Finite State Machine Patterns

Sherif M. Yacoub; Hany H. Ammar
Computer Science and Electrical Engineering Department
West Virginia University, Morgantown
West Virginia, WV26506

INTRODUCTION

Finite state machines (FSM) are widely used in many reactive systems to describe the dynamic behavior of an object based on its state. The theoretical concepts of FSMs and an object's specifications, in terms of state transition diagrams, have long been used. Here, we show how to solve several recurring design problems in implementing a state machine. We present a basic design pattern for finite state machines whose design evolves from the general understanding of state machines functionality. Then we extend the basic pattern to support solutions to several design problems that are commonly thought of by system designers. These design decisions include the state-transition mechanisms, the design structure, state-instantiation techniques, and the machine type. Since finite state machines are frequently applicable to areas of concurrent and real-time software, thus it is useful for the system designer to search a catalog of classified state machine patterns. The patterns cover the three-layer FSM pattern by Robert Martin[5] and extend the set of patterns described by Paul Dyson and Bruce Andreson [2].

The next section summarize the patterns for finite state machine and show those that were addressed by Erich Gamma *et al*[1] , Robert Martin[5], and Paul Dyson and Bruce Anderson[2]. The following sections describe the other patterns addressed in this work.

SUMMARY OF THE PATTERNS

The set of patterns presented here constitutes a pattern language of finite-state machines. First, the patterns are classified based upon frequent design decisions that are usually addressed in the finite state machine context. Figure (1) shows the classification and describes how they are related to each other. The inheritance sign shows that a pattern extends another by providing a solution to another design problem. The double-headed arrow, with "X" label, indicates that only one of the patterns will appear in the design because they are contradicting from their motivation or solution facets. The dotted arrow shows that one pattern motivates and leads to the use of another, the arrowhead shows the direction of motivation.

The finite state machine (FSM) pattern is an extension of the basic state pattern of Gamma *et al.*[1], also referred to as State Object[2], it adds implementation of the state transition diagram specifications in terms of actions, events, and a state transition mechanism. In figure(1), the labeled single-headed solid arrow indicates the classification of the FSM pattern, according to state transitions, into two basic types: Owner Driven and State-Driven patterns. The two patterns are in tension with each other as discussed by Paul *et al.*[2]

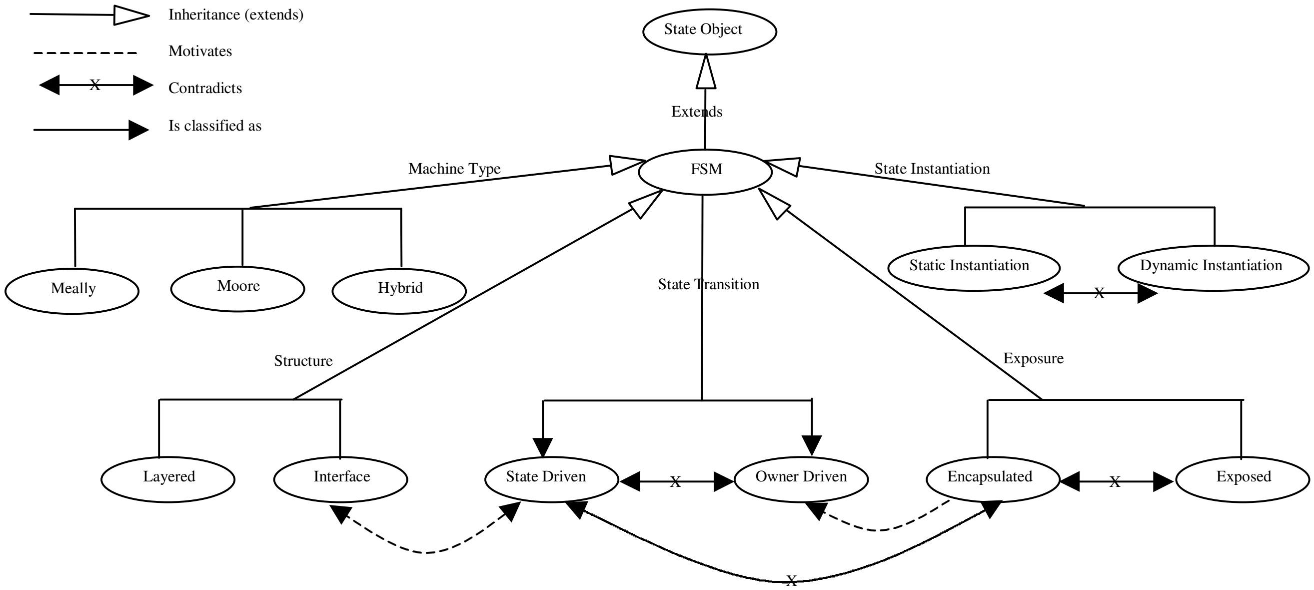

Figure 1 Relationships between State Machine Patterns

According to the mechanism for producing outputs; i.e. machine type, the basic FSM pattern is extend into Meally, Moore or Hybrid, which describes whether the outputs are dependent on the object's current state only or dependent on the events as well.

State instantiation can be either static or dynamic. In Static-Instantiation pattern, all the object's states are created at the initialization phase, while in the Dynamic Instantiation pattern, the states are created dynamically during run time. Only one of the two patterns will be incorporated in your state machine design, depending on the number of states, the required response time on state transitions, and the availability of memory as discussed later.

The FSM design can be structured into Layered and Interface organizations, the layered organization splits the behavior and the logic transitions so that the machine can be easily maintained and comprehended. The interface organization allows the design to be embedded into the overall application design and facilitate communication between other entities and the machine design.

From the exposure side, the current state of the FSM can be either exposed to other application entities to have direct invocation to the state class methods; i.e. Exposed State pattern, or it is encapsulated inside the object only; Encapsulated State pattern. You will only use one of these two patterns by choosing either to expose your object's current state or prevent access to it.

Usage of one pattern may lead to usage of another. If you decide to use the Encapsulated state pattern thus you will need to secure access to the state object, therefore you will use an owner driven transition, because using a state driven necessitates having an assessor method to the object's state in the owner which violates the encapsulation motivation. Also, usage of interface leads to thinking of applying a state-driven transition and vice versa, because moving the state transition logic to the states is a step in simplifying the object's interface to other application entities.

Table (1), summarizes the patterns as problem/solution pairs.

Design Decisions	Pattern Name	Problem	Solution	Ref.
	State Object Pattern	How can you get different behavior from an object if it differ according to the object's state?	Create states for the object, describe its behavior in each state, attach a state to the object and delegate the action from the object to its current state.	Erich Gamma *et al.*[1] Paul Dyson *et al.*[2]
	Basic Finite State Machine Pattern	Your object's state changes according to events in the system. The state transitions are determined from the object specification. How can you implement the object behavior in your design?	Use the state object pattern and add state transitions methods in response to state transition events. FSM pattern = State Object Pattern + State Transition Mechanism	Robert Martin[5] Paul Dyson *et al.*[2] *
State-Transition	State-Driven Transition FSM pattern	How to hide the state transition logic from the object class in a state machine pattern?	Have the states of the object initiate the transition from self to the new state in response to the state-transition event.	Paul Dyson *et al.*[2] *
State-Transition	Owner-Driven Transition FSM Pattern	You want your states to be simple and shareable with other objects and you want the object to have control on its current state. How can you achieve this?	Make the object respond to the events causing the state transitions and encapsulate the transition logic in the object	Paul Dyson *et al.*[2] Robert Martin[5]
Structure	Layered-Structure FSM Pattern	You are using a FSM pattern, how can you make your design maintainable, easily readable and eligible for reuse?	Organize your design in a layered structure that de-couples the logic of states transition from the object's behavior defined in terms of actions and events	Robert Martin[5] *
Structure	Interface Organization FSM Pattern	How can other application entities communicate and interface to an object whose behavior is described by a finite state machine?	Encapsulate states and logic inside the machine and provide a simple interface to other application entities that receives events and dispatches them to the current state.	*

Machine Type	Meally FSM Pattern	How do you activate the FSM outputs if they should be produced at specific events while the object is in a particular state?	Make the concrete event method of each state initiate the required (output) action in response to the event.	*
	Moore FSM Pattern	How do you activate the FSM outputs if they are produced only at the state entry, and each state has a specific set of outputs?	Implement an output method in each state that calls the required actions. Make the state transition mechanism call the output method of the next upcoming state.	*
	A Hybrid FSM Pattern	What do you do if some FSM outputs are activated on events and some other outputs are activated as result of being in a particular state only?	Make the event method of each state produce the event-dependent outputs and make the state transition mechanism call an output method of the upcoming state to produce the state-dependent output.	*
Exposure	Exposed FSM Pattern	You want to allow other external entities in your application to know of your object's state and have access to call some of the state's methods.	Provide a method that exposes the state of the object and allows access to the current state.	Paul Dyson *et al.*[2]
	Encapsulated FSM pattern	Your FSM should follow a sequence of state changes that should not be changed by other application entities. How can you ensure no state changes are enforced to your object?	Encapsulate the current state inside the object itself and keep the state reference as a private attribute. Only the object itself can change its state by handling the events causing the state change but still delegate the behavior to the current state.	*
State Instantiation	Static State Instantiation FSM pattern	Your application is small and it has few states. Speed is a critical issue in state transitions. How do you instantiate your object's states?	Create instances of all possible states on the object instantiation. Switch from current to next state by altering the reference to the next state	Robert Martin [5]
	Dynamic State Instantiation FSM Pattern	Your application is large and you have too many states. How do you instantiate the states in your application?	Don't initially create all states, make each state knowledgeable of the next upcoming states. Create instances of upcoming states on state entry and delete them on state exit.	*

Table 1 State Machine Patterns

* Addressed in the following sections

BASIC FINITE STATE MACHINE PATTERN

Context

Your application contains an object whose behavior depends on its state. The object's state changes according to events in the system and the state transitions are determined from the object specification.

Problem

How can you implement the behavior of the object in your design?

Forces

In many reactive systems, the behavior of the object depends on its state. The behavior is described by the actions that the object takes in response to events, and the object's state may change that allow it to handle upcoming events as of the new state. You can implement the object using a single class and use an attribute to keep track of the object's state. For example, if you have a coin machine object and you want to insert a coin to pass, the machine could be operating (Locked or UnLocked) or broken (due to equipment failure). The object should check its current state to allow the person to pass (if it is working) or display an out of order message and prevent him from passing (when it is broken). This could be implemented in the event method of coin insertion using a conditional check statement that would look like:

```
Coin_Machine::coin()
{ switch(MyState) {
       case Broken :   // Display out of order message
       case Locked :   // Unlock the machine's gate
       ----       };
}
```

This is a traditional approach that is easy to implement, but you will have to implement this state checking in all of the object's methods because you cannot make the object take any actions unless it is in a correct state. This will make the class methods look cumbersome and will not be easily maintained.

Moreover, you will also prefer to have a design that clearly distinguish the states, the events, and the actions because that is usually what you will think of in terms of an object's behavior as related to the application environment. Figure (2) shows the behavior specification that we would like to imitate in our design.

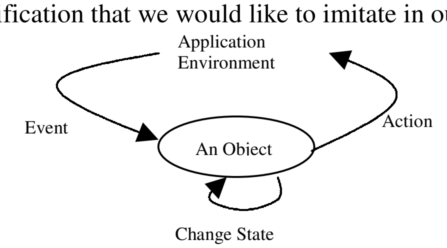

Figure 2 The object behavior in an application environment

Solution

Implement the object's behavior using a design that distinguishes:

- *The Object*: The entity whose behavior is to be described. e.g. A Coin Machine
- *States*: A state of the object describes the behavior and reaction of the object to events occurring while the object is in that specific state.
- *Events*: A finite state machine object reacts to stimulating events in the application environment. The events may be triggered by external real-world entities or by other application entities.

- *State Transitions*: The change of the object's state results by an event. The state transitions are driven by events but not all events produce state transitions.
- *Actions*: The actions taken by the object that affect the external world and/or other application entities. Generally, there will be some state-dependent actions and some state-independent ones.

To design the FSM based on the above model we:

- Construct an abstract state class "`AState`" which contains static methods implementing the actions taken by all states of the object.
- Inherit the possible concrete classes representing the actual states of the object. Create virtual methods for all possible events in the "`AState`" class. The concrete classes implement these methods as specified for the behavior of the object in each state and invoke of the actions that affects the application environment.
- Create a class for your object that contains a current state of type "`AState`". Delegate all events received by the object to it. Choose a state transition mechanism, State-Driven or Owner-Driven Transitions.

Example

Consider the coin machine example described by Robert Martin [5] in the three level FSM, figure (3) summarizes the specification using a state- transition diagram.

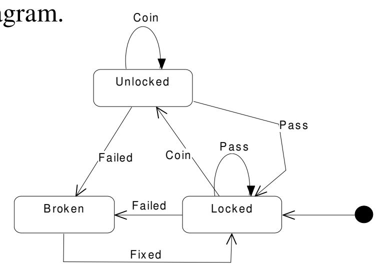

Figure 3 Example: The state transition diagram of a coin machine

Given the above specification for the coin machine, how do you use the FSM pattern to implement its behavior:

1. Identify and create the concrete state classes "`Locked`", "`Unlocked`" or "`Broken`"
2. For each concrete state class, implement the event methods: "`Coin`" method for coin insertion, "`Pass`" method for person passage, a "`Failed`" method for machine failure and a "`Fixed`" method after being fixed. All the events are virtual methods in the abstract class "`AState`" and are implemented on each state accordingly. You need not implement all events in every state only those which are identified from the state diagram, for example, the "`Pass`" event need not be implemented in the "`Broken`" state of the machine.
3. Implement static methods for all possible actions in the "`AState`" class. The specification shows that the following actions are taken by the state machine in various states: "`Unlock`" method allows a person to pass, "`Lock`" prevents a person from passing, a method to display a "`Thankyou`" message, another to give an alarm "`Alarm`", display an out-of-order message "`Outoforder`", and "`Inorder`" method when the machine is repaired.

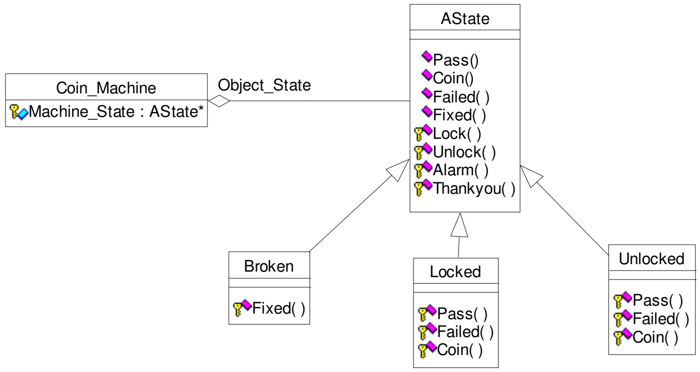

Figure 4 FSM Pattern for coin machine

Discussion

We used a FSM pattern to translate the model of a standard state transition diagram (STD) to an object-oriented design. The translation process maps the elements thought of and described in the STD to classes and methods in an OO design. Since STDs are used frequently by system analysts to describe the behavior of an object, then the FSM pattern will help a designer map the analyst work into implementation. The state transition mechanism is deferred to another design decision that is Owner-Driven or State-Driven transitions.

STATE-DRIVEN TRANSITION FSM PATTERN

Context

You are using a basic state machine pattern.

Problem

What do you do to hide the state transition logic from the object class?

Forces

When you are using a basic state machine pattern, you will need to complete the object's behavior implementation by adding a state-transition mechanism to your design. Since the object holds a reference to its current state class for the purpose of delegating events, then you can intuitively implement the state transition inside your object. For example, in the coin machine you can implement the state transition from "`Locked`" to "`Broken`" in the object itself when it receives a "`Failure`" event that will look like

```
Coin_Machine::EventHandler(EVENT event)
{ switch(event)
  {  case Failure : delete Machine_State ;  // delete the old state object
                    Machine_State = new Broken(); // create the new state object
   }
}
```

This has a disadvantage of implementing every condition that combines the current state and the event that causes state transitions in the object itself. This will increase the complexity and add several condition checks in the object. But you want the object to be simple that is why you first chose to use a state machine pattern and delegate the event processing to the current active state class. Therefore, you will think of delegating the state transition logic to the current state instead of processing the event twice; once for the logic transition in the object and another for the behavior implementation in the state class.

Solution

Delegating the state transition logic to the state classes, make each state knowledgeable of the next upcoming states and have the concrete states of the object initiate the transition from self to the new state. In this case, the concrete states have to know their object. Thus, create a static pointer in the abstract class "AState" to be shared by all concrete states and accessible via a static method for state changes that can be invoked from any concrete state class. The object current state has to be exposed to the state change mechanism encapsulated in the "AState"

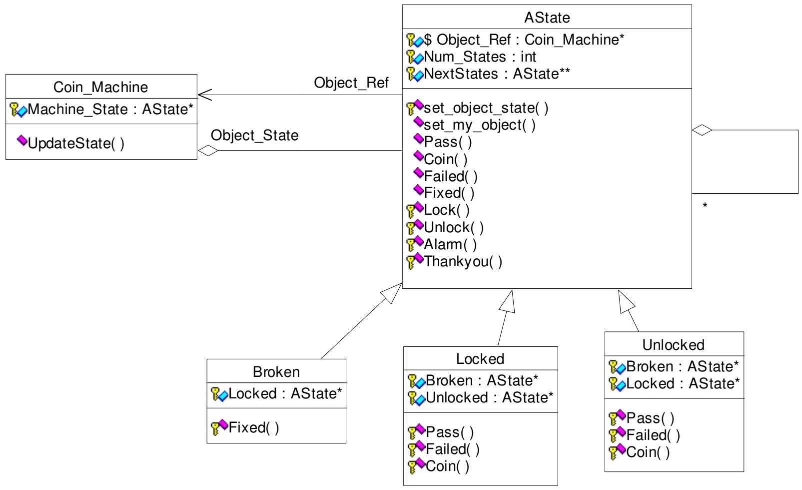

Figure 5 State-Driven transition FSM pattern

Example

So, how can you use the pattern to solve the following problem: you don't want the "Coin_Machine" to know about the transitions from one state to another; it just dispatches the events to the current state in the event handler method. Make each state knowledgeable of the next coming state; for example the "Broken" class has pointers to the "Locked" class. We use the pointer to self "NextStates" in the abstract class to provide generic pointers to upcoming states. We added the "Object_Ref" to point to the coin machine object and use the "set_object_state" to change its current state. From the specification of the problem (STD in figure 3), we identify the transitions from source states to destination states, and in the implementation of the event causing the transition in the source state, we call the "set_object_state" with the destination state as an argument. For example, in the implementation of the "Fixed" event we change the objects state by calling set_object_state(Locked).

Discussions

Although the state driven transition simplified the coin machine class implementation it added the burden to the concrete state classes which requires more instantiation and declaration efforts to ensure that each concrete state class points correctly to the next states. Another liability is that sharing the states between several objects will not be feasible in the design because there is only one object the concrete states are pointing to.

INTERFACE ORGANIZATION FSM PATTERN

Context

You are using a basic finite state-machine pattern to implement the behavior of an object.

Problem

How can other application entities communicate and interface to your object?

Forces.

Your object may not be a standalone design but rather it would be embedded in an application. You can embed the object's design in the overall application design, but then how will other application entities interact with the object? Should these other entities keep references to the object and all of its states? For example, assume the coin machine application has another class for the machine maintenance, when the maintenance is done to the machine, the maintenance class should notify the coin machine by a "`Fixed`" event, another example is the coin insertion event received from the sensors of the machine. Therefore, you want to define an interface to the object to interact with, this necessitates decoupling the object interface and the states' logic. The transition logic and implementation of events can be hidden from the object interface, which simplifies the object's interface to other parts of the application environment. The interface role will be receiving events and dispatching them to its state-implemented behavior.

Solution

Encapsulate the logic transition in the states and hide it from the object interface i.e. use a state-driven transition mechanism. Design the FSM to distinguish the interface that receives events and the states that handle events, invoke actions, and maintain the correct current state of the object.

Example

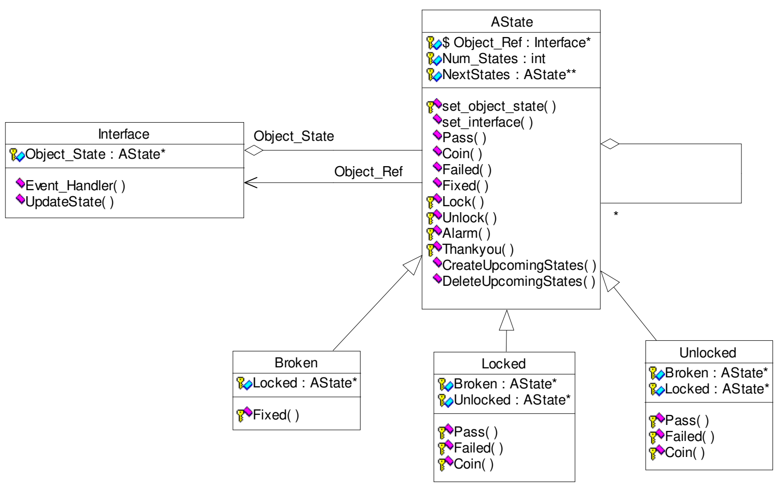

Figure 6 Interfaced Organization FSM Pattern

In the coin machine example, we create an "`Interface`" class and an "`Event_Handler`" method to handle and dispatch the events. The state transitions are implemented as state-driven FSM. The object "`Interface`" class acts as interface to the logic encapsulated in the FSM. The Interface knows which state the object is currently in and so, it handles the incoming events and invokes the appropriate state event method. The "`Event_Handler`" captures events in the application environment and calls the state implementation of the event accordingly. Separation of the interface and the actual logic and implementation of the state machine ease the design reuse and the interface to the other parts of the application.

LAYERED ORGANIZATION FSM PATTERN

Context

You are using the basic finite state-machine pattern to implement the behavior of an object.

Problem

How can you make your design maintainable, easily readable and eligible for reuse?

Forces

When you use the basic finite state-machine pattern to describe the behavior of the object, you find that the events and the actions are all defined in the "`AState`" class. For example, the "`AState`" class in the coin machine contains a large set of methods, this often occurs when you try to simplify the object's interface and encapsulate the actions, events and state transitions in the state classes, which makes them more complex even in a simple example as the coin machine.

Earlier figure (2) showed that the interaction of the object with the environment describes its behavior as events received by the object and actions taken by it, but you cannot clearly distinguish this behavior in the basic state machine pattern because both are defined in the state classes. Therefore, you would rather separate the events and the actions in different design layer; i.e. the object's behavior layer. This allows users of the pattern to understand and easily modify the design. Understandability of the design and its maintainability are the main forces deriving a layered approach.

Solution

Organize your design in a layered structure that de-couples the logic of states transition from the object's behavior defined in terms of actions and events

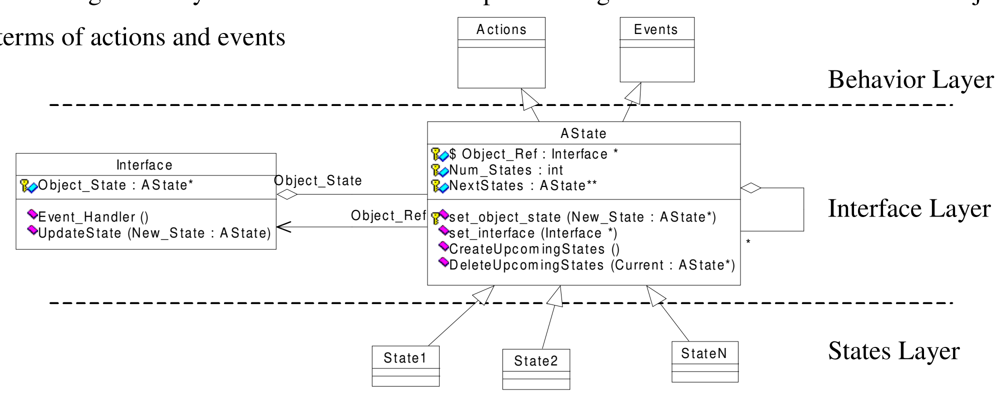

Figure 7 A Layered Organization FSM Pattern

The structure has three layers:

- *The Behavior layer*: The behavior of the state machine is described as "`Actions`" and "`Events`".
- *The Interface layer*: This layer has the interface of the pattern that reacts to external events and calls the attached state to behave accordingly.
- *The States layer*: This layer describes the concrete states of the machine.

The "`Event`" class contains all events that occur in the environment and the FSM responds to. It is an abstract class; the response to each event will differ according to the current state, thus each concrete state will implement the adequate functionality for that particular event. The "`Actions`" class contains all the methods that can be executed in the state machine and will affect the application environment or invoke another event. These actions describe the outputs of the state machine called by event methods in the concrete classes. In many cases, we will have one implementation of actions used by several classes.

Example

In the coin machine example, the events are distinguished as "`Coin`", "`Pass`", "`Failed`" and "`Fixed`" methods. The possible actions are "Lock" to prevent passengers from passing, "`Unlock`" to allow them to pass, "`Thankyou`" a displayed message and others. Thus the layered solution is used and the events and action classes will contain these methods.

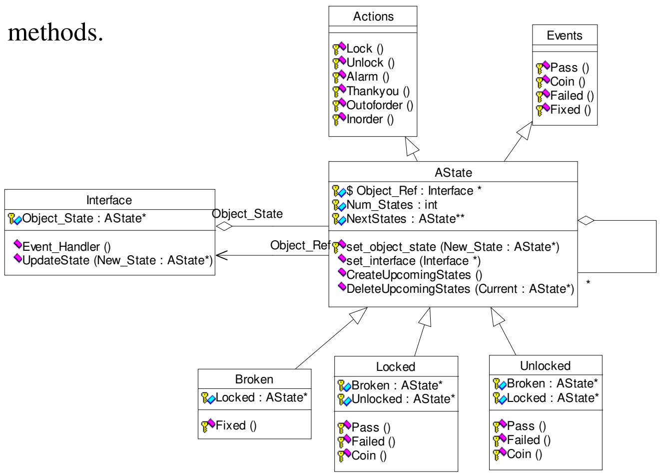

Figure 8 Layered Organization for FSM Patterns

So, how does this structure help the maintainability of the design? Assume that the designer will change the method implementation of the "`Lock`" method due to installation of new locking mechanism. Instead of getting lost in large number of methods, he can easily consult the layered organization design for "`Lock`" method in the "`Action`" class and modify it. The structure of the machine separates actions, events and state transition that eases its maintenance and reuse. The multiple inheritance followed by several single inheritances shows a three-layer architecture that simplifies the state machine design.

THE MEALLY FSM PATTERN

Context

You are using a finite state machine pattern. The machine responds to events in the application environment by producing outputs (actions).

Problem

How do you activate the FSM outputs?

Forces

You are describing the behavior of the object using a state machine pattern and part of the behavior is to identify the output actions of the machine and when to produce them. You have identified these outputs (actions), but you find that they should only be produced in response to specific events depending on the object's state. Thus, you want the actions taken by the object to be associated with the object's present state and the current inputs affecting it [3]. For example, the coin machine should produce a "`ThankYou`" message if a coin is inserted and it is in the "`UnLocked`" state, which is an action associated with the event "`Coin`" and the state "`UnLocked`". In a design context, the inputs are those events occurring in the applications domain, thus you will need to associate the activation of outputs with the event handling in each state.

Solution

Let the states produce the outputs in response to events. In each concrete state, implement the calls to the necessary action methods from the concrete method implementation of the events.

Example

In the coin machine example, a "`Thankyou`" message will appear each time the machine is unlocked and a coin is inserted, and similarly other event/output pairs as specified in the following Meally version specification:

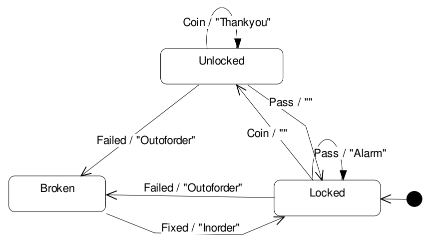

Figure 9 A Meally machine example

As an example, in the event handling of "`Failed`" in class "`Unlocked`", a call to "`Outoforder`" will be implemented as follows:

```
Void Unlocked::Failed() { Outoforder(); //
// The Outputs associated with input event "Failed" in state "Unlocked" }
```

Whenever an event is required to produce an output, the output actions can be called from inside the event method of that specific state.

THE MOORE FSM PATTERN

Context

You are using a finite state machine pattern. The machine produces specific set of outputs depending on its current state only.

Problem

How do you activate the state machine outputs?

Forces

You have identified the set of outputs (actions) that the machine produces and implemented them as methods in the state machine pattern. You can then apply the Mealy machine pattern to implement the calls to these outputs; but then you find that you are replicating calls to these output method because you want to produce the output for the machine in a given state and thus you will have to check all the state entry conditions from other states and add the calls to the output method in each one of them. For example, if a warning lamp is required to be lit on each time the coin machine is in the "`Broken`" state, thus you will have to call the output routine to light on the lamp in two situations, one in the "`Failed`" event for the "`Locked`" state and the other in the "`Failed`" event in the "`UnLocked`" state. This has two disadvantages: the first is that you will have many calls to the output method which increases as the number of state entry increases. The second disadvantage is the maintenance of the design, you are calling outputs associated with being in a state from the event methods of other states. Therefore, you would rather associate the actions taken by the object with the object's present state only [3]. In a design context, this is translated as producing the outputs at the state entry and hence you don't have to worry about calling outputs from the event methods of other states.

Solution

Create an output method for each concrete state, implement the calls to the required actions in the output method, and make the state transition mechanism call an output method of the next upcoming state. In a state-driven transition FSM, the machine changes state by calling the `"set_object_state"` method. Thus we add to the previous design a method called `"Output()"` which is *specific* for each state and is called by the `"set_object_state"` routine using the new state as the caller. In an owner driven transition FSM, the output method will be invoked from the owner after each transition condition is satisfied.

Example

We will use an example of the state-driven transition. For instance, in our example, a lamp will be turned on whenever the machine is broken and off whenever it is operating in either locked or unlocked state. A Moore machine version specification is shown in figure (10).

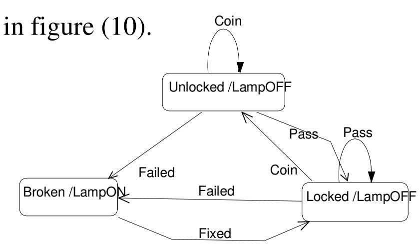

Figure 10 A Moore machine example

Thus the `"Output()"` method of the `"Locked"` and `"Unlocked"` states will set the lamp off , and the `"Output()"` method of the "`Broken`" state will turn it on. The call to the "`Output()`" method will be invoked as follows:

```
AState::set_object_state(AState* New_State){
New_State->Output(); // Call the output of the upcoming state }
```

Whenever a Moore machine is required, the output method and its calls can be added. Figure (11) shows the design using a Moore FSM.

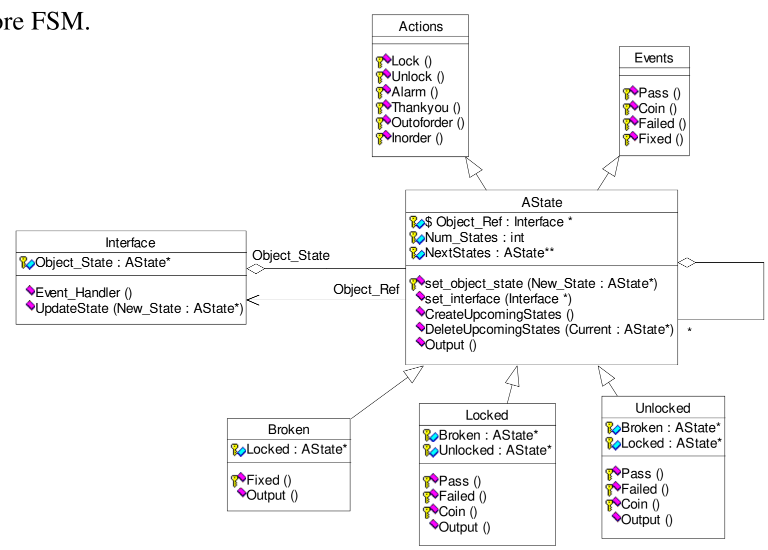

Figure 11 The Moore machine pattern structure

HYBRID FSM PATTERN

Context

You are using a finite state machine pattern. The machine produces some outputs in response to events and some other outputs are associated with the object's state.

Problem

How do you activate the state machine outputs?

Forces

You are using a state-machine pattern and you have decided to use a Meally machine, but you find that some outputs are dependent on the states only, however, you cannot use a pure Moore machine because some other outputs are dependent on the events response. But does the implementation of a Meally machine contradict that of a Moore one? As discussed in their solution sections, they are not. Therefore, you can use a hybrid machine where some actions taken by the object are associated with the object's present state only (A Moore behavior), and some other actions are constricted by both the object's state and an event in the application (A Meally behavior).

Solution

Use a combination of the Meally and Moore FSM patterns, the pattern solutions provided in each of them are not contradicting and in fact can be used together.

Example

In the coin machine example, we desire the "`Lamp`" output to be associated with the states and the "`Thankyou`" message is generated only on the "`Coin`" insertion event while in the unlocked state. Thus, in the event handling of "`Failed`" in class "`Unlocked`", the call to "`Outoforder`" method is placed as shown in the Meally pattern and we also add the "`Output()`" method which will be called by the "`set_object_state`" method using the new state as the caller. The "`Output()`" method of the "`Locked`" and "`Unlocked`" states will set the lamp off , and the "`Output()`" method of the "`Broken`" state will turn it on as explained in the Moore pattern.

ENCAPSULATED FSM PATTERN

Context

You are using a finite state machine pattern. The sequence of state changes is defined in the object's specification.

Problem

How can you ensure no state changes are enforced to your object?

Forces

Paul Dyson *et al.* [2] discussed the exposed state pattern that allows other application entities to access and retrieve the object's state. This was shown of benefit to prevent the owning class from having large number of methods that are state-specific and state-dependent, but this allows other application entities to know about the object's state and can possibly change it. This might contradict with your desire to keep the states known to the object only, a situation that often occurs for safety purposes. This arises when the specification of the object's behavior necessitates the sequence of state transitions are to follow the causes(events) only. For example, in an automated train control system, you want to open the train's door if and only if the train has stopped, thus you cannot expose the train state because an external entity can accidentally inject an event causing doors to open.

Solution

Encapsulate the current state inside the object itself and keep the state reference as a private attribute. Only the object itself can change its state by handling the events causing the state change but still delegate the behavior to the current state. Thus the owner driven transition[2] should be implemented and the concrete state reference should be private or protected.

Example

In the coin machine example, the "`Object_State`" is declared as protected, and the event handler will not only delegate the handling to the concrete state implementation but also changes the reference to the new concrete state.

Related Patterns

The Encapsulated pattern is related to the Owner-Driven transition pattern from the solution perspective, however, the problems and forces are different that is the rational behind documenting them as different patterns.

DYNAMIC INSTANTIATION FSM PATTERN

Context

You are using a finite state machine pattern to implement your object's behavior. The application in which you are using the object is large and the object has many states.

Problem

How do you instantiate the states in your application?

Forces

You can statically instantiate all the states of the object at the initialization phase as described in the three level machine by Robert Martin[5], whose implementation would look like:

```
Class Coin_Machine {
       static Broken BrokenState;
       statIc Locked LockedState;
       static UnLocked UnLockedState;
}
```

But since the number of states is enormous in large applications, this will consume large memory space thus decreasing the availability of free memory. Therefore, it is preferable to keep few instances of the states loaded at a time such as the current state and the possible upcoming states. Thus the number of states kept loaded is small and varies according to the current state.

Solution

Upon state transitions, load the upcoming state and unload the current state then update the object’s state with the new one. This design decision has several implementations depending on the selected state-transition technique of the machine. As an example, when each state is knowledgeable of the next upcoming state, i.e the State-Driven transition mechanism, let the old state create the next state object and the invocation of the state change method in the object deletes the previous state.

Example

In the coin machine example, if required to dynamically instantiate the states *, we add the two methods "`CreateUpcomingStates`" and "`DeleteUpcomingStates`" which are called from an "`UpdateState`" method of the coin machine as follows:

```
void Coin_Machine::UpdateState(AState* New_State){
   Object_State->DeleteUpcomingStates(New_State);
   delete Object_State;
   Object_State = New_State;
   Object_State->CreateUpcomingStates();};
```

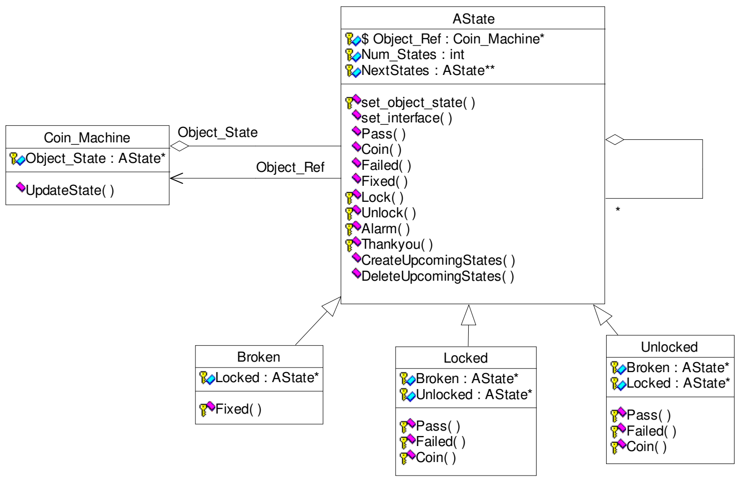

Figure 12 Dynamic Instantiation of states

These two methods are implemented for each concrete state to create and delete its "`NextStates`". For example, the "`Locked`" state in our example will the following implementations:

```
void Locked::DeleteUpcomingStates(AState* CurrentState)
         { for(int I =0; I < Num_States; I++)
              { if(CurrentState != NextStates[I])
                delete NextStates[I]; }
              delete NextStates;     }
void Locked::CreateUpcomingStates()
{   Num_States =0 ;
   NextStates = new AState*[2];
   NextStates[Num_States] = new Broken();   Num_States++;
   NextStates[Num_States] = new Unlocked();   Num_States++;   }
```

Discussion

The dynamic instantiation mechanism has the advantage of keeping few state objects loaded at a time, however it has a drawback of slowing down the state transition operation, because we delete the previous states and create objects for the upcoming states. This makes the dynamic instantiation pattern not applicable for real time systems and hence static instantiation is recommended.

* In this example, the number of states are small and hence static instantiation is more suitable, however, we used this simple example for illustration purpose.

SUMMARY AND KNOWN USES

A pattern language of finite state machine is discussed in which the relationship between the patterns is identified. Other patterns related to State Object are discussed by Paul *et al.* which can be integrated to this catalog. They include solutions to reusing state objects (Pure State pattern), initializing default states (Default State pattern), and allocating data members in the design objects (State Member pattern).

Finite state machines are widely used in many reactive systems, their design represents a general problem to be addressed by system designers. They are often used in communication systems in which the status of the link between two or more communicating entities limits the behavior of the above application layers. FSMs are widely used in control systems such as motion control system of automated trains, elevators control, and automated train door control. Gamma *et al.*[1] have pointed some known uses in graphical user interfaces. Paul *et al.*[2] have also pointed out their usage in library applications. Automated Teller Machines are one of the most known and frequently used illustrative examples for an application whose state plays a major role in the flow of operations.

ACKNOWLEDGEMENT

We would like to thank Andreas Rueping, who was our EuroPLoP shepherd, for his valuable comments and feedback which helped us to identify the exact problem/solution pairs in the work and to sharpen various sections of the patterns.

REFERENCES

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

[2] Paul Dyson and Bruce Anderson, "State Patterns". In Robert Martin, Dirk Riehle, and Frank Buschmann (edt.) "*Pattern Languages of Program Design 3*", Addison Wesely Longman Inc 1998, chapter 9, pp125

[3] Charles Roth, "Fundamentals of Logic design", P203, West Publishing Co. 1975

[4] UML Resource Center. http://www.rational.com/uml/index.html

[5] Robert Martin, "THREE-LEVEL FSM". In Jim Coplien, Douglas Schmidt, (edt.), *Pattern Languages of Program Design.* Addison-Wesely,1995, chapter 19, pp383