A Pattern Language of Statecharts

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

INTRODUCTION

Finite state machines and their extension to statecharts are widely used in reactive systems. David Harel [Harel87] has introduced statecharts as an extension of finite state machines to describe the complex behavior of an entity. The formalized concepts of statecharts and their specifications have been used in many applications. Here, we show how to solve recurring design problems in implementing statechart specification of an entity in an object-oriented application. The statecharts' patterns represent solutions to frequent design problems that are commonly thought of by system designers, which include how to deploy hierarchy, orthogonality, and broadcasting in a statechart's object oriented design. Since statecharts are frequently applicable to software applications, thus it is helpful for the system designer to directly exercise the statechart pattern language in his application design.

In the next section, a quick background of statechart is presented, then a pattern map summarizes the statechart patterns and their relation to finite state machine patterns. The rest of the sections describe the patterns themselves.

BACKGROUND

David Harel [Harel87,Harel88] introduced statecharts to extend finite state machine for complex behavior and to describe a visual formalism. Objectcharts were then presented by Derek Coleman *et.al.*[CHB92] 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 an implementation mechanism for *Recursive Control*. Alternatively, we perceive that statecharts are frequently used as a behavior description technique and using an object-oriented design of statechart provides flexibility at the design level and facilitates the design maintainability. We explain how to deploy reusable design solution to frequent statecharts concepts. It is recommended that readers unfamiliar with statechart consult the visual formalism by David Harel [Harel87,Harel88] before proceeding in reading the following patterns, a summary of statecharts principles and properties is presented in appendix (B).

PATTERNS SUMMARY

Several articles have been presented to document solutions of recurring design problems in implementing finite state machines. Erich Gamma *et.al.*[GHJV95] 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 Andreson [DA98] 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.

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 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 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 the orthogonal behavior to support event broadcasting. Finally, the statecharts' history specification is sometimes used in the superstates of the *Hierarchical* pattern; *History-State* addresses the problem.

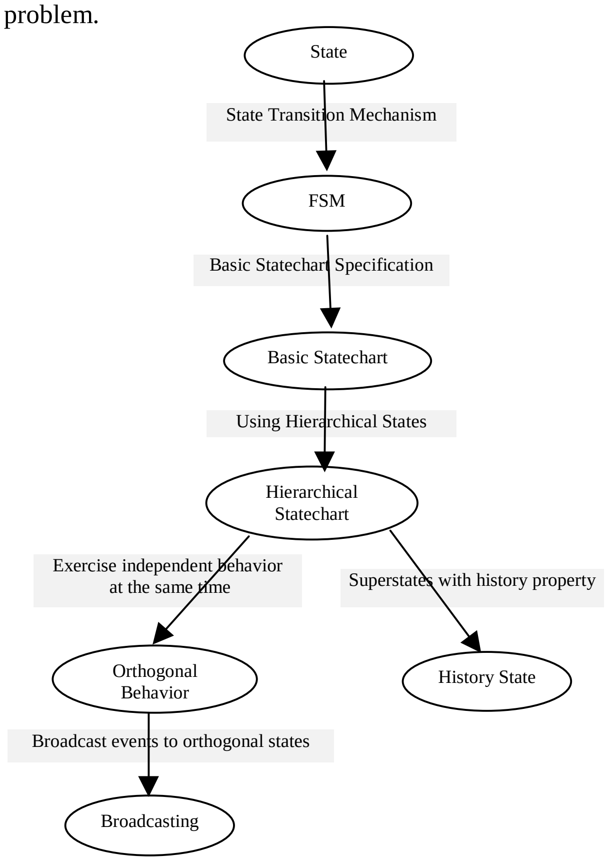

State
State Transition Mechanism
FSM
Basic Statechart Specification
Basic Statechart
Using Hierarchical States
Hierarchical Statechart
Exercise independent behavior at the same time
Superstates with history property
Orthogonal Behavior
History State
Broadcast events to orthogonal states
Broadcasting

Figure 1 Patterns Map

The turnstyle coin machine specification discussed by Robert Martin [Martin95] is extended one step at a time to illustrate the various patterns usage. The example is selected for its illustrative capability to show how the pattern language is used to solve several design problems. Appendix (A) summarizes the problem/solution pairs addressed by the statechart pattern language.

BASIC STATECHART

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. You are using a statechart to specify the entity's behavior.

Problem

How do you implement the statechart specification into design?

Motivations

A statechart is a specification language which 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 that encapsulates the state of the entity into separate classes which correspond to the states defined in the specification. Distinguish the events, conditions, actions, entry and exit procedures in each state class. Figure (2) shows the design solution structure in UML notation [UML98].

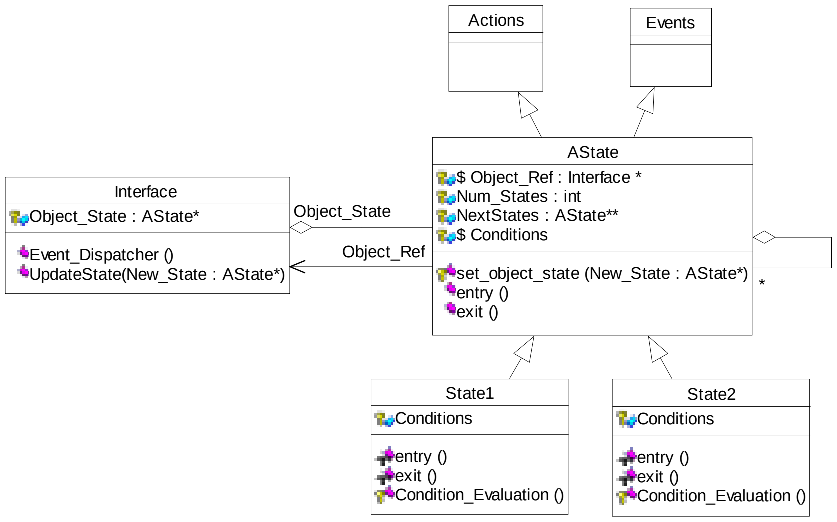


Figure 2 The structure of the basic statechart pattern

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 event class and each state class implements the adequate functionality required in response to that particular event. 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 by the state classes' event methods implementation. Generally, action methods are static methods inherited by the state classes. 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 state-driven transition [DA98] mechanism, the entity's state will be changed when the "`set_object_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 [DA98], in this case the 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.

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_object_state`" method.
- Host state specific conditions which can be implemented by:

  a. Using a Boolean data member of a state class. For example if we want to implement whether a link is established between two communicating entities, we 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 can invoke the exit method of the old state and the entry method of the new one.

Interface

The "`Interface`" class acts as an interface to the logic encapsulated in the statechart pattern. The 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_object_state`" method.

Example

Consider the example of the turnstyle coin machine in its simplest form. The specification is shown in the following chart. The design participants are identified as follows:

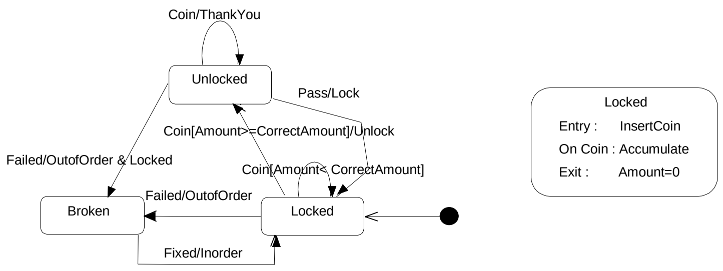


Figure 3 The turnstyle coin machine specification

1. The *state* classes "`Locked`", "`Unlocked`" and "`Broken`"
2. The *events* "`Coin`" method for coin insertion, "`Pass`" method for person passage, a "`Failed`" method for machine failure and a "`Fixed`" method after being fixed.
3. *Actions;* 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. The "`Actions`" class methods implements the various actions.
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, we 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(4) 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.

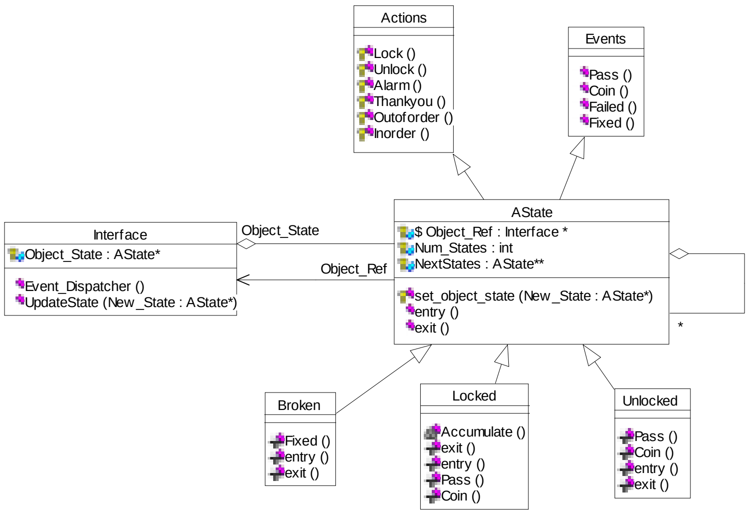


Figure 4 The coin machine design using the basic statechart pattern

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 [DA98], and Alexander Ran [Ran96]. 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. One instance of each state class will be instantiated at a time, and hence they can be considered *Singletons* [GHJV95] as well.

HIERARCHICAL STATECHART

Context

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

Problem

How do you implement the states hierarchy in your design?

Motivations

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 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 modify 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 Simple State* : 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 part of a superstate but doesn't have any child state (has a parent but has no children).
- *A Top 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 state class types are subclassed (inherited) from "`AState`" class. We use two pointers to implement the child/parent relationships, the "`MySuperState`" and the "`CurrentState`". Figure(5) shows the extensions to *Basic Statechart* to support hierarchy.

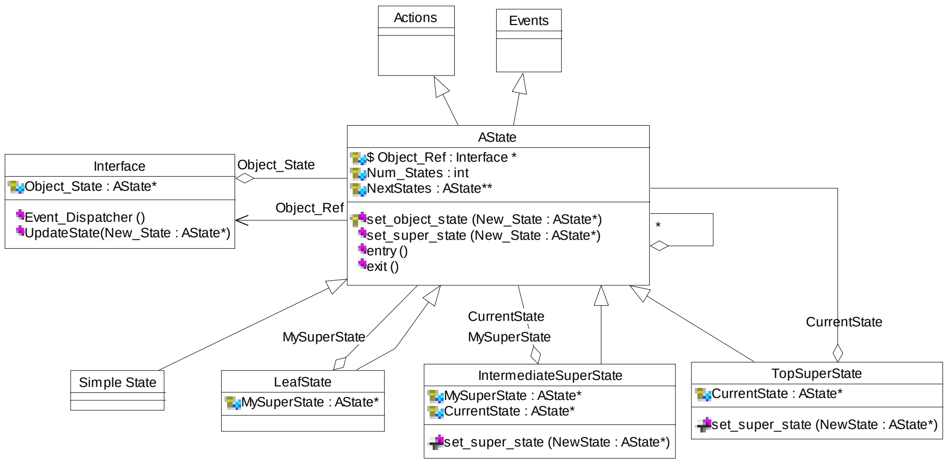


Figure 5 The Hierarchy statechart pattern

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 TopSuperState and the LeafState.

TopSuperState

- Keeps track of the current state of the group of states 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:

- Uses a "`MySuperState`" pointer to change the current active state of its parent class.

Example

Consider the same example of the coin machine (although the example is simple and need not be hierarchical, we will use it for illustration), the superstate "`S_Functioning`" is introduced as a superstate for the "`Unlocked`" and "`Locked`" states. Figure(6) 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 TopSuperState. The conditions and actions are not shown.

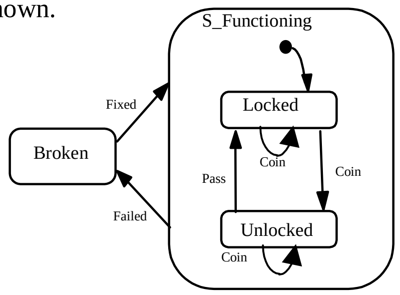
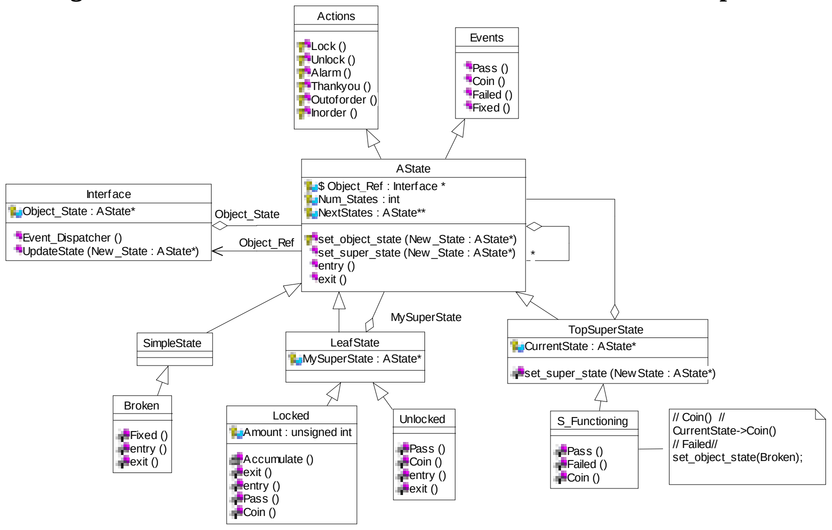


Figure 6 Hierarchical statechart for the coin machine example



```
// Coin()  //
CurrentState->Coin()
// Failed//
set_object_state(Broken);
```

Figure 7 The coin machine design using the hierarchical statechart pattern

Related Patterns

Hierarchical Statechart can be considered a *Composite* pattern [GHJV95] in which a composite class (SuperState) is composed of child classes (LeafStates or other SuperStates) of similar abstract type (AState).

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?

Motivations

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 the orthogonal behavior and hence you have to add it to the design.

Solution

Consult your statechart specification to identify those superstates that run orthogonaly and dispatch the events to each of those states. Define a "*Virtual superstate"* as a collection of superstates that process the same events for the entity. The designer groups these superstates in a virtual superstate whose event method will call all the event method of the attached superstates. Figure (8) 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". Using a virtual state class has the advantage of giving freedom to the designer about the implementation of 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.*[HG97] and provides flexibility in implementation. We also 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(9) describes the structure of the solution.

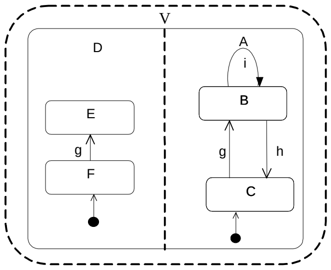


Figure 8 Example of a virtual superstate

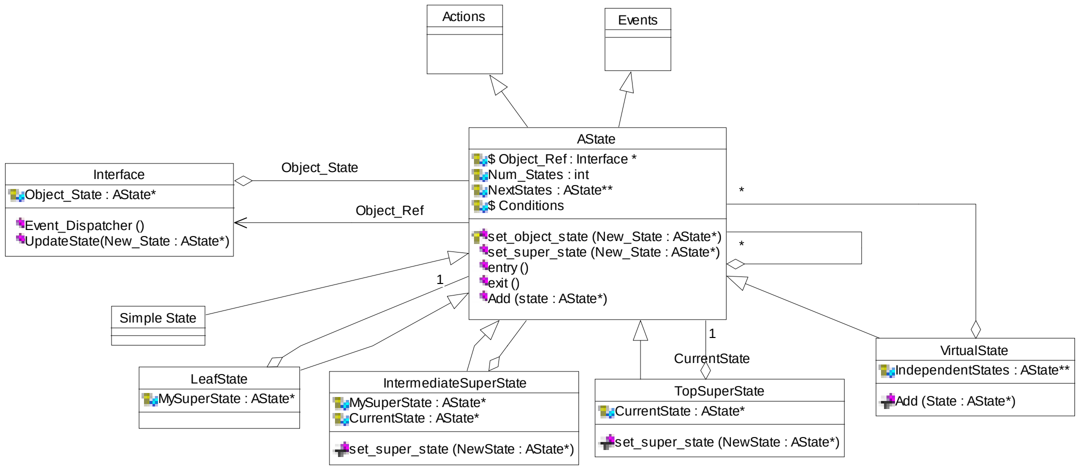


Figure 9 The Orthogonal Statechart pattern

Example

Consider the coin machine example, we 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, we have independent behavior describing the warning operation as shown in figure (10). To map this to a 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(11) shows the class diagram, other states such as Broken, Locked, etc. were removed from the diagram for simplicity.

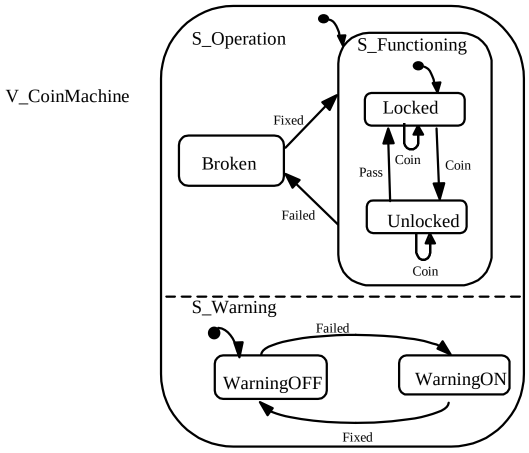


Figure 10 Orthogonality in a coin machine

Related Patterns

The *Orthogonal Behavior* pattern can be considered a *Composite* pattern [GHJV95] in which a composite class (Virtual class) is composed of several children (SuperState classes) of the same abstract type (AState).

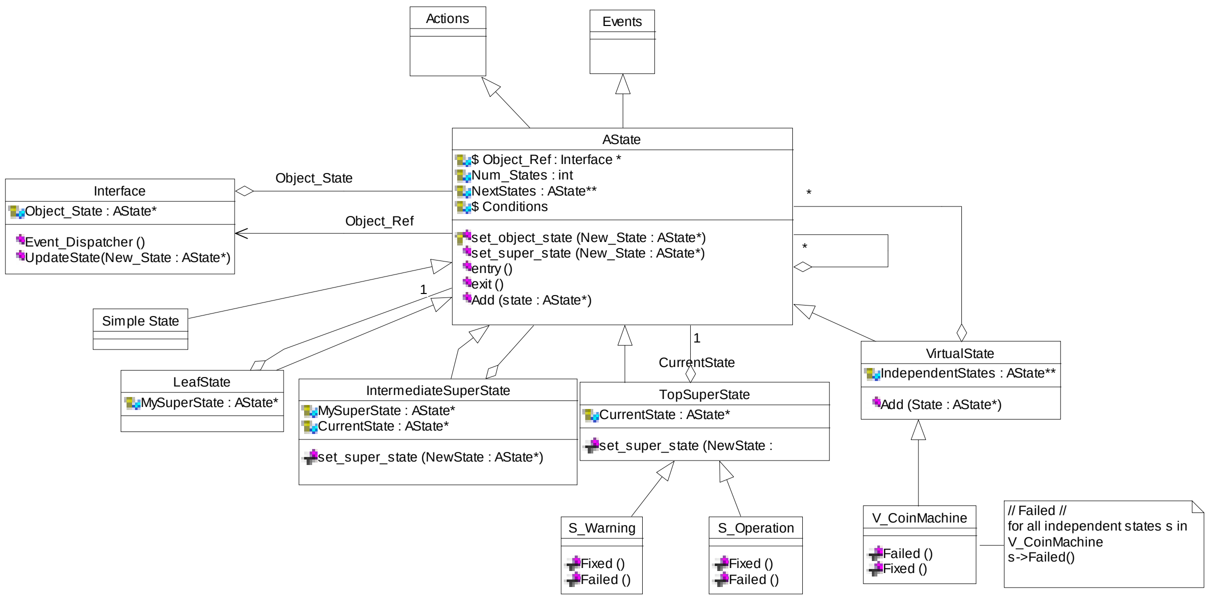


Figure 11 The coin machine design using the orthogonal statechart pattern

BROADCASTING

Context

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

Problem

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

Motivations

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 stimulates 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 "`Interface`" entity with the new event.

Example

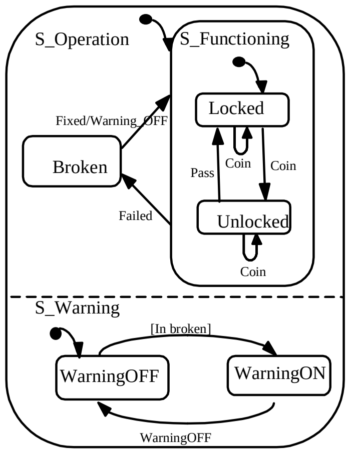


Figure 12 Coin machine statechart with broadcasting

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(){
{ //….
Object_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.

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?

Motivations

When you transition from a superstate to another (super)state and then back to the original superstate, you may want to reenter 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.

Example

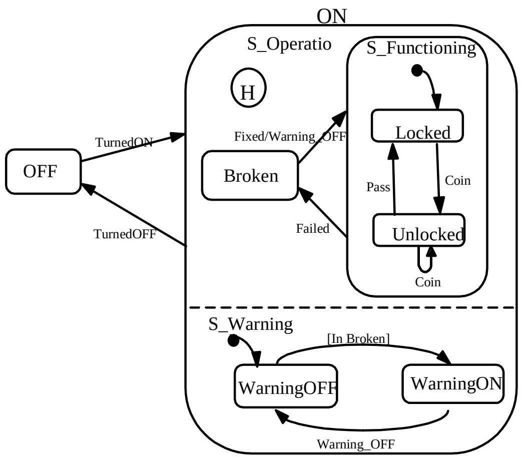


Figure 13 The coin machine with history property

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*.

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, elevators control, and automated train door control. Gamma *et.al.*[GHJV95] have pointed some known uses in graphical user interfaces. Dyson *et.al.* [DA98] 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.

Several patterns for finite state machines were previously presented in the literature [DA98,Martin95,Ran96], 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 Paul Dyson and Bruce Anderson [DA98] might also be applicable to state classes in statecharts. An integrative work of *State* pattern [GHJV95,DA98], *Finite State Machine* patterns [Martin95,Ran96], and *Statechart* patterns would provide a comprehensive pattern language to solve problems in implementing an entity's behavior.

ACKNOWLEDGEMENT

We would like to thank Dennis DeBruler, our PLoP shepherd, for his valuable comments and extensive help in revising the patterns and improving their quality.

REFERENCES

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software” Addison-Wesley 1995

[DA98] 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

[UML98] UML Resource Center. http://www.rational.com/uml/documentation.html

[Martin95] Robert Martin, "THREE-LEVEL FSM". In James Coplien, Douglas Schmidt, (edt.), Pattern Languages of Program Design. Addison-Wesely,1995, chapter 19, pp383

[CHB92] Derek Coleman, Fiona Hayes, and Stephen Bear, "Introducing Objectcharts or How to Use Statecharts in Object-Oriented Designs," IEEE Transactions on Software Engineering, Vol 18, No. 1, January 1992.

[Harel87] David Harel, "Statecharts: a Visual Formalism for Complex Systems," Science Computer Program, Vol 8, pp 231-274, 1987

[Harel88] David Harel, "On Visual Formalism", Communications of the ACM, Vol 31, No 5, May 1988

[Selic98] Bran Selic, "Recursive Control". In Robert Martin, Dirk Riehle, and Frank Buschmann (edt.) "Pattern Languages of Program Design 3", Addison Wesely Longman Inc 1998, chapter 10, pp147

[HG97] David Harel and Eran Gery, "Executable Object Modeling with Statecharts", IEEE Computer magazine, July 1997, pp31-42

[Ran96] Alexander Ran, "MOODS: Models for Object-Oriented Design of State", In John M. Vlissides, James O. Coplien, and Norman L. Kerth (edt.) Pattern Languages of Program Design 2, Addison Wesely Longman Inc 1996, Chapter 8, pp119-142

APPENDIX (A): SUMMARY OF STATECHART PATTERNS

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 which 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 *Basic Statechart*. 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 *Composite* pattern [GHJV95] 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 *Hierarchical Statechart*. 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 "*Virtual superstate"* as a collection of superstates that process the same events, dispatch the events to each state.
Broadcasting	You are using the *Orthogonal Behavior*. 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, do not reinitialize it on the superstate entry method.

APPENDIX (B): PRINCIPLES OF STATECHARTS

Statecharts Elements

States : Describes an entity's behavior in a given state. They are the static elements of statecharts. States have entry and exit procedures, and process events as they occur in the state.

Triggers: They are the dynamic elements of the statecharts that cause state transitions or state reactions. Events, conditions or both can be triggers

Events: Events occur at specified instances of time. They can be generated internally in the statechart or externally.

Conditions: Boolean expressions, valued TRUE or FALSE. Conditions can be primitive elements or compound elements that express a set of boolean operations such as ANDs or ORs.

Actions: Operations performed as a result of events in the statecharts.

Activities: Operations that are performed by the entity in a given state.

State Entry: Special event type that describes what the entity should do on entering a particular state.

State Exit: Special event type that describes what the entity should do on exiting a particular state.

Statecharts Principles

Statecharts extends finite state machines to support the following basic principles:

Hierarchy

The hierarchy principle introduces a more global state, referred to as a *superstate*, that includes other entity's states. It is sometimes called `"Depth"` principal as well as the `"XOR-Decomposition"` because only one state of the superstate describes its behavior at a time. As an example, figure (14), an entity in superstate "A" can be in either simple state "B" or "C".

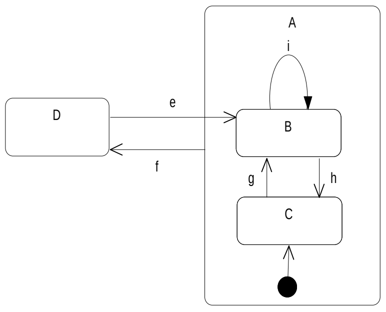


Figure 14 Hierarchy Principle of Statecharts

Orthogonality

This is the principle of Statecharts that allows several behaviors to be experienced at the same time. An event may cause a state transition in several superstates at the same time. This Principle is sometimes referred to as the `"AND-Decomposition"` principle as changes may occur in several superstates. As an example, figure (15), an event "g" will cause the entity's behavior in superstate "A" to change from "C" to "B" and the entity's behavior in superstate "D" to change from "F" to "E" as well.

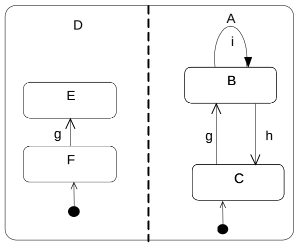


Figure 15 The Orthogonality Principle of Statecharts

Broadcasting

A state transition inside a superstate may cause another event, which in turn triggers state transition in another superstate. As an example, figure (16), the event "g" will produce output event "k" which will change the state in superstate "A" from "C" to "B"

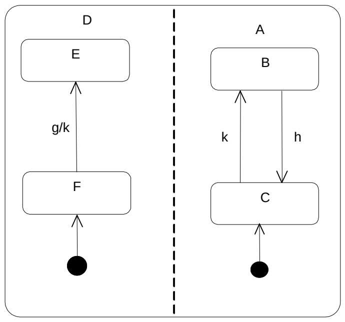


Figure 16 Broadcasting Principle of Statecharts

History

After transition from a superstate to another (super)state and the return back to the original superstate, it is sometimes required to return to the previous state of the original superstate rather than the default state. As an example, figure (17), assume that the current state is "B" and the event "m" causes the transition to state "E". Then event "p" occurs, in default state situation the current state will be "C", but since superstate "A" has history of its last internal state then the current state will be "B"

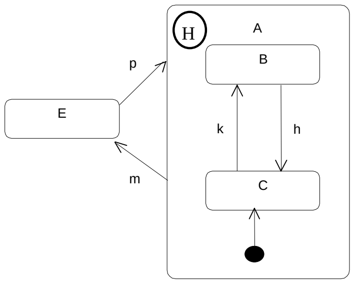


Figure 17 History Principle of Statecharts

Thus; Statecharts = State Machine + Depth + Orthogonality + Broadcasting + History