design 3*, edited by Robert Martin, Dirk Riehle, and Frank Buschmann, Addison-Wesley, 1998.

[Gamma+1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[Henney1999] Kevlin Henney, "Collections for States", *Proceedings of the 4th European Conference on Pattern Languages of Programs*, 1999, `http://www.curbralan.com`.

[Henney2002] Kevlin Henney, "State Government", *C/C++ Users Journal* C++ Experts Forum, June 2002, `http://www.cuj.com/experts/2006/henney.htm`.

[Yacoub+1998] Sherif M Yacoub and Hany H Ammar, "Finite State Machine Patterns", *Pattern Languages of Program Design 4*, edited by Neil Harrison, Brian Foote, and Hans Rohnert, Addison-Wesley, 2000.