State Action Mapper

Günther Palfinger (gp@sdm.de)
sd&m GmbH Co. KG
Thomas-Dehler-Straße 27
81737 Munich, Germany

Abstract

This Pattern is an extension of the well-known *State* pattern [Gam+95], which allows an object to change its behavior depending on the internal state of the object. In *State Action Mapper* the behavior is defined by events, whose transformation to actions depends on the object state. This pattern introduces a way to manage state actions.

Example

In applications with graphical user interfaces (GUI), it's very common that the appearance and behavior of dialog windows change during their lifetime. Dialog windows have various states. Each state defines the appearance and behavior of the dialog window.

For example, some of the edit fields and action buttons may be disabled (i.e. grayed out) in a particular dialog state. Changing the dialog state (e.g. by selecting an element) will activate these edit fields and action buttons.

To illustrate this, consider the customer dialog shown in Figure 1.

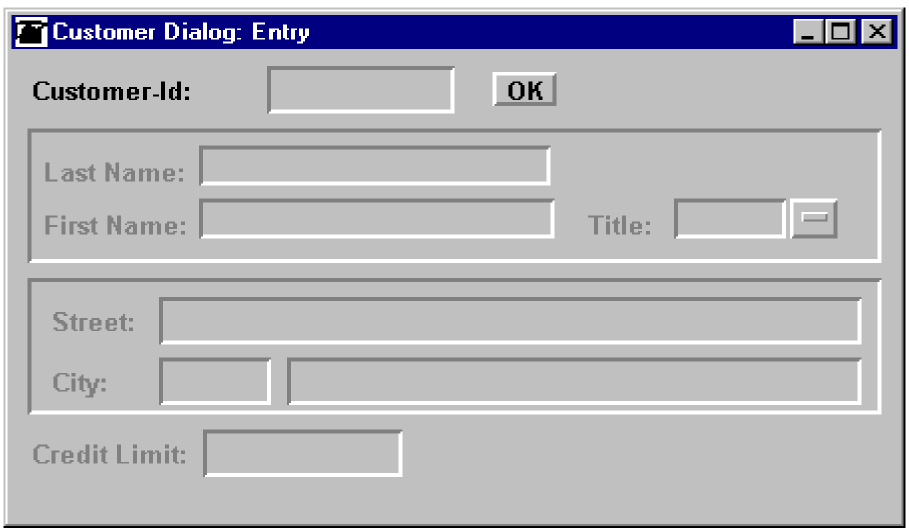

Figure 1: Customer dialog; current state is "Entry"

The dialog has three states as summarized in Table 1.

State	**State of Customer Fields** (other than Customer Id)	**Meaning of OK Button**	**Transitions**
Entry	Disabled	Insert a new customer or update an existing customer.	User enters a new Customer Id, and hits OK; go to *Insert* state. User enters an existing Customer Id. and hits OK; go to *Update* state.
Insert	Enabled	Insert a new customer record into the database.	User hits OK; go to *Entry* state.
Update	Enabled	Update the customer's record in the database.	User hits OK; go to *Entry* state.

Table 1: State Table for Customer Dialog

Context

An object needs to have different behavior during its lifetime. The behavior can be described in the form of events, whose transformation to actions depends on the object's state. It may be necessary extend the object's behavior by adding new event/action pairs during runtime.

Problem

How do you implement an object that changes its behavior depending on its internal state, while allowing new behavior to be added in a simple and flexible way?

Forces

The following forces are focused by the GOF *State* pattern:

- Implementing state-specific behavior in state-specific classes may make the resulting software easier to understand by logically grouping action code according to states (i.e. all code for a given state is grouped together within one class). However implementing state-specific behavior in state-specific classes may increase coding effort due to the potential need to define many state classes. Furthermore creating an instance for each state may waste a lot of memory-space.
- Implementation of actions within auxiliary (i.e. state) classes, may result in breaking encapsulation of the state-dependent object (or the need for a special provision, such as the "friend class" mechanism in C++) in order to allow state class methods to access the state-dependent object's instance variables. On the other hand the implementation of state-specific behavior using "if" or "switch" statements may result in code which is more difficult to maintain and modify. For example, adding a new state or action may require searching through the code for multiple places where modifications are required.

The following are additional forces focused by *State Action Mapper*:

- Hard-coded approaches (i.e. state classes, "if" statements, or "switch" statements) are not able to support situations where mappings between events and actions for a given state need to be modified dynamically at runtime.
- The State-class approach, as discussed in the GOF *State* pattern, introduces a dependency to a particular abstract *State* class (providing a common method-interface for all subclasses). Hence, if you extend the behavior by adding a new method in an existing or new subclass, you have to change the abstract baseclass and recompile all the code. Further, each state class implements certain state-dependent actions (hard-coded), which can't be modified dynamically at runtime.

Solution

For each state, define a *State Mapper object* that maps events to actions. Allow *State Mapper* objects to delegate execution of actions back to the state-dependent object. *State Mapper* doesn't hold an instance variable to the state-dependent object; consequently a *State Mapper object* my be shared between different dependent objects (given that they have identical behavior). This may be achieved by making the dependent object an argument of the mapping method of *State Mapper*.

The dispatching code (i.e. executing the action mapped to the event) can be factored into the *State Mapper* class as in a table-driven approach. Each *State Mapper* has a list of event/action pairs in the form of a data structure (i.e. a dictionary of events and actions). This allows state action pairs to be added/modified/deleted at runtime.

Structure

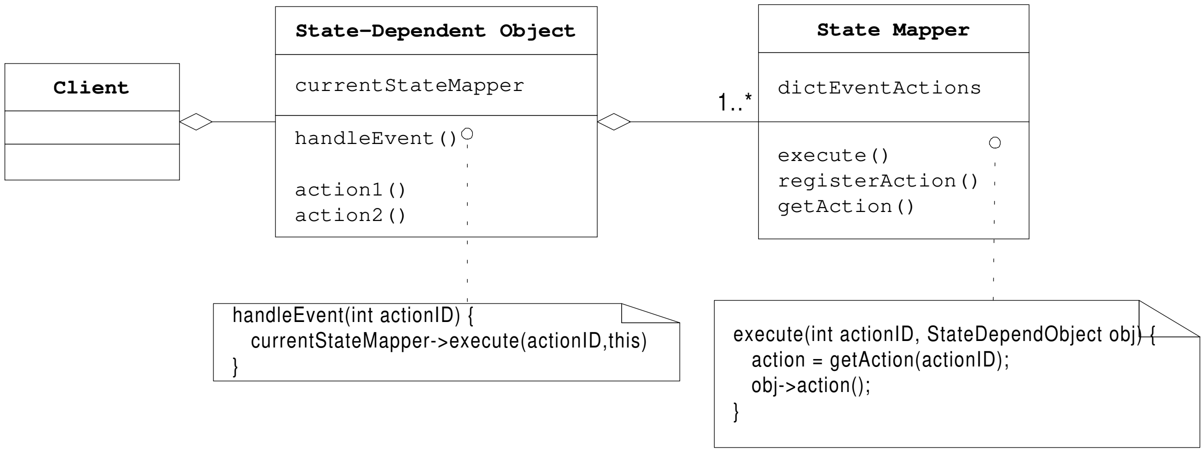

Figure 2: Structure of the State Action Mapper

Participants

- **Client**
  - uses the interface of *State-Dependent Object* for event handling
- **State-Dependent Object**
  - defines the interface for event handling
  - holds an instance of the current *State Mapper* and defines an interface to change it
  - implements the actions that are registered in the *State Mapper*
- **State Mapper**
  - holds the mapping of events (as key) and actions (as value) defined in *State-Dependent Object*
  - defines an interface for registering and finding the actions for given events
  - delegates execution of actions to the *State-Dependent Object*

Collaborations

Figure 3 shows the dynamic collaboration of *State-Dependent Object* and *State Mapper*: *State-Dependent Object* handles a client event and directs a request to its current *State Mapper* object, which evaluates the action and delegates the execution back to the *State-Dependent Object.*

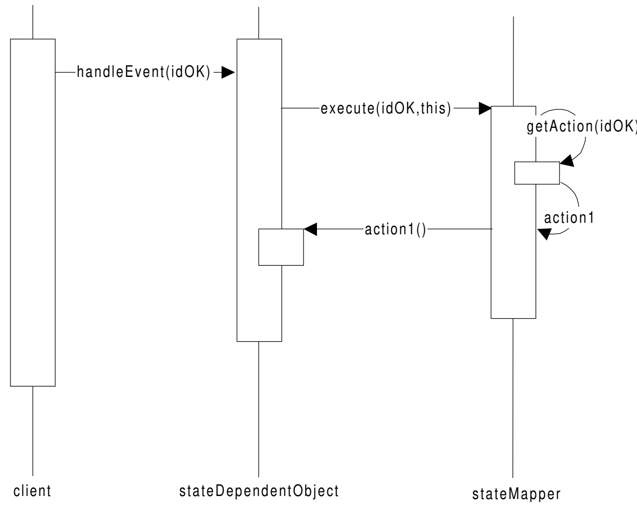

Figure 3: Dynamic behavior of State Action Mapper

Consequences

- *This approach uses a single class State Mapper* with an instance for each state as opposed to the State-pattern which requires a class for each state. This is because *State Mapper* uses a data-driven approach for action mapping instead of hard-coding the actions in state-classes methods. Using an instance for each state may waste a lot of memory-space. To overcome this problem you may share *State Mapper objects* as discussed in the next paragraph.
- *Instance Sharing of State Mapper objects*: Supposing that multiple *State-Dependent Objects* have identical behavior, it is possible to share the common state mapping behavior between them. This is because *State Mapper* doesn't hold an instance variable to its *State-Dependent Object*. For this purpose you may implement shared *State Mappers* as *Singletons*.
- *Code Sharing:* The pattern locates the mapping of events to actions in class *State Mapper*. As a consequence, it is easy to place some general policies in this class. For example you might trace the execution of actions (in a log-file).
- *Maintaining encapsulation*: This approach doesn't break the encapsulation of the state-dependent object, because the implementation of actions is not separated into auxiliary-classes as opposed to the *State*-pattern which uses state classes for each action.
- *Avoiding multiple "if" and "switch" statements*: The introduction of *State Mapper* object avoids the if-cascades or switch-statements for state checking code in *State-Dependent Object*. This will be achieved by simply substituting the state-dependent code with a method call to a *State Mapper object*.
- *Flexible approach for adding or modifying behavior for a given state*: Due to the table-driven nature of this approach, you may add new behavior by registering event/action pairs for a given state at runtime.
- *Increased Complexity*: Introducing class *State Mapper* as additional indirection increases complexity, because you have to look at two classes in order to understand *State-Dependent Object's* behavior.

Example resolved

Figure 4 shows an excerpt of the dialog control from the application framework *Frammento* [Alb96].

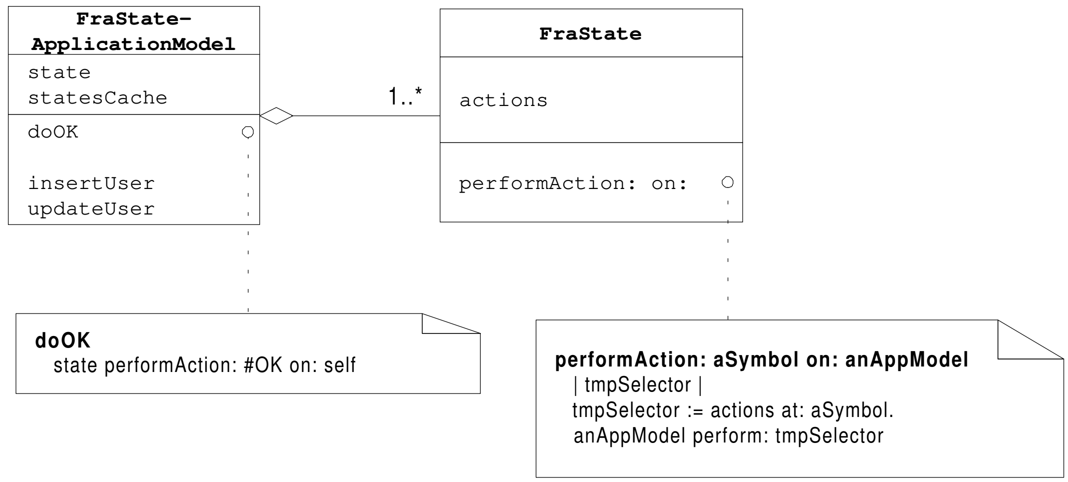

Figure 4: Dialog control of Frammento

`FraStateApplicationModel` defines the event `doOK` and implements the actions `insertUser` and `updateUser`. These actions are placed for brevity in this class. Usually events and actions are defined in a concrete subclass of this class.

For mapping the events and actions, `FraState` uses an `IdentityDictionary` (actions).

The method `performAction:on:` first evaluates the action and delegates the execution back to `FraStateApplicationModel`.

Implementation

Dynamic registering of actions against events is a central point of this approach. This can be done using unary method selectors in Smalltalk or method pointers in C++. In Java, you can use *Action* objects or *Command* objects as in [Gam+95]. This is achieved by simply encapsulating the method-invocation in a class.

```
public abstract class Action {//abstract baseclasse for actionobjects
    public abstract void doIt(ActionContext ac);//abstact method-call
}
public class Action1 extends Action {//concrete action1-object
    public void doIt(ActionContext ac) {
        ac.action1(); //invoke the action
    }
}
public class StateMapper {
    ...
    public void register(int id, Action act) {
        actions[id] = act;
    }
    public void execute(int id, ActionContext ac) {
        actions[id].doIt(ac);
    }
}

stateMapper.register(idOK, new Action1()); //register Aktionsobj.
```

The disadvantage is the increased number of classes because each action requires its own class.

Variants

- *Action filter*: You can create subclasses of *State Mapper* which provide a filter-interface where you can control filtering of events and actions. For example, depending on a special event (system-event, concurrent user-access, security-restriction, etc.) the filtering *State Mapper* object can forbid the execution of some registered actions (i.e. actions with write-access). This may be achieved by adding additional tables for event/disable-action and event/enable-action pairs.
- *Simplified variant using nested dictionaries*: This variant uses one dictionary for all states (mapping from state to an action dictionary) and one action dictionary for each state (mapping from event to action). These dictionaries can be implemented as instance variables in the State-Dependent Object.

  But there are also some drawbacks: You may share the dictionaries (as data) between different objects, but you are not able to share common code as provided by a *State Mapper* class. For instance see *Code Sharing* in the *consequences* section. Nested dictionaries may also be harder to maintain, because they are less explicit than a state mapping class and sometimes harder to debug in some C++ environments on UNIX (especially if you use an older gnu debugger which doesn't let you evaluate instance variables of a complex type; or if you don't have the debug version of the class library containing the dictionary; in those cases programmers usually wrap these containers and facilitate them with tracing and debug support).

  Nevertheless this may be a good choice for simple action models. (This was the original implementation of *Frammento's* dialog control).
- *An even simpler variant*: Use a single dictionary with a two-part key (i.e. map state/event to action). In Smalltalk the two-part key can be implemented using the *Association* class, or you may want to create your own key class in the interests of clarity.

Related patterns

State Action Mapper is an extension of the *State* pattern [Gam+95]. Class *State Mapper* plays the role of *State* in the original pattern. Essentially there are three differences:

1. *State Mapper* allows the dynamic registration of event/action pairs during runtime. This is because of the data-driven approach for action mapping instead of hard-coding the actions in methods of state classes as opposed to the *State* pattern.
2. This pattern uses a single class *State Mapper* as opposed to the original *State* pattern which requires a class for each state. Furthermore it's possible to share a *State Mapper* instance between multiple state-dependent objects, if they have identical behavior.
3. *State Mapper* delegates the execution, and hence the implementation, back to the *State-Dependent Object*.

A good choice for implementing a state transition strategy is a combination with Dyson's state transition patterns described in *State Patterns Language* [Dys+96]. The *State-Driven Transitions* pattern deals with state objects (in our pattern the *State Mapper objects*) being responsible for the transition from itself (the current state) to the new state object. By using this pattern, you are not able to share *State Mapper objects*, if you have different state-transition profiles of the state-depending objects.

The *Owner-Driven Transitions* pattern deals with an alternative approach where the owning object (in our pattern the *State-Depending Objects*) implements the finite state machine. This pattern enables you to share the *State Mapper objects* in case you have different state-transition profiles.

Alexander Ran offers in his event-centered Architecture pattern MOOD [Ran95] a very good classification of *Event* patterns. The design decision "*For State Machine use Transition Methods*" gives some hints, on how to implement the finite state machine in conjunction with the previously discussed *Owner-Driven Transitions* pattern.

Known uses

The application framework *Frammento* [Alb96] uses this pattern to implement the dialog control. The pattern solves two aspects of the dialog control: the one already mentioned here is control of *dialog behavior* and the second is the *control of appearance* (enabling and disabling of widgets).

The framework is actually used in two projects of the German railway company [DaRT] (database for locomotives and wagons) and TIV (locomotive maintenance and servicing). There are plans to use this pattern for an upcoming enterprise Java Framework.

Acknowledgments

This pattern was mined from the application framework *Frammento* written by Gerhard Albers. He reviewed the German and the English versions of the pattern and improved content and style with his profound know how.

My shepherd Jim Doble, Jens Coldewey and the "OO Techniques" workshop group at the PLoP'97 (Bobby Woolf, James Noble, Ian Chai, Fernando D. Lyardet, James Newkirk, F. Balaguer, and G. Dombiak) gave me valuable hints to improve the structure and the content of the paper.

References

[Alb96] **Gerhard Albers:** *Frammento Handbuch;* sd&m, Munich 1996.

[DaRT] **Project DaRT:** *Datenbank für Reisezugwagen und Triebfahrzeuge,* sd&m; information at http://www.sdm.de/e/www/aktuelles/

[Dys+96] **Paul Dyson, Bruce Anderson:** *State Patterns;* Proceedings of EuroPLoP '96.

[Gam+95] **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:** *Design Patterns, Elements of Reusable Object-oriented Software*; Addison-Wesley, 1995.

[Ran95] **Alexander Ran:** MOODS: Models for Object-Oriented Design of State; Proceedings of PLoP '95.