Crossroads
The ACM Student Magazine

Modeling Object States and Behaviors Using a State Action Manager

by *Gunther Palfinger*

Introduction

State dependent behavior is human nature. (For example, if it is cold, John uses a hat; if it is very cold, John uses gloves too. But does Jim behave like John at the same temperatures?) In computer science this is a common pattern as well. In functional and procedural programming languages, it often leads to inflexible and uncomfortable `if-then-else` constructs. In object-oriented programming languages, like Java and C#, one can implement this pattern close to human abstraction level. In our example, this corresponds to giving Jim and John different state dependent behavior. (Jim is from Florida, while John is from Alaska, hence they have different chill levels, and Jim does not even own a pair of gloves.)

The layout of this paper follows the structure of design pattern descriptions, with the following sections: Abstract, Example, Context (with Problem and Forces), Solution (with Consequences, Structure, Participants, Collaboration, Example Solved, and Implementation),Variants, Related Patterns, Known Uses, and References.

Abstract

This design pattern is an extension of the well-known state pattern [2], which allows an object to change its behavior depending on the internal state of the object. 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, it is common that the appearance and behavior of dialog windows change during their lifetime. Dialog windows define various states, and at any time, the dialog window is in a certain state that defines the appearance and behavior of the window.

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

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

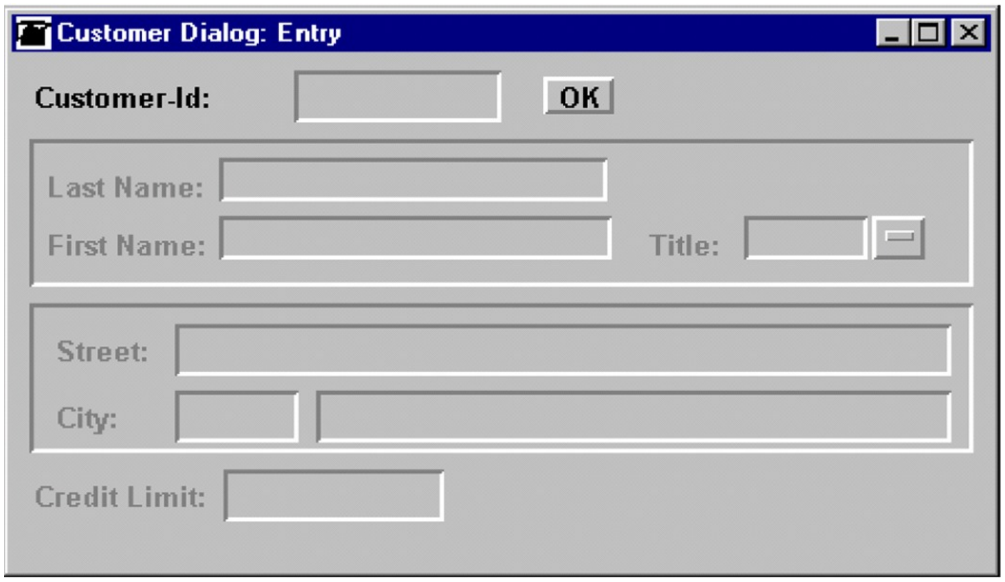


Figure 1. Customer dialog in the *Entry* state.

This dialog has three states as summarized in Table 1.

Table 1. The states of the customer dialog.

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.

Context

The behavior of an object can be described in the form of events, whose transformation into action depends on the object's state. In some cases it is necessary to 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

Forces acting on implementation aspects include the following:

- 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).
- 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 memory space.
- Implementation of actions within auxiliary (i.e., state) classes may result in breaking the 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.
- Implementing 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.

Forces affecting design aspects include the following:

- 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 state pattern in [**2**], 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 base class and recompile all the code. Further, each state class implements certain state dependent (hard-coded) actions, which cannot be modified while the program is running.

Solution

For each state, define a **state manager** object that maps events to actions. Allow state manager objects to delegate execution of actions back to the state dependent object. The state manager does not need to hold an instance variable to the state dependent object; consequently a state manager object may be shared between different dependent objects (given that they have identical behavior). This is achieved by providing the dependent object as an argument when invoking the mapping method of the state manager.

The dispatching code (i.e., executing the action mapped to the event) can be factored in to the state manager class using a table-driven approach. Each state manager has a list of event/action pairs in an appropriate data structure (e.g., a dictionary of events and actions). This allows state action pairs to be added, modified, and deleted at runtime.

Consequences

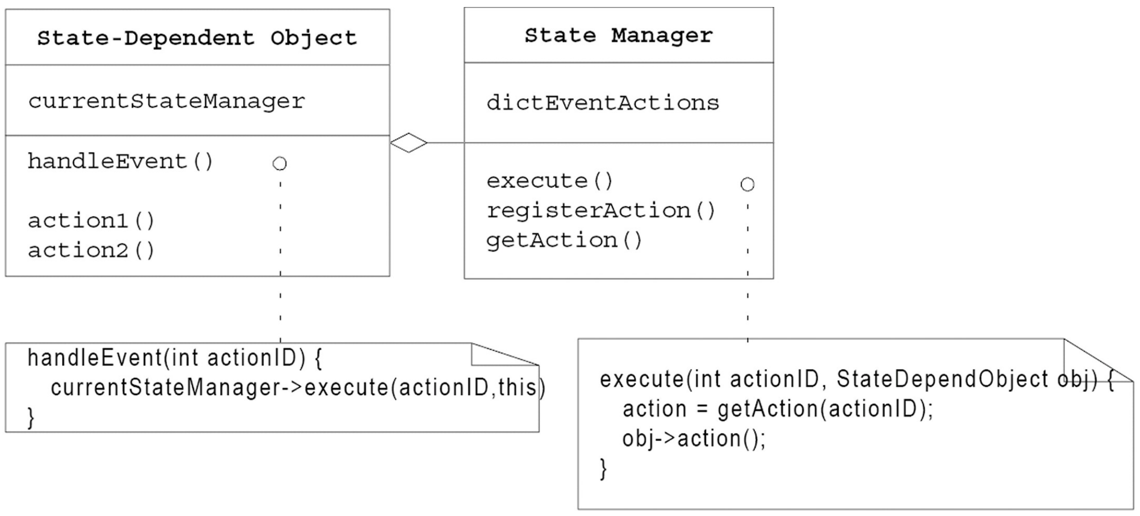
The solution has the following consequences for design and implementation:

- This approach uses a single class, the state manager, with an instance for each state as opposed to the state pattern, which requires a class for each state. This is because the state action manager pattern uses a data-driven approach for action mapping instead of hard-coding the actions in state classes. Using an instance for each state may waste a lot of memory space. To overcome this problem, you may share state manager objects as discussed in the next point.
- Supposing that multiple state dependent objects have identical behavior, it is possible to share the common state mapping behavior between them. This is because the state manager does not hold an instance variable to any state dependent object. For this purpose you may implement shared state managers following the singleton pattern.
- Because the state action manager pattern locates the mapping of events to actions in the state

manager class, it is easy to place any general policies in this class, thus sharing the code. For example, you could trace the execution of actions in a log file.

- This approach does not break the encapsulation of the state dependent object, because the implementation of actions is not separated into auxiliary classes. In contrast, the state pattern uses state classes for each action.
- The introduction of a state manager object avoids the `if` cascades or `switch` statements used for state checking in the state dependent object. This is achieved by simply substituting the state dependent code with a method call to a state manager object.
- Flexibility exists for adding or modifying behavior for a given state. Due to the table-driven nature of this approach, you can add new behavior at runtime by registering new event/action pairs for a given state.
- Introducing a state manager class as an additional indirection increases complexity. Now you have to look at two classes in order to understand the state dependent object's behavior.

Structure



```
handleEvent(int actionID) {
   currentStateManager->execute(actionID,this)
}
```

```
execute(int actionID, StateDependObject obj) {
   action = getAction(actionID);
   obj->action();
}
```

Figure 2. Structure of the state action manager.

Participants

The *state dependent object*

- holds an instance of the current state manager, and
- implements the actions that are registered in the state manager.

The *state manager*

- holds the mapping of events (as key) and actions (as value) defined in the state dependent object,
- defines an interface for registering and finding the actions for given events, and
- delegates execution of actions to the state dependent object.

Collaboration

Figure 3 shows the dynamic collaboration between the state dependent object and the state manager: the state dependent object directs a request (event OK) to its current state manager object, which evaluates the action and delegates the execution back to the state dependent object.

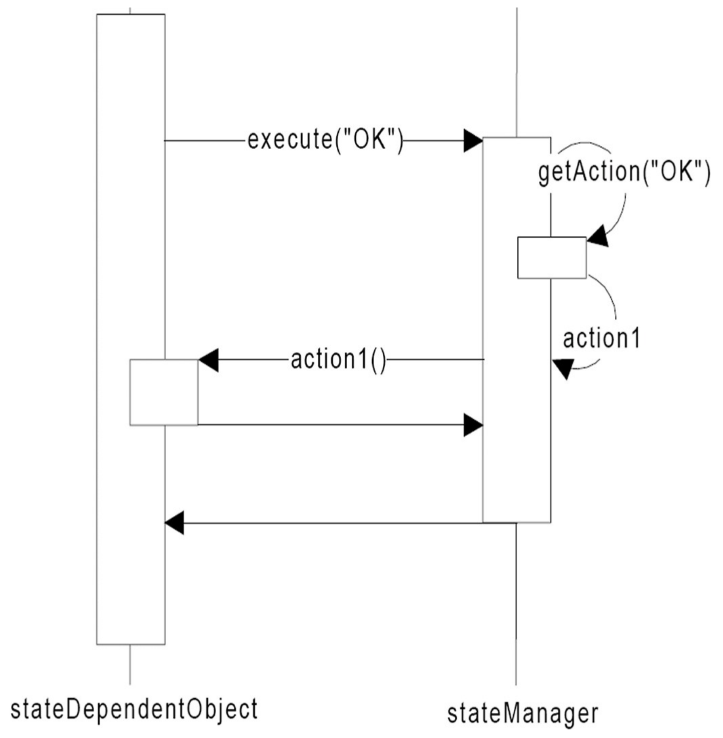


Figure 3. Dynamic behavior of the state action manager.

Example Solved

Figure 4 shows an excerpt of the dialog control from an application framework.

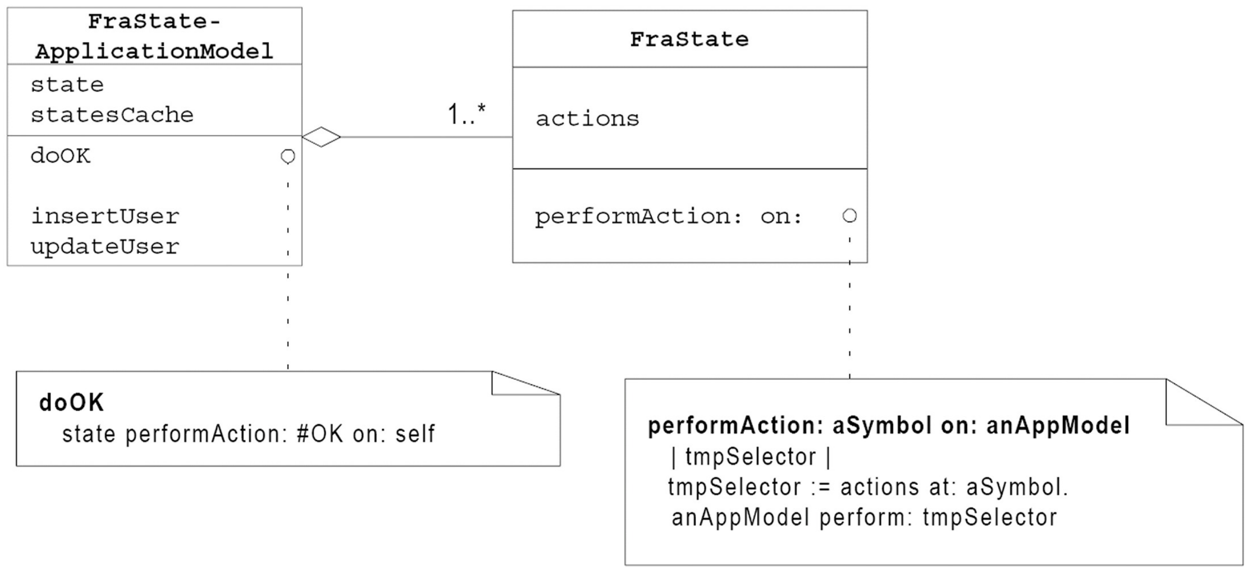

FraState-ApplicationModel
state
statesCache
doOK
insertUser
updateUser
1..*
FraState
actions
performAction: on:
doOK
state performAction: #OK on: self
performAction: aSymbol on: anAppModel
| tmpSelector |
tmpSelector := actions at: aSymbol.
anAppModel perform: tmpSelector

Figure 4. Dialog control.

`FraStateApplicationModel` defines the event `doOK` and implements the actions `insertUser` and `updateUser`. (For brevity of this example, these actions are placed in this class. Usually events and actions are defined in a concrete subclass of this class.) For mapping the events and actions, `FraState` uses a dictionary containing the actions.

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

Implementation

Dynamic registering of actions and 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 objects after the action or command patterns in [2]. This is achieved by simply encapsulating the method invocation in a class.

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

Variants

Action Filter

You can create subclasses of the state manager that 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 manager object can forbid the execution of some registered actions (e.g., actions with write access). This may be achieved by adding additional tables for event/disable-action and event/enable-action pairs.

Nested Dictionaries

This variant uses one dictionary for all states (mapping from a 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.

There are, however, some drawbacks to this approach. You can share the dictionaries (as data) between different objects, but you are not able to share common code as provided by a state manager class. Nested dictionaries may also be harder to maintain, because they are less explicit than a state mapping class and sometimes harder to debug.

Nevertheless, this may be a good choice for simple action models (this was the original implementation of a dialog control).

Single Dictionary

An even simpler variant is to 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, though you may want to create your own key class in the interest of clarity.

Related Patterns

The state pattern [2] is closely related, as the state action manager pattern is an extension of it. State manager plays the role of state in the original state pattern. Essentially there are three differences:

- The state manager allows dynamic registration of event/action pairs at runtime. This is a result of the data-driven approach used for action mapping, instead of hard-coding the actions in methods of state classes as done in the state pattern.
- This pattern uses a single class, the state manager class, as opposed to the original state pattern which requires a class for each state. Furthermore, it is possible to share a state manager instance between multiple state dependent objects, assuming they have identical behavior.
- The state manager delegates the execution, and hence the implementation, back to the state dependent object.

A good choice for implementing a state transition strategy can be found in the state transition patterns described in the volume State Patterns Language [1]. The state-driven transitions pattern involves a state object (in our pattern, a state manager object) being responsible for the transition from itself (the current state) to the new state object. When using this pattern, you are not able to share state manager objects if you have different state transition profiles for each state dependent object.

The owner-driven transitions pattern deals with the alternative approach, which is for the owning object (in our pattern, the state dependent object) to implement the finite state machine. This pattern enables you to share state manager objects even when you have different state transition profiles.

In his event-centered architecture pattern MOOD [3], Alexander Ran offers a good classification of event patterns. He also gives some hints on implementing the finite state machine in conjunction with the owner-driven transitions pattern.

Known Uses

The state action manager pattern is used for realizing the dialog control of an application framework. The pattern solves two aspects of dialog control: one already mentioned is control of dialog behavior, and the second is control of appearance (enabling and disabling widgets).

Dialog control with the state action manager pattern is actually used in two projects of the German railway company: a database for locomotives and wagons, and a locomotive maintenance and servicing project. There are plans to use this pattern for an upcoming enterprise Java framework.

References

1
Paul Dyson, Bruce Anderson: State Patterns; Proceedings of EuroPLoP ′96.

2
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns, Elements of Reusable Object-oriented Software; Addison-Wesley, 1995.

3
Alexander Ran: MOODS: Models for Object-Oriented Design of State; Proceedings of PLoP ′95.

Biography

Gunther Palfinger (palfinger@acm.org) is a software architect at, and since 2000, the managing director of, eMundo, an IT-consulting company in Munich, Germany.