State Patterns

Paul Dyson and Bruce Anderson

Abstract

As our understanding of how we build software grows, we document recurrent design decisions in the form of patterns. As our understanding of each pattern grows, we extend and refine these patterns so that the advice they give is more concrete and comprehensive. Here, we present seven patterns that refine and extend the *State* pattern found in the GOF [GHJ+94] book: the refinements are concrete advice for some of the decisions mentioned, but not dealt with in detail, in the original pattern; the extensions are new advice for making decisions that we have repeatedly had to make while implementing the *State* pattern. The splitting of the one large pattern into a pattern-language also makes it more accessible and easier to understand.

Acknowledgement

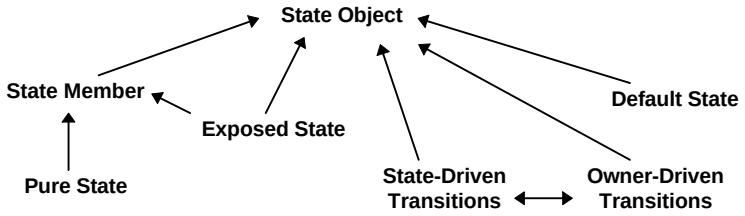
Many thanks to Walter Zimmer who was the EuroPLoP shepherd for this paper and gave some excellent and concrete advice on how it might be improved.

Introduction

Here we present seven patterns that deal with the implementation of state-dependant behaviour. The patterns are all extensions or refinements to the *State* pattern found in 'Patterns: Elements of Object-Oriented Software' [GHJ+94] (referred to here as the Gang-of-Four - or GOF - book), with the first pattern - *State Object* - representing what we see as the core of the GOF pattern: the delegation of state-dependant behaviour to an object which encapsulates state. This pattern is the central pattern to the language. If, in a given set of circumstances, the reader decides that *State Object* is not a suitable pattern for their needs then none of the other patterns will be of much use. If, however, the reader decides that they should use *State Object*, the other patterns give concrete advice on how to implement its key concepts. By splitting what could be a very large single pattern up into a language, the reader can make a choice based on the prime motivation for using the pattern (*State Object*) and then gradually proceed through the language to obtain more detail about the decisions required to implement such a solution.

Patterns Summary

The patterns presented as a language - they don't stand alone but rather have dependencies on the other patterns in the language. These dependencies are shown below with the single-headed arrows indicating preceding patterns and the double-headed arrow indicating two patterns in tension with each other:



State Object represents the core of the *State* pattern as presented in the GOF [GHJ+94] book: the encapsulation of state as an object to which state-dependant actions are delegated.

The *State Member* pattern deals with whether data members should be placed in the owning object or in the *State Object*. This is an extension of the GOF *State* pattern.

The absence of any *State Members* leads to a *State Object* with *Pure State*: *Pure State* is a special case of *State Member*. This pattern shows how to share such *State Objects* between a number of owning objects. *Pure State* is a refinement of the GOF patterns section on 'sharability'.

Usually, the use of *State Object* to implement the state-dependant behaviour of an owning object will be encapsulated within the owning object's class. However, sometimes it is necessary to allow external objects direct access to the owning object's *State Object*. *Exposed State* details when such an implementation is appropriate. This is an extension to the GOF pattern.

State-Driven Transitions and *Owner-Driven Transitions* are two patterns in tension with each other. *State-Driven Transitions* deals with the *State Objects* being responsible for the transition from one state to another. *Owner-Driven Transitions* deals with the alternative approach which is for the owning object to implement the finite state machine. These two patterns are refinements of the GOF pattern which deals mainly with the benefits and consequences of having the *State Objects* drive the state machine and only briefly mentions making the owning object responsible.

The final pattern, *Default State* deals with the creation of the correct initial *State Object* when creating a new owning object. This is an extension of the GOF pattern.

A summary of the problems dealt with by the patterns, and the solutions they provide, is presented below:

Pattern Name	Problem	Solution
State Object	How do you get different behaviour from an object depending on its current state?	Encapsulate the state of the object in another, separate, object. Delegate all state-dependant behaviour to this *State Object*.
State Member	How do you decide whether a data member belongs in the owning class or in the *State Object* class?	If a data member is only required for a single state then place it in the corresponding *State Object* class. If the data member is required for some, but not all, states then it should be placed in a common superclass. If the data member is state-independent, place it in the owning class and pass it to the *State Object* if necessary.
Pure State	You have a lot of *State Object*s, is it possible to cut down on the number required?	When a *State Object* has no *State Member*s, it can be said to represent *Pure State* - nothing but state-specific behaviour. A single *Pure State* object can be shared amongst any number of owning objects, drastically reducing the number of objects required.
Exposed State	How do you prevent the owning class having an excessive number of state-specific, state-dependant methods?	If a data member is only required for a single state then place it in the corresponding *State Object* class. If the data member is required for some, but not all, states then it should be placed in a common superclass. If the data member is state-independent, place it in the owning class and pass it to the *State Object* if necessary.
State-Diven Transitions	How do you get the *State Object* to change when the owning object's state changes?	Have the *State Object* initiate the transition from itself (the current state) to the new *State Object*. This ensures transitions are atomic and removes state-dependant code that determines the next state depending on the event and the current state.
Owner-Driven Transitions	How do you reuse *State Object* classes amongst owning classes with different state-transition profiles?	If *State Object* classes are to be used by more than one owning class, and those owning classes have different FSMs, have the owning class initiate the transition between states.
Default State	When creating a new owning object, how do I ensure that it has the correct initial *State Object*?	Have a method, called in the initialize method, that returns the default *State Object*. Redefine this method in a subclass if a different default state is required.

Pattern Template

Each pattern follows the form:

Pattern Name

Preceding patterns.

Problem (in bold)

Forces acting on the problem, including examples.

Solution (in bold)

Solution example.

Subsequent patterns.

State Object

How do you get different behaviour from an object depending on its current state?

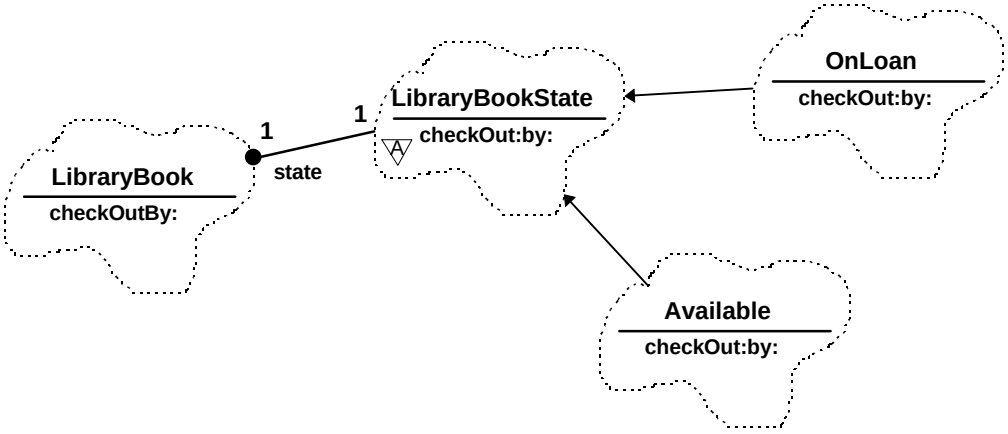
We often want an object to behave differently depending on what state it is currently in. For example, if we have a LibraryBook object and we wish to borrow it, we want it to react differently to the checkOutBy: message (the LibraryBook should be able to check itself out to a LibraryUser) depending on its current loan state: if it is available for loan then it can be checked out, if it is already checked out then an error has occurred. We could set a flag in the LibraryBook object that indicates whether it is currently being borrowed or not. If we took this approach, the checkOutBy: method would look like:

```
checkOutBy: aUser
"Check self out to aUser"

(currentlyBorrowed = 1)
       ifTrue: [^self error: 'The book is currently out on loan.'].

"Check the book out"
```

There are two problems with this approach (as described in the *Choosing Message* pattern [Bec95]):

1. Adding new states that the LibraryBook may have requires more flags and more conditional tests.
2. This kind of logic will have to be repeated in other methods. This means that several methods have to be maintained.

Encapsulate the state of the object in another, separate, object. Delegate all state-dependant behaviour to this *State Object*.



We create an abstract class: LibraryBookState, which represents the state of the LibraryBook object and declares a checkOutBy: method. We then inherit from this class to create two new concrete classes: Available and OnLoan. These classes define checkOut:by: to perform in a manner appropriate to the state they represent.

LibraryBook now delegates checking itself out to its *State Object*:

```
LibraryBook>>checkOutBy: aUser
    state checkOut: self by: aUser
```

LibraryBookState defines checkOut:by: to be implemented by its subclasses and the concrete state classes, Available and OnLoan, define the method according to the state they represent:

```
LibraryBookState>>checkOut: aBook by: aUser
    ^self implementedBySubclass
```

```
Available>>checkOut: aBook by: aUser
    “Check out aBook to aUser”

OnLoan>>checkOut: aBook by: aUser
    ^self error: ‘Book is already out on loan’
```

With a hierarchy of states, adding a new state is now simply a matter of inheriting from LibraryBookState and redefining checkOut:by: to perform an action appropriate to that new state. Suppose we want to allow a book to be reserved, where reservation is equivalent to a copy of the book being placed on a ‘reserved’ self in the library until the reserving user comes to pick it up. This is a new state because it is mutually exclusive of the existing states: the book is neither available for loan nor currently out on loan. To represent this new state we create a new subclass of LibraryBookState (Reserved) and redefine checkOut:by:

```
Reserved>>checkOut: aBook by: aUser
    (aUser = reservingUser)
        ifTrue: [“Check out aBook to aUser”]
        ifFalse: [^self error: ‘The book is currently reserved’]
```

State Object is the central pattern to this language, all the other patterns are subsequent to this one.

State Member

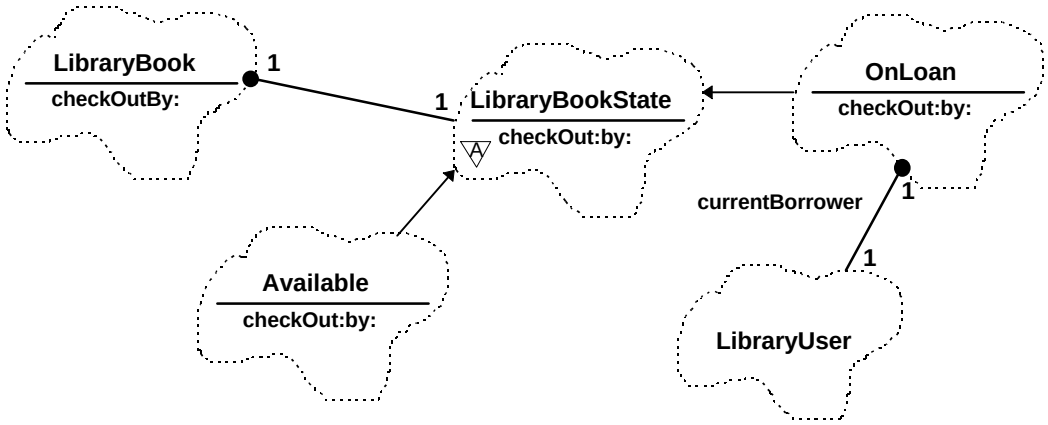
You are using *State Object*.

How do you decide whether a data member belongs in the owning class or in the *State Object* class?

Data members form part of an object’s state, but not all of them belong in the *State Object* and, for those that do, it is sometimes difficult to determine which *State Object* class they should be placed in. How do we choose whether to place a data member in the *State Object* class or in the owning class, passing it as a parameter in a message when required?

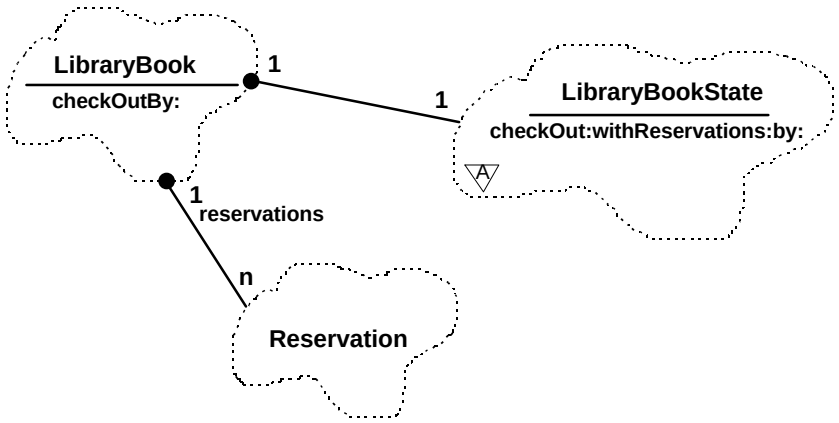
Returning to the previous example of the LibraryBook object, we wish to reference the LibraryUser who has checked out the book. Does this reference go in the LibraryBook class, or one of the LibraryBookState classes? We also wish to introduce a more sophisticated version of reservations, where the book can be reserved like a hotel room: the reservation is made for a period of time from a particular date and the book can still be borrowed outside of the reservation period. Multiple reservations can be placed upon a single book as long as they don’t overlap.

If a data member is only required for a single state then place it in the corresponding *State Object* class. If the data member is required for some, but not all, states then it should be placed in a common superclass. If the data member is state-independent, place it in the owning class and pass it to the *State Object* if necessary.



We only wish to have a reference to the user borrowing the book if it is actually out on loan: having a reference to a borrowing user is unnecessary if the state is anything but OnLoan. Hence, we make borrowingUser a *State Member* of the OnLoan *State Object* class:

Unlike currentBorrower, a list of reservations is state-independent: the reservations remain valid whether the LibraryBook is Available or OnLoan. Because of this, we make the list of reservations a member of LibraryBook, but we have to pass the list to the *State Object* as part of checking the book out, as the validity of the action of borrowing a book is now affected by the reservations made upon it:



The new methods for checking a book out now look like:

```
LibraryBook>checkOutBy: aUser
    state checkOut: self withReservations: reservations by: aUser

LibraryBookState>>checkOut: aBook withReservations: someReservations by: aUser
    ^self implementedBySubclass

Available>>checkOut: aBook withReservations: someReservations by: aUser
    “If checking out aBook doesn’t invalidate any of the reservations,
     check out aBook to aUser”

OnLoan>> checkOut: aBook withReservations: someReservations by aUser
    ^self error: ‘Book is already out on loan’
```

A *State Object* that has no *State Members* is an example of *Pure State*.

Introducing *State Members* may require the use of *Exposed State*.

Pure State

You are using *State Object* but not *State Member*.

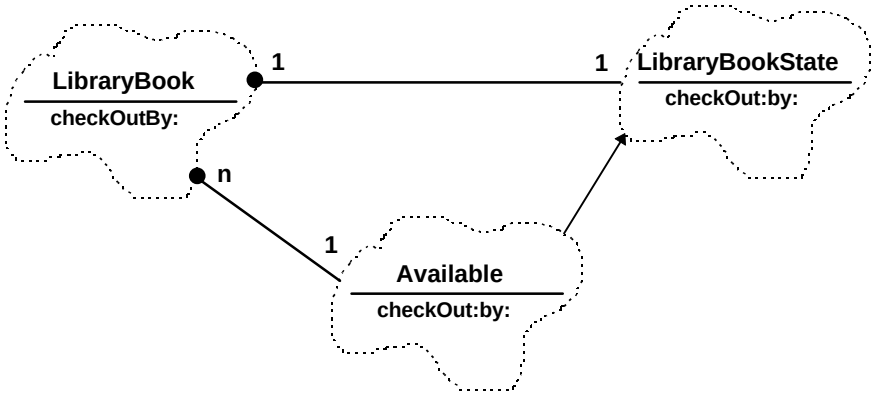
You have a lot of *State Objects*, is it possible to cut down on the number required?

Following the *State Object* pattern requires that two objects are used (owning and *State Object*) where one used to be sufficient. This increases complexity and can be inefficient in terms of memory usage.

When a *State Object* has no *State Members*, it can be said to represent *Pure State* - nothing but state-specific behaviour. A single *Pure State* object can be shared amongst any number of owning objects, drastically reducing the number of objects required.

The Available *State Object* is *Pure State*, it simply implements the state-specific behaviour for checking a LibraryBook out. Because of this, any number of LibraryBook objects can share a single Available object,

they only need separate OnLoan objects because each OnLoan object records some state-specific data (on which LibraryUser is borrowing the book):



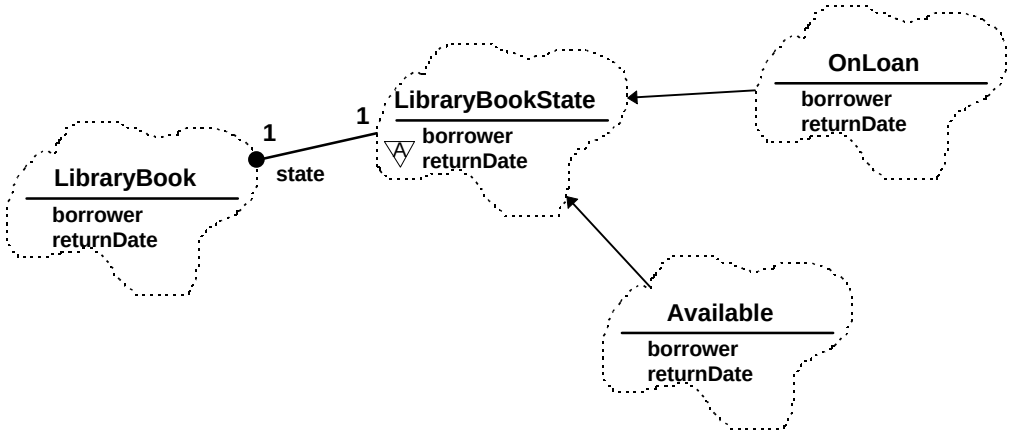
A *Pure State* object is an example of a *Flyweight* [GHJ+94], and is also often a *Singleton* [GHJ+94].

Exposed State

You have a *State Object* encapsulated within an owning object.

How do you prevent the owning class having an excessive number of state-specific, state-dependant methods?

State Object is an implementation pattern - it describes how we might implement state-specific behaviour in a separate object. Following the principle of encapsulation, the owning object's class provides all the state-dependant methods in its interface: we don't see that these messages are actually delegated to another object. This can cause a problem when a lot of the state-dependant messages are also state-specific:



LibraryBook declares two methods: borrower and returnDate, which return the user currently borrowing the book and the date it is due to be returned, respectively. These are state-specific enquiries, they have no meaning when the book is available for loan:

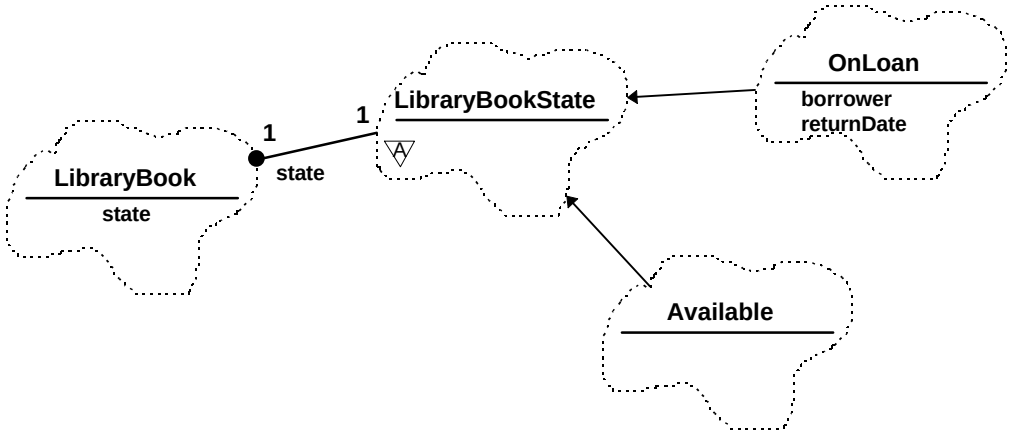
```
LibraryBook>>borrower
    ^state borrower

LibraryBook>>returnDate
    ^state returnDate

LibraryBookState>>borrower
    ^self implementedBySubclass

LibraryBookState>>returnDate
    ^self implementedBySubclass
```

```
Available>>borrower
    ^self error: 'Book not out on loan'

Available>>returnDate
    ^self error: 'Book not out on loan'

OnLoan>>borrower
    ^currentBorrower

OnLoan>>returnDate
    ^dueDateOfReturn
```

Another problem arises when we add a new state to the hierarchy. If we wish to add a Reserved state, which has a reservingUser *State Member*, we need to update the interfaces of LibraryBook, LibraryBookState, Available and OnLoan to accept the reserver message, with Available's and OnLoan's versions generating an exception.

Expose the *State Object* by defining a method in the owning class that returns a reference to it. Make state-specific enquiries directly to the *State Object*.

This would change the example above to look like:



LibraryBook's state method simply answers the *State Object*:

```
LibraryBook>>state
    ^state
```

and state-specific enquiries are made directly to it. Using this pattern, Reserved could be added and enquires made about the user making the reservation without needing to update the interfaces of the other *State Object* classes. However, the major drawback of using *Exposed State* to help control interface complexity is that we need to be sure that the owning object is in the required state to make state-specific enquires. If the state method returned an Available object, a doesNotUnderstand exception would be generated in response to the borrower message.

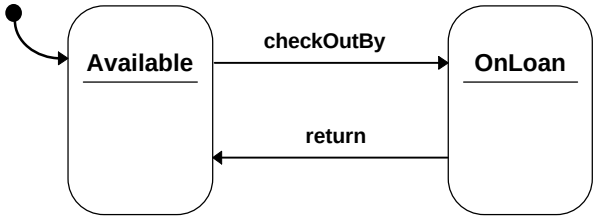
Exposed State is usually required when we have a large number of *State Members*, and so a large number of methods in the *State Object* classes.

State-Driven Transitions

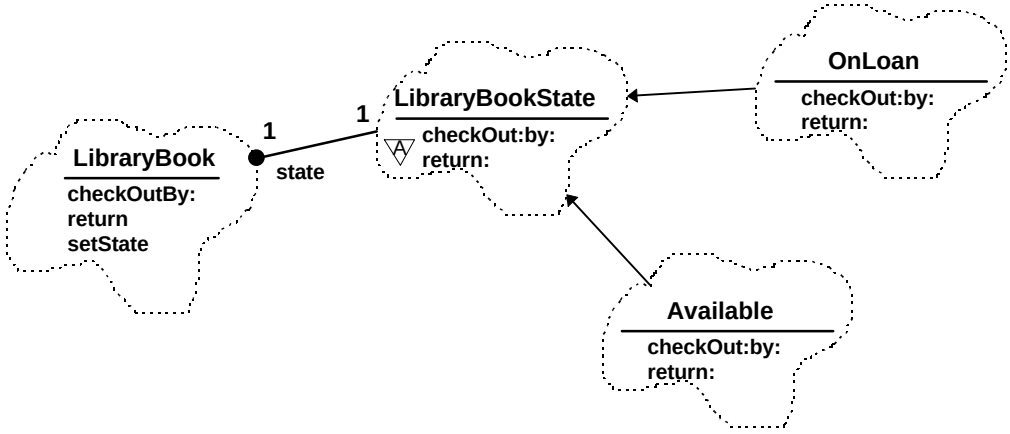
You are using a number of *State Objects* to implement a state machine.

How do you get the *State Object* to change when the owning object's state changes?

When we can express the behaviour of an object in terms of the states it can be in, and the actions which affect the transitions between those states, we can use *State Object* to implement a 'finite state machine'. The (simplified) state diagram for LibraryBook looks like:



Have the *State Object* initiate the transition from itself (the current state) to the new *State Object.* This ensures transitions are atomic and removes state-dependant code that determines the next state depending on the event and the current state.



LibraryBook's interface is extended to allow its state to be changed, which the *State Object* does when a transition is required:

```
LibraryBook>>checkOutBy: aUser
    state checkOut: self by: aUser

LibraryBook>>return
    state return: self

LibraryBook>>setState: aBookState
    state := aBookState

Available>>checkOut: aBook by: aUser
    "Check out aBook to aUser"
    aBook setState: (OnLoan to: aUser)  "a new OnLoan State Object"
```

```
OnLoan>>return: aBook
    "Register aBook as being returned"
    aBook setState: (Available new)
```

State-Driven Transitions is in tension with *Owner-Driven Transitions*.

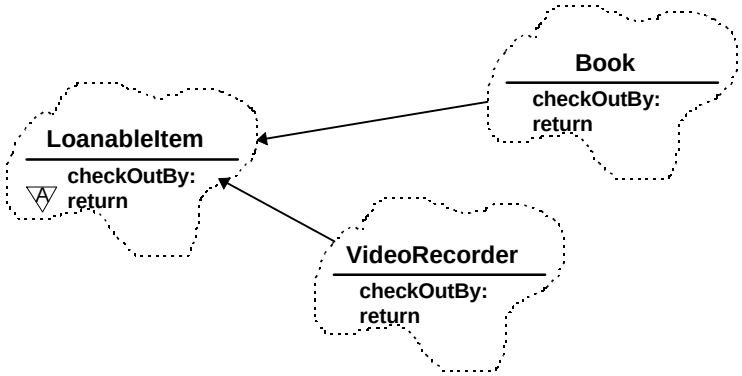
Owner-Driven Transitions

You are using a number of *State Objects* to implement a state machine. You are considering using *State-Driven Transitions*.

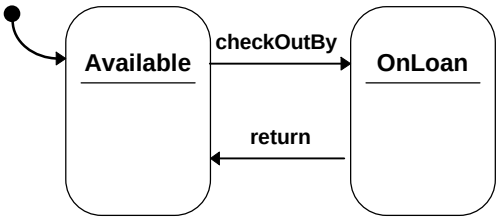
How do you reuse *State Object* classes amongst owning classes with different state-transition profiles?

State-Driven Transitions describes how to implement a finite state machine using *State Objects*, and have those objects manage the transitions between states. If you wish to use those *State Objects* for another owning object, the new owning object must follow the same state machine otherwise you need a new set of *State Objects*.

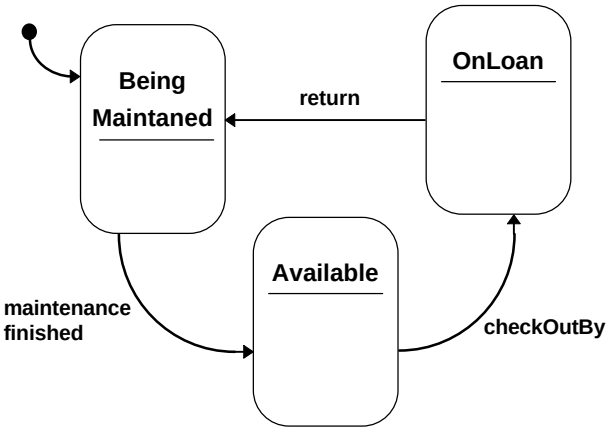
Let's widen the example of the Library system to include other things available for loan such as audio/visual equipment for teaching purposes (if this is too fanciful, imagine a teaching resources centre that also loans out books and manuals). We can create an abstract superclass: LoanableItem, and inherit Book and VideoRecorder from it:



The FSMs for these two subclasses of LoanableItems are different and look like:



FSM for Book



FSM for VideoRecorder

When a VideoRecoder is returned after being checked out, it is sent for maintenance to check that it hasn't been damaged while out on loan. Once the checks and any required work have been carried out, the maintenance has finished and the VideoRecorder is available for loan again.

In this example, the code for checking out and returning the VideoRecorder is the same as that for the book. If, because we were using *State-Driven Transitions*, we created a new set of *State Objects* for VideoRecorder, we would be repeating a lot of what is already in Available and OnLoan, even though the only difference between VideoRecorderOnLoan and BookOnLoan would be the state transition in return:

```
BookOnLoan>>return: aLoanableItem
    "Return aLoanableItem"
    aLoanableItem setState: (Available new)

VideoRecorderOnLoan>>return: aLoanableItem
    "Return aLoanableItem"
    aLoanableItem setState: (BeingMaintained new)
```

In addition to the missed reuse, the design has become more confused because of the extra classes, and we have introduced coupling between owning and state classes at the concrete rather than the abstract level.

If *State Object* classes are to be used by more than one owning class, and those owning classes have different FSMs, have the owning class initiate the transition between states.

Rather than have the state transitions encoded into Available, OnLoan, and BeingMaintained, Book and VideoRecorder take responsibility for changing their own state:

```
Book>>checkOutBy: aUser
    state checkOut: self by: aUser.  "if no exceptions have been raised: the book has been
                                      checked out..."
    self setState: (OnLoan to: aUser)

Book>>return
    state return: self.
    self setState: (Available new)

VideoRecorder>>checkOutBy: aUser
    state checkOut: self by: aUser.
    self setState: (OnLoan to: aUser)

VideoRecorder>>return
    state return: self.
    self setState: (BeingMaintained new)
```

This pattern is in tension with *State-Driven Transitions* which should be used as a default unless it is anticipated that *State Objects* will be used in different state machines. Even if some of the *State Objects* are to be used across different FSMs, it may still be possible to use *State-Driven Transitions* by using a *Template Method* [GHJ+94] to avoid redundancies between different implementations of the *State Objects*. This should only be considered when there are only slight changes in transition or when there are only a few occurrences of re-using the *State Objects*.

The owning object in *Owner-Driven Transitions* is taking on the role of a *Mediator* [GHJ+94].

Default State

You are creating a new object which is implemented with *State Object*.

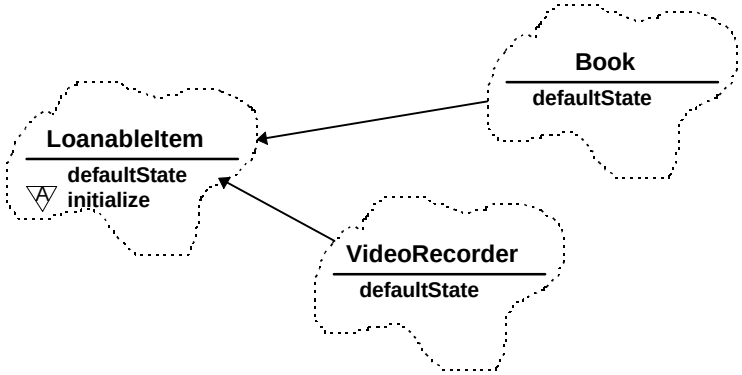
When creating a new owning object, how do I ensure that it has the correct initial *State Object*?

Creating a new instance of an owning object requires that its state is initialised. The default state could be written into the initialize method:

```
LoanableItem>>initialize

    state := Available new
```

but what happens if we need to change the default state? We could subclass and re-write initialize, but the state would still be initialised to Available, only to be re-initialised to the new default state. With a non-trivial *State Object* this could be very expensive. Another approach would be to pass the initial state as a parameter to the constructing method but this would require the explicit creation of a *State Object* every time a controller object is created.

Have a method, called in the initialize method, that returns the default *State Object*. Redefine this method in a subclass if a different default state is required.



The initial state for Book is Available but for VideoRecorder it is BeingMaintained (we can assume that new books are okay, but we have to check that new video recorders work okay). Having the initialize method as a *Template Method* [GHJ+94] ensures that each new class must define what its *Default State* is, but doesn't dictate as to what that should be:

```
LoanableItem>>initialize
    "Some initialisation"
    state := self defaultState

LoanableItem>>defaultState
    ^self implementedBySubclass

Book>>defaultState
    ^Available new

VideoRecorder>>defaultState
    ^BeingMaintained new
```

Default State is an example of *Modifying Super* [Bec95].

References

[Bec95] Kent Beck, *Smalltalk Best Practice Patterns Volume 1: Coding (draft review copy)*, First Class Software Inc., 1995.

[GHJ+94] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Object-Oriented Software*, Addison Wesley, 1994.