The Anthology of the Finite State Machine Design Patterns

Paul Adamczyk

University of Illinois at Urbana-Champaign
Department of Computer Science
1304 W. Springfield Ave, Urbana, IL 61801
email: padamczy@uiuc.edu

ABSTRACT

No Finite State Machine (FSM) is an island. Every aspect of an FSM depends on its context, limitations of the programming language, system requirements, and many other factors. In more advanced systems, many independent FSMs cooperate together. The design describing their relationships is not an archipelago either, but rather a tightly coupled structure of classes or class hierarchies. This paper brings together many Object-Oriented FSM designs and compares them from the perspective of added flexibility as well as the cost associated with it. The final product is summary of the competitive advantages of different FSM designs in a specific set of conditions (e.g. problem domain, user expectations). This paper is intended to assist software engineers faced with the task of designing an optimal FSM implementation in a specific context. The uniform format of pattern summaries is intended to allow the designers to compare different patterns, based on their key characteristics – applicability, advantages, and disadvantages.

INTRODUCTION

Software designers have befriended finite state machines from the very dawn of time. But with the advent of the Object-Oriented technology, this friendship grew colder, because FSMs could not be easily implemented using objects. Many nominally Object-Oriented FSM designs were in fact structural designs. As the designers grew more accustomed to the Object-Oriented paradigm, they have developed several methods to model FSM using this new methodology.

The most popular non-Object-Oriented implementations of FSMs are cascading if statements [Pal97], nested switch statements [vGB99], state transition tables [Car92] [Sam02], and generated code using goto statements. They are popular because they provide fast execution, but at the cost of weaker readability and maintainability. Some of the Object-Oriented designs described in this paper, motivated by the execution speed rather than clear separation of design elements, borrow from these implementations. On the other extreme, the pure Object-Oriented FSM designs require "complete reification" i.e. making every element of the state machine an object [Ack95]. FSM design patterns described in this paper occupy the entire spectrum of design decisions bound by these two extremes.

This paper describes known finite state machine patterns as a progression of cost vs. benefit design choices: speed vs. flexibility, readability vs. overhead, or ease of initial development vs. extensibility. Each pattern is presented in the context that takes advantage of its benefits and is not limited too much by its drawbacks. Patterns presented here are not original, but rather come from well-known research. The contribution of this

paper is to combine them into a continuum of extensions that result in more involved designs that are rooted firmly in the original ideas of the State DP[1] [GHJV95].

CATALOGUE OF PATTERNS

The patterns analyzed in this paper fall into three groups. The first group contains the State DP and pattern languages that further describe consequences and implementation alternatives presented in State DP. The second group lists patterns that build on the State DP to create larger, more flexible designs. The third group consists of object-oriented patterns that present alternative implementations of finite state machines.

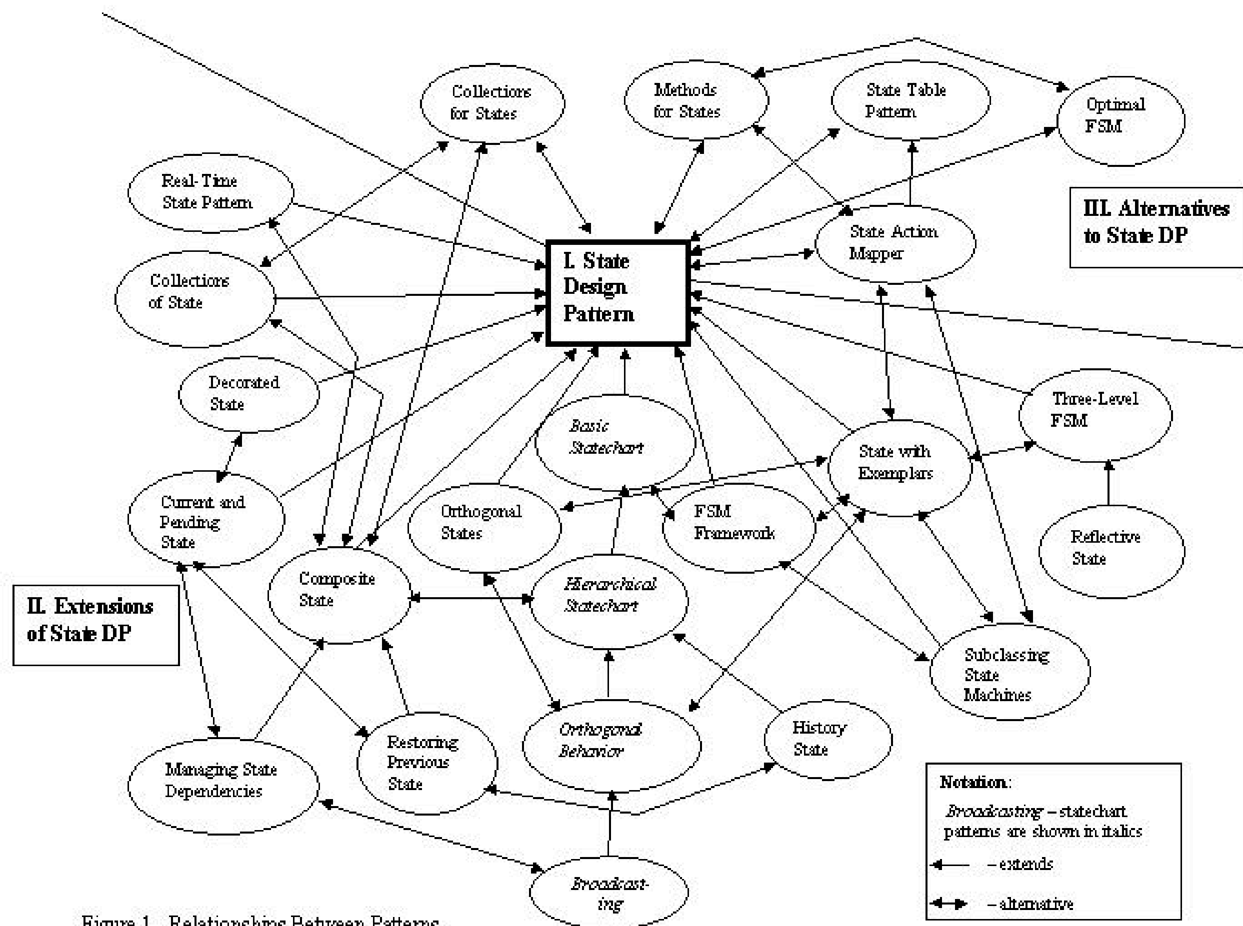

Figure 1. Relationships Between Patterns

Figure 1, "Relationships Between Patterns" contains the complete listing and depicts dependencies between all patterns described in this paper. The first type, Variations of State pattern is enclosed in the State DP rectangle, because these patterns are not providing any additional knowledge, just explain in more detail how the State pattern works. Patterns that present alternative solutions (for a part of or the entire problem) are connected with a double-headed arrow. All patterns classified as Alternatives to State DP

[1] The name "State DP" is as much a convenience (shorter than "Objects for States" and other names given to this pattern) as a necessity to distinguish it from other State design patterns.

demonstrate that property. Patterns that build on other patterns use a single arrow to point to the pattern they extend. Most Extensions to State DP extend its design directly, but they are also related to other patterns. Since many patterns described here do not have a name (or their original author did not even consider them patterns), some of them are named by the author of this paper.

I. STATE DP AND ITS MANY ELABORATIONS

This section describes the major characteristics of State DP along with other patterns (listed in Related Patterns) that further address specific aspects of State DP. These patterns do not describe solutions different from State DP, but analyze specifically all major decisions required to understand this pattern.

State DP [GHJV95, pp. 305]

Applicability

An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
Operations have large, multi-part conditional statements that depend on the object's state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure. The State DP puts each branch of the conditional in a separate class. This lets you treat the object's state as an object in its own right that can vary independently from other objects [GHJV95, 306].

Solution

Each state is encapsulated in a separate class that implements the common State interface and defined behavior to handle all external events. Context class is responsible for maintaining the knowledge of the current state and other data. External entities communicate with the system via the Context class that forwards requests to the current state object.

Consequences

- ✦ State-specific behavior is localized and partitioned between different states.
- ✦ State transitions are performed explicitly in the code.
- ▭ If an object has many states, many classes need to be created, which may result in an excessively large class hierarchy. The Decorated State [OS96] pattern described below solves this specific problem.
- ▭ The Context class delegates all external events to corresponding methods in the interface of the State class. It is responsible for providing the correct mapping. This results in a tight coupling between the Context class and concrete events.

Related Patterns

Two pattern languages that address in detail many specific design and implementation decisions raised by State DP are "State Patterns" [DA96] and "Finite State Machine Patterns" [YA98b]. Neither pattern language describes patterns that are extensions to State DP, but they provide a very detailed explanation of choices to be made for each concrete implementation of State DP. "State Patterns" explain how to allocate state

members to the appropriate class (Context vs. State), list optimizations that can be done to state classes without members, discuss methods of exposing state classes to the outside world, discuss tradeoffs of allocating responsibilities for invoking state transitions, and emphasize the importance of the initialization of state machine with a meaningful default state. “Finite State Machine Patterns” explore in even more details the choice of state machine types, structures, state transitions, tradeoffs of the exposure of state classes, and the responsibility of state instantiation.

II. EXTENSIONS OF STATE DP

This section list various patterns that build upon State DP to provide larger, more general or more specialized patterns. Each pattern is described in reference to State DP as well as other FSM patterns related to it.

Three-Level FSM [Mar95]

Applicability

Applies in any context where behavior may be controlled by more than one finite state machine and logic-independent behavior should be reusable across multiple FSMs.
When behavior needs to be overridden and/or extended through inheritance.
When State DP is too restrictive because it couples behavior directly with incoming events.

Solution

Encapsulate FSM in three separate levels, summarized in Figure 2.

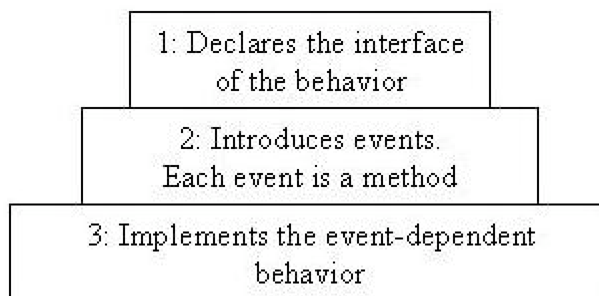

Figure 2. Levels of abstraction in Three-Level FSM pattern

Level 1 provides the complete behavioral interface of the state machine. It also may define an implementation of the portion of the interface that is event-independent, but often it has only a default, empty implementation [see Mar02, pp. 192]. Level 2 contains the concrete implementation of State DP where the Context class inherits from level 1. Level 2 introduces concrete events (modeled as methods) and an appropriate handling of them. Level 3 inherits from the Context class of level 2 and implements the behavior declared in level 1 that is event-dependent (i.e. it depends on events defined in level 2).

Consequences

✦ Clearly separates the behavior from concrete events.

- ✦ Logic of level 2 (State DP) could be generated.
- ▭ Use of virtual functions adds small run-time overhead.
- ▭ Level 2 may consist of a lot of classes (if there are many states).
- ▭ The Context class delegates all external events to corresponding methods in the interface of the State class. It is responsible for providing the correct mapping. This results in a tight coupling between the Context class and concrete events.

Related Patterns[2]

Three-Level FSM uses State DP to implement its level 2.
State with Exemplars provides a different model to abstract out events – each event is encapsulated in its own class that knows how to handle that event. Similarly to Three-Level FSM, the interface for the outside world is event-independent in State with Exemplars.

Reflective State [FR98]

Applicability

When the number of states is relatively large.
When State DP is not flexible enough, because the large number of states dictates that control aspects related to states and their transitions should be separated from the functional aspect (i.e. behavior).
When separation of control from behavior is more important than abstracting implementation-independent behavior from application specific one (as in the Three-Level FSM).

Solution

Separate the application into two levels – the finite state machine (FSM) level and the application level. The FSM level corresponds to the meta level of the Reflective architecture, while the application level corresponds to the base level of the architecture.
Follow the Reflection architectural pattern [BMRSS96] to design the meta level. The Context class and the State classes, similarly to State DP, implement the base level.

Consequences

- ✦ State-specific behavior is localized and partitioned between concrete State classes.
- ✦ Control aspects of a state machine are separated from functional aspects.
- ✦ State and Context class hierarchies are independent and can be extended separately. Consequently, there is no coupling between the Context class and the incoming events.
- ▭ But even with this flexibility, it is not possible to subclass State classes as Subclassing State Machines does.

---

[2] Martin lists the Strategy pattern as an alternative to State DP to implement the level 2 of the FSM. It may seem plausible since Strategy and State have the same class structure. However, in Strategy, the Context class selects an algorithm only once, at the beginning of processing, while in State DP, changes between concrete State classes occur often. Use of Strategy, instead of State DP in level 2 would make state changes impossible. Thus Strategy does not seem to be a good alternative to the State DP here.

- Modifications at the meta level may produce negative results.
- Increased number of components.
- Lower efficiency – many levels of indirection.
- Not all potential changes to the software are supported.
- Not all programming languages support reflection.
- Does not address the problem of communication between multiple FSMs.

Related Patterns
State DP and Reflection [BMRSS96] are combined to provide a very flexible reflexive interface. State DP provides the implementation of the base level, the Reflection pattern describes how to model the meta level.
In Three-Level FSM, behavior is logic-independent. Reflective State is even more flexible as it decouples behavior completely from the FSM constructs.

FSM Framework [vGB99]

Applicability
Modeling complex FSMs.
When the design requires all FSM elements to be modeled as objects.
When FSM components should be configurable.

Solution
FSMContext class holds the current state and all state-specific data (in a repository). State is represented by a single class and contains a set of transition-event pairs. Transition class has a reference to the target state and the corresponding FSMAction class. FSMAction class plays the role of Command class in the Command pattern.
FSM object receives the incoming events and responds to them by allocating a FSMContext instance. FSMContext object forwards the request to the State object that looks up the transition corresponding to the incoming event and executes it. The Transition object knows which FSMAction to execute and resets the current state on the FSMContext object.

Consequences
- ✦ States and events are configurable.
- ✦ Context can be a single component. There is no coupling between the Context class and concrete events.
- ✦ Actions can share behavior through inheritance.
- ✦ Actions can delegate execution to other Actions; thus each event needs to be mapped to a single Action only.
- ✦ Mapping between FSM design and implementation is more direct, because states, actions, events, and transitions are implemented as classes.
- ✦ All FSM elements except for Actions can be configured with a configuration tool.
- The use of the Context repository causes slower performance compared to direct access to state-specific data.

- The Context repository does not provide any interface to update the state-specific data so Action classes can make uncontrolled changes to the data.

Related Patterns
The FSM Framework extends State DP by modeling all FSM elements as classes.
Command [GHJV95, pp. 233] is used to implement a part of the design. FSMAction classes are Commands, the Transition class is the Invoker, and the FSMContext class is the Receiver.
State with Exemplars models events (but not actions or transitions) as classes.
Subclassing State Machines is similar to FSM Framework, but it displays tighter coupling between the Context class and concrete events.
Basic Statechart also encapsulates the actions and events into a separate class, but it combines all events in one class and all actions in another class.

State with Exemplars[3] [Ack95]

Applicability
When State DP is not flexible enough, because the external interface should be independent of the incoming events.
When the design requires events and FSM actions to be modifiable at run time.
When performance is not as critical as maintainability.

Solution
The Context class from State DP is replaced by two class hierarchies: Targets (containing the state-independent data) and Events (defining the interface to respond to external requests).
FSM container (un)registers concrete Events (exemplars) and Targets at run-time.
An externally generated event is encapsulated (as a struct/class) and forwarded by the FSM container to a concrete Event class that is a wrapper for that event.

Consequences
- ✦ The decision of which class should process the incoming event is encapsulated in the event class hierarchy. FSM container just queries the instances registered with it.
- ✦ Supports orthogonal states – FSM container will forward the event to the current state object of each currently registered Target.
- ✦ Loose coupling between State and Target. Use of the Event class hierarchy allows Targets to be oblivious to all actions. A Concrete State class is responsible for updating the data stored in the Target.
- The search of the concrete Event instance to handle the incoming event is slow (linear).

[3] Exemplars are objects used as models to construct other objects. Prototype [GHJV95] is a sample use of exemplars. Coplien [Cope92] describes exemplars in C++ in much detail.

Related Patterns

State DP encapsulates states as classes. State with Exemplars encapsulates both states and events as classes.

In Three-Level FSM, the external interface is also independent of events.

State Action Mapper also provides the dynamic registration of events, but can only support one State-Dependent Object (i.e. Context) at a time.

FSM Framework also encapsulates Events as classes. It models all FSM entities as classes.

Orthogonal Behavior and Orthogonal States – State with Exemplars supports the orthogonality by allowing many Contexts to register with the FSM container.

State with Exemplars is similar to the Command [GHJV95, pp. 233] pattern. Event classes play the role of Commands, State classes are Receivers, and the FSMContainer is the Invoker.

Subclassing State Machines attempts to achieve similar flexibility of design by allowing subclassing of State classes to override their handling of new events and actions.

Subclassing State Machines [SC95]

Applicability

Object-oriented design that permits a design of complex state machines by incrementally modifying and combining other independently designed state machines.

When State DP does not apply, because we would like to subclass State hierarchy to handle new events without affecting the existing code.

Solution

Subclass concrete State classes to add more specialized behavior/event handling. To handle these new events, the Context class needs to be updated to dispatch them to the current State object. The State superclass is also modified to provide an empty default implementation of these events.

Instead of returning a programmed constant (or reference to an object), return the next state indirectly, by looking it up in the table called StateMap.

Consequences

- ✦ Can add State subclasses that handle more events (i.e. extend the public interface) without affecting existing State classes, while reusing the default handling of the existing events.
- ✦ StateMap lookup is similar to a virtual table lookup and does not increase the run-time overhead.
- ✉ The Context class delegates all external events to corresponding methods in the interface of the State class. It is responsible for providing the correct mapping. This results in a tight coupling between the Context class and concrete events.

Related Patterns

State DP does not support adding new events to new classes without modifying all classes in the State hierarchy.

State Action Mapper also uses a table, but fails to provide the ability to extend the behavior this pattern provides, because it focuses on limiting the number of classes and uses only a single class to model the state.
State with Exemplars provides a flexible FSM design by dynamically registering Event and Target objects. Subclassing State Machines provides flexibility in the form of new State subclasses that override the default behavior and default transitions in the StateMap class.
FSM Framework does not support the subclassing of State classes, but provides even more flexibility by encapsulating all FSM elements as classes.

Real-Time State Pattern [Dou98, pp. 648]

Applicability
Real-Time system that requires fast performance.
When nested switch statement solution is not flexible enough.
When state machine consists of multiple levels of states. It has many light-weight child-level state transitions, but few more expensive top-level state transitions.
When State DP needs to be extended to support nesting of state classes.

Solution
Context object is the container of concrete state objects. Each concrete state class may have its own state machine for its child states. Each child state has its own algorithm to perform state transitions.

Consequences
- ✦ The separation of states into multiple levels makes each level less complex, which makes transitions at each level less expensive to calculate.
- ✦ If space is more critical than time, concrete state objects may be created when they are transitioned to and deleted when exited. More commonly, they are created and destroyed at the same time as the Context object is.
- ✦ With nested switch statement implementation, if a subclass specializes the state machine, the entire nested switch must be reimplemented. With Real-Time State Pattern, only the modified state must be reimplemented.
- ▭ The Context class delegates all external events to corresponding methods in the interface of the State class. It is responsible for providing the correct mapping. This results in a tight coupling between the Context class and concrete events.

Related Patterns
In State DP, the Context class holds a reference to the current state and does not need to manage other state objects. Real-Time State Pattern keeps one instance of each state in a container so that state changes never result in a creation or deletion of objects.
The Context class acts like a Composite [GHJV95, pp. 163].
Composite State manages multiple active state objects (one per hierarchy), while Real-Time State Pattern's container holds many objects, but only one of them is active (i.e. responds to events) at a given time.

The following is the list of state patterns compiled by Odrowski. All patterns described in that listing are present here. In addition, they are described in relation to other patterns. Each pattern's Solution section specifies which patterns are combined to produce its design.
Note: Since all of the patterns described in this collection are using State DP, they all share the disadvantage that the Context class is tightly coupled with the concrete event handlers.

Composite State [OS96]

Applicability
When multiple state objects should be treated uniformly so that it is possible to execute a single command (group transition) that will cause all of them to change their state.

Solution: State DP + Composite
Use Composite pattern to store all state objects in a uniform structure that provides the same interface to invoke the transition request on composite objects as well as an individual one.

Consequences
- ✦ Both a single state object and a collection of objects provide the same uniform interface for the external entities.
- ✦ When a component rejects the transition request due to an error, tolerate the fault and continue processing.

Related Patterns
State DP and Composite [GHJV95, pp. 163] are combined to implement this pattern. State DP provides the implementation of the state behavior of each object. The Composite pattern provides the uniform interface to build collections of state objects and to accommodate their group transitions with a single method invocation.
Collections for States and Collections of State also provide means of managing lists of state objects, but Composite State can hold objects that are in different states.
If a failed transition request must result in a back out of all changes in all components, use Restoring Previous State instead of this pattern.
Real-Time State Pattern also maintains many state objects at once, but only one of them is active (i.e. responds to events) at a given time.
Hierarchical Statechart uses the Composite pattern to simplify the design. Composite State uses it to support group transitions.

Managing State Dependencies [OS96]

Applicability

When the Composite State is used and there is a complex algorithm to pass an event to multiple objects in the Composite structure.

When change of one object's state can result in changes of other objects and the algorithm to notify them is complex.

Solution: State DP + Observer

State classes communicate their events using a variant of the Observer pattern, Change Manager. ChangeManager class encapsulates the algorithm for notifications between the Context class and State subclasses. Context and State can be both Subjects and Observers, but instead of communicating directly, they send all communication via the ChangeManager object that is responsible for determining how to handle each notification.

Consequences

- ✦ Rules to resolve dependencies between State classes are encapsulated in one class, ChangeManager.
- ✦ ChangeManager class can be subclassed to provide variations on the notification algorithm.
- ▭ Since the Context and State classes cannot invoke methods on each other directly, each such invocation using the ChangeManager object requires an additional level of indirection.

Related Patterns

Managing State Dependencies builds on the Composite State.

ChangeManager [GHJV95, pp. 299] is a variant of the Observer pattern [GHJV95, pp. 293] that decouples Subjects from Observers and allows a more complex update algorithm to be encapsulated in the ChangeManager object.

Broadcasting pattern addresses a similar issue in the context of orthogonal state hierarchies. It solves the problem of disseminating events generated in one state machine hierarchy to another, unrelated hierarchy. Managing State Dependencies models broadcasting of state changes in a single hierarchy where all Contents states collectively can cause a change of their Container's state, or the Container can cause all its Contents to change their states.

As with the Composite State, when failure on state transition requires a complete back out, the Restoring Previous State should be the preferred solution.

Current and Pending State also uses a single object (StateManager) to mediate between the Context class and State subclasses. StateManager encapsulates the knowledge of the current state. The ChangeManager encapsulates the algorithm for disseminating notifications between the Context class and State subclasses.

Decorated State [OS96]

Applicability
When the implementation using State DP would result in a large number of possible state classes, but many of these states differ only in the additional events that they need to handle.
When the ability of a state to handle specific events is added and removed at run-time.

Solution: State DP + Decorator
Encapsulate these events in Decorator classes. At run-time, use Decorators to add and remove events that should be handled at a given point of execution. The Decorator object handles the decorated events and forwards all other events to the State object, which it encapsulates.

Consequences
- ✦ Reduces the number of state classes. Can decorate many classes with one Decorator.
- ▭ A decorator and its component are not identical (see [GHJV95, p. 178]).

Related Patterns
State DP and Decorator [GHJV95, pp. 175] are combined to produce the lowest number of classes that model all states with State and Decorator classes.
Current and Pending State could use Decorated State to model pending states as independent entities.

Orthogonal States [OS96]

Applicability
When State DP is not sufficient, because the state object exhibits several independent behaviors and exercises them at the same time. Each behavior should be encapsulated as a separate state hierarchy.

Solution: State DP^n
Define a new state class that encapsulates all other states. Use State DP to implement the outer state as well as all the states it encapsulates. State DP can be used recursively for problems where state can vary within state.

Consequences
- ✦ The uniform high-level design is easy to understand.
- ▭ Low-level design is more complex. There are many dependencies between state classes not addressed in this pattern.

Related Patterns
State DP is used to model each individual orthogonal state.
Orthogonal Behavior presents an alternative implementation of the solution. It uses Composite [GHJV95, pp. 163] to represent each hierarchy of states.

State with Exemplars can also handles many states at once. It forwards each incoming event to the current state object of all Targets registered with the FSM container.

Current and Pending State [OS96]

Applicability
When current object's state is not the only indicator of the actual state (because the object has also pending states).
When State DP needs to be extended to provide a list of states that may be entered in the future (pending) and to log the list of states previously entered.

Solution: State DP + Mediator
Use a StateManager class as a mediator between the Context and the State hierarchy. The Context still delegates all requests to the current State object, but indirectly, because it knows only about the StateManager object. The StateManager object is responsible for delegating the request to the appropriate state object.

Consequences
- ✦ The current state can be a function of an arbitrarily large collection of states.
- ▭ Since the Context and State classes cannot invoke methods on each other directly, each such invocation using the StateManager object requires an additional level of indirection.

Related Patterns
State DP provides the generic Context and State hierarchy framework.
Mediator [GHJV95, pp. 273] pattern is the basis for the State Manager class.
Collections of State and Collection for States focus on all objects in a specific state. In contrast, the Current and Pending State focuses on all (past, current, and pending) states of an object.
Alternatively, pending states could be modeled as independent entities that generate an event to the current state at the time of their "maturity." This would force potentially all states to be able to handle these events. Use the Decorated State pattern to solve this problem.
Restoring Previous State provides the ability to back out state transitions, rather than just store them.
Managing State Dependencies also uses a single object (ChangeManager) to mediate between the Context class and State subclasses. The ChangeManager encapsulates the algorithm for disseminating notifications between the Context class and State subclasses. StateManager encapsulates the knowledge of the current state.

Collections of State [OS96]

Applicability
When State DP is not applicable, because it cannot provide the location information about all objects in a specific state.
When the design relies on collections of objects in a specific state and the states of individual objects are of lesser importance.
When state objects maintain additional object-dependent data.

Solution
Collect all objects that have the same state in a single container class. Provide a separate container for each state. When an object changes its state, it also moves to the container for the new state.

Consequences
- ✦ It is easy to find all objects in a specific state.
- ✦ All objects still retain the entire interface provided by State DP.
- ✦ Containers for specific states can be created dynamically when an object enters a specific state. They can be destroyed when the last object exits that state.
- ✦ Requests to find all objects in multiple states require the concatenation of multiple containers.
- ▭ Deletion rules must be enforced so that an object that is deleted (or changes its state) informs the container of the change.

Related Patterns
State DP provides the generic state functionality that is augmented in this pattern by the ability to hold collections of objects in the same state.
If objects in the collection share common transitions, not common states, use Composite State to facilitate group transitions.
If objects do not contain any object-dependent data (i.e. pure states), Collections for States is a better alternative. The overhead of State DP (additional classes and indirection) is avoided in this case.
Use Observer [GHJV95, pp. 293] to enforce correct propagation of deletion information.

Restoring Previous State [OS96]

Applicability
When it is necessary to support the ability to revert to previous state (i.e. undo), which is not a part of the State DP.

Solution: State DP + Memento
On each state change, store current state and (optionally) state-specific data in a separate object. To perform an undo operation, revert to the old data stored in that object.

Consequences

- ✦ Provides backup and back out capabilities for state transitions.
- ▭ Storing backup copies creates a memory overhead, especially if large amount of object-specific data needs to be stored on each transition.

Related Patterns

Builds on Composite State and serves as its better alternative to handle complete back out of a failed group transition.
State DP and Memento [GHJV95, pp. 283] are combined to allow revertible state changes. The Memento pattern is applied to basic State DP objects (called Originators in the context of Memento pattern).
History State provides a means of remembering the previous state on subsequent entrances to it, but does not provide undo capabilities.
Current and Pending State provides a simpler alternative in cases when previous states are to be logged rather than undoable.

The following is the list of statechart patterns compiled by Yacoub. His work is based on the concept of statecharts developed by Harel [Har87]. All patterns described in Yacoub's pattern language are present here. They are also described in relation to other patterns.
Note: The Interface class (i.e. the Context) delegates all external events to corresponding methods in the interface of the AState class. It is responsible for providing the correct mapping. This results in a tight coupling between the Interface class and concrete events. The benefit of this approach is that the external entities need to be only aware of the Context class while other classes are completely separated from the outside world.

Basic Statechart [YA89b]

Applicability

When State DP needs to be extended to model statechart elements, e.g. exit and entry events, guards.

Solution

Encapsulate Events and Actions in separate classes. They are superclasses of the AState interface. Methods inherited from Events class are virtual and can be overridden as needed, while methods in the Actions class are static. They define interfaces for events and actions and provide default handling of events. Concrete State subclasses may override the default event and action handling.

Consequences

- ✦ Statechart elements are modeled explicitly to simplify the maintenance and increase the readability of the design.

Related Patterns

State DP does not implement all FSM elements explicitly.

FSM Framework implements all FSM elements as classes, which makes it more flexible then the Basic Statechart.

Hierarchical Statechart [YA89b]

Applicability
Large applications that use the Basic Statechart and their states have a hierarchical class structure.

Solution
Differentiate between high-level states (TopSuperState class hierarchy), intermediate states (IntermediateSuperState class hierarchy), and leaf staes (Leaf class hierarchy). Have all hierarchies contain references to all states they need to be aware of (e.g. their immediate superstate). Classify all state classes into these three hierarchies.

Consequences
- ✦ Improves understanding and readability of the design. Hierarchies of states are shown explicitly.
- ✦ State classes that do not display a hierarchical relationship with other classes remain subclasses of AState and need not be forced into this design.

Related Patterns
Basic Statechart is extended to support hierarchical structure of relationships between states.
SuperState classes are Composites [GHJV95, pp. 163].
Composite State uses the Composite pattern to support group transitions. Hierarchical Statechart uses it to simplify the design.

Orthogonal Behavior [YA89b]

Applicability
Entity modeled by the Hierarchical Statechart that exhibits several independent behaviors and exercises them at the same time.

Solution
Define a VirtualSuperstate class that is a composite of all superstate states that model a specific behavior. VirtualSuperstate will dispatch events to all the components it holds.

Consequences
- ✦ Separate state diagrams model each independent behavior.
- ✦ Dispatching the event to all independent states can be implemented either sequentially or concurrently, depending on the operating system and platform support.

- The design separates orthogonal behaviors, but in reality they model one entity and may need to communicate. Broadcasting pattern solves this specific problem.

Related Patterns

Virtual Superstate is a Composite [GHJV95, pp. 163].

Hierarchical Statechart uses the same technique (Composite pattern) to model state hierarchies. Orthogonal Behavior builds on class hierarchies created using Hierarchical Statechart and combines them (using the Composite pattern) to provide concurrency (i.e. many active states, one in each hierarchy).

Orthogonal States uses State DP instead of Composite to implement hierarchies of states. In both cases, two respective patterns are applied twice. Orthogonal States uses State DP to model the hierarchy of the container state (called VirtualSuperstate in Orthogonal Behavior). It also uses State DP to model hierarchies of concurrent (i.e. orthogonal) states. Orthogonal Behavior uses the Composite pattern to model both steps.

State with Exemplars can also handles many states at once. It forwards each incoming event to the current state object of all Targets registered with the FSM container.

Broadcasting [YA89b]

Applicability

Design uses the Hierarchical Statechart with Orthogonal Behavior where events occurring in one state trigger other events in orthogonal superstates.

When orthogonal behaviors of an entity, modeled as separate hierarchies, need to communicate.

Solution

To process any event, invoke it on the entity Interface (i.e. the Context class). The interface forwards the event to the VirtualSuperstate, which forwards it to the concrete SuperStates.

Consequences

- ✦ Internal and external events are handled uniformly.
- Inefficient; multiple indirections are required to handle an event.

Related Patterns

Broadcasting solves the main drawback of Orthogonal Behavior – lack of communication between orthogonal states.

Managing State Dependencies encapsulates relationships between FSMs in a single object (ChangeManager). Broadcasting distributes the generation of events causing other classes to respond among multiple classes. The solution in Managing State Dependencies is more extensible. Consider for example class A that generates event E_1 upon the reception of event E_2. If another class B expects that event as E_3, one of these two classes needs to be changed to solve the problem. In Managing State Dependencies, the responsibility to change event E_2 to E_3 before forwarding it to class B is encapsulated in the ChangeManager class.

History State [YA89b]

Applicability
When the Hierarchical Chart is used and subsequent re-entries into a state should not result in resetting of the inner state to the default state.
When a superstate should have a memory of which active state it was in last just before exiting the whole superstate.

Solution
Initialize the current state reference of a Superstate class once on creation and do not reinitialize it on each invocation of the entry() method. This will keep the value of the Superstate class the same as it was on the last execution of the exit() method.

Consequences
- ✦ Preserves the knowledge of the innermost current state prior to a global transition.
- ▭ Loses the flexibility of re-initializing data on state entry.

Related Patterns
Current and Pending State and Restoring Previous State are somewhat similar to History State. Current and Pending State provides means of logging previous states, but without the ability to undo them. Restoring Previous State provides a design that allows a back out of state transitions.

III. ALTERNATIVES TO STATE DP
This section describes patterns that produce FSM designs different from State DP. Some of them are applicable to small designs, many apply to specific domains (such as real-time systems), but this fact does not make them less important than other FSM designs described above. Each pattern is also compared to other patterns that address its main concern(s) using different solutions, in the Related Patterns section.

Collections for States [Hen00]

Applicability
When State DP results in too much overhead, because Pure State objects need to be managed as collections of objects (that share the same state), rather than as individual objects.

Solution
Represent each state by a separate collection that knows about all objects in that state. The Manager object holds all the state collections and is responsible for managing the lifecycle (i.e. state changes) of the objects. When an object changes its state, the manager moves it from the collection of objects in the source state to the collection of objects in the target state.

Consequences

- ✦ Less time required to access all objects that meet specific criteria with each criterion modeled as a state.
- ✦ The size of each object is decreased by not storing its current state explicitly in the object, but rather implicitly in the collection.
- ✦ The Manager object is the only design element that requires changes when the behavior changes.
- ✦ Can support many collections concurrently.
- ✦ Orthogonal state models can be modeled with multiple managers and no updates of state objects are required).
- Cannot easily manipulate other state-independent data in state objects.
- Not feasible when state changes occur frequently.

Related Patterns

The Collections for States is an alternative to the State DP in the cases when Pure State objects are analyzed or processed as groups.
When objects can initiate state changes, use the Observer pattern [GHJV95, pp. 293] to keep the Manager up to date.
When objects contain state-independent data, they cannot be stored using Collections for States. Use Collections of State instead. Note that Collections of State uses all the features of State DP and consequently its implementation incurs a similar overhead.
To hold collections of objects that are in different states, use Composite State.

Methods for States [Hen03]

Applicability

When the solution using State DP is too complex to model a simple object with few states that could otherwise be implemented with few conditional checks and few methods.

Solution [Hen03]

Encode method calls in a table within the class. Have appropriate interface methods invoke the methods in the table. Upon each state change, set the next entry in the table to be the method that represents to the new state.

Consequences

- ✦ Implementation is much simpler than State DP.
- ✦ Requires less code and only one class.
- This design is not extensible.

Related Patterns

State DP becomes a viable alternative again when it becomes necessary to add more states to the object implemented using Methods for States.
State Action Mapper also uses a transition table, but it provides a run-time registration and deregistration of events/actions in the table.

Optimal FSM also models states as methods and its design is more applicable to larger systems.

State Table Pattern [Dou98, pp. 650]

Applicability
When state machine contains a large number of states and transitions with minimal state-independent data.
When State DP does not apply, because its storage overhead cannot be tolerated in real-time and other embedded systems that have a limited amount of memory.
Safety-critical and other systems that define state transitions in a tabular form.

Solution
States and Transitions are modeled as classes. Context contains a State Table instance that provides references/callbacks to concrete State and Transition objects. State Table encapsulates the transition table of size *num_states* * *num_transitions* that can be accessed in constant time.
The Context object sends an external event, encapsulated as a constant, to the Transition class that returns the resulting state. Next, the Context delegates the processing of the event to that state object.

Consequences
- ✦ Good performance, O(c).
- ▭ State table is expensive to construct and difficult to maintain.
- ▭ High initialization overhead (need to initialize a large table). This problem can be avoided by using compile-time constructs (e.g. structs in C++) as elements of the table.

Related Patterns
State DP provides a similar solution, but is not concerned with minimizing the memory usage or optimizing performance (since it uses virtual functions).

Optimal FSM [Sam02, pp. 69]

Applicability
When State DP implementation is too slow, because of its use of indirections.
Embedded real-time systems.

Solution
Encapsulate states as methods. Have the Context class store the current state as the pointer to the state method. Subclass the Context class to provide the concrete implementation of the state methods.

Consequences

- ✦ It is simple to implement.
- ✦ State specific behavior is encapsulated in specific state methods.
- ✦ Since state methods are implemented in subclasses of the Context class, there is no encapsulation problem. State methods have direct access to all data in the Context class.
- ✦ Small memory footprint – only one function pointer is required to implement the state machine.
- ✦ Promotes code reuse (can subclass Context further, but without virtual functions).
- ✦ Makes state transitions efficient – one assignment of state pointer.
- ✦ Good performance – O(log n), where n is the number of cases in the switch statement. However, the switch can be replaced by look-up table in selected (critical) state handlers to provide even faster performance.
- ⌨ The design is scalable and flexible during the original development, but not during maintenance.
- ⌨ The code is not easy to follow, because it is located in one large class.

Related Patterns

State DP models states as classes rather than methods.
Methods for States also uses a pointer to state methods, but they are private methods.

State Action Mapper [Pal97]

Applicability

When creating a class per state is too expensive (e.g. many small and similar states).
When flexibility to manage (add/remove) events and their corresponding actions at run-time is more important than modeling states as independent classes.

Solution

Instead of a hierarchy of State classes, use one concrete State Mapper class that maintains a current event-action table and delegates the execution of the concrete behavior to the State-Dependent Object (i.e. the Context class). On a state change, replace the state object reference to another pre-allocated instance of a State Mapper object that contains the transition table for that state.

Consequences

- ✦ An instance of State Mapper object can be shared by multiple State-Dependent Objects.
- ✦ Code sharing – since State Mapper class contains the mapping algorithm it is a good candidate to implement other general policies.
- ✦ Maintains encapsulation of the State-Dependent Object.
- ✦ Avoids multiple if and switch statements.
- ✦ Behavior of a given state can be easily modified.
- ⌨ Increased complexity and an additional level of indirection – State Mapper class delegates requests to the State-Dependent Object.

Related Patterns

There are three differences between State Action Mapper and State DP. First, the State Mapper class allows the dynamic registration of event/action pairs during run-time. State DP does not describe that option, but it would be possible to add it. Second, the State Action Mapper pattern uses one State Mapper class as opposed to the State class hierarchy in State DP. Third, the State Mapper class delegates the execution, and the implementation, back to the State-Dependent Object (i.e. the Context class).

State Action Mapper pattern extends the State Table pattern, which contains a state transition table that cannot be modified at run-time.

Methods for States also uses a transition table, but it is not modifiable at run-time.

State with Exemplars also provides dynamic registration of events. In addition, it supports dynamic registration of Targets, which allows it to support multiple objects with states at the same time.

Subclassing State Machines encodes the transition table in the StateMap class hierarchy. It also allows the modification of behavior by implementing a parallel hierarchy of State classes. Since State Action Mapper has only one state class, it cannot provide this flexibility.

RELATED WORK

One well-known work on Object-Oriented state machines not included in the catalogue above is MOODS [Ran95], which presents an alternative technique of selecting the most optimal design among different state machine patterns, using DDT (design decision tree). The MOODS paper focuses primarily on generic problems (e.g. complex object behavior, events cause state changes) that are prerequisites of State DP. Thus only a small subset of MOODS could be applicable to this paper.

Other related work is analyzed in this paper. To date, only Yacoub [YA98a, YA98b] has attempted to combine multiple state machine design patterns into a cohesive unit. This work is a further extension of these efforts.

CONCLUSION / FUTURE WORK

Design patterns summarized in this paper represent a wide range of known FSM designs. Each one is an optimal solution in a specific design context. Collectively, they address a wide range of problems, but no pattern can solve all the problems. In addition to their applicability and consequences of their use, all patterns are analyzed specifically with regard to the coupling between the class that handles requests from the outside world (e.g. the Context class in State DP) and the entity that provides the concrete response (e.g. event-handling methods in State DP). Patterns that implement loose coupling between these two elements are more complex and require more levels of indirection than their less extensible alternatives, but they produce a cleaner and more flexible design. Patterns described in this paper can be grouped in three types of solutions:

1. Modeling FSM elements as classes.
2. Modeling interactions between different FSM classes.
3. Reusing and extending FSM behavior by subclassing.

FSM Framework [vGB99] is the most extensible example of the first type. It models all FSM elements as classes. But it does not provide a good model to handle multiple FSMs. The Reflective State Pattern [FR98] provides a flexibility of changing FSM structure at run-time using reflection, but it does not address the inter-FSM communication either. State with Exemplars [Ack95], which gives lesser flexibility in FSM design (only events and states are classes) does provide an effective means of communication between FSMs.

Two patterns that describe the most effective design of communication between multiple, different, and possibly nested FSMs are Managing State Dependencies [OS96] and Broadcasting [YH99b]. Neither is as flexible in modeling FSM elements as patterns that encapsulate FSM elements as classes, but their communication model is very good.

The third type, represented by Subclassing State Machines [SC95], provides a method of adding new State subclasses without changing the existing code. In this respect, this pattern is unique among all patterns described in this paper. It remains to be seen to what extent these three types of solutions can be combined into a single design.

This paper shows how different design patterns solve different problems given a specific context and a set of expectations (e.g. flexibility of design, loose coupling between elements, performance, etc). The uniform format of presentation aims to help software designers select the FSM most appropriate for their needs.

Although the listing described here is significantly more comprehensive than the individual papers is draws upon, it does not provide a complete comparison. Diagram "Relationships Between Patterns" shows very clearly that relationships between state patterns other than State DP have not been explored sufficiently yet. Except for "The Pattern Language of Statecharts" [YA98a], design patterns show at most two levels of dependencies. The author's future goal is to perform a complete comparison of all design patterns described here using a single set of terminology, modeling technique, programming language, and a common example. The example should evolve from a simple single-class design into a fully extensible design.

ACKNOWLEDGEMENTS

The author wishes to thank Dr. Ralph Johnson for his guidance during the creation of this paper. Many thanks go also to Sherif Yacoub for his review of the initial version.

A big 'Thank you' that requires a separate paragraph goes to my shepherd, Joel Jones, for his timely and precise feedback that was on target in both big and small matters.

BIBLIOGRAPHY

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

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

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

[Cope92] Coplien, J., “Advanced C++, Programming Styles and Idioms,” Addison Wesley, 1992.

[DA96] Dyson, P. and B. Anderson, "State patterns," PLoPD3. Also available at: http://www.cs.wustl.edu/~schmidt/europlop-96/papers/paper29.ps.

[Dou98] Douglas, B. P., "Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns," Addison Wesley, 1998.

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

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

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

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

[Hen03] Henney. K., “Methods for States,” VikingPLoP 2002, updated March, 2003. Also available at: http://www.two-sdg.demon.co.uk/curbralan/papers/MethodsForStates.pdf.

[Mar95] Martin, R., "Three Level FSM," PLoPD, 1995. Also available at: http://cpptips.hyperformix.com/cpptips/fsm5.

[Mar02] Martin, R., “UML for Java Programmers,” Prentice Hall 2002. Also available at: http://www.objectmentor.com/UMLFJP/UMLFJP.

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

[Pal97] Palfinger, G., "State Action Mapper," PLoP 1997, Writer’s Workshop. Also available at: jerry.cs.uiuc.edu/~plop/plop97/Proceedings.ps/palfinger.ps.

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

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

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

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

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

[YA98b] Yacoub, S. and H. Ammar, "Finite State Machine Patterns," EuroPLoP 1998. Also available at: http://www.coldewey.com/europlop98/Program/Papers/Yacoub.ps.