Chapter 1 : Finite State Machine Patterns

1.1 Introduction

Finite state machines (FSMs) are widely used in many reactive systems to describe the dynamic behavior of an entity. The theoretical concepts of FSMs and an entity's specification, in terms of state transition diagrams, have long been used. This chapter presents an FSM pattern language that addresses several recurring design problems in implementing a state machine in an object-oriented design. The pattern language includes a basic design pattern for FSMs whose design evolves from the general understanding of state machines functionality. The basic pattern is then extended to support solutions for other design problems that commonly challenge system designers. These design decisions include state-transition mechanisms, design structure, state-instantiation techniques, and the machine type. Since FSMs are frequently applicable to areas of concurrent and real-time software, it is useful for the system designer to consult a catalog of classified state machine patterns. The pattern language presented in this chapter covers the three-layer FSM pattern by Robert Martin [Martin95] and extends the set of patterns described by Paul Dyson and Bruce Anderson [Dyson+98]. Discussion on nested and concurrent states (i.e. statecharts) can be found in Chapter 11.

The following section provides an overview on the new pattern language and its relationship to other patterns of state. A pattern road map is presented to illustrate the semantics relationship between patterns, that is, how they coexist or contradict. We then describe a turnstyle coin machine example that is adopted from [Martin95] and is used through out our discussion. The rest of the chapter describes the state machine patterns and how the example is redesigned as patterns from the pattern language are applied.

1.2 Road map of the Patterns

The set of patterns presented here constitutes a pattern language of FSMs. Figure 1.1 shows the set of patterns and how they are related. The patterns address design issues related to the machine type (*Meally*, *Moore*, or *Hybrid*), the design structure (*Layered* or *Interface Organization*), exposure of an entity's internal state (*Exposed* or *Encapsulated State*), and the instantiation technique of state objects (*Static* or *Dynamic State Instantiation*).

The extend symbol (we used the inheritance symbol [UML99]) shows that a pattern extends another by providing a solution to an additional design problem. The double-headed arrow, with the "X" label, indicates that only one of the patterns will appear in the design because they contradict each other according to their motivation or solution facets. The dotted arrow shows that one pattern motivates

(leads to) the use of another, the arrowhead shows the direction of motivation. A labeled single-headed solid arrow indicates the classification according to a certain design decision.

The basic FSM pattern (*Basic FSM*) is an extension of the *State* pattern of Erich Gamma *et. al.* [Gamma+95], also referred to as *State Object* [Dyson+98]. It adds implementation of the state transition diagram specifications such as actions, events, and a state transition mechanism. The *Basic FSM* is classified, according to the state transition mechanism, as: *Owner-Driven Transitions* and *State-Driven Transitions* which are in tension with each other.

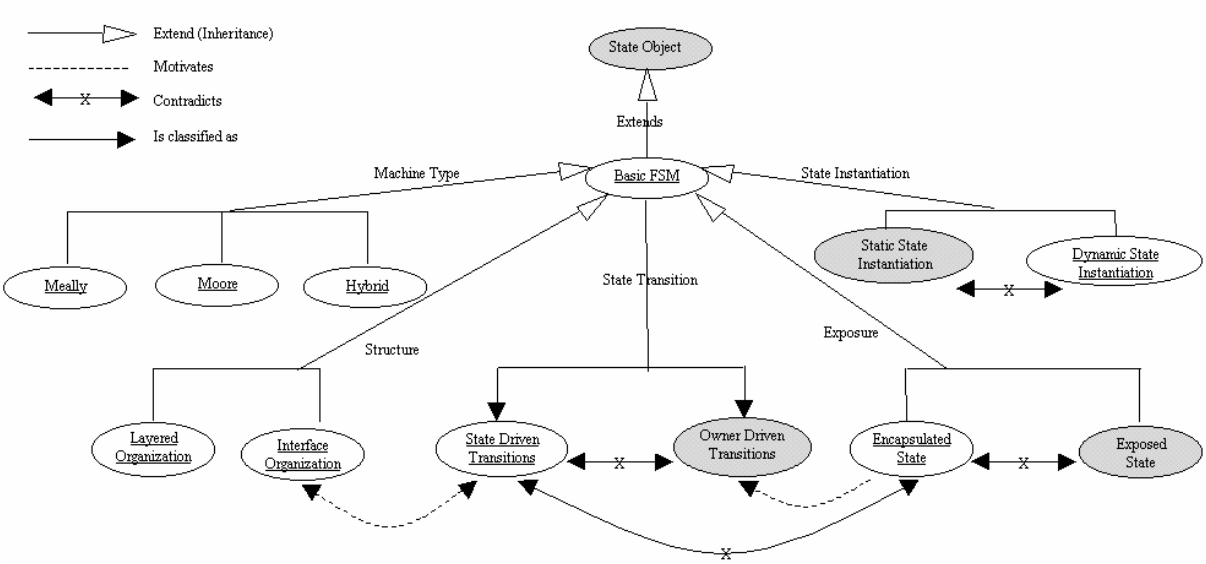

Figure 1.1 Relationship among state machine patterns the underlined patterns are those addressed in this chapter.

For maintainability purposes, the *Basic FSM* design can be structured into *Layered Organization* and *Interface Organization*. 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 facilitates communication between other entities and the machine design.

According to the mechanism for producing an FSM output, i.e. the machine type [Roth75], the *Basic FSM* is extended into *Mealy*, *Moore,* or *Hybrid* to describe whether the outputs are dependent only on the entity's current state or on the events as well.

The entity described by an FSM has a particular state at a given time. The current state of the entity can be exposed to other application entities to allow direct invocation of the state class methods; i.e., *Exposed State.* It also can be encapsulated inside the entity, and no access is permitted from other application entities, i.e. *Encapsulated State*. The designer will only use one of these two patterns by choosing either to expose the entity's current state or prevent access to it.

The *Basic FSM* considers a state class for each state of the entity, and thus you would need a mechanism to instantiate the state objects. State instantiation can be either static or dynamic. In *Static State Instantiation*, all the state objects are created at the initialization phase, while in the *Dynamic State Instantiation* the states are created dynamically during runtime. 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, which will be discussed later in this chapter.

Usage of one pattern may lead to usage of another. If you decide to use the *Encapsulated State* you will need to secure access to the state object, therefore you may use an *Owner-Driven Transitions*. Usage of *Interface Organization* leads to application of the *State-Driven Transitions* and vice versa, because moving the state transition logic to the states is a step in simplifying the entity's interface to other application entities.

The *State Object*, *Owner-Driven Transitions*, and *Exposed State* patterns are discussed by Paul Dyson *et al.* [Dyson+98]. Robert Martin [Martin95] discussed the *Static Instantiation.* We will not further discuss these patterns (shaded ellipses in Figure 1.1). Section 1.14 summarizes the patterns as problem/solution pairs and provides references to those that are addressed in other literature.

1.3 Example

We will consider applying the state machine pattern language to the turnstyle coin machine example described by Robert Martin [Martin95] in the three-level FSM. Figure 1.2 summarizes the machine specifications using a state transition diagram.

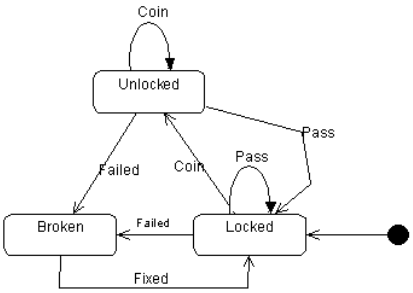

Figure 1.2 The state transition diagram of a coin machine

The machine starts in a locked state (`Locked`). When a coin is detected (`Coin`), the machine changes to the unlocked state (`UnLocked`) and open the turnstyle gate for the person to pass. When the machine detects that a person has passed (`Pass`) it turns back to the locked state. If a person attempts to pass while the machine is locked, an alarm is generated. If a coin is inserted while the machine is unlocked, a `Thankyou` message is displayed. When the machine fails to open or close the gate, a

failure event (`Failed`) is generated and the machine enters the broken state (`Broken`). When the repair person fixes the machine, the fixed event (`Fixed`) is generated and the machine returns to the locked state.

A direct traditional implementation of the example in an object-oriented design would use a class called `CoinMachine` and keep track of the entity's state as an internal member attribute of the class. For each event received by the machine class a conditional check would be implemented to act according to the current present state. For example, the processing of the coin event would differ if the machine is locked or unlocked. The person would be allowed to pass (if it is locked) or the `Thankyou` message would be display (if it is unlocked). A sample implementation would look like:

```
enum State = { Locked, UnLocked, Failed };
class Coin_Machine
{      State CurrentState;

       void coin()
              { switch(CurrentState) {
                     case Broken :   // Display an "out of order" message
                     case UnLocked:   // Display a "Thank You" message
                     case Locked :   // Unlock the machine's gate
                     ----       };
              }

};
```

The above example will be redesigned using the patterns presented in the rest of this chapter.

1.4 Basic FSM

Context

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

Problem

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

Forces

Understandability of the Design

The traditional approach of using one class is easy to implement, but you will need to replicate state checking statements in the methods of that class because you cannot make the entity take any action unless it is in a correct state. This will make the class methods look cumbersome and will not be easily understood by other application designers.

Traceability from Specification into Implementation

The specification of an entity's behavior is normally described in terms of a state transition diagram as that shown in Figure 1.2. State transition diagrams clearly distinguish the states, the events, and the actions of an entity behavior as related to the application environment. The implementation of the entity's behavior should possess the same characteristics to ease the traceability of the specification to implementation.

Flexibility and Extensibility

Implementation of state machines using a single-class or tabular implementation would localize the behavior description in one implementation unit. However, this would limit the extensibility of the design. A good model would imitate the behavior of the entity as related to the application environment. Figure 1.3 shows the behavior to be mapped in the design model.

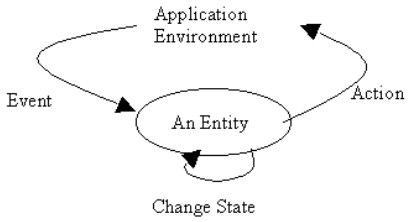

Figure 1.3 The entity behavior in an application environment

Solution

Implement the entity's behavior using a design model that distinguishes the entity, its states, events, state transitions, and actions.

Structure

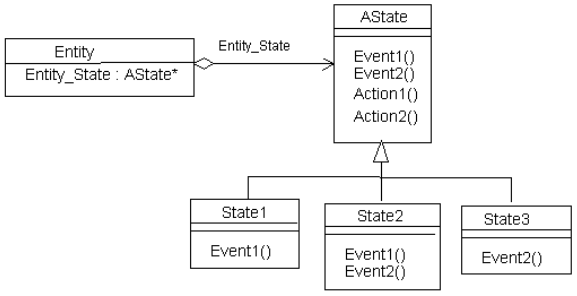

Figure 1.4 Structure of the Basic FSM pattern

- Construct an abstract state class `AState` that contains static methods implementing the actions taken by all states of the entity.

- Have all the possible concrete classes inherit from the abstract class. Create virtual methods for all possible events in the `AState` class. The concrete classes implement these methods, as specified for the behavior of the entity in each state, and invoke the actions that affect the application environment.
- Create a class for your entity that contains a current state of type `AState`. Delegate all events received by the entity to the current state using the `Entity_State` reference. Choose a state transition mechanism, *State-Driven* or *Owner-Driven Transitions* patterns.

Example Resolved

How do you use the *Basic FSM* pattern to implement the coin machine behavior?

1. Identify and create the concrete state classes `Locked, Unlocked,` and `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. Events are specified as virtual methods in the abstract class `AState` and are implemented on each state accordingly. However, you need not implement all events in every state, only those that are identified from the state diagram; for example, the `Pass` event need not be implemented in the `Broken` state of the machine. Thus, you can provide default implementation for events in the `AState` class.
3. Implement static methods for all possible actions in the `AState` class. The specification shows that the following actions are taken by the FSM in various states: allow a person to pass (`Unlock`), prevent a person from passing (`Lock`), display a `Thankyou` message, give an alarm (`Alarm`), display an out-of-order message (`Outoforder`), and indicate that the machine is repaired (`Inorder`).

Figure 1.5 shows the class diagram of the example.

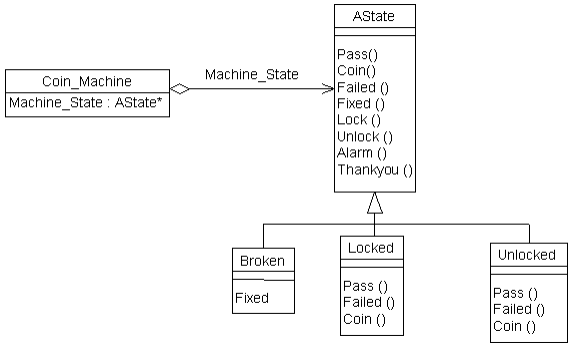

Figure 1.5 The coin machine resolved using the Basic FSM pattern

Consequences

- The design is understandable because the pattern maps the elements thought of and described in the state transition diagram to classes and methods in an OO design and hence eases the traceability from the state transition diagram (Figure 1.2) into a design (Figure 1.4). For example, the coin insertion event is mapped to coin methods implemented in each state to give particular implementation according to the current state.
- The model of interaction of an entity with the environment, in terms of actions and events, is mapped to methods implementation in OOD designs that give an implemented image of the practical model.
- The model is flexible enough to handle the addition of new states as well as other events. However as the number of states increases, the design becomes more complex because a state class is required for each state. For such cases, *Statechart* patterns (Chapter 11) can be used to simplify the design.

Related Patterns

The *Basic FSM* pattern should possess a state transition mechanism that either could be the *Owner-Driven Transitions* [Dyson+98] or *State-Driven Transitions* pattern.

1.5 State-Driven Transitions

Context

You are using the *Basic FSM*. You need to specify a state transition mechanism to complete the entity's behavior implementation of the *Basic FSM*.

Problem

How would you implement the state transition logic but yet keep the entity class simple?

Forces

Reusability of State Classes versus Complexity of the Entity

Since the entity holds a reference to its current state class, you can intuitively implement the state transition inside your entity. This has the disadvantage of increasing the complexity of the entity because you would implement every condition that combines the current state and the event that causes state transitions in the entity class itself. But it has the advantage of reusing the state classes. However, you want the entity implementation to be simple, which is why you first chose to use a state machine pattern and delegate the event processing to the current active state class.

Concatenating the Event Processing

The entity state delegates the event processing to its current state. If you implement the state transition in the entity, then you have split the processing of the event into two, one for the state transition in the entity and the other for the event processing and action activation in the state class. You would rather delegate the state transition logic to the current state instead and have one unified processing mechanism for events.

Solution

Delegate the state transition logic to the state classes, make each state knowledgeable of the next upcoming state, and have the concrete states of the entity initiate the transition from self to the new state.

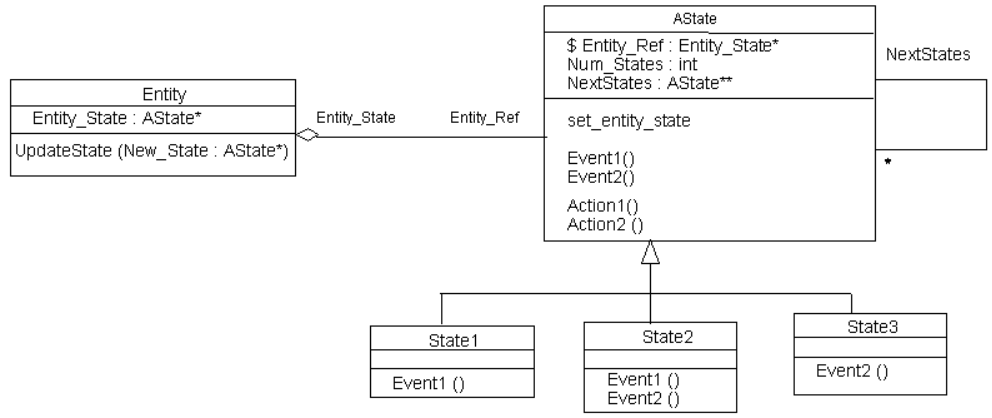

Figure 1.6 The structure of the State-Driven Transitions pattern

- Use the pointer to self `NextStates` in the abstract class `AState` to provide generic pointers to upcoming states.
- The `Entity_Ref` is added to point to the coin machine entity and use the `set_entity_state` to change its current state. The concrete states have to know their entity. Thus, create a static pointer in the abstract class `AState` that can be shared by all concrete states and that is accessible via a static method for state changes that can be invoked from any concrete state class. The entity current state has to be exposed to the state change mechanism encapsulated in the `AState`.

A variation of the solution would be to modify each event method to return the state for which the entity should change or return itself in case of no transition. This would remove the circular reference between the `Entity` and the `AState` classes, however, you have to update the current state of the entity after each event delegation. The first solution is assumed for the rest of the patterns.

Example Resolved

How can you use the pattern to solve the following problem: you don't want the `Coin_Machine` class to know about the transitions from one state to another; it just dispatches the events to the current state object. Make each state knowledgeable of the next coming state; for example the `Broken` class has pointers to the `Locked` class. From the specification of the problem (state transition diagram in Figure 1.2), you identify the transitions from source states to destination states, and in the implementation of the event causing the transition in the source state, you call the `set_entity_state` with the destination state as an argument. For example, in the implementation of the `Fixed` event, the entity's state is changed by calling `set_entity_state(Locked)`.

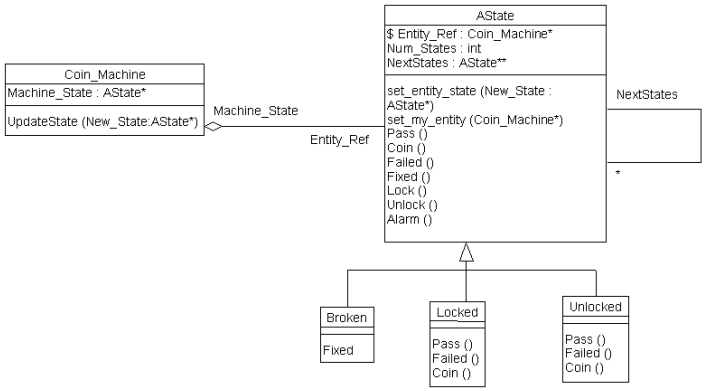

Figure 1.7 The coin machine resolved using State-Driven Transitions pattern

Consequences

- The *State-Driven Transitions* pattern simplified the coin machine class implementation as it delegates the event processing to the concrete state class. However, it added the burden to the state classes, which requires more instantiation and declaration efforts to ensure that each concrete state class points correctly to the next states.
- All the processing related to an event is contained in the event method implementation of the state classes, this localization is important for good maintainability of the event processing.

Related Patterns

The *State-Driven Transitions* is in conflict with the *Owner-Driven Transitions*, only one state transition mechanism should be used in the design.

Since we have chosen the implementation using an entity's reference in the state class `AState`, the *State-Driven Transitions* is also in conflict with the *Encapsulated State* as discussed later in this chapter.

The *Factory Method* pattern [Gamma+95] can be used to manage the creation of the state objects.

1.6 Interface Organization

Context

You are using the *Basic FSM* to implement the behavior of an entity.

Problem

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

Forces

Interface-Centric Design

Your entity may not be a standalone design but rather one that is embedded in an application. When you embed the entity behavior design in the overall application design, you think of the way other application entities interact with the entity and whether they keep references to the entity only or they can have access to its state classes. Therefore, you want to define an interface for the entity.

Simple Interfaces versus Delegation and State Class Complexity

To simplify the interfaces of the entity behavior design, consider decoupling the entity interface and the states transition logic and behavior. This necessitates delegating all processing to the state classes, which makes these classes more complex, but on the other hand leaves the design with a simple interface with which to interact. Then the interface role is to receive events and dispatch them to its state implementation.

Solution

Encapsulate the transition logic in the states and hide it from the entity 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 entity.

Structure

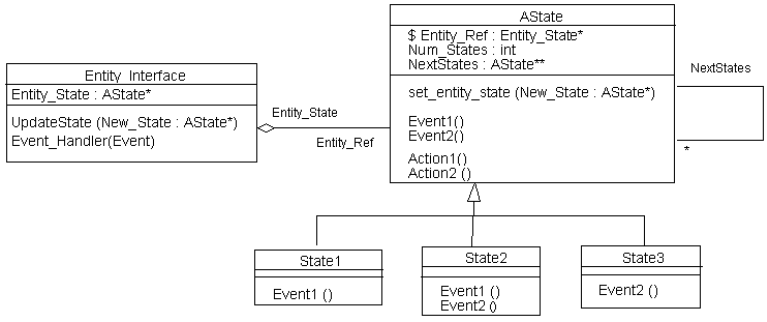

Figure 1.8 The structure of the Interface Organization pattern

Example Resolved

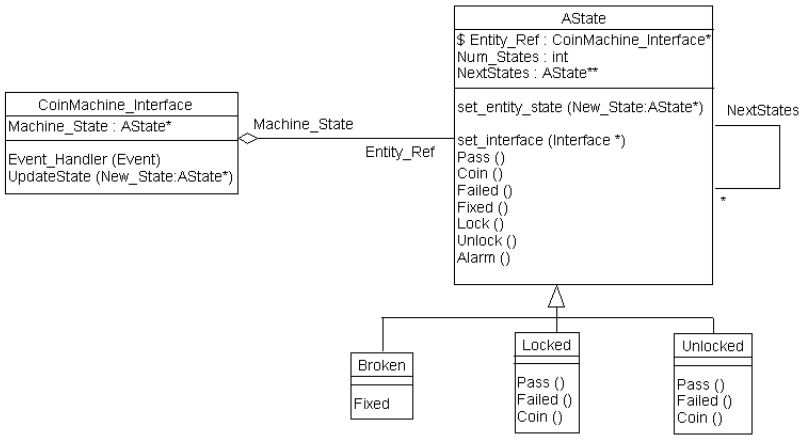

Figure 1.9 The coin machine resolved using the Interface Organization pattern

In the coin machine example, you create a `CoinMachine_Interface` class and an `Event_Handler` method to handle and dispatch events. The `CoinMachine_Interface` class acts as an interface to the logic encapsulated in the design. The interface knows which state the entity is currently in and thus, it handles the incoming events and invokes the appropriate state event method. The `Event_Handler` receives events from the application environment and calls the state implementation of the event accordingly.

Consequences

Using the `Entity_Interface` class clarifies the interaction of the entity with the other classes of the application and hence separates the interface and the actual logic and implementation of the state machine.

Related Patterns

The *Interface Organization* pattern motivates the designer to use *State-Driven Transitions* to simplify the tasks required from the interface by delegating the state transition logic to the states themselves.

1.7 Layered Organization

Context

You are using the *Basic FSM* to implement the behavior of an entity.

Problem

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

Forces

Understandability of the Design

When you use the *Basic FSM* to describe the behavior of the entity, 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 entity's interface and encapsulate the actions, events and state transitions in the state classes, which makes them more complex, even in an example as simple as the coin machine.

Maintainability of the Application

Earlier, Figure 1.3 showed that the interaction of the entity with the environment describes its behavior as events received by the entity and actions taken by it, however you cannot clearly distinguish this behavior in the *Basic FSM* because both are defined in the abstract state class. This impedes the maintainability of the design because it becomes difficult to distinguish events and actions methods and to add new ones. Therefore, you will want to separate the events and the actions in a different design layer i.e., the entity's behavior layer.

Solution

Organize your design in a layered structure that decouples the logic of state transitions from the entity's behavior as it is defined by actions and events.

Structure

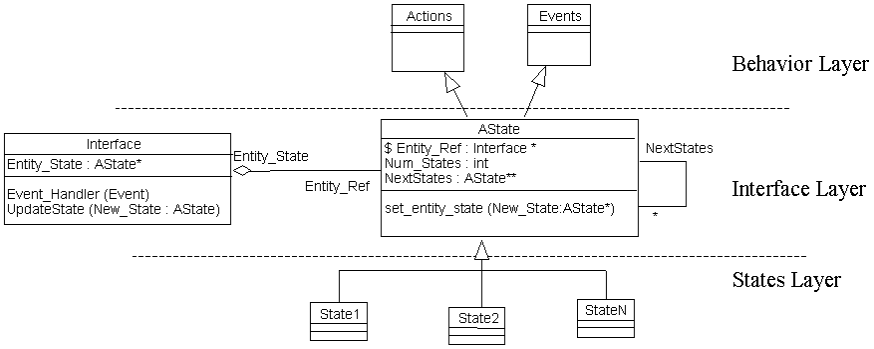

Figure 1.10 The structure of the Layered Organization pattern

The structure has three layers (Figure 1.10):

- *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 to which the FSM responds. It is an abstract class with virtual method declaration; 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, you will have one implementation of actions used by several classes.

Multiple inheritance from `Events` and `Actions` compose the behavior of the machine such that any concrete state class encapsulates the behavior specification.

Example Resolved

In the coin machine example, the events are distinguished as `Coin`, `Pass`, `Failed`, and `Fixed` methods. The possible actions are `Lock`, `Unlock`, and `Thankyou`. When the layered solution is used the events and action classes will contain these methods.

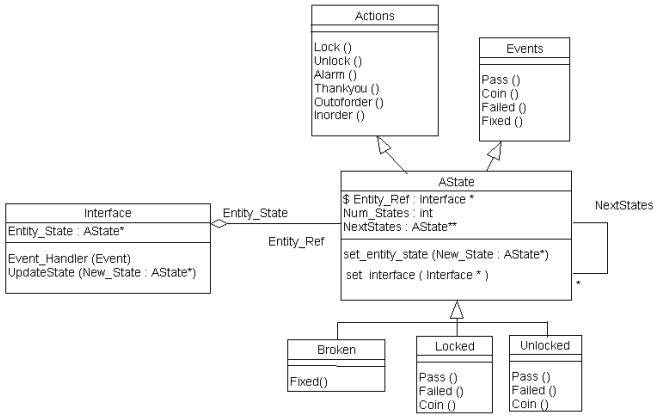

Figure 1.11 The coin machine resolved using the Layered Organization pattern

Consequences

- How does this structure facilitate the maintainability of the design? Assume that the designer will change the 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 the `Lock` method in the `Action` class and modify it.
- The structure of the machine separates actions, events, and state transitions which eases its maintenance and reuse. The multiple inheritance followed by several single inheritances shows a three-layer architecture that simplifies the state machine design. Multiple inheritance is used for

mixing [Gamma+95, pp16] the functionality of the two classes (`Action` and `Events`) into `AState` class. This is different from the traditional "is-a" meaning of inheritance relationship.

1.8 Meally

Context

You are describing output actions of the machine and when to produce them. The requirements specify that the outputs should be produced only in response to specific events depending on the entity's state.

Problem

How do you activate the FSM outputs?

Forces

Explicit Generation of Outputs on an Event/State Combination

You want the actions taken by the entity to be associated with the entity's present state and the current inputs affecting it [Roth75]. 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 class.

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 Resolved

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

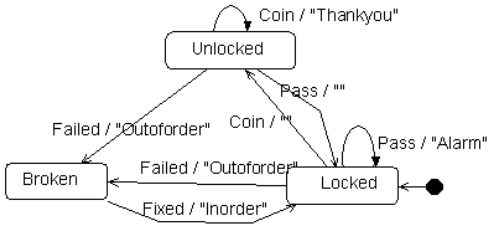

Figure 1.12 The state transition diagram of a Meally machine example

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" }
```

Consequences

Whenever an event is required to produce an output, the output actions can be called from inside the event method of that specific state. This associates the outputs with the event/state combination.

Related Patterns

The *Moore* and *Hybrid* patterns are the alternatives used to generate the FSM outputs.

1.9 Moore

Context

You are using an FSM. You have identified the set of outputs (actions) that the machine produces. The machine produces these outputs depending on its current state only.

Problem

How do you activate the state machine outputs?

Forces

Avoid Code Replication as the Number of States Increases

You could consider applying the *Meally* pattern to implement the calls to the outputs, but then you will find that you are replicating calls to these output methods. This is 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 turned on each time the coin machine is in the `Broken` state, you will have to call the output routine to turn the lamp on in two situations. The first one is in the `Failed` event of the `Locked` state and the other in the `Failed` event in the `UnLocked` state. This will require many calls to the output method, which will increase as the number of state entries increase.

Maintainability 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 entity with the entity's present state only [Roth75]. In a design context, this is translated as producing the outputs on entering the state,

and hence you don't have to worry about calling outputs from the event methods of other states. Therefore, you can easily maintain the outputs of a particular state.

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 Transitions*, the machine changes state by calling the `set_entity_state` method. Thus, a method called `Output()` is added to the previous design, which is *specific* for each state and is called by the `set_entity_state` routine using the new state as the caller. In an *Owner-Driven Transitions*, the output method will be invoked from the owner after each transition condition is satisfied.

Example Resolved

In a state-driven transition design for the coin machine, consider that a lamp will be turned on whenever the machine is broken and turned off whenever it is operating in either the `Locked` or `Unlocked` state. A Moore machine version specification is shown in Figure 1.13.

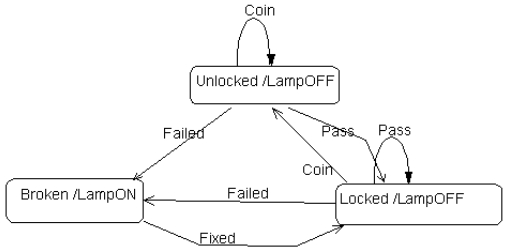

Figure 1.13 The state transition diagram of a Moore machine example

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

Figure 1.14 shows the design using a Moore FSM.

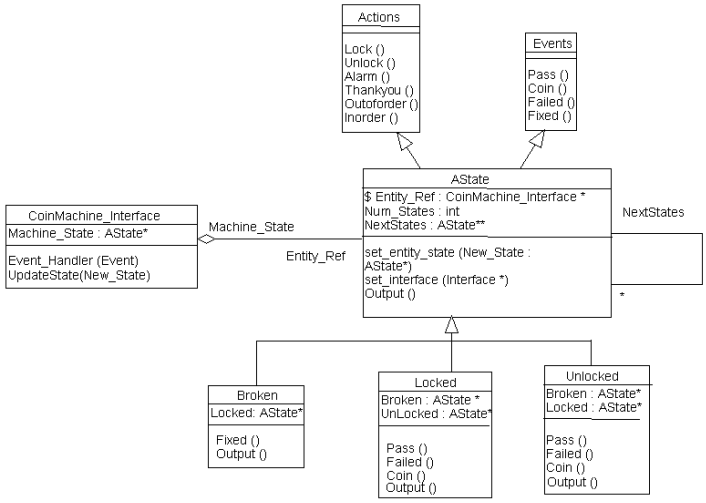

Figure 1.14 Coin machine resolved using the Moore pattern

Consequences

The output method of the state class produces all the actions associated with that particular state and hence provides a focal method for maintaining these outputs.

Related Patterns

The *Meally* and *Hybrid* patterns are the alternatives used to generate the FSM outputs.

1.10 Hybrid

Context

You are using an FSM pattern. The machine produces some outputs in response to events, and some other outputs are associated with the entity's state.

Problem

How do you activate the state machine outputs?

Forces

FSMs can be a Combination of *Meally* and *Moore*

When you consider using a *Meally* pattern, you will find that some outputs are dependent on the states only, however, you cannot use a pure *Moore* pattern because some other outputs are dependent on the events response. But does the implementation of a *Meally* contradict that of a *Moore*? As discussed in their Solution sections, they are not. Therefore, you can use a *Hybrid* machine by which some actions taken by the entity are associated with the entity's present state only (*Moore* behavior), and some

other actions are constricted by both the entity's state and an event in the application (*Meally* behavior).

Solution

Use a combination of the *Meally* and *Moore* FSMs the pattern solutions provided for each of these patterns do not contradict one another and, in fact, can be used together.

Example Resolved

In the coin machine example, it is desired that the activation of the `Lamp` output is associated with being in a particular state while the `Thankyou` message is generated only on the `Coin` insertion event while in the `Unlocked` state. Thus, in the event handling of the `Coin` method in class `Unlocked`, the call to the `Thankyou` method is placed as shown in the Meally pattern. You also add the `Output()` method that is called by the `set_entity_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 was in the Moore pattern.

Related Patterns

The *Meally* and *Moore* patterns are parts of the solution of the *Hybrid* pattern.

1.11 Encapsulated State

Context

You are using an FSM. The sequence of state changes is defined in the entity's specification.

Problem

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

Forces

State Transitions should not be Forgeable

Paul Dyson *et al.* [Dyson+98] discussed the *Exposed State* pattern that allows other application entities to access and retrieve the entity's state. This was shown 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 entity's state and to possibly change it. This might contradict with your desire to keep the states known to the entity only, a situation that often occurs for safety purposes. This arises when the specification of the entity'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 the doors to open.

Solution

Encapsulate the current state inside the entity itself and keep the state reference as a private attribute. In our implementation, only the entity 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 Transitions* [Dyson+98] would be used and the concrete state reference should be private or protected. However, you can use *State-Driven Transitions*, but in this case, the implementation of the methods should return a reference to the new state instead of having the abstract state class refer to the entity interface.

Example Resolved

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

Related Patterns

The *Encapsulated State* pattern is in tension with the *Exposed State* pattern [Dyson+98].

1.12 Dynamic State Instantiation

Context

You are using an FSM pattern to implement your entity's behavior. The application in which you are using the entity is large, and the entity has many states.

Problem

How do you instantiate the states in your application?

Forces

Limited Availability of Memory versus Performance

You can statically instantiate all the states of the entity at the initialization phase as described in the three-level machine by Robert Martin [Martin95], but since the number of states is enormous in large applications, this will consume large memory space and thus decrease the availability of free memory. Therefore, it is preferable to keep few state objects loaded in memory such as the current state and the possible upcoming states. This will possible slow down the state transition process because of the

amount of time required to create and delete state objects. However, the number of states kept loaded is small which will occupy smaller memory size.

Solution

Upon state transitions, load the upcoming state and unload the current state then update the entity'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 Transitions*, let the old state create the next state object and then the invocation of the state change method in the entity will delete the previous state.

Example Resolved

In the coin machine example, if you are required to dynamically instantiate the states[*], you will 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){
   Entity_State->DeleteUpcomingStates(New_State);
   delete Entity_State;
   Entity_State = New_State;
   Entity_State->CreateUpcomingStates();};
```

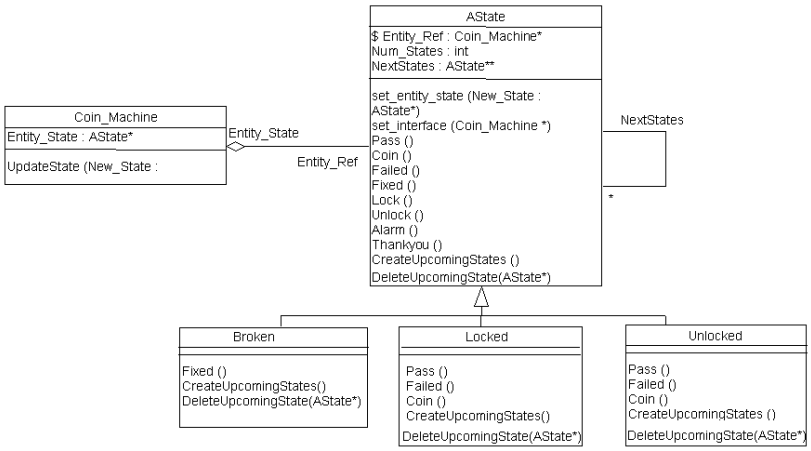

Figure 1.15 The coin machine resolved using the Dynamic State Instantiation pattern

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

```
void Locked::DeleteUpcomingStates(AState* CurrentState)
        { for(int I =0; I < Num_States; I++)
```

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

```
              { 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++;  }
```

Consequences

The dynamic instantiation mechanism has the advantage of keeping few state objects loaded at a time, however its disadvantage is that is slows down the state transition operation because you delete the previous states and create objects for the upcoming states. Thus, the *Dynamic State Instantiation* pattern is not applicable to real time systems for which *Static State Instantiation* pattern is recommended.

Related Patterns

The *Static State Instantiation* is an alternative to *Dynamic State Instantiation* to instantiate the state objects during the entity initialization.

1.13 Known Uses

FSMs are widely used in many reactive systems and 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, the motion control system of automated trains and elevator controls. Erich Gamma *et al.*[Gamma+95] have identified some known uses in graphical user interfaces. Paul Dyson *et al.* [Dyson+98] have also addressed their usage in library applications. Automated Teller Machines are one of the most widely known and frequently used illustrative examples for an application whose state plays a major role in the flow of operations.

1.14 Summary of FSM Patterns

	Pattern Name	Problem	Solution	Ref.
	State Object	How can you get different behavior from an entity if it differs according to the entity's state?	Create states classes for the entity, describe its behavior in each state, attach a state to the entity, and delegate the action from the entity to its current state.	[Gamma +95] [Dyson+98]
	Basic FSM	Your entity's state changes according to events in the system. The state transitions are determined from the entity specification. How can you implement the entity behavior in your design?	Use the *State Object* pattern and add state transition mechanisms in response to state transition events. FSM pattern = State Object pattern + State Transition Mechanism	[Martin95] [Dyson+98] *
State-Transition	State-Driven Transitions	How would you implement the state transition logic but yet keep the entity class simple?	Have the states of the entity initiate the transition from self to the new state in response to the state-transition event.	[Dyson+98] *
	Owner-Driven Transitions	You want your states to be simple and shareable with other entities, and you want the entity to have control on its current state. How can you achieve this?	Make the entity respond to the events causing the state transitions and encapsulate the transition logic in the entity	[Dyson+98] [Martin95]
Structure	Layered Organiza-tion	You are using an FSM pattern, how can you make your design maintainable, easily readable, and eligible for reuse?	Organize your design in a layered structure that decouples the logic of state transition from the entity's behavior, which is defined by actions and events	[Martin95] *

	Interface Organiza-tion	How can other application entities communicate and interface to an entity whose behavior is described by an FSM?	Encapsulate the states classes and state transition logic inside the machine and provide a simple interface to other application entities that receive events and dispatch them to the current state.	*
Machine Type	Meally	How do you activate the FSM outputs if they should be produced at specific events while the entity is in a particular state?	Make the concrete event method of each state call the required (output) action method in response to the event.	*
	Moore	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.	*
	Hybrid	What do you do if some FSM outputs are activated on events and some other outputs are activated as the 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 State	You want to allow other external entities in your application to know of your entity's state and have access to call some of the state's methods.	Provide a method that exposes the state of the entity and allows access to the current state.	[Dyson+98]
	Encapsulated State	Your FSM should follow a sequence of state changes that should not be changed by other application entities. How can you ensure that no state changes are enforced to your entity?	Encapsulate the current state inside the entity itself and keep the state reference as a private attribute. Only the entity itself can change its state by handling the events causing the state change but still delegate the behavior implementation to the current state.	*

State Instantiation	Static State Instantiation	Your application is small and it has few states. Speed is a critical issue in state transitions. How do you instantiate your entity's states?	Create instances of all possible states on the entity instantiation. Switch from current to next state by altering the reference to the next state	[Martin95]
	Dynamic State Instantiation	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.	*

- Addressed in this chapter

Table 1-1 Summary of FSM patterns