Chapter 1 : A Pattern Language of Statecharts

Statecharts have been introduced to model the behavior of complex systems. Statecharts are used in specifying the DICOIM UL in the Medical Informatics system discussed later in Chapter 14. The formalized concepts and specifications of statecharts have been used in many applications. This chapter discusses a design pattern language to solve recurring design problems in implementing a statechart specification in an object-oriented design. The pattern language offers solutions to frequent design problems that commonly confront a system designer. These problems include how to implement specifications containing hierarchy, orthogonality, and broadcasting. The pattern language maps the specification into object oriented design and hence code will be generated. Since statecharts are frequently applicable to software applications, it is helpful for the system designer to directly exercise the statechart pattern language in his application design.

In the next section, a brief background on statecharts is presented, then a pattern map summarizes the statechart patterns, their relationships, and how they relate to finite state machine patterns. The rest of the sections describe five new statechart patterns namely, *Basic Statechart, Hierarchical Statechart, Orthogonal Behavior, Broadcasting,* and *History State.* Section 1.10 summarizes the patterns as problem/solution pairs, and Appendix C of the thesis gives a summary of basic statecharts principles.

1.1 Background

David Harel [Harel87, Harel88] introduced statecharts to extend finite state machines for complex behavior and to describe a visual formalism for behavior specification of an entity. Statecharts extend finite state machines with new concepts such as hierarchy, orthogonality, broadcasting and history states. The hierarchy principle introduces a more global state, referred to as a "superstate", that includes other entity's states. The principle of statecharts that allows several behaviors to be experienced at the same time is called orthogonality. The broadcasting principle allows events produced in one state to be broadcasted to orthogonal states.

Objectcharts were then presented by Derek Coleman *et.al.* [Coleman+92] as an extension to describe how to use statecharts in object-oriented environment to describe the lifecycle of an object. Statecharts were used by Bran Selic [Selic98] as a mechanism for *Recursive Control*.

Several authors have documented solutions of recurring design problems in implementing finite state machines. Erich Gamma *et.al.*[Gamma+95] documented the *State* as a basic pattern which delegates the state-dependent behavior of an entity to separate state classes, and encapsulates the current state object in the entity's interface. Paul Dyson and Bruce Anderson [Dyson+98] extended and refined this

basic pattern and presented several patterns to solve problems of assigning data members to state classes, exposing an entity's state, implementing a default state, and alternatives in implementing state transition mechanisms.

Alexander Ran [Ran96] discussed a family of design patterns to implement an entity with complex behavior. He discussed how possible design decisions form a tree for models of object oriented design of states (MOODS). The tree considered decomposition of methods and abstract states, possible implementation of state transitions using conditions or state transition methods, and described how single and multiple inheritance can be used to compose the entity's state classes. However, all these patterns addressed problems related to state machines and none has been extended to support statecharts principles.

Alternatively, we perceive that statecharts are frequently used as a behavior-description technique. This chapter defines statecharts as design components that provide flexibility at the design level and facilitates the design composition and maintenance. We explain how to deploy reusable design solution to frequent statecharts concepts. A summary of statecharts principles and properties is presented in Appendix C.

1.2 Road map of the Patterns

As a further extension to finite state machine patterns, a set of solutions to problems encountered in implementing statecharts is presented. The *Basic Statechart* translates the elements of statechart specification into an object-oriented design. Based on the statechart's hierarchy principal, the *Hierarchical Statechart* extends the basic pattern to support hierarchical states in which a superstate is composed of other states. Sometimes the entity's behavior is described using orthogonal non-contradicting behaviors by means of the statechart's AND-decomposition specification; thus the *Hierarchical Statechart* is extended to *Orthogonal Behavior* to support handling events to orthogonal states. Using an *Orthogonal Behavior* may lead to the possibility of broadcasting the effect of an event in a state to another orthogonal state, *Broadcasting* extends *Orthogonal Behavior* to support event broadcasting. Finally, the statecharts' history specification is sometimes used in the superstates of the *Hierarchical* pattern; *History State* addresses this problem. Figure 1.1 shows the relationship between the statechart patterns in this language.

1.3 Example

The turnstyle coin machine specification discussed by Robert Martin [Martin95] is extended one step at a time to illustrate the various patterns usage. Figure 1.2 summarizes a simplified version of the machine specifications using a state transition diagram.







Figure 1.2 Example: 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 through (Pass) it turns back to the locked state. If a coin is inserted while the machine is unlocked, a (Thankyou) message is displayed. When the machine fails in opening or closing the gate, a failure event (Failed) is generated and the machine enters the broken state (Broken). When the machine is repaired, a fixed event (Fixed) is generated and the machine returns to the locked state.

1.4 Basic Statechart

Context

Your application contains an entity whose behavior depends on its state. You are using a statechart to specify the entity's behavior.

Problem

How do you implement the statechart specification into design?

Forces

What factors motivate you to describe your entity's behavior in statecharts semantics?

- Limited capabilities of state transition diagrams for large systems: Finite state machines are used to describe many reactive systems, but as the complexity of the system increases, the complexity of state transition diagrams also increases because of their flat nature. These diagrams become unreadable and difficult to comprehend.
- Orthogonal behaviors of an entity: Sometimes it is necessary to have orthogonal description of non contradicting behaviors of an entity, and state transitions take place in each independent behavior in response to the same event. State diagrams sequentially describe the behavior of an entity and don't reveal the concurrent nature.

What factors motivate you to use the Basic Statechart pattern?

• **Traceability from specification into design**: Statechart is a specification language that enables you to construct a model and further check it. Some case tools allow you to generate code from your specification. Using the generated code might not be useful because it is usually bulky and not comprehensible for the designer. Thus you want to translate the specification into design that allows you to have a higher level of maintenance and to embed this design into your overall application design.

Solution

Implement the statechart specification into object oriented design, define a state class for each entity's state defined in the specification. Distinguish the events, conditions, actions, entry and exit procedures in each state class.

Structure

Figure 1.3 illustrates the design solution structure. The design evolves around the FSM patterns documented in Chapter 10. It has a layered organization separating the behavior (Actions and Events) from the logic of the machine. A state-driven transition is chosen to implement the transition mechanism.



Figure 1.3 The structure of the Basic Statechart pattern

Participants

Events

The Events class hosts declaration of events that the entity responds to. The response to each event may differ according to the entity's current state. Events, that are handled differently by each state, are specified as virtual methods in the Events class and each state class implements the adequate functionality required in response to that particular event (ex. Event1() and Event2()). Common events, that have the same implementation for the entity's states, can be implemented in this base class and hence its implementation is sharable by all state classes (ex. Error Handlers).

Actions

The Actions class contains the output methods that can be executed by the entity and affect the application environment. Action methods are invoked from event method implementations in each concrete state class. Generally, action methods are static methods inherited by the state classes (such that all class objects share the same implementation instance). Private actions that are only meaningful in a particular state class context are declared within that state class scope.

AState

It is the general abstract state class that groups the actions and events, and encapsulates the statedriven transition (Chapter 10) mechanism. The entity's state will be changed when the set_entity_state method is invoked, this provides a high-level state transition procedure that is invoked from any state class. The class possesses pointers to self NextStates that allows each state class to point to next possible states to be used in state transitions. The designer can also use an owner driven transition [Dyson+98], in this case these pointers will not be needed in the *Basic Statechart*. Multiple inheritance from Actions and Events is used to build the behavior of the states. These classes can be merged with the AState class. However, it is preferred to separate them for large systems where the number of events and actions are enormous and distinction between outputs and events makes the design more comprehensive and easily maintainable.

States

These are the actual implementation of the entity's states, they inherit from the general AState class and have the following tasks:

- Implement event methods that the state responds to and invoke appropriate actions accordingly.
- Keep knowledge of possible upcoming states and perform the state transition by invoking the set_entity_state method.
- Host state specific conditions which can be implemented by:
 - a. Using a Boolean data member of a state class. For example if you want to implement whether a link is established between two communicating entities, you can use a simple Boolean data member LinkEstablished.
 - b. Using data member variables of a state class. For example, a simple conditions such as [Amount>=50] can be implemented using a data member Amount and a condition check statement (If (Amount>=50){..}). The condition check statement is implemented in the event method of the state class that needs to evaluate the condition to take action accordingly, as shown latter in the coin insertion event in the coin machine.
 - c. Compound conditions, which are longer expressions and require more effort in evaluation, can also be evaluated in the same manner. But if the expression is oftenly checked then to simplify the implementation you would rather evaluate it in a Condition_Evaluation method that returns whether the expression is true or false and hence save the effort of repeating the

code in several other methods. For example, an IsEmpty method of a queue class can be considered a condition evaluation method that checks whether a queue is empty or not.

You can also use static data members in the AState class for conditions that are common to all state classes.

• The entry and exit specification of each state are implemented in its entry and exit methods, which are called on transitions to enter or leave a state. The UpdateState method in the entity's interface invokes the exit method of the old state and the entry method of the new one.

Entity_Interface

The Entity_Interface class acts as an interface to the logic encapsulated in the statechart pattern. The Entity_Interface holds the current entity's state, the Event_Dispatcher method receives events from the application environment and calls the state implementation of that event accordingly. The entity's state is updated when UpdateState is invoked by the set_entity_state method.

Example

Consider the example of the turnstyle coin machine in its simplest form. The specification is shown in the following chart



Figure 1.4 The turnstyle coin machine specification

So, how do you use the *Basic Statechart* pattern to design the coin machine?

- 1. Identify the state classes Locked, Unlocked, and Broken
- 2. Implement the events methods Coin for coin insertion, Pass for person passage, a Failed method for machine failure and a Fixed method after being fixed.
- 3. Identify action methods. 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. Implement these specified methods in the Actions class.

- 4. Implement the entry and exit specification as methods in each state class. For example, the coin machine should keep track of the amount of coins inserted. So, in the Locked state the machine keeps counting the amount inserted using Accumulate() method. On entering the Locked state the machine displays a message telling the user to insert coins to pass, thus on the entry() method the message is displayed. Each time the machine leaves the lock state it should clear the amount of accumulated amount to zero, thus the exit() method clears the amount.
- 5. Identify the *conditions* in each state class. For example, the condition Amount>= CorrectAmount is true whenever the accumulated sum is greater than a predefined value CorrectAmount. Thus, in the Locked state, you declare the attribute Amount to hold the accumulated sum and the condition checking is implemented in the coin insertion event (coin() method). If the condition is true, the person is allowed to pass; i.e. the Unlock action method is called and the transition to UnLocked state is activated, otherwise the machine will still maintain its Locked state.

Figure 1.5 shows the statechart design of the coin machine based on the basic pattern structure. Only a simplified portion of the full design is presented for illustration purposes.



Figure 1.5 The coin machine design using the Basic Statechart pattern

Consequences

- **Understandability:** The clear role of each participant in the design facilitates the understandability of the design. The use of multiple inheritance to mix the Actions and Events and encapsulate the behavior for each state class makes the design more comprehensible and easily maintainable.
- **Traceability**: The design is easily traceable because the pattern maps statechart specification into classes, attributes, and methods in an OO design, and hence eases the traceability from a statechart (Figure 1.4) to design (Figure 1.5).
- **Complexity**: The model is flexible to 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 would to be required for each state.

Related Patterns

Basic Statechart is related to state machine patterns from a behavior perspective. Finite state machine patterns are described by Robert Martin in the *Three-Level FSM* [Martin95], Paul Dyson and Bruce Anderson [Dyson+98], Alexander Ran [Ran96], and our FSM patterns discussed in Chapter 10. However, *Basic Statechart* focuses on some formalized elements of statecharts such as the state exit and entry methods, and how and where to implement the conditions specification, these issues were not addressed in finite state machine patterns.

1.5 Hierarchical Statechart

Context

You are using the *Basic Statechart*. The application is large and the states seem to have a hierarchical nature.

Problem

How do you implement the states hierarchy in your design?

Forces

• Understandability of a complex system behavior. You are using a statechart to describe an entity's behavior and you find that a flat description of states is not illustrative because of their large number. Thus you decided to make use of the hierarchical nature of the statechart specifications to simplify the visual presentation and make it more understandable. The use of statecharts hierarchy introduces superstates, which contains other simple states. *Basic Statechart* doesn't support the concept of hierarchy thus you have to extend the design to allow enclosure of states inside superstates.

Solution

To implement hierarchy in your design, you have to distinguish different types of states:

- A SimpleState : a state that is not part of any superstate and doesn't contain any child state. (no parent and no children)
- A Leaf State: a state that is child of a superstate but doesn't have any children (has a parent but has no children).
- A Root SuperState: a state that encapsulates a group of other states (children) but has no parent.
- An Intermediate SuperState: a state that encapsulates a group of other states (children) and has a parent state.

All these state classes are subclassed (inherited) from AState class and the child/parent relationship is then implemented.

Structure

We use two pointers to implement the child/parent relationships, the MySuperState and the CurrentState. Figure 1.6 shows the extensions to *Basic Statechart* to support hierarchy.



Figure 1.6 The Hierarchical Statechart pattern

Participant

In addition to the participants of Basic Statechart, Hierarchical Statechart has:

SimpleStates

They are the *States* participants of the *Basic Statechart*.

IntermediateSuperState

Does the functionality of both the RootSuperState and the LeafState.

RootSuperState

- Keeps track of which of its children is the current state using CurrentState
- Handles event addressed to the group and dispatches them to the current state to respond accordingly.
- Produces common outputs for children states, and it can also implement the common event handling methods on their behalf.
- Performs state-driven transitions from self to the next upcoming states.
- Implements the entry and exit methods for the whole superstate.

LeafState

Does the same functionality as a SimpleState and additionally uses a MySuperState pointer to change the current active state of its parent class.

Example

Consider the coin machine, the superstate S_Functioning is introduced as a superstate for the Unlocked and Locked states. Figure 1.7 shows the statechart of the example in which we distinguish: Broken as a SimpleState class, Locked and Unlocked as LeafState classes, and S_Functioning as RootSuperState. The conditions and actions are not shown.



Figure 1.7 A hierarchical statechart for the coin machine example



Figure 1.8 The coin machine design using the Hierarchical Statechart pattern

Consequences

- **Understandability and Traceability:** The use of the hierarchical principle of statecharts simplified the understandability of the design. Mapping the hierarchical states into design facilitate the traceability from specification into implementation and hence maintainability of the design.
- Ease of Instantiation: The implementation of the application specific states is achieved through the extension of the pattern solution structure by inheriting from the SimpleState, LeafState, RootSuperState, and IntermediateSuperState. This facilitates the instantiation process of the pattern in application specific examples.
- Visual Hierarchy: The major drawback of applying the pattern is that the visual hierarchy of states in a statechart specification is not mapped into visual hierarchy in the design. For example, it is not visually clear from the design class diagram in Figure 1.8 that the S_Functioning state encapsulates the Locked and UnLocked states.

Related Patterns

Hierarchical Statechart can be considered a *Composite* pattern [Gamma+95] in which a composite class (SuperState) is composed of child classes (LeafStates or other SuperStates) of similar abstract type (AState).

1.6 Orthogonal Behavior

Context

You are using *Hierarchical Statechart*. Your entity has several independent behaviors that it exercises at the same time.

Problem

How can you deploy the entity's orthogonal behaviors in your design?

Forces

• An entity may exercise multiple independent behaviors: You want to simplify the behavior of an entity by explaining it as groups of independent state diagrams whose individual behavior describes one aspect of the overall behavior. You cannot use conventional state diagrams because they are poorly sequential as only one state can be active at a time. Therefore, you have used statechart orthogonality principle to accomplish your task. *Hierarchical Statechart* doesn't support orthogonal behavior and hence you have to extend its design.

Solution

Consult your statechart specification to identify those superstates that run orthogonal and dispatch the events to each of those states. Define a *Virtual superstate*¹ as a collection of superstates that process same events. Then group these states in a virtual superstate whose event method will call all the event method of the attached superstates. Figure 1.9 shows an example, virtual state "V" will receive the event "g" from the entity interface and will dispatch it to both superstates "A" and "D".



Figure 1.9 Example of a virtual superstate

¹ Note that the name virtual doesn't necessarily mean that the state doesn't really exist in the specification, it can be a meaningful superstate, however the name virtual is chosen as it is virtually grouping the orthogonal states together.



Figure 1.10 The structure of the Orthogonal Behavior pattern

Example

Consider the coin machine example, you want to add a warning system that operates when the machine is broken. The warning system may turn on a warning lamp, display a message or send notification to the operator. Thus, you have independent behavior describing the warning operation as shown in Figure 1.11. To map this to design, a virtual class called V_CoinMachine is created, which contains the two superstates S_Warning and S_Operation, events received by the interface is dispatched to the V_CoinMachine class which dispatches them to both the S_Operation and S_Warning classes. Figure 1.12 shows the class diagram, other states such as Broken, Locked, etc. were removed from the diagram for simplicity.



Figure 1.11 An orthogonal statechart of the coin machine



Figure 1.12 The coin machine design using the Orthogonal Behavior pattern

Consequences

Flexibility in implementation: Using a virtual state class has the advantage of giving freedom to the designer in implementing orthogonality. Sequential implementation can be achieved by calling one orthogonal state at a time. Concurrent implementation can be achieved by firing events to each state object which is running as a thread of operation (if supported by the underlying operating system). This addresses the concern of orthogonal states and concurrent objects as discussed by Harel *et.al.*[Harel+97] and provides flexibility in implementation.

Related Patterns

The *Orthogonal Behavior* pattern can be considered a *Composite* pattern [Gamma+95] in which a composite class (VirtualState) is composed of several children (SuperState classes) of the same abstract type (AState).

1.7 Broadcasting

Context

You are using *Orthogonal Behavior*. Some events occurring in one state trigger other events in orthogonal superstates.

Problem

How do you broadcast the produced event to the listener state?

Forces

• Relationship between orthogonal states: The orthogonal behaviors of the entity are still logically related to each other because they describe the behavior of one entity. When you use the orthogonality of statecharts, you sometimes find that the actions generated in one superstate stimulate an event in another superstate. None of the previous statechart patterns explicitly deals with this situation, however you can make use of their solution structure to broadcast the stimulated event to the orthogonal superstate.

Solution

When a new event is stimulated, the broadcasting state would inject the event directly to the entity interface to handle it. The event propagates though virtual superstates and hence it reaches the orthogonal superstate of the broadcasting state. The structure of *Orthogonal Behavior* is used and only those state that need to broadcast event calls the Entity_Interface with the new event.

Example



Figure 1.13 The coin machine statechart with broadcasting capability

In the example of the coin machine, assume that the Fixed event will stimulate another event Warning_OFF which will turn the superstate S_Warning from being in WarningON to WarningOFF state. In this case, the design structure will be the same as *Orthogonal Behavior*, but the S_Warning state will not implement the Fixed event but it will implement a Warning_OFF event. The Fixed method of the Broken state will broadcast the Warning_OFF event by calling the entity interface that would look like:

```
void Broken::Fixed(){
{ //....
Entity_Ref->Event_Dispatcher(Warning_OFF);
}
```

Discussion

You use *Broadcasting* to transfer the events to orthogonal superstates. The source of event generation differs, some events are generated as actions of a former event, some are due to a condition becoming true in one state that affect the orthogonal state, and some events are due to being in a new state (In(mode) event). We only discussed one example and one solution. However, other solutions, such as queuing the new event until the current transition is done, can be accommodated in the pattern.

1.8 History State

Context

You are using *Hierarchical Statechart* and you find that a superstate should have memory of which active state it was last in just before exiting the whole superstate.

Problem

How do you keep the history of a superstate in your design?

Forces

• Using the default state versus the last active state: When you transition from a superstate to another (super)state and then back to the original superstate, you may want to re-enter the original superstate with its previous state rather than its default state. So, you use the history property of the statechart specification but when you implement it, you will need to keep the superstate object knowledgeable of its last state object. Without history, the entry of a superstate will initialize its CurrentState pointer to the default internal state object which would look like:

```
Void superstate::entry()
```

{ CurrentState = DefaultState; }

And the default state is initialized on the superstate constructor method. Using this for superstates with history removes the knowledge of a superstate of its latest internal state.

Solution

Initialize the CurrentState pointer of a superstate class once on creation, do not reinitialize it on the superstate entry() method. This will keep the value of the pointer on entry as that of the last exit.





Figure 1.14 The coin machine statechart with history property

In the coin machine example, let us assume that the machine can be turned on and off and reserves its previous state, thus it should return to Broken if it was last out of order and should return to S_Functioning if it was operating correctly before shutting down. In this case, the entry() method as S_Operation superstate will not initialize its CurrentState pointer as would be the case in the default state implementation in *Hierarchical Statechart*.

1.9 General Discussion

The design of finite state machines is a common problem 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, elevator controls, and automated train door control. Gamma *et.al.*[Gamma+95] have pointed some known uses in graphical user interfaces. Dyson *et.al.* [Dyson+98] have also pointed out their usage in library applications.

Several patterns for finite state machines were previously presented in the literature [Dyson+98, Martin95, Ran96, and Yacoub+98c], some of which are also applicable to statecharts as they both share some common properties. For example, the *Default State*, *Exposed State* and *Pure State* patterns described by Dyson *et. al.* [Dyson+98] might also be applicable to state classes in statecharts. An integrative work of *State* pattern [Gamma+95,Dyson+98], *Finite State Machine* patterns [Martin95, Ran96, Yacoub+98c], and *Statechart* patterns could provide a comprehensive pattern language to solve problems in implementing an entity's behavior.

| Pattern Name | Problem | Solution |
|----------------------------|--|---|
| Basic Statechart | Your application contains an entity whose behavior depends on its state. You have decided to use statechart's specifications to specify the entity's behavior. How do you implement the statechart specification into design? | Use an object oriented design that encapsulates the state of the entity into separate classes that correspond to the states defined in the specification. Distinguish the events, conditions, actions, entry and exit activities in each state class as methods and attributes of the state classes. |
| Hierarchical Statechart | You are using the <i>Basic Statechart</i> . The application is large and your states seem to have a hierarchical nature. How do you implement the states hierarchy in your design? | Use superstates classes that are inherited from the abstract state class. Use the <i>Composite</i> pattern [Gamma+95] to allow the superstate to contain other states. Keep the superstate knowledgeable of the current active state and dispatch events to it. |
| Orthogonal Behavior | You are using the <i>Hierarchical</i> <i>Statechart.</i> Your entity has several independent behaviors that it exercises at the same time. How do you deploy the entity's orthogonal behaviors in your design? | Identify the superstates that run independently in your specification, then define a " <i>Virtual superstate</i> " as a collection of superstates that process the same events, dispatch the events to each state. |
| Broadcasting | You are using the <i>Orthogonal</i> <i>Behavior</i> . How can you broadcast a stimulated event produced from another event occurring in an orthogonal state? | When a new event is stimulated, make the broadcasting state inject the event directly to the entity interface which dispatches it to the virtual superstate. Eventually, the virtual supertate dispatches the event to all of its orthogonal states. |
| History State | If one of the superstates has a history property, how do you keep its history in your design? | Initialize the current active state class pointer of the superstate object once on creation, use it throughout the entity's lifetime, and do not reinitialize it on the superstate entry method. |

1.10 Summary of Statechart Patterns

Table 1-1 Summary of Statechart patterns