Selected Patterns for Implementing Finite State Machines

Paul Adamczyk
University of Illinois at Urbana-Champaign
Department of Computer Science
email: padamczy@uiuc.edu

INTRODUCTION

This paper describes design patterns for writing Finite State Machines (FSMs) by hand. While there exist many techniques for generating code of FSMs, software engineers still implement their own simple state machines from the ground up. And each time they face the same challenges – trying to write efficient code while preserving the structure of the FSM for maintainability. In many instances, software engineers have no access to tools that generate FSM code or find them too generic and not appropriate for their work. It is often easier to implement simple, tailor-made state machines manually, especially with the aid of good design patterns. There are quite a few patterns available, which poses a difficulty, because it is not easy to determine which pattern fits best. This paper attempts to address this problem by comparing well-known FSM patterns directly, using the same criteria and illustrated with the same example. Another problem is that FSMs are multi-dimensional entities – e.g. modifying a state likely means changing events and state transitions and all of these changes require modifications of different places in the code – but the code that implements them is a one-dimensional sequence of statements. To address this problem, the description of each pattern focuses on only one dimension of the FSM (e.g. state, event, action) and discusses it in the context of a complete FSM.

This paper builds on the author's previous work on classifying design patterns for FSMs [Ada03]. Patterns described in this paper provide different solutions depending on the reader's answers to the following questions: How frequently is the FSM implementation going to change? Does the FSM implementation need to be efficient? How much state-specific data is needed? How complex is the transition function (i.e. is input event and current state sufficient or is there coupling to something outside the FSM)? Does the FSM definition have to change at run-time?

Many other FSM-related questions are beyond the scope of this work. This paper is not concerned with comparing advantages offered by specific programming languages for implementing FSMs. All patterns are implemented in C++, but if other languages provide a better solution, they are mentioned as well. This paper does not address the advantages and disadvantages of implementing FSMs by hand rather than by using code generators. Also, all patterns described in this paper apply to a single FSM. Interactions between multiple FSMs produce many additional patterns, which are not discussed in this paper. See "Conclusion and Future Work" section for more on this.

This paper contains a subset of patterns described in a larger body of work. The most recent version of complete work can be found at *http://pinky.cs.uiuc.edu/~padamczy/fsm*.

MOTIVATING EXAMPLE

Consider the traffic lights at an intersection[1]. The intersection could be modeled as an FSM. However, it is easier to implement it as a composition of FSMs. A simple, two street intersection, for example, is easier to model with two FSMs, one to control the lights on the main street and another one for the lights on the side street. Assume that both lights are equipped with sensors to detect the incoming traffic. The FSMs handle the input generated by the sensors and communicate with each other by exchanging messages.

Picture from www.joe-ks.com.

Figure 1 illustrates a typical sequence of events handled by these FSMs. The main light is initially green and the side light is red, as indicated by the black circles. When the sensor of the side light FSM detects that the level of incoming traffic exceeds a predefined threshold, it sends a signal (IncomingTraffic) to the side light FSM. In response, the side light FSM sends a SideLight_GetGreen message to the main light requesting a permission to turn green. If the sensor of the main light does not report any incoming traffic, the message from the side light FSM causes the main light to change to yellow light, and then (after the YellowLightTimeout occurs) to red light. At this point, the main light responds to the side light with a message (MainLight_TurnedRed) indicating that it has turned red. The message makes the side light change from red to green.[2] A similar sequence of events occurs when the main light determines that it should turn to green when it receives the IncomingTraffic signal from its sensor or when RedLightTimeout occurs.

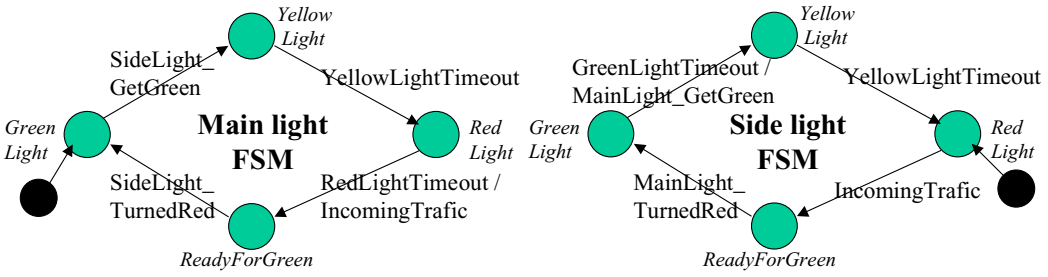

Figure 1. Traffic lights FSMs – main light (left) and side light (right).

[1] The traffic light model used here is borrowed from [Spa00].

[2] Note that this traffic light follows the American model. In Europe, for example, yellow light is displayed also on change from red to green light, resulting in the sequence: green, yellow, red, red and yellow, green.

In Figure 1, **states** are depicted as circles, **transition functions** are arrows that show the direction of the transitions and are labeled with the **inputs** causing them. The main light FSM has four **states** (GreenLight, YellowLight, RedLight, and ReadyForGreen). ReadyForGreen is a logical state that does not correspond to a physical light. It indicates that the main light is still red, but it will turn green as soon as the side light turns red. The **input alphabet** of the FSM consists of five entries (YellowLightTimeout, RedLightTimeout, IncomingTraffic, SideLight_TurnedRed, and SideLight_GetGreen). SideLight_TurnedRed indicates that the side light FSM has just turned red. Inputs with names prefixed with "SideLight_" are generated by the side light FSM. The other three inputs are generated by main light FSM internally (by its sensors or by the timer). The **transition functions** map all inputs to state changes, shown by arrows in Figure 1. The same information, in a tabular form is presented in Table 1.

	GreenLight	**YellowLight**	**RedLight**	**ReadyForGreen**
SideLight_GetGreen	YellowLight	I	I	E
YellowLightTimeout	E	RedLight	E	E
RedLightTimeout	E	E	ReadyForGreen	E
IncomingTraffic	I	I	ReadyForGreen	I
SideLight_TurnedRed	I	I	I	GreenLight

Table 1. State transitions of the main light FSM

In the table, **states** are listed as columns, **inputs** are listed as rows and each pair of state and input corresponds to next state. For example, if the main light FSM is in GreenLight state, it can only transition to the YellowLight state, but only after receiving SideLight_GetGreen event. For each state, the main light FSM transitions to a different state in response to at most two inputs. The other inputs are either ignored (marked with 'I'), because they don't give the FSM any new information or they indicate errors (marked with 'E').

The error combinations of states and events should not occur. For example, it is not possible to receive the RedLightTimeout event in the GreenLight state, because when the FSM is in the GreenLight state the timer is not running. Note that from the RedLight state, the main light FSM will transition to the ReadyForGreen state on two internal inputs – when its timer expires or when an incoming traffic arrives. In addition to the state transition (to ReadyForGreen), the main light FSM also sends MainLight_GetGreen message to the side light FSM.

The side light FSM consists of the same four states (GreenLight, YellowLight, RedLight, and ReadyForGreen). Its **input alphabet** contains a set of inputs similar to the main light FSM. One difference is that it receives MainLight_GetGreen and MainLight_TurnedRed events originated by the main light. Moreover, it receives GreenLightTimeout rather than RedLightTimeout. This is because the side light is red by default and it stays green only for a specific period of time. The GreenLightTimeout event is generated when that time expires. The **transition functions** map all inputs to states according to Table 2 below, following the same rules as in Table 1.

	GreenLight	YellowLight	RedLight	ReadyForGreen
MainLight_GetGreen	YellowLight	I	I	E
GreenLightTimeout	YellowLight	E	E	E
YellowLightTimeout	E	RedLight	E	E
IncomingTraffic	I	I	ReadyForGreen	I
MainLight_TurnedRed	I	I	I	GreenLight

Table 2. State transitions of the side light FSM

To implement the traffic light FSMs, it is not sufficient to define **states**, **input alphabet**, and **transition functions**, because the FSMs are more complex. The processing of an input message may result in generating more input messages (e.g. YellowLightTimeout in the main light FSM generates MainLight_TurnedRed input to the side light FSM). To make adding such code possible, the software models of FSMs usually define two more elements – actions and FSM data. **Action** is a sequence of steps performed by the FSM upon the reception of a specific input in a specific state. **FSM data** is additional information stored by the FSM. This data can arrive along with the input (e.g. SideLight_GetGreen message could also contain the number of cars waiting for the green light) or can be generated internally by the FSM (e.g. the amount of time that the light is green per hour). These two elements complement the three original components of the FSM – states, input alphabet (henceforth called **events**), and transition functions – to complete its definition.

CODE FOR THE TRAFFIC LIGHTS EXAMPLE

The two traffic light FSMs described above present simplified models of traffic lights. But even in this simple form, they are not trivial to implement. Let's consider only one of the FSMs, the main light. The code below is a procedural-style implementation of the FSM in C with some details omitted for brevity. For example, there is no distinction between ignorable (I) and erroneous (E) events. Both types are considered "unexpected" events.

```
//////////////////////
// main_light_fsm.h //
//////////////////////
enum FSMEvent{
  SideLight_GetGreen,
  YellowLightTimeout,
  RedLightTimeout,
  IncomingTraffic,
  SideLight_TurnedRed
};

enum FSMState{
  GreenLight,
  YellowLight,
  RedLight,
  ReadyForGreen
};

enum FSMTimer{
  YellowLightTimer,
  RedLightTimer
};

#define YELLOW_LIGHT_TIME 1.2 /* seconds */
#define RED_LIGHT_TIME 20.0

// Functions defined
int handleUnexpectedEvent(FSMEvent input);
```

```
int setTimer(FSMTimer timer);
int timeout();
int processEvent(FSMEvent input);


//////////////////////
// main_light_fsm.c //
//////////////////////
FSMState currentState = GreenLight;

int handleUnexpectedEvent(FSMEvent input){
  // collect data about erroneous event and store it persistently
  return SUCCESS;
}

int setTimer(FSMTimer timer){
  switch(timer){
    case YellowLightTimer:
      // pass YELLOW_LIGHT_TIME to the timer facility
      return SUCCESS;
    case RedLightTimer:
      // pass RED_LIGHT_TIME to the timer facility
      return SUCCESS;
    default:
      // log data about erroneous timer and store it persistently
  }
  return FAILURE;
}

int timeout(){
  // invoked by the timing facility
  switch(currentState){
    case YellowLight:
      return processEvent(YellowLightTimeout);
    case RedLight:
      return processEvent(RedLightTimeout);
    default:
      // log the erroneous timeout and store it persistently
  }
  return FAILURE;
}

int processEvent(FSMEvent input){
  switch(currentState){
    case GreenLight:
      switch (input){
        case SideLight_GetGreen:
          // side light FSM requested light change - transition to
          // yellow light state if there is no incoming traffic
          // on the main road
          currentState = YellowLight;
          return setTimer(YellowLightTimer);
        default:
          return handleUnexpectedEvent(input);
        }
      break;
    case YellowLight:
```

```
      switch (input){
        case YellowLightTimeout:
          currentState = RedLight;
          // send MainLight_TurnedRed input to the side light FSM
          sideLightFsm->processEvent(MainLight_TurnedRed);
          return setTimer(RedLightTimer);
        default:
          return handleUnexpectedEvent(input);
        }
      break;
    case RedLight:
      switch (input){
        case RedLightTimeout:
        case IncomingTraffic:
          currentState = ReadyForGreen;
          // send MainLight_GetGreen input to the side light FSM
          sideLightFsm->processEvent(MainLight_GetGreen);
          return SUCCESS;
        default:
          return handleUnexpectedEvent(input);
        }
      break;
    case ReadyForGreen:
      switch (input){
        case SideLight_TurnedRed:
          // side light is now red, switch to green light
          currentState = GreenLight;
          return SUCCESS;
        default:
          return handleUnexpectedEvent(input);
        }
      break;
    default:
      // log data about invalid state and store it persistently
      break;
  }
  return FAILURE;
}
```

The work in the code above is performed primarily in the **`processEvent()`** function, which uses the incoming input **`event`** and the **`currentState`** to determine which code to execute. On the surface, it appears that this implementation is extensible. The function uses nested switch statements although the second level of switches handles at most two events and could have been implemented as an if statement. Don't be fooled. This code is likely to break on the first attempt to extend it, because any change requires updating multiple places in the code.

Let's focus on the implementation of each FSM element. Both **states** and **events** are enums. It is easy to add new states and events, but adding the code corresponding to each new value is difficult. Each new state requires new definition in the header file, a new case in the switch statement (possibly also in the **`timeout()`** function), and modifications of existing cases to add transitions to the new state. Adding a new event also requires an update of the header file and adding new cases to (potentially all) second levels of switch statements in the **`processEvent()`** function. **Actions** are not structured at all, they are

blocks of code corresponding to event/state pairs. Although **`setTimer()`**and **`handleUnexpectedEvent()`** functions seem to provide some code reuse, all actions could be implemented in a more structured manner (e.g. as separate functions). This implementation of the main light FSM does not contain any **FSM data**, but if it did, the data would most likely be stored in a global struct. Lastly, **state transitions** are implemented inside actions in a somewhat haphazard way. The code does not enforce that the state change occur as the first or the last step of the processing, or perhaps somewhere in the middle. In short, this code will work well only as long as there are no changes to the FSM.

But this FSM does not implement many features of traffic lights. Before it can be useful, it needs to be extended. Here are some examples. To support left turn lights, two more states (**`GreenLeftArrow`**, **`YellowLeftArrow`**) and two more events (**`Yellow-`** and **`GreenLeftArrowTimeout`**) should be added. Lights for pedestrians require new states (**`Walk`**, **`DoNotWalk`**) to exist concurrently with **`GreenLight`** state and implementing them means adding two new events (**`WalkLightRequested`** and **`WalkLightTimeout`**). Mechanical failures that cause traffic lights to go out of service also require a new state (**`BlinkingRed`**) and a new event (**`MechanicalFailure`**) that all states need to handle in a uniform way. Supporting night-time lights requires adding **`BlinkingYellow`** state for the main light FSM (the side light FSM could reuse the **`BlinkingRed`** state). The list is long. So, what could be done to make this code capable of changing easily?

Many flexible FSM designs enable a graceful incorporation of such changes. This paper explores some of them in detail. Features of traffic lights mentioned above serve as examples throughout this paper.

OVERVIEW OF THE PATTERNS

The remainder of this paper describes different patterns for implementing FSMs. Each pattern focuses on a single FSM element. Patterns are divided into four categories corresponding to four FSM elements: states, events, actions, and state transitions[3]. Each category is self-contained and different FSM elements are analyzed independently. In a simple FSM, it is easy to separate its elements and to analyze them individually. In a more complex one, interdependencies between elements make the distinction less clear. But even in such cases describing separate elements is useful, because it allows comparing FSMs by focusing on individual elements.

The first set of patterns describes ways to implement states. The **state** is the essential and also the easiest to understand part of FSM, because it summarizes the externally visible characteristic of the FSM at a point in time (e.g. the main light FSM is in green light state now). This paper presents three ways to implement states: as enumerated values, methods, and classes. If an FSM implements its states as enumerated values, then other entities (events or actions) are responsible for changing the value of the current state. If each state has its own method, then all processing associated with that state (including state change)

[3] The fifth element, FSM data, does not have any specific patterns

can be encapsulated in the method. Implementing a state as a class offers an even higher level of encapsulation of state-specific processing.

The second set of patterns describes ways to implement events. **Events** are inputs that the FSM can recognize and act upon. A combination of the current state and the incoming event dictates the specific behavior (action) that the FSM needs to perform. One way to visualize the relationship between states and events is to see them as rows and columns in a two-dimensional array. Another way is to see each state as an array of functionality and events as indices to the array. Conversely, it is also possible to consider each event as an array of functionality and states as indices into the array. Figure 2 illustrates these relationships in more detail. In (a) states and events are indices into a complete array. In (b) each state is a separate entity (a method or a class) whose behavior depends on the index provided by the corresponding event. In (c) state acts as an index into the event array, but there are individual arrays for each event.

(a)

	State 1	**State 2**	**State 3**	**State 4**
Event 1	New state	New state	New state	New state
Event 2	New state	New state	New state	New state
Event 3	New state	New state	New state	New state

(b)

	Event 1	**Event 2**	**Event 3**
State 1	New state	New state	New state

(c)

	State 1	**State 2**	**State 3**	**State 4**
Event 1	New state	New state	New state	New state

Figure 2. Relationship between states and events.

Because of these similarities, events, like states, can be implemented as enumerated values, methods, and classes. The major difference between them is that states are internal to the FSM and events are usually generated by the external entities (but they can also be generated internally by the FSM).

The third set of patterns describes ways to implement **actions**, the behavior executed in response to the reception of an event in a specific state. In simple FSMs, an action is a single statement that performs a state transition by modifying the value of the variable `currentState`. As such, it can be implemented anywhere in the code. At the other end of the spectrum, it is possible to build complex actions and to combine sequences of actions to implement other actions, which suggests implementing them as classes. This paper describes three patterns for implementing actions: as unstructured code, methods, and classes. Unstructured actions are implemented as part of state- or event-specific code. Action methods are named after the actions they perform and contain the code to perform the action. Action class encapsulates the processing of an action inside a class, which is named after the action.

The last set of patterns describes ways to implement **state transitions**, methods to change the current FSM state. In many FSMs, state transitions are performed by actions. Only more complex FSMs separate actions and transitions. Three patterns for state transition definitions are table, state-driven transition, and class. Transition table is a two-dimensional table mapping current state and incoming event to a new state. State-driven

transition describes ways to combine state-changing with other FSM behavior. Transition class encapsulates the transition table inside a class to enable its modifications at run-time.

Figure 3 below lists all the patterns discussed in this paper, grouped according to the element they describe. Pattern names shown in ***italics*** are described in this paper. Other patterns are summarized in the Appendix. Patterns are described starting with the simplest and evolving into more complex ones. The simplest designs from each category are used in the example FSM code in the introduction. The example uses ENUMERATED STATE, ENUMERATED EVENT, UNSTRUCTURED ACTIONS, and STATE-DRIVEN TRANSITION patterns.

State	Event	Action	Transition
Enumerated	Enumerated	Unstructured	Table
Methods	Methods	Methods	State-Driven
Class	***Class***	***Class***	***Class***

Figure 3. FSM patterns discussed in this paper. Patterns in italics are described in detail.

This table shows that there are 81 possible combinations of patterns. Fortunately most of them are not valid or not practical. The patterns described in this paper illustrate the most likely combinations and discuss which combinations are not valid.

Enumerated State

The main light FSM example, although written in C, could be easily adapted to an object-oriented C++ implementation by encapsulating all the functions above as methods in the class that implements the FSM. In the example code, states are defined as enums, which is sufficient for the purpose of that example. When an FSM has only few states, such as the main light FSM, defining a state as anything more complex than a simple data type seems wasteful. There is nothing more to a state than its value (or name). State attribute is simply a data value that indicates the current state of the FSM object. The least complex way to refer to a state is to assign it a symbol. Fortunately, compilers make our life easier by assigning numbers as symbols.

Problem
How to define states as simply as possible?

The FSM needs to address the following forces. FSM's behavior is simple; it consists of changing the value of the current state. Moreover, the FSM is small – it has a few states, responds to a few events, and is not likely to change, because all states (and possibly events) are already defined. Small memory footprint and optimal performance are the primary goals of the implementation.

Solution
Define states as enumerated values or integers. To check the value of the **`currentState`** variable, compare its value with a predefined set of constants.

```
enum FSMState{
  GreenLight,
  YellowLight,
  RedLight,
  ReadyForGreen
};
```

Side note on other languages
Note that defining constants as enums guarantees that only valid values of **`FSMState`** can be used, because the compiler performs the check. But not all languages support enums. A corresponding definition in Java would look as follows:

```
static final int S_MIN_STATE = -1;
static final int S_GREEN_LIGHT = 0;
static final int S_YELLOW_LIGHT = 1;
static final int S_RED_LIGHT = 2;
static final int S_READY_4_GREEN = 3;
static final int S_MAX_STATE = 4;
```

Two additional values, MIN and MAX are added to facilitate explicit checks to determine that a value of the state variable is larger than **`S_MIN_STATE`** and smaller than **`S_MAX_STATE`**.

An alternative way to encapsulate enums in Java is to implement a special class for them, as described in [Arm97]. Implementing them in a class ensures that the values, unlike in C++, are typesafe.

To associate a state with a specific behavior, use a switch statement, as in the example code in the introduction, or an if statement. The original example code shows the association of enumerated states and their corresponding behavior. The action is responsible for setting the next state. Also, the code has similar structure for all input events.

Consequences

+ *State definition is simple and concrete*. Each state has a well-defined name and a corresponding value. The compiler guarantees that the values are immutable (const), while names make code more readable. Imagine what would happen if states were implemented as numbers and a new initial state was added. Some (if not all) of the states would need to be renumbered (effectively, renamed) to preserve the ordering of states. Having both names and values for states solves this problem. If state number changes, the change is isolated to only one place, where the mapping of name to value is defined.

+ *Good performance*. Although the use of ENUMERATED STATE alone does not guarantee that the FSM implementation is fast, it is a good indicator that the FSM design strives to be efficient. The actual performance depends on the implementation of other FSM elements, but they are usually implemented efficiently as well. The section Building Complete FSMs lists the patterns for other FSM elements that provide this efficiency.

- *Changes of state definitions may affect compilation time*. States defined as ints and enums are usually stored in header files so that they can be included in multiple files. Modifying them is likely to result in recompiling many files. It is possible to avoid this problem by defining state values in implementation files instead, but that makes them harder to find without proper IDE support.

- *Not all compilers check for completeness of switching over enums*. It is helpful when compilers issue warnings if not all cases are handled in a switch, but not all compilers do so. But even if they do, having a default case in the switch eliminates the compiler's support, because all undefined cases are implicitly covered in the default case.

- *States are not modeled as independent entities*. This pattern separates the value of a state from the state behavior. The behavior is associated with other FSM elements. Most programming languages do not support associating behavior with enums or ints the way data and behavior is associated with objects. This separation makes understanding the behavior in each state harder limiting the applicability of this pattern to FSMs with small number of states that are not likely to change.

Building Complete FSMs

Complete FSMs that implement states as enums are usually small and implement other FSM elements with the minimum overhead as well. Actions usually follow UNSTRUCTURED ACTION pattern, while Events are defined using ENUMERATED EVENT or EVENT METHODS. In the first case, the implementation of the complete FSM looks exactly like the motivating example. In the second case, the monolithic

`processEvent()` function is broken into smaller methods, one for each event, which eliminates the second level of switch statements, but forces the Client to invoke different methods for each event.

In the context of ENUMERATED STATE, it does not make sense to implement events using EVENT CLASS for two reasons. The first is purely pragmatic. FSMs that use ENUMERATED STATE and UNSTRUCTURED ACTIONS are designed for simplicity, so events should also be designed as simply as possible. The second reason is aesthetic. Consider the code for mapping states to actions in UNSTRUCTURED ACTIONS. If this code were to be moved to the event class, there would only be one method in the event class. Having a class with one method is ugly and should only be done when necessary. It is not necessary in this case.

FSMs using ENUMERATED STATE are often found in applications that have small memory requirements, such as embedded applications.

State Methods

One of the shortcomings of the traffic light FSMs, as described in the motivating example, is that the same algorithm is used to change lights during the day as well as at night. But the night-time traffic is usually light and many traffic lights are not needed. More sophisticated traffic lights solve this problem by entering a night mode – the main light blinks yellow (informing the drivers to proceed with caution) and the side light blinks red (which is equivalent to a stop sign).

Adding this feature to the main light FSM requires a significant extension of the original design. The main light FSM needs to support new state, **`BlinkingYellow`**, and the side light FSM needs to support **`BlinkingRed`** state. Two new events, **`EnterNightMode`** and **`ExitNightMode`**, indicating to enter into the new state and to exit it, need to be added to both FSMs. To implement these changes, make states explicitly bound to their behavior by encapsulating the behavior of each state in a method.

Problem
How to add new states easily? How to add new behavior to existing states?

The FSM needs to address the following forces. The complete behavior of the FSM should be encapsulated in one class to avoid exposing its internals to the Clients. The FSM has a few states, responds to a few events, stores small amount of data, and has a relatively simple behavior. It is not likely to change extensively, but small-scale changes are almost certain to occur. Although good performance is essential, is should be achieved without sacrificing maintainability.

Solution
Implement the behavior of each state in a separate method of the FSM class.

Table 3 shows the state transition table for the main light FSM with the support for the night mode highlighted in red. Note that none of the existing transitions need to be modified. Instead, existing states are extended to support two new events, **`EnterNightMode`** and **`ExitNightMode`**, and a new state, **`BlinkingYellow`**, is added.

	GreenLight	YellowLight	RedLight	ReadyForGreen	*BlinkingYellow*
SideLight_GetGreen	YellowLight	I	I	E	*I*
YellowLightTimeout	E	RedLight	E	E	*E*
RedLightTimeout	E	E	ReadyForGreen	E	*E*
IncomingTraffic	I	I	ReadyForGreen	I	*I*
SideLight_TurnedRed	I	I	I	GreenLight	*I*
EnterNightMode	*BlinkingYellow*	*BlinkingYellow*	*BlinkingYellow*	*BlinkingYellow*	*E*
ExitNightMode	*E*	*E*	*E*	*E*	*GreenLight*

Table 3. Main light FSM with support for night mode. Changes made to Table 1 are shown in red italics.

To implement states as methods, encapsulate all behavior related to each state in one method. Since the behavior of each state depends on the incoming event, structure the

method so that it is easy to determine which event was received. In Table 3, there are two possible transitions out of the **GreenLight** state – one to **YellowLight** state and another (newly added) to **BlinkingYellow** state following the reception of **SideLight_GetGreen** and **EnterNightMode** events. Code below shows the state method for **GreenLight** state that defines the handling of these two events.

```
void MainLightFSM::greenLight(FSMEvent incomingEvent){
  switch (incomingEvent) {
    case SideLight_GetGreen:
      // Side light FSM requested light change - transition to
      // yellow light state if there is no incoming traffic on
      // the main road.
      currentState = &MainLightFSM::yellowLight;
      setTimer(YellowLightTimer);
      break;
    case EnterNightMode:
      // If the sensor does not report incoming traffic,
      // change the light to blinking yellow
      if (!isIncomingTraffic()){
        currentState = &MainLightFSM::blinkingYellowLight;
      } else { // if not, try again, later
        setTimer(RetryNightMode);
      }
      break;
    // other events
}
```

Note that the code has an additional condition for changing the lights to the night mode. In the case of heavy traffic, more checks are required to ensure that the light does not change from green to blinking yellow, because it will appear to the drivers as a regular yellow light for a period of time long enough to cause problems. A possible solution for this is to add a timer that generates the **EnterNightMode** event at discrete intervals (specified by the **RetryNightMode**) until the state change is performed successfully.

To keep track of the current state, define a variable **currentState** as a pointer to the current state function. Here is an example showing how to define the type of this variable.

```
typedef void (MainLightFSM::*Function)(FSMEvent event);
Function currentState;
```

In addition, define **processEvent()** method on the FSM. This method is invoked by the Client to pass an event to the FSM. Implement in it the invocation of the current state method passing in the received event.

```
void MainLightFSM::processEvent(FSMEvent event) {
  (this->*(currentState))(event);
}
```

Note that the use of function pointer as the indicator of the current state makes it impossible to directly query the current state variable for its name.

Side note on other languages
This is not a problem in languages that fully support reflection, e.g. Smalltalk. In Smalltalk, the state can be stored as a symbol that can be invoked as a method. Here are the two methods from above implemented in Smalltalk.

```
MainLightFSM>>processEvent: anEvent
  self perform: currentState withArgs: (Array with: anEvent).

MainLightFSM>>greenLight: anEvent
  anEvent = #SideLight_GetGreenEvent ifTrue
    [self setimer: #YellowLightTimer.
     self currentState: #yellowLight.].
  "more events to handle here"
```

When **`currentState`** variable is set to **`#greenLight`**, message **`perform: withArgs:`** in the **`processEvent:`** method invokes the corresponding **`greenLight`** method.

Recall an earlier discussion of the relationship between states and events. One would expect this pattern to explicitly use events as indices into implementations of state methods, however this is not exactly the case here. A variant of this pattern, described later follows that idea more explicitly.

Consequences[4]
+ *The complete state behavior is defined explicitly in one place in the code*. The behavior corresponding to each state is encapsulated in a separate method. Moreover, explicit mapping of events to actions is encoded in the state definition. All the FSM data and behavior is implemented in one class.
+ *Often an optimal implementation of states*. In FSMs where the state-dependent behavior is not too complex, the use of state method results in a very compact implementation.
+ *Good performance*. Only two levels of indirection are required to handle each incoming event; one to select the current state and one to process the specific event. This implementation is as efficient as ENUMERATED STATE, but easier to maintain.
- *Adding new states and events affects the existing code*. Adding new events means modifying each concrete state. While definitions of new states are isolated from the existing code, at least one existing state needs to be modified to add a state transition to the new state. Thus it requires access to all the source code of the FSM.
- *This design does not handle behavior associated with transitions*. FSMs can define behavior that occurs specifically during the state transition (i.e. when the state is not defined). It is not possible to implement such behavior here.

[4] See [Hen03] for the complete discussion of consequences.

Variant - State Functions Struct [Hen03]

A small variation on the concept of state methods is to define each state as a collection of methods, where each method corresponds to the processing of a specific input event in that state. Since each state needs to handle all events, define the state as a list of function pointers corresponding to functions for all the events, listed in a consistent order. In C++, the best data structure for such a list is a struct of function pointers, shown below. Note that the keyword **`const`** indicates that the function pointers can only be assigned once, at initialization.

```
// state type declaration – function pointer
typedef void (MainLightFSM::*Function)();

struct MainLightState {
  const Function sideLight_GetGreen,
                 yellowLightTimeout,
                 redLightTimeout,
                 incommingTraffic,
                 sideLight_TurnedRed,
                 enterNightMode,
                 exitNightMode
};
```

Define the current state of the FSM class as the pointer to the appropriate instance of the **`MainLightState`** struct. Declare one static instance of **`MainLightState`** struct for each concrete state.

```
static const MainLightState green_light;
static const MainLightState yellow_light;
```

Note that each method inside the **`MainLightState`** struct is specific not to the current state, but to the combination of state and the incoming event. This has a significant impact on the overall design of FSMs that implement states using state method struct. See Building Complete FSMs section below for detailed discussion of the interrelations between different FSM elements imposed by the use of this pattern.

Additional Consequences

Here are additional consequences introduced by the variant. All the consequences described above apply to it as well. For the complete discussion of consequences related to this pattern, see [Hen03].

+ *Short methods*. All actions and events are implemented in separate methods, which make them simple to understand. This, however, does not translate into simplified understanding of the behavior from the perspective of each state, as noted in the last liability.

- *Adding new states and events affects the existing code*. Adding new events means modifying each concrete state and modifying the definition of the state struct. While definitions of new states are isolated from the existing code, at least one existing state needs to be modified to add a state transition to the new state. Thus it requires access to all the source code of the FSM.

- *Poor code readability*. It is difficult to understand the behavior of each state, because its implementation is spread out over many methods that are not tied explicitly to the state. The state is encoded in a pseudo-tabular structure that shares the disadvantages of the tabular form (behavior logic and transitions cannot be viewed together) and the disadvantages of describing states explicitly (behavior and state changes are mixed together). Only small FSMs can benefit from this explicitness.

Building Complete FSMs

Any FSM design that implements states as methods indicates clearly that its main goal is performance rather than flexibility. Thus, it is unlikely that any of the FSM elements would be implemented as classes. Moreover state methods encapsulate also the functionality to change the state (using STATE-DRIVEN TRANSITIONS). This leaves only two decisions left to the designer. Should we use enumerated events or event methods? What about actions? Should they be unstructured or implemented as methods?

The main version of the pattern is likely to work better with ENUMERATED EVENT, because events are passed as inputs to the **`processEvent()`** method, but actions can be implemented either way. Implementing a complete FSM with state method struct variant poses more restrictions on actions and events. Implement both elements as short methods. EVENT METHODS provide the initial interface for the Client. Their only goal is to find the ACTION METHODS to invoke, based on the current state.

The state struct, described in the variant section, is the heart of the complete FSM. Encode in it the mapping between states, events, and actions. For each state, define a structure such as the one below, based on the transitions defined in Table 3. The code below shows the struct for the **`green_light`** state. Looking back at Table 3, there are only two events that require handling in this state, **`sideLight_GetGreen`** and **`enterNightMode`**. Other events are either ignored or indicate an error, so they are mapped to generic **`ignore()`** and **`error()`** ACTION METHODS.

```
const MainLightFSM::MainLightState MainLightFSM::green_light =
{
  &MainLightFSM::setYellowTimer,
  &MainLightFSM::error,
  &MainLightFSM::error,
  &MainLightFSM::ignore,
  &MainLightFSM::ignore,
  &MainLightFSM::tryEnteringNightMode,
  &MainLightFSM::error
};
```

Implement event methods as dispatchers that find and invoke the action method corresponding to the current state and incoming event. Implement the action methods to perform the work. Some action methods are very generic, such as **`ignore()`** and **`error()`**. However most of them correspond to one event and one state, which means that they have very specific roles and their implementation is usually short.

Here is a concrete example of event and action methods required for the main light FSM. Initially, the Client invokes an event method, such as **`enterNightMode()`** whose implementation is shown below.

```
void MainLightFSM::enterNightMode(){
  (this->*(currentState->enterNightMode))();
}
```

The event method invokes the method corresponding to the location of **`enterNightMode`** event in the **`MainLightState`** structure. So event methods are indices into the state struct. In this case the action method is resolved to **`tryEnteringNightMode()`**, implemented below. For details of the method itself, refer back to the Solution section.

```
void MainLightFSM::tryEnteringNightMode(){
  if (!isIncomingTraffic())
    currentState = blinkingYellowLight;
  else
    setTimer(RetryNightMode);
}
```

Since the Client invokes event methods, declare them as public. Action methods are internal to the FSM and should be declared private.

FSMs using STATE METHODS are often found in applications that require fast processing and are likely to require extensions, but no major redesign, such as network protocols.

State Class [GHJV95]

Another major feature missing in the traffic light FSMs is the left-turn light. Intersections often have a separate left-turn lane, which is controlled by two separate lights – green turn arrow and yellow turn arrow. Adding them requires more extensions to the original FSMs. Turn lights require adding two new states (**`GreenLeftArrowLight`** and **`YellowLeftArrowLight`**) and corresponding events.

Note that adding more features results in adding more states to the FSM. This is hard if the code does not cleanly separate the behavior of each state. But if the code for each state is separated from other states (e.g. in a separate source file), adding a new state will not affect existing ones. To isolate a state from other states most effectively, encapsulate each one in its own class.

Problem
Adding a new state results in changing the code of existing states. How can we protect the existing code from unnecessary changes?

The FSM needs to address the following forces. The organization of the logic that manages the FSM's state should be able to scale up to many states without becoming difficult to maintain. Adding new states should not affect the existing code, yet new states should be able to reuse the existing code easily. The FSM should be able to handle a large number of events. The behavior of each state may be arbitrarily complex and may require storing significant amount of FSM data. The design should shield the Clients from the changes of implementation.

Solution
Implement each state as a class. Build a state class hierarchy that shares a common interface, corresponding to the incoming events.

Before we can discuss the implementation, let's update the main light FSM from Figure 2 to support the two new states. It is also necessary to add a new event that indicates the green arrow light timeout. The implementation of the yellow arrow light can reuse the event and the timer used for the yellow light, because both yellow lights should have the same length. These new states are inserted between **`ReadyForGreen`** and **`GreenLight`** states. Figure 4 contains the updated state diagram of the main light FSM.

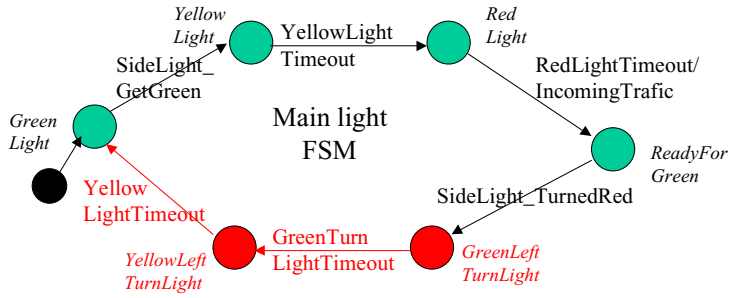

Figure 4. State diagram of the main light FSM with support for left turn signals. Changes made to Figure 2 are shown in red.

	GreenLight	YellowLight	RedLight	ReadyForGreen
SideLight_GetGreen	YellowLight	I	I	E
YellowLightTimeout	E	RedLight	E	E
RedLightTimeout	E	E	ReadyForGreen	E
IncomingTraffic	I	I	ReadyForGreen	I
SideLight_TurnedRed	I	I	I	*GreenLeftArrow*
GreenTurnTimeout	*E*	*E*	*E*	*E*

	GreenLeftArrowLight	***YellowLeftArrowLight***
SideLight_GetGreen	*I*	*I*
YellowLightTimeout	*E*	*GreenLight*
RedLightTimeout	*E*	*E*
IncomingTraffic	*I*	*I*
SideLight_TurnedRed	*I*	*I*
GreenTurnTimeout	*YellowLeftArrow*	*E*

Table 4. Main light FSM with support for left turn signals. Changes made to Table 1 are shown in red italics.

Adding new states requires changing state transitions. Table 4 includes the new transitions needed to support the updated FSM. Note that, with the exception of one transition (in ReadyForGreen state upon the reception of SideLight_TurnedRed event), the original transition table remains unchanged. All the existing states treat the new event as an error. This is not always the case. For example, adding night-lights, described in STATE METHODS, would require specifying new transitions for each of the existing states.

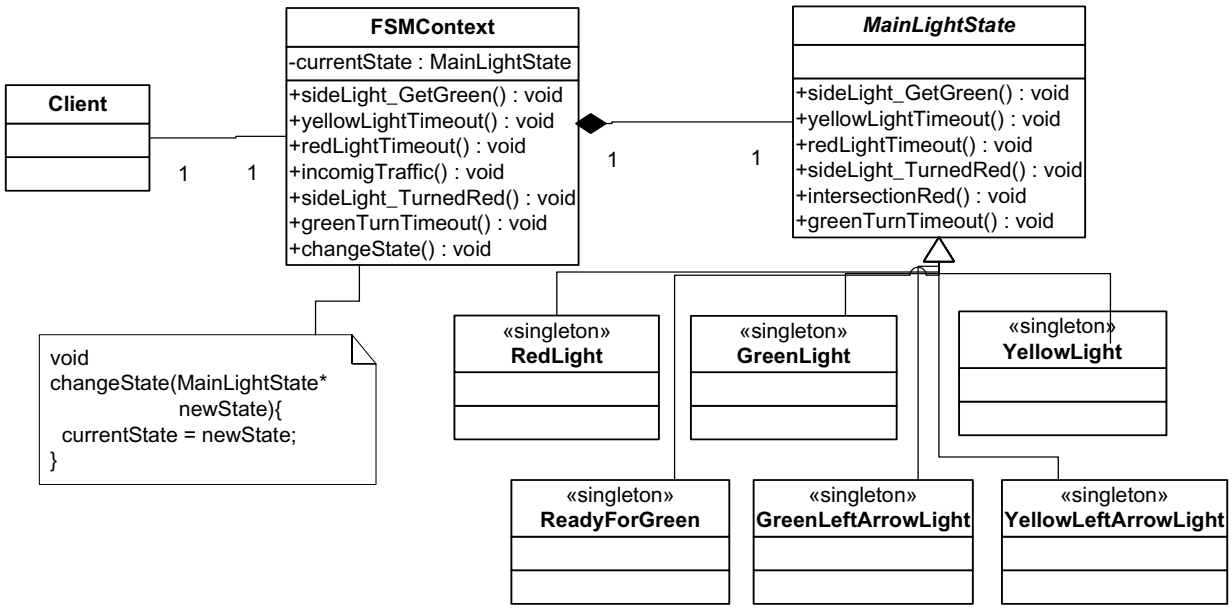

Figure 5. Class diagram of the State Class

Having defined the transitions, we are ready to implement the states. First, define an abstract base class for all states, **`MainLightState`**. In its interface, define one method for each event handled by the FSM. Make these methods abstract. Next, define concrete subclasses, one for each state of the FSM. Implement the handling of each event (including state changes) in the appropriate methods of the concrete classes. Lastly, define a class representing the FSM to the outside world, usually called **`FSMContext`**. In this class, provide methods corresponding to all events so that the Clients can invoke them.

Figure 5 depicts the complete class diagram of the State Class. All subclasses of the State class hierarchy inherit all the methods from the base class. These methods are omitted from the figure for readability. The figure shows also additional details of this pattern. The **`FSMContext`** class keeps track of the current state (the object of the concrete FSM class) in the **`currentState`** variable. Since information about next state is encapsulated inside state classes, the Context allows them to modify the current state via the **`changeState()`** method. In general, the **`FSMContext`** class contains other state-independent functionality, such as handling timers, and storing FSM-specific data.

Despite having so many responsibilities, the **`FSMContext`** class is easy to implement. Its public methods, invoked by the Client, forward the requests to the current state passing all the inputs and the reference to the Context. This is required for callbacks, to enable state classes to set next state and to modify the FSM-specific data.

```
void FSMContext::incomingTraffic(FSMEvent event) {
  currentState->incomingTraffic(*this, event);
}
```

Implementing each state as an object can result in increased memory usage. Moreover, continuously creating and deleting state objects on each state change is likely to decrease performance. But since all FSM data is stored in the Context, State classes are context-free and only implement the state-specific behavior. As a result, it is sufficient to have one instance of each concrete State class (using the Singleton pattern [GHJV95, p. 127]) shared by multiple FSMs. To instantiate a new FSM instance, create only an instance of the Context object. Single instance of each concrete State class can be shared by multiple Context instances, according to the Flyweight pattern [GHJV95, p. 195]. Alternatively, instances of the State class can be created and destroyed as needed, if this overhead is acceptable. Use this approach when not all states may be entered during the execution of the FSM and state changes do not occur frequently.

Some events may be handled the same way by most (or all) classes. For example, **`handleUnexpectedEvent()`** method in the original code of main light FSM handles many cases. Such behavior can be implemented in the **`MainLightState`** class and overridden by the concrete state classes that require different processing. One drawback of this approach is that it is easy to forget to implement the different behavior if the default one is already provided. Keeping only abstract methods in the **`MainLightState`** class allows the coder to delegate the completeness check to the compiler.

If implementing FSM states as objects is too costly for you, consider a less memory-intensive solution described in the Variant section.

Consequences

+ *State-specific behavior is localized and partitioned between states*. Code changes specific to a state are completely isolated from other states.

+ *State transitions are performed explicitly in the code*. Each state knows exactly what is the next state to transition to in each situation.

+ *Adding new states is easy*. Add a new subclass of State and provide implementation of all the actions. If base class provides reasonable default behavior, override only those actions that require different handling.

+ *Standardized interface*. All State classes are guaranteed to provide the handling for all events. Otherwise, the code will not compile.

+ *Having the Context class increases encapsulation*. The client only needs to know about the Context, regardless of how complex the internal FSM implementation is. The implementation can change without affecting the Client.

- *This pattern makes the FSM implementation more complex*. The state element is implemented in two classes (State and Context), which may be too complex for a simple FSM. Consider for example an FSM with 3 states and 4 events. With STATE METHODS, this FSM would be implemented in one class with 4 methods (one method per state and the **`processEvent()`** method). In contrast, State Class implementation requires 5 classes (3 concrete state classes, base State class, and the Context class) and 20 methods (each class implements 4 methods, one for each event) to accomplish the same result.

- *Additional space requirements*. New objects take up extra space. In most cases, however, state objects can be shared and only one instance of each State class is needed. Multiple instances of the Context class can interact with the same State instance, because all the instance-specific data is stored in the Context, while the State class only implements context-free behavior.

- *The Context class is responsible for providing the correct mapping between incoming events and the behavior defined by the State class hierarchy*. The context class delegates all external events to corresponding methods in the interface of the State class. This results in a tight coupling between the Context class and concrete events.

- *Adding new states and events affects the existing code*. It is not possible to add a new subclass of the State class without modifying the existing classes. At least one existing class must be modified to add a state transition to the new state. Adding a new event can potentially affect all existing classes. Even in the case when it is handled in the same way by all states and the handling is implemented in the base class, the Context class still needs to be updated to invoke the new method. Thus it requires access to all the source code of the FSM.

Variant - Single State Class [Pal97]
An alternative FSM implementation of states as classes is to define one state class and to create one object of the class for each state. The role of states in this variant is to encapsulate the mapping of events to actions in state objects. The processing is implemented in the Context class.

This variant is summarized in Figure 6. To implement the states of the FSM as Single State Class, define a State class (called **`StateMapper`** in the figure) with a table that maps incoming events to action methods. In each instance of **`StateMapper`** class define different mapping of events to actions specific to that state. In the Context class (called **`State-Dependent Object`** in the figure), store a reference to the current state and provide the event-handling interface (here, **`handleEvent()`** method) for the Client. In addition, define all action methods in the Context.

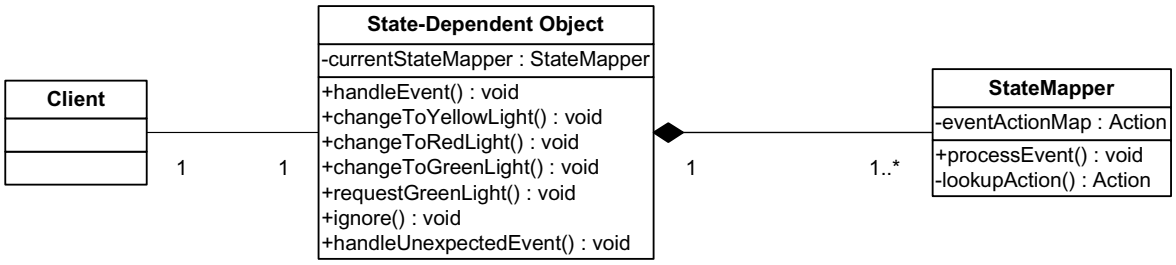

Figure 6. Class diagram of Single State Class

Note that one minor change of design (moving the main processing from states to the Context class) produces a static diagram strikingly different from Figure 5. But the run-time dynamics of this variant are quite similar to typical STATE CLASS. First, the Client invokes **`handleEvent()`** method, passing an event as input. Then the Context forwards the request to the object corresponding to the current state.

```
void StateDependentObject::handleEvent(FSMEvent event) {
  currentState->processEvent(*this, event);
}
```

The State object then looks up the method that corresponds to the incoming event in **`eventActionMap`**. Lastly, it invokes the method back on the Context[5].

```
void StateMapper::processEvent(StateDependentObject& context,
                               FSMEvent event) {
  Action action = lookupAction(event);
  // Lookup returns an action function, execute it
  (context.*(action))();
}
```

In the process of executing an action, **`State-Dependent Object`** may replace the current **`StateMapper`** object with a new one, corresponding to the new state.[6]

[5] This technique, called double-dispatch, was originally described in [Ing86].

Variant-specific Consequences
+ *State transitions are performed explicitly in the code*. Each state knows exactly what is the next state to transition to in each situation.
+ *Adding new states is easy*. Add a new subclass of State and provide implementation of all the actions. If base class provides reasonable default behavior, override only those actions that require different handling.
+ *Having the Context class increases encapsulation*. The client only needs to know about the Context, regardless of how complex the internal FSM implementation is. The implementation can change without affecting the Client.
- *Adding new states and events affects the existing code*. It is not possible to add a new subclass of the State class without modifying the existing code in the Context. Thus it requires access to all the source code of the FSM.
- *Difficult to match state transitions with events*. Since the code implementing state changes resides in the Context, it is difficult to map the state transitions to the event-state mapping in the **`StateMapper`** class.

Building Complete FSMs
FSMs that define states as classes provide good maintainability and extensibility. It is easy to extend them and easy to map FSM elements to specific design elements. In such FSMs states play the most important role and other FSM elements are complementing the state.

Let's give Table 4 a closer look. Each cell in the table corresponds to exactly one state and one event. In the code, each cell corresponds to a body of each concrete method in the state class. These are EVENT METHODS containing UNSTRUCTURED ACTIONS. Each method corresponding to a cell specifying a state transition will call **`changeState()`** method. This is one way to map actions to events, known as STATE-DRIVEN TRANSITION. An alternative is to encode the transition table directly in the code as described in TRANSITION TABLE.

The design of State Class is based on the assumptions that {event, state} pairs correspond to unique actions. But as we can see in Table 4, most pairs are invalid and map to generic actions that are not state-specific. Defining them inside state-specific methods is unnecessary. The Single State Class variant takes advantage of this observation and separates actions from states. It puts all actions in the Context class to make code more compact and easier to follow. The FSMs built using the variant use ENUMERATED EVENTS, ACTION METHODS, and TRANSITION TABLE. However, this design is less modular and will not scale well as new states and events are added. Adding many new action methods to the Context class will make it monolithic and difficult to maintain. For a more flexible solution of this problem, see ACTION CLASS, which defines actions independently from other FSM elements.

FSMs using STATE CLASS are mostly found in large, not performance-critical applications with unpredictable growth requirements, such as graphical tools.

---

[6] This pattern bears resemblance to Command and Chain of Responsibility (both [GHJV95]), because the classes delegate requests to each other. While these patterns have similar intents, the details of the solution described here are sufficiently different from both of them to warrant this as a distinct pattern.

Event Class [HJE95]

Traffic lights are to make the flow of traffic as smooth as possible. But if the traffic lights were to determine their light changes in isolation, driving would be more difficult, because on average every other light would be red. To synchronize light changes, each light should communicate with nearby lights by exchanging status messages. In addition, there are cases when traffic lights should not follow the indications of their sensors. For example, if one road is blocked because of a passing train, the light sensor will be reporting that many cars are waiting for the green light, but the traffic light should ignore this and give the green light to the crossing, unblocked traffic. This is possible only if the traffic light receives external messages informing it of this condition.

Implementing such message exchange does not require adding transition states to the FSM. Instead, the algorithm to determine the next state requires modifications, because the new state is now determined by examining data received from other traffic lights. That data can be encoded as events that carry additional, event-specific information. To associate the events with their processing, encapsulate each event in a class.

Problem
How to design well-encapsulated events that can be passed within an FSM and between FSMs?

The FSM needs to address the following forces. The number of events is significantly larger than the number of states and it is more likely to fluctuate. Events should be implemented so that they can be added and removed at run-time (while the number of states is unchanged). Events should know how to execute the behavior associated with them.

Solution
Build a hierarchy of event classes. Define only one public method, **`apply()`**, to initiate the processing of the event. For each concrete event, specify how to process it in each state of the FSM. Depending on the current state, the event should be processed, ignored, or treated as an error. Declare these three cases as methods in the abstract interface of the base class of all events. These methods are likely to differ between event classes, so don't provide any default implementation in the base class. One possible exception may be the method to ignore an event, whose empty default implementation can be provided in the base class.

```
void FSMEvent::ignore(FSMContext& ctx, FSMState& state){}
```

Figure 7 illustrates a portion of the **`FSMEvent`** hierarchy with the methods corresponding to all the cases that each Event class needs to handle. Note that the **`FSMEvent`** superclass stores a **`sender`** variable, a reference to the FSM that originated the event. In cases that require the FSM to respond back to the sender of the event (for example, when an error occurs), the event object can create another event and send it to the sender FSM. The **`MsgOutOfOrderEvent`** class corresponds to the new event designed specifically for such

a communication between FSMs. The **SideLightGetGreenEvent** class encapsulates the processing of the event to allow the side light FSM to turn green.

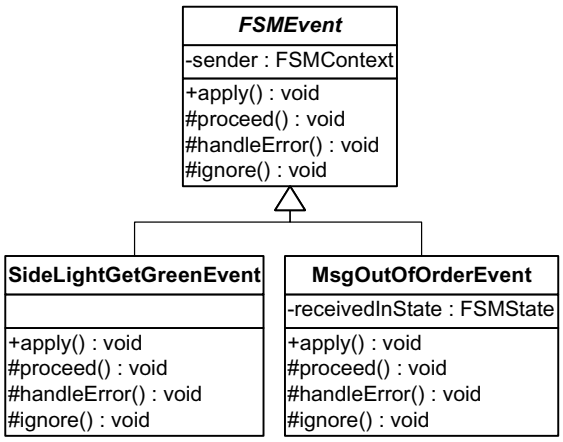

Figure 7. Class diagram of the Events class hierarchy.

The implementation of **SideLightGetGreenEvent** class shown below, provides a good example of implementing the event-related behavior. Only concrete event classes know how they can be processed in each FSM state, so implement the **apply()** method in each derived class. Depending on the current state, the event may be allowed to continue (in the **proceed()** method), treated as error (in the **handleError()** method), or ignored. SideLightGetGreen event is expected in the GreenLight state. In other states it should be ignored, except in ReadyForGreen state, where it indicates an error.

```
class SideLightGetGreenEvent: public FSMEvent{
  void processEvent(FSMContext& ctx, FSMState& state){
    if (state == GreenLightState::instance())
      proceed(ctx, state);
    else if (state == ReadyForGreenState::instance())
      handleError(ctx, state);
    else
      ignore(ctx, state);
  }
  void proceed(FSMContext& ctx, FSMState& state){
    state.sideLightGetGreenEvent(this, ctx);
  }
  void handleError(FSMContext& ctx, FSMState& state){
    // sender should not have sent this message now
    MsgOutOfOrderEvent* event = new MsgOutOfOrderEvent(ctx);
    sender.process(*event);
  }
};
```

This is the only FSM pattern that requires the Client of the FSM to be actively involved in the processing. The Client is responsible for creating the appropriate event object, for populating its data, and for passing it to the appropriate FSM. Typically, Clients should not have so much responsibility. However, in this case, Clients are other instances of the FSM, so they are already implementing this processing.

Consequences

+ *Mapping of the incoming event to its processing is encapsulated in the Event class hierarchy*. Each incoming event is initially analyzed in the corresponding Event class. The class knows how to process the event.

+ *Adding new events does not require changing existing code*. Events know how to process themselves and most of the processing is encapsulated inside of them. This is useful when building FSMs that have a constant number of states, but need to support ever-expanding set of events.

+ *Two FSMs running different versions of the code can communicate.* FSM need not know that it is processing a newly defined event, because that event's processing is completely encapsulated in the event object. If the event object interfaces correctly with the FSM, it can process itself. This is useful for software upgrades.

- *Too complex for most FSMs*. Implement Events as classes only if the number of valid events can vary at run-time. Otherwise, events are best modeled as data that is analyzed from the perspective of the current state and processed by actions.

- *Exchanging too many events can affect performance*. While receiving notifications from more FSMs is likely to increase the precision of decisions made by any one of them, exchanging too many messages may unnecessarily complicate the processing. Each additional message produces processing overhead of generating and analyzing their data as well as memory overhead for storing, buffering and sending them.

Building Complete FSMs

FSMs that define events as classes provide significant flexibility and ability to pass a lot of data between FSMs. Designing a complete FSM with event classes makes the other FSM elements adhere to specific constraints. Events are responsible for recognizing whether they can be processed or not. States are implemented exactly as in STATE CLASS. The only minor difference is that the ACTION METHODS defined in the states are invoked by the Events, rather than by the Context. The STATE-DRIVEN TRANSITIONS are also implemented in the States. Refer to Figure 8 for a more detailed picture.

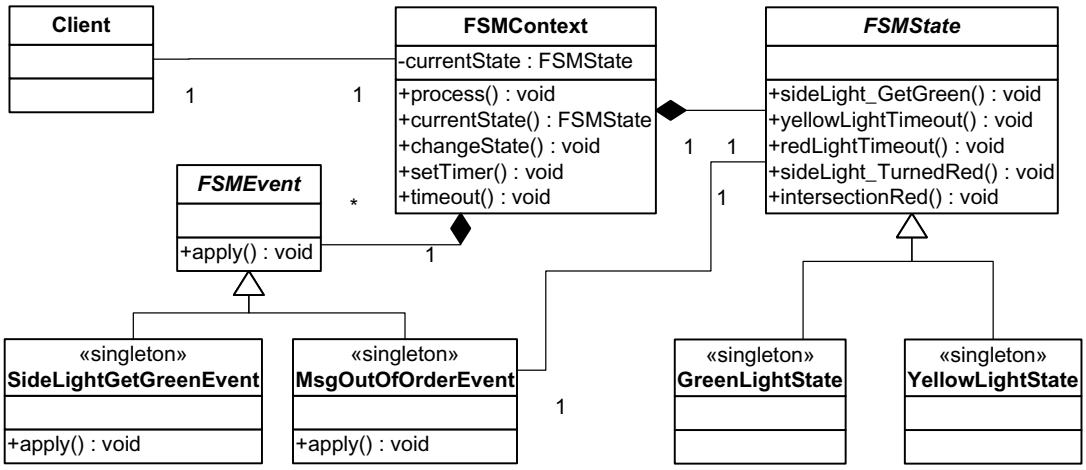

Figure 8. Class diagram of the main light FSM with Events modeled as classes

Note how balanced this FSM design is. The processing is separated between two class hierarchies, Events and States. Event classes know whether or not they can be processed, depending on the current state of the FSM. State classes implement the actual behavior of the FSM, such as handling the data associated with each event and performing the state transitions.

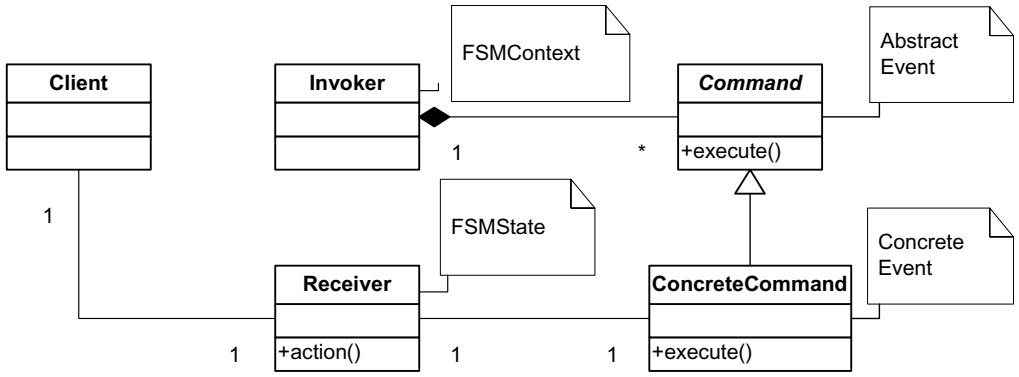

Figure 9. The Command pattern applied to Event Classes

This design is a classic example of the Command pattern [GHJV95, p. 233]. Figure 9 shows how the FSM classes correspond to the elements of the Command Pattern. In the terminology of the Command pattern, all Events play the role of Commands. The **`FSMContext`**, class plays the role of the Invoker, because it initiates the execution of Events. **`FSMState`** class is the Receiver, because the Event class invokes the appropriate method on the **`FSMState`** class to perform the actual processing of the event.

FSMs implemented using EVENT CLASS may seem too slow for real-world applications, but this pattern is used in commercial network protocol software [HJE95]. However, typically, FSMs that require such flexibility are defined using code generators.

Action Class [vGB99]

A recent addition to traffic lights is the ability to control them remotely from emergency vehicles (e.g. ambulances). This feature is especially helpful during the rush hour when intersections are packed with cars and emergency vehicles have less ability to maneuver in the traffic. By communicating with the light from inside the vehicle, it is possible to change the light from red to green faster. It is also possible to keep the green light on longer than usual.

To implement this feature correctly, it is necessary to bypass (possibly many) intermediate FSM states. It is not feasible to expect that the operator of the emergency vehicle would send in multiple events sequentially. Instead, one event sent from the vehicle should produce a (possibly long) sequence of light changes (actions). The traffic light FSM itself needs to invoke the actions in the appropriate order. In the simplest case, all intermediate actions will be empty; the FSM will pass through all the states until it reaches the green light. But it is also possible to perform additional processing in the intermediate states.

One way to implement the support for the emergency vehicles is to associate the incoming event with a sequence of actions. To accomplish this, actions of the FSM should be implemented so that they can be easily combined into sequences, can be undone, and replayed without duplicating the code. To do so, implement actions as classes.

Problem
How to design an extensible model for actions so that they can be reused, undone, composed into sequences and/or attached to other FSM elements.

The FSM needs to address the following forces. The FSM encapsulates a lot of complex behavior that can be broken up as individual actions. It should be easy to reconfigure actions, to add new actions, and to undo previously performed actions at the cost of lower performance. To limit code duplication, it should be possible to define new actions as sequences of existing ones. It should also be possible to keep the implementation of actions unchanged while the number of states, events, and state transitions fluctuates.

Solution
Encapsulate each action in a separate class. Combine all new classes into the Action class hierarchy. This hierarchy is likely to be large, because FSMs usually have more actions than states. Fortunately, actions tend to be small – their interface consisting of as few as two methods, **`executeAction()`** that performs the processing associated with the action and **`undoAction()`** that undoes that processing in a safe manner. Here is a definition of a base class, **`MainLightAction`**.

```
class MainLightAction {
  virtual void executeAction(FSMContext& ctx,
                             FSMData* data) const = 0;
  virtual void undoAction(FSMContext& ctx,
                          FSMData* data) const { };
};
```

Note that both action methods receive two inputs, the **FSMContext** object and the incoming **EventData** object, so that they can modify FSM data stored in **FSMContext** with the new input. Thus Acion classes do not need to store any FSM-specific data and can be implemented as Singletons [GHJV95, p. 127]. For more details on structuring FSMs containing Action classes, see section Building Complete FSMs.

This pattern is applicable to many types of actions: actions executed when entering a state (state entry), when exiting a state (state exit), and on state transitions. From the perspective of the action class, it is irrelevant which type of action it implements. Each one can be used as an entry, exit, or transition action. Name each **MainLightAction** subclass after the action it performs so that it can be used in any case. Here is a sample action to support emergency vehicles:

```
class BypassStateAction: public MainLightAction {
  void executeAction(FSMContext& ctx, FSMData* data) const {
    // Optionally, store info about this emergency vehicle,
    // passed in FSMData.
  }
};
```

The **BypassStateAction** action is applicable to many state transitions. In contrast, **GreenToYellowLightAction** action below is specific to only one condition, changing from green to yellow light. When the action is executed, it sets a timer. To undo it, simply cancel the timer, because main light FSM can have at most one timer running at a time.

```
class GreenToYellowLightAction: public MainLightAction const {
  void executeAction(FSMContext& ctx, FSMData* data) {
    TimerObject* t = new TimerObject(YELLOW_LIGHT_TIMER);
    t->startTimer();
    ctx.storeTimer(*t);
  }
  void undoAction(FSMContext& ctx, FSMData* data) {
    ctx.cancelTimer();// only one timer is running at a time
  }
};
```

It is also possible to build composite actions by reusing implementation of other actions. For example **EmergencyGreenLightAction** action below reuses an existing action to set the green light and performs additional processing.

```
class EmergencyGreenLightAction: public MainLightAction {
  void executeAction(FSMContext& ctx, FSMData* data) const {
    ctx.cancelTimer(); // if any
    GreenLightOnAction::instance()->executeAction(ctx, NULL);
    // possibly disable other actions until the emergency
    // vehicle passes
  }
};
```

Consequences

+ *Actions are independent entities.* Modifications to the code of one action are isolated from all other code. Various types of actions (on state entry, on state exit, on transition) can be reused in all contexts without changes. For example, an entry action of one state can be used as an exit action of another state if they have the same behavior.

+ *Actions can share behavior through inheritance.* It is possible to subclass actions and to extend their behavior incrementally.

+ *Actions can delegate execution to other actions*. Every incoming event handled by the FSM is usually mapped to a single initial action, but multiple actions may be executed before the processing completes. Smaller, self-contained the actions are more reusable.

- *This design is too complex for most systems*. Building a hierarchy of classes so that they implement one method is usually not needed, unless actions are complex and can be reused in many contexts.

- *Additional levels of indirection*. Separating the behavior into many action classes results in many function calls between classes. In the worst case, one exit action, one transition action, and one entry action needs to be executed on each state transition, which may have a negative effect on the performance.

Building Complete FSMs

FSMs that define actions as classes are built for extensibility and reconfigurability at the cost of slower performance. In such FSMs, actions perform all the processing while other FSM elements are responsible for invoking appropriate actions at the correct time.

One such action-centric implementation of the FSM is described in [vGB99] and summarized in Figure 10. Action objects perform all the processing. Other FSM elements merely link actions together and invoke them in the proper order. **`FSMTransition`** class (described in detail as a variant of TRANSITION TABLE CLASS) executes actions related to each transition. Each **`FSMState`** class stores actions to enter and exit the state.

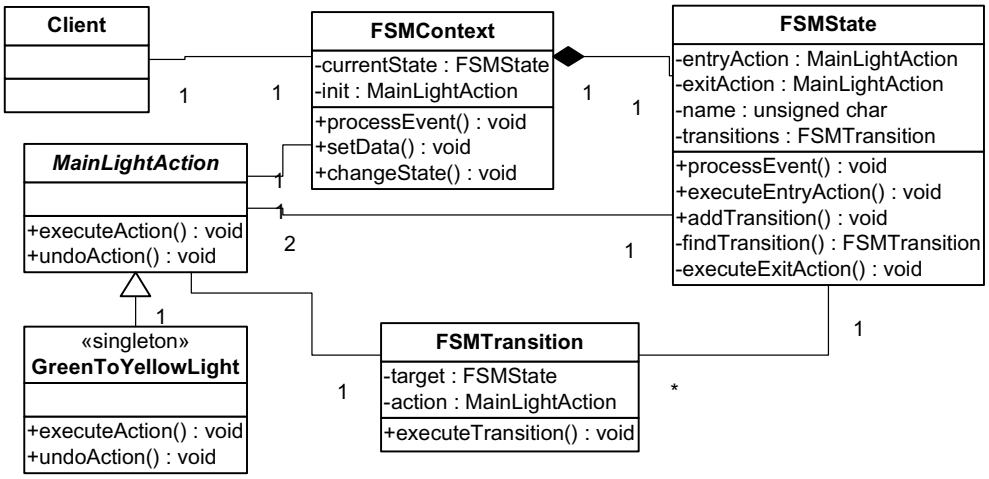

Figure 10. Complete FSM design using Action classes

Since actions can be combined into sequences, stored, and replayed, the best way to implement them is to follow the Command[7] pattern [GHJV95, p. 233], illustrated in Figure 11. In the terminology of the Command pattern, all Actions play the role of Commands. The **`FSMContext`**, **`FSMState`**, and **`FSMTransition`** classes play the role of Invokers, because they initiate the execution of Actions. **`FSMContext`** invokes the initial action, **`FSMState`** invokes entry and exit actions, and **`FSMTransition`** invokes an action on transition. **`FSMContext`** class is also the Receiver when Actions invoke it to update the FSM data it stores.

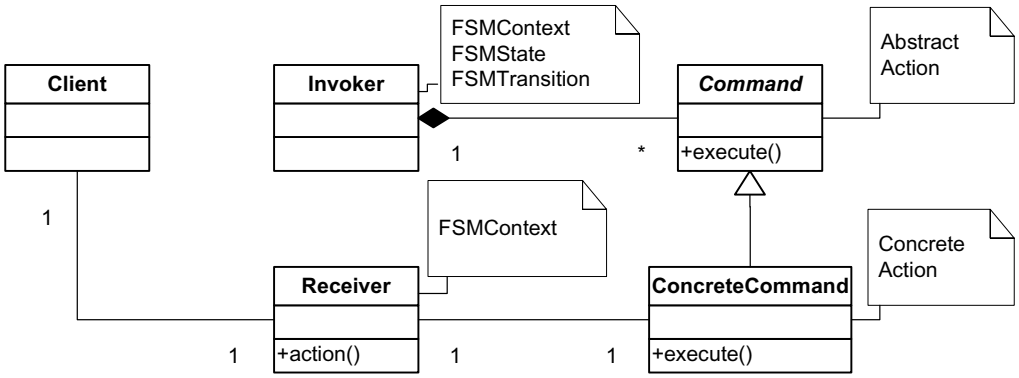

Figure 11. The Command pattern applied to Action Classes

Figure 12 shows the sequence diagram for a typical state change. Only the **`FSMContext`** class contains instance-specific data. The reference to the **`FSMContext`** object is passed along in each step so the other classes are context-free.

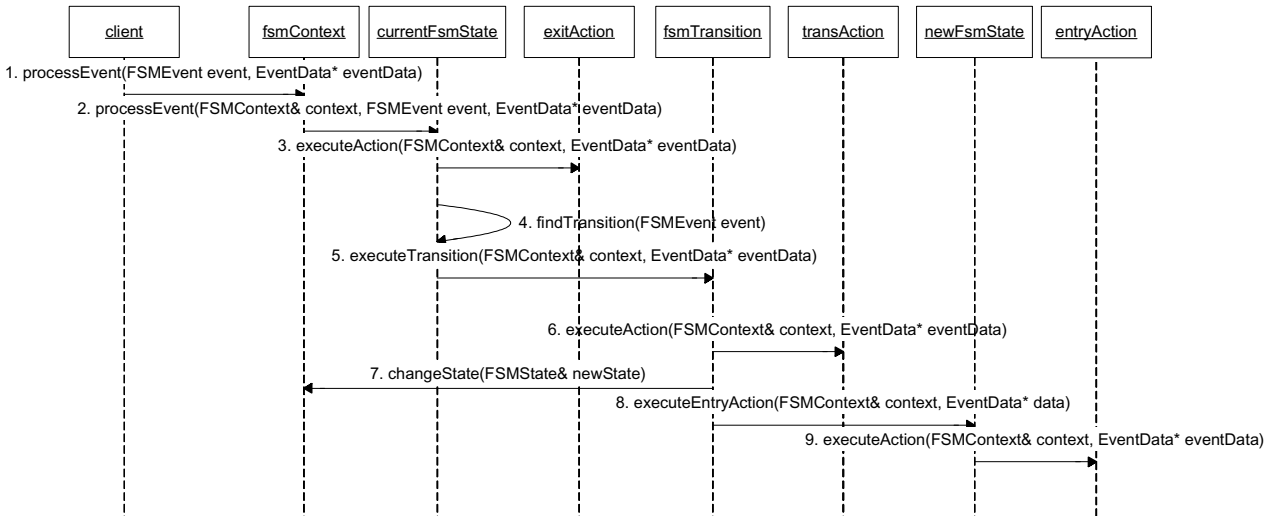

Figure 12. New event causes a state change and an invocation of 3 action methods

[7] This pattern may seem to be a Strategy, but it is not. If Actions were Strategies, the State or Transition class would select one of them from a list of choices on each invocation. Instead, they are context-free and operate only on the data stored in the Context class. Each action processes one command, not a complete algorithm.

The **FSMContext** object knows only about the concrete **FSMState** objects, so it delegates the Client's request (step 1) to the current instance of the **FSMState** class (step 2). The state class (after invoking its exit action in step 3) finds which transition is required (step 4) and delegates it to the **FSMTransition** object (step 5). The current **FSMTransition** object invokes a concrete action corresponding to the transition (step 6), sets the new state (step 7), and invokes its entry method (step 8). **FSMState** executes its entry Action to complete the processing.

Note that the implementation of **FSMState** used here is an extension of the Single State Class variant of STATE CLASS. There is a single state class, **FSMState**, and each state is an instance of that class. Each instance defines its transitions as mappings of the incoming ENUMERATED EVENT to a **FSMTransition** object (rather than an Action), because each state transition is associated with a set of actions (exit old state, perform transition, enter new state). Actions are not defined as methods in the Context class (which is the main drawback of Single State Class), but rather are encapsulated in separate classes.

The author is not familiar with any systems that implement FSMs using ACTION CLASS. The original description of this pattern [vG99], does not describe any known uses. That paper advocates code generation to produce parts of the system. FSMs that implement actions as classes are usually complex and may be easier to implement with code generators.

Transition Table Class [Pal97] [vGB99]

Let's reconsider the problem of implementing a communication between traffic lights. Under special circumstances, each traffic light needs to communicate with other traffic lights. Such communication requires additional data transmission between traffic lights. Fortunately, this situation does not occur in all cases (for example, all traffic problems cease to exist during the night) and does not produce additional processing overhead in such times.

The solution described in EVENT CLASS adds a new set of events that are used specifically for adding communication between FSMs. All events (and their handling) are encapsulated in classes and the additional events are instantiated only to support additional communication. An alternative is to design the communicating FSM from the perspective of transitions, rather than events. Add simple events for communication between FSMs, but put the logic for handling them inside the state transition mechanism that reconfigures itself when handling of such events is necessary. If the transitions are hard-coded (in a TRANSITION TABLE or inside STATE-DRIVEN TRANSITION), they cannot be easily modified. Were state transitions encapsulated in a separate class, the FSM would be able to register and deregister them at run-time. So, encapsulate the state transitions in a class.

Problem
How to define a flexible state transition mechanism that can be manipulated at run-time?

The FSM needs to address the following forces. The FSM should be flexible and easy to modify, even at run-time. State transitions should be implemented separately from the behavior, because the number of transitions fluctuates as states and events are added, removed and modified. It should be possible to add and remove transitions at run-time. Transitions should know how to execute the behavior associated with them.

Solution
Define a single transition table class to encapsulate the state transition mechanism. In the class, provide methods to invoke existing transitions (**`performTransition()`**) and to modify them by adding and removing transitions (using **`addTransition()`** and **`removeTransition()`**) at run-time. Here is a sample implementation of such a class.

```
class TransitionTableClass {
  public:
    void performTransition(FSMContext& ctx,
                           FSMEvent event){
      Action action = lookupAction(event, ctx.getCurrentState());
      // lookupAction() returns an action function – execute it
      (context.*(action))();
    }
    // run-time updates of the state change mechanism
    void addTransition(Transition t);
    void removeTransition(Transition t);
```

```
  private:
    // map event and state to an action function
    Action lookupAction(FSMEvent e, FSMState s) const;
    // internal representation of the transitions
    Transition eventActionMap[];
};
```

When Client (e.g. current state class) invokes **`performTransition()`** method, **`TransitionTableClass`** looks up the action (corresponding to the incoming event and the current state) and invokes it. Note that the action to invoke can be defined in any class of the FSM, not necessarily in the **`TransitionTableClass`**.

With the transition algorithm encapsulated in a small and stable interface of the **`TransitionTableClass`**, it is possible to change the internal representation of the state change mechanism without affecting the external interface. Transitions can be stored as linked lists or multidimensional arrays.

There are many other ways to implement state transitions as classes. Variant section below presents one alternative approach.

Consequences
+ *The interface is separated from the implementation of the state transition mechanism*. Can easily change the protocol of state transitions without affecting the interface that adds and removes them.
+ *New transitions can be easily added and removed at run-time*. It is possible to build an FSM with completely new set of transitions at run-time. In the extreme case, all transitions may be changed. Of course, all the transitions need to be defined earlier. It is not possible to link new code dynamically.
+ *Support for actions on transitions*. This design allows associating actions with transitions so that during a state change a behavior can be executed (e.g. to verify that the state change can occur). This is a major feature of FSMs that other transition patterns do not support.
+ *Smaller memory footprint of transitions*. Only the transitions that are currently active are stored in memory, so the overall size of transitions, compared to TRANSITION TABLE, is smaller.
- *Too complex for most FSMs*. When no transitions on actions are defined, each transition class performs limited functionality. It is wasteful to create new classes to implement a one-line function.
- *Difficult to verify validity and completeness of reconfigured transitions*. During the reconfiguration, it is difficult to check whether adding or removing transitions produces a valid FSM. The FSM cannot assist the user in ensuring that the resulting transition setup is valid.

Variant – Actions on Transitions [vGB99]
If the FSM contains many transitions, encapsulating them in a single class may turn that class into a bottleneck of the system, especially if adding and removing transitions occurs often. Rather than having one class for the entire transition table, define a class to encapsulate a single state transition. Instantiate concrete state transitions as instances this

class. This approach makes extending the behavior of each transition easy – just associate an action with it. As a part of processing the transition, the transition instance will execute the action, if one is defined, set the new state (on the Context), and invoke the state's entry method.

```
class FSMTransition
{
  const FSMState& newState;
  const FSMAction* transitionAction;

  void executeTransition(FSMContext& context,
                         FSMData& data) const {
    if (transitionAction)
      transitionAction->executeAction(context, data);
    context.changeState(newState);
    newState.enter(context, data);
  }
};
```

Note that this variant is not essentially different from the original version. They share the same consequences.

Building Complete FSMs

This implementation of transition table is similar to one in [Pal97] (refer back to Figure 6), which uses the Single State Class (called **`StateMapper`**) to encapsulate transitions of each specific state. But that design requires having one instance of **`StateMapper`** per state. While the description above suggests using one state transition object per FSM, it is possible to define instances of **`TransitionTableClass`** with more flexibility. A single state transition class can be defined for multiple FSMs, or individually for each state (or any other combination that is convenient for a given problem). This pattern assumes the use of ACTION METHODS, and ENUMERATED EVENT, but does not require any specific definition of states.

The Actions on Transitions variant is used in the FSM implementation described in ACTION CLASS. Note that this variant does not have a central repository of all transitions such as **`TransitionTableClass`** in the main version. Instead, the functionality to add and remove transitions is distributed among **`FSMState`** classes. Each state contains references to all the transitions that are valid for that particular state. Adding new states requires adding new instances of **`FSMTransition`** class. For details of the complete FSM, refer to ACTION CLASS.

The author is not aware of any commercial products that use TRANSITION TABLE CLASS. The only use of this approach known to the author, but implemented slightly differently than described here (see [SC95]), is Choices, an object-oriented operating system from the University of Illinois.

CONCLUSION AND FUTURE WORK

This paper describes patterns for designing individual FSM elements. It explains how to build a more complex implementation of each element and how to combine them into complete FSMs. It is a survey of patterns for software engineers, who are building FSMs incrementally. The typical sequence of patterns for each element is also incremental. The first design pattern is unstructured; next ones add more structure by defining methods, classes and class hierarchies.

This paper presents a new, uniform way of analyzing complex, multi-dimensional design patterns. Without this approach, it is difficult to make any comparisons between different FSM design patterns, because they usually differ in the implementation of multiple FSM elements. A single design pattern can address changes to only one dimension well.

Many aspects of FSMs have not been addressed here. The sole focus of this paper is the set of patterns for a single FSM. Designs that require substates [Dou98a], statecharts [Har87] [YA98a], and interactions between multiple FSMs [Odr96] are not addressed yet. Similarly, incorporating the concurrent, embedded [Sam02], and real-time [Dou98b] FSM patterns is a part of future work.

Another way to extend this work is to provide a more generic format for describing design patterns for complex models (such as FSMs). Such a format should be applicable to other areas that deal with complex, multi-dimensional models. This goal is the hardest to achieve, but it could potentially be the most meaningful contribution of this paper.

ACKNOWLEDGEMENTS

The author would like to thank Prof. Ralph Johnson for his continuous support and constructive criticism. Many thanks go to Joel Jones for his detailed review of several versions of this paper. Also, to the members of SAG group at the University of Illinois for their comments on an earlier version of this paper. Baris Aktemur and Andres Fortier also provided feedback on this paper. So did Lars Grunske, the shepherd of this paper for PLoP 2004. Thanks to everyone.

A rewrite of this paper was prompted by the comments received during the workshop at PLoP 2004. The author would like to thank Joel Jones, Crutcher Dunnawant, Bob Blakley, Craig Heath, Bob Hanmer, and others who participated in the workshop for their insightful observations and suggestions.

BIBLIOGRAPHY

[Ack95] Ackroyd, M., "Object-oriented design of a finite state machine," Journal of Object-Oriented Programming, pp. 50, June 1995.

[Ada03] Adamczyk, P., "The Anthology of the Finite State Machine Design Patterns," Pattern Languages of Programming conference, PLoP 2003. Available at: http://pinky.cs.uiuc.edu/~padamczy/docs/plop03.pdf.

[Arm97] Armstrong, E., "Create enumerated constants in Java," Java World, July 1997.

[BMRSS96] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, "Pattern-Oriented Software Architecture: A System of Patterns," John Wiley and Sons, 1996.

[Car92] Cargill, T., "C++ Programming Style," Addison Wesley, 1992.

[DA96] Dyson, P. and B. Anderson, "State patterns," European Pattern Languages of Programming conference, EuroPLoP 1996. Published in PLoPD3, ch. 9, p. 125-142. Also available at: http://www.cs.wustl.edu/~schmidt/europlop-96/papers/paper29.ps.

[Dou98a] Douglas, B. P., "Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns," Addison Wesley, 1998.

[Dou98b] Douglas, B. P., "Real-Time UML, Developing Efficient Objects for Embedded Systems," First Edition, Addison Wesley, 1998.

[FR98] Ferreira, L. and C. M. F. Rubira, "The Reflective State Pattern," Pattern Languages of Programming conference, PLoP 1998. Available at: http://jerry.cs.uiuc.edu/~plop/plop98/ final_submissions/P42.pdf.

[GHJV95] Gamma E., R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Object-Oriented Software," Addison Wesley, 1995.

[Har87] Harel, D. "Statecharts: a Visual Formalism for Complex Systems," Science of Computer Programming, Vol. 8, pp. 231-274, 1987.

[Hen00] Henney, K., "Collections for States," Java Report, August 2000. Also available at: http://www.two-sdg.demon.co.uk/curbralan/papers/CollectionsForStates.pdf.

[Hen02] Henney, K., "From Mechanism to Method: State Government," C/C++ Users Journal, June 2002. Also available at: http://www.cuj.com/documents/s=7982/cujcexp2006henney.

[Hen03] Henney. K., "Methods for States," Nordic Pattern Languages of Programming conference, VikingPLoP 2002. Also available at: http://www.twosdg.demon.co.uk/curbralan/papers/MethodsForStates.pdf.

[HJE95] Huni, H., R. Johnson, and R. Engel, “A Framework for Network Protocol Software,” Proceedings of Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA) 1995, pp. 358-369.

[Ing86] Ingalls, D., “A simple technique for handling multiple polymorphism,” Proceedings of Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA) 1986, pp. 347-349.

[Mar95] Martin, R., "Three Level FSM," Pattern Languages of Programming conference, PLoP 1995. Published in PLoPD. Also available at: http://cpptips.hyperformix.com/cpptips/fsm5.

[OS96] Odrowski, J., and P. Sogaard, "Pattern Integration - Variations of State," PLoP96. Available at: http://www.cs.wustl.edu/~schmidt/PLoP-96/odrowski.ps.gz.

[Pal97] Palfinger, G., "State Action Mapper,” Pattern Languages of Programming conference, PLoP 1997. Available at: http://jerry.cs.uiuc.edu/~plop/plop97/Proceedings.ps/palfinger.ps.

[Ran95] Ran, A., "MOODS, Models for Object-Oriented Design of State," Pattern Languages of Programming conference, PLoP 1995. Published in PLoPD. Also available at: http://www.soberit.hut.fi/tik-76.278/alex/plop95.htm.

[Sam02] Samek, M., “Practical Statecharts in C/C++,” CMP Books, 2002.

[SC95] Sane, A. and R. Campbell, "Object-Oriented State Machines: Subclassing, Composition, Delegation, and Genericity," OOPSLA ’95. Also available at: http://choices.cs.uiuc.edu/sane/home.html.

[Spa00] Spaanenburg, L., “Digital Systems” notes from: http://www.cs.rug.nl/~rudy/onderwijs/DT/Lectures/Lecture_2.ppt.

[vGB99] van Gurp, J. and J. Bosch, “On the Implementation of Finite State Machines,” Proceedings of the IASTED International Conference, 1999. Also available at: http://www.xs4all.nl/~jgurp/homepage/publications/fsm-sea99.pdf.

[YA98a] Yacoub, S. and H. Ammar, "A Pattern Language of Statecharts," Pattern Languages of Programming conference, PLoP 1998. Also available at: http://citeseer.nj.nec.com/yacoub98pattern.html.

[YA98b] Yacoub, S. and H. Ammar, "Finite State Machine Patterns," European Pattern Languages of Programming conference, EuroPLoP 1998. Published in PLoPD4, ch. 19, p. 413-440. Also available at: http://www.coldewey.com/europlop98/Program/Papers/Yacoub.ps.

APPENDIX

Here are summaries of other FSM patterns referenced in this paper that appear in the complete version on the web site at: http://pinky.cs.uiuc.edu/~padamczy/fsm.

Enumerated Event

The main light FSM example provides a simple FSM implementation with all FSM elements implemented using the least complex model. The least complex way to refer to an event is to assign it a symbol (a number or a name).

How to define events as simply as possible?
There are several simple solutions.
Define events as integers, as enumerated values, or as strings.

Event Methods

Let us reconsider Ignored and Error transition functions from the main light FSM example. Most {event, state} pairs map to these two transition functions. Also each event can result in only two transitions, which is much smaller than the number of states. In such FSMs, it is simpler to consider transitions as event- rather than state-dependent. This means that events should be encapsulated as separate functional entities. So define separate methods for each event.

You would like to associate behavior corresponding to specific events with the invocation of these events.
Define one method for each (internal and external) event.

Unstructured Actions

The main light FSM example provides a simple FSM implementation with all FSM elements implemented using the least complex model. The least complex way to define an action is to implement it as a sequence of statements that is associated with an event or a state, e.g. as a case statement in a switch.
How to define actions as simply as possible?
Implement actions as a sequence of steps corresponding to specific combinations of current state and incoming event. Actions could be implementations of clauses in if statements or cases within switch statements.

Action Methods

Being mechanical devices, traffic lights sometimes malfunction. This happens usually during bad weather conditions, which compounds the resulting chaos. In response to a mechanical failure, traffic lights change to blinking red lights and every intersection becomes a four-way stop.

Handling malfunctioning traffic lights in the software is a complex, multi-step process. First, store all available error-related information in a permanent storage. Then reset all statistical data. Next send an alarm to the traffic control office. Finally transition to the BlinkingRed state. To support

this sequence of steps effectively, it is necessary to handle the MechanicalError event in every state. Rather than duplicating the handling in every state, encapsulate the complete action as a method.

This situation seems similar to the one described in EVENT METHODS. The difference is that here, the action method has many steps so it is important to encapsulate them in one place in the code. The example in EVENT METHODS does not contain multiple steps. Instead, it suggests separating behavior based on the event (rather than state), because that produces a more coherent design for that specific case.

How to give structure to unstructured actions?
Define a method for each action.

Transition Table [Car92]

The description of main light FSM example uses two graphical representations of the FSM – a table and a state diagram. The tabular representation is much more structured than the State diagrams. It is more readable and easier to change, especially if new transitions occur between states that are far apart in the state diagram. The tabular format is easier to read and follow, because each event corresponds to one row that defines its relationship to each FSM state. Encode this tabular description in the FSM code.

Mixing the code that implements FSM behavior with the logic for changing states makes state changes hard to follow. Need to define a separate structure for state transitions.
Localize the mapping of states and events to a specific action in a transition table. In the table, map {current state, event} pair to {new state, action function} pair.

State-Driven Transition [DA96]

Let us look at the main light FSM example again. In each state, there are at most two events that cause a transition to a different state. Other events are either handled without changing the state, or ignored. In such a kind of FSM, defining state changes explicitly in the code is a good strategy. State transitions are not frequent and defining a separate entity to encapsulate them is not necessary. In this case, the best approach is to define state transitions as a part of the behavior of the state.

Need to define state transitions as the integral part of the flow of control.
Implement state transitions in the same code where the behavioral logic is defined. Treat state transitions as a natural extension of the FSM logic.