

# The Library

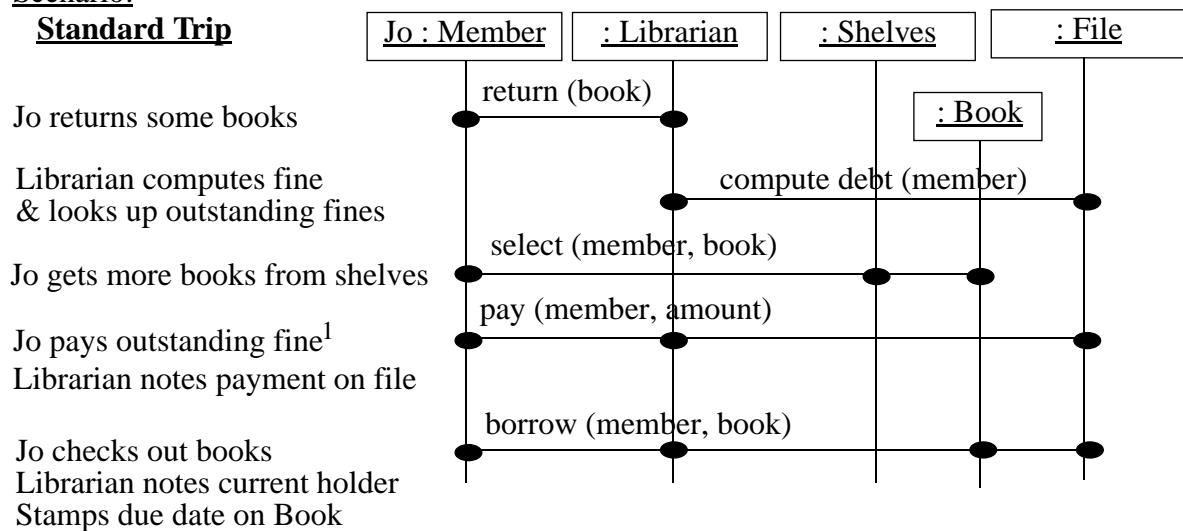
## Business model

### Observation of typical use by members

This is the standard scenario for a trip to the library:

Scenario:

#### Standard Trip



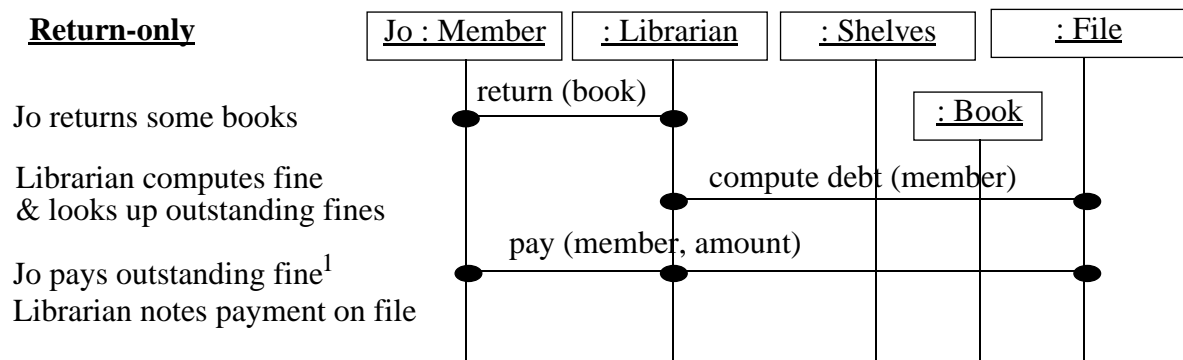
<sup>1</sup> Sometimes, librarians opt to put fines on account, for payment later; or may waive fines altogether

We should note that this involves the tail end of one loan and the start of another.

Sometimes a Member just wants to return books:

Scenario:

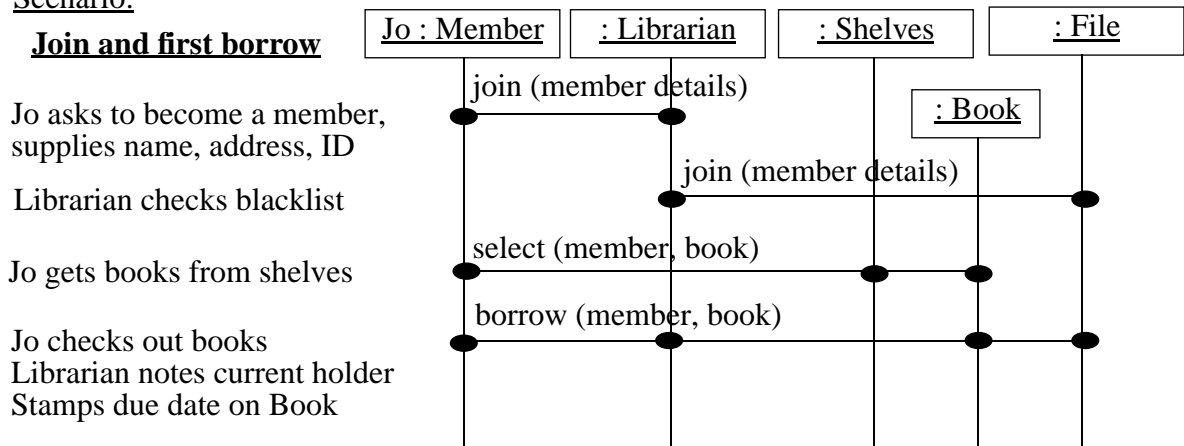
#### Return-only



A new member usually borrows some books immediately:

Scenario:

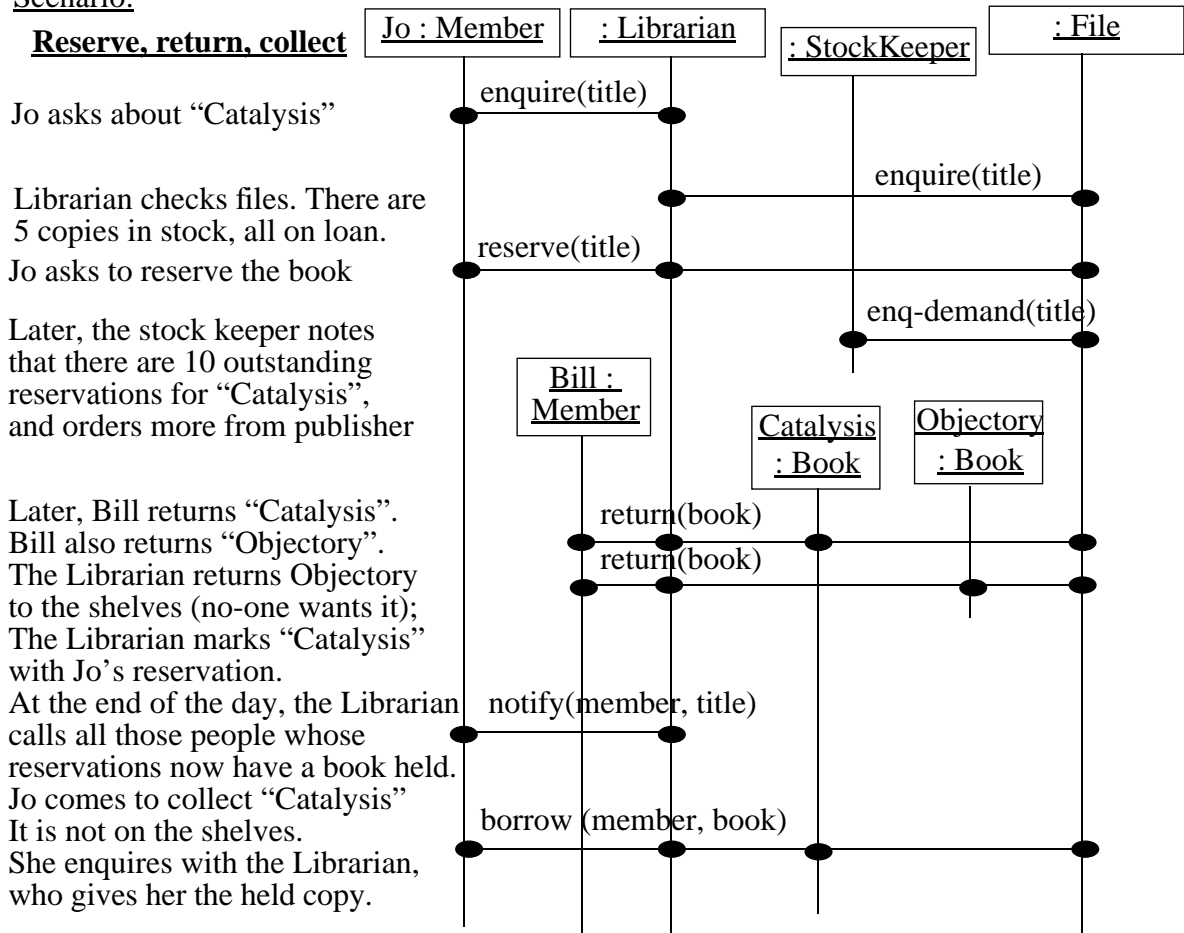
**Join and first borrow**



Reservations introduce extra complications:

Scenario:

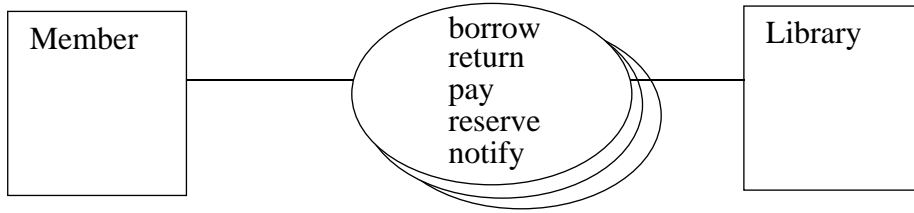
**Reserve, return, collect**



## Business use-cases

*These are the use-cases pertaining to the Library as a whole — not any one computerised system within it.*

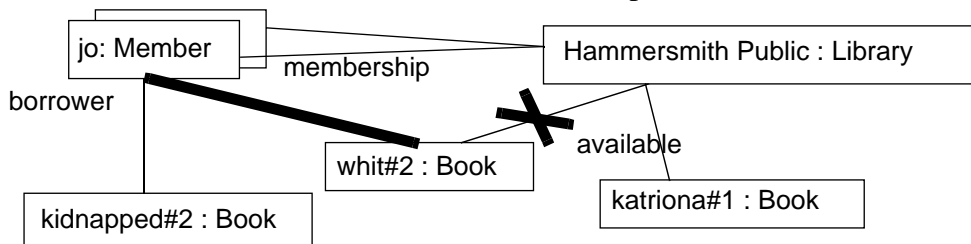
The interactions between the members and the library seem from the scenarios to be:



## Borrow — from shelf

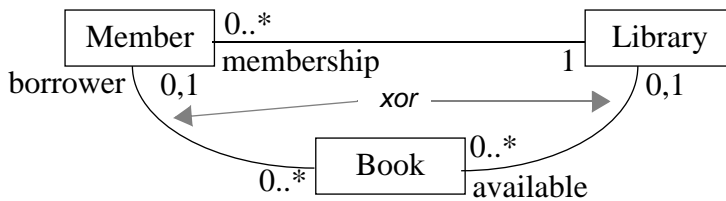
Borrowing a book means it is in the charge of the Member. A Book may be borrowed only if it is available in the Library to which the Member belongs. A Book can't be borrowed and available at the same time.

To illustrate the borrow use-case, we can draw a snapshot like this:



## Dictionary:

- Library — an organisation that lends Books
- Library::membership — the Members entitled to borrow books from the Library
- Member — an entitlement to borrow Books
- Book — any item lent by a Library
- Library::available — the Books the Library currently can lend to Members
- Book::borrower — the Member currently in charge of a Book



<u>use-case</u>	borrow (member, book)	
<u>goal</u>	book.borrower = member	— the member is now the borrower of the book
<u>pre</u>	member.library.available includes book	— the book has to be available in the library

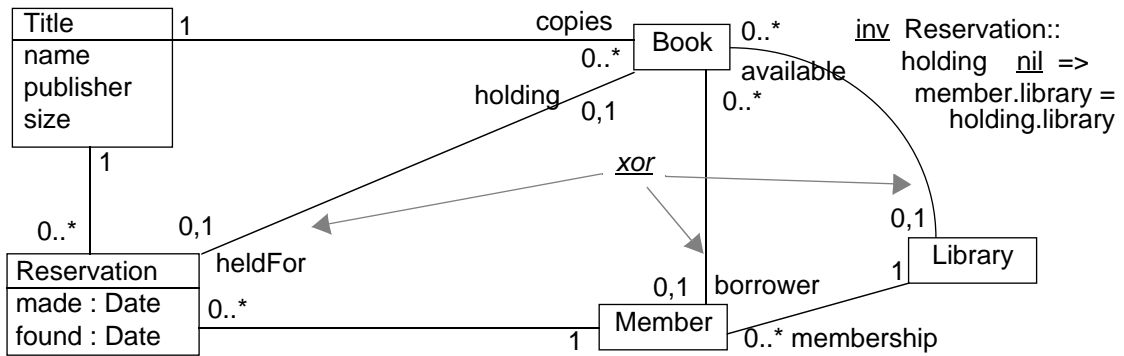
## Reserve

Actually, you don't reserve a Book: you reserve a Title. A Book is a particular copy of a literary work; a Title is our name for that work.

A reservation goes through a couple of states: when it is made, it is waiting for a Book to be found for it. Once a Book is found, it is waiting for the Member to come and collect it. The library records how long a reservation has been in the latter state, and drops the reservation if the book is not collected within a few days.

A book that is being held for a reservation is neither available nor borrowed; this extends the 'xor' to encompass available, borrowed, and held for a reservation. When a book is held for a

reservation, it has to be a book that is available in the library to which the member belongs.



use-case reserve (member, book)  
goal let res belong to new\*Reservation  
and res.member = member and res.book = book  
— a new Reservation exists for this member and book  
and res.made = today

For any given Title and Library, either

- No copies of the Title are available in the Library; or
- There are no reservations for this Title for members of this Library which are unfulfilled. (Unfulfilled means that no Book is being held for it.)

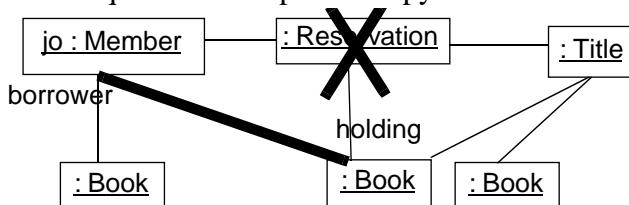
inv Title title, Library lib ::  
(title.copies \* lib.available) = 0 or  
(title.reservations[member.library=lib and holding=nil]) = 0

- Any use-case that sets the holding link should also set the ‘found’ Date.  
effect Reservation :: (holding nil and @pre.holding = nil ) => found = today  
— Any use-case that results in a holding link where there was none before should set ‘found’

*An effect is conjoined with all the postconditions of all the use-cases. However, only those that bring about the changes described to the left of ‘=>’ are affected. The reserve and the return use-cases are affected.*

## Collect (when reserved)

Now we can see that a book can be borrowed even if it is not available on the shelves — but only by the person who holds the reservation for which it is being held. We shall choose to give this a different name, “collect”: because the Member has to know that, having a reservation, they should enquire for the specific copy at the desk.



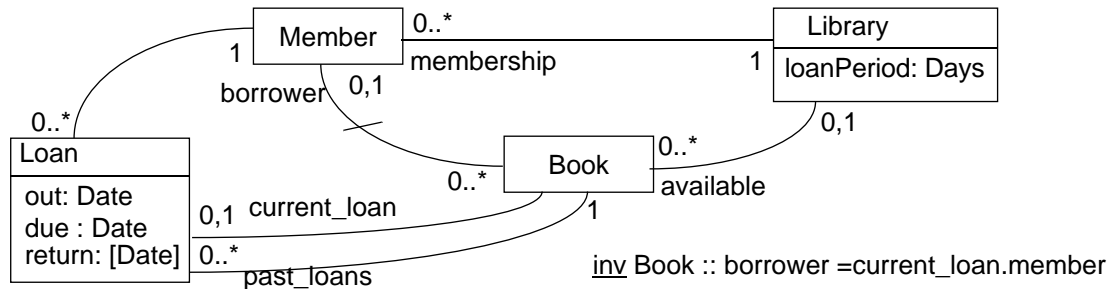
use-case collect (reservation, m : Member)  
goal reservation@pre.holding = m  
— the member is now the borrower of the book  
and deleted (reservation)  
pre reservation.holding nil and reservation.member = m

## Return

When a Member returns a book, a variety of things happen:

- If the book is overdue, a fine is added to their account.
- If there are unfulfilled Reservations for this title (in this library), the book is held for one of them; otherwise, it goes back on the shelves.

But first: we can't make sense of 'overdue' unless we know what the date of the loan was. We shall therefore reify the 'borrower' association. We don't need to delete 'borrower' — just make it a derived simplified version of the Loan object:



We'll model past loans as well. A late interview with the Stock Keeper suggests that the history of borrowing of books helps decide what books to buy.

We should also extend our borrow use-case to define the details of the Loan:

```

use-case borrow (member, book)
goal let loan belong new*Loan
and loan.out = today and loan.due = book.library.loanPeriod + loan.out
and loan.member = member and loan.book = book
  
```

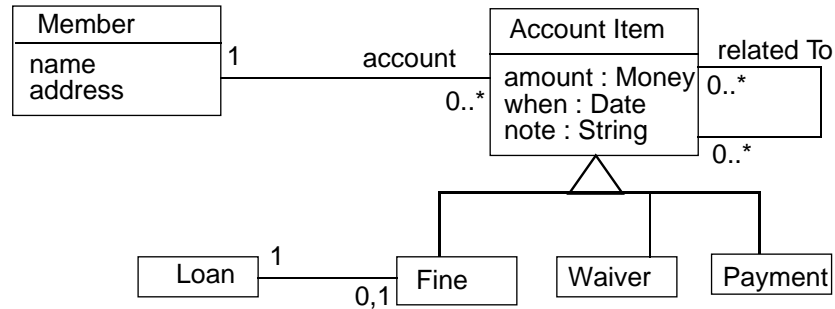
Now the return use case can be defined:

```

use-case return (library, book)
goal book@pre.current_loan.return = today
  — record the date of return
and book.past_loans += book@pre.current_loan and book.current_loan = nil
  — now no current loan, and loan has been added to list of past loans
and ( if book.title.reservations@pre[holding = nil] 0
  — if the set of reservations for this books title
    [ which did not have specific books holding for them ] is not empty
  then book.holdingFor@pre.holding=nil
  — this book is now held for a Reservation that previously had no holding
  else book += library.available )
  — this book is added to the available set of the library
and if today > book.current_loan.due
  then member.account += new Fine (loan)
  — this member's account has a new Fine associated with this Loan
pre book.borrower.library = library
  — only makes sense if this book is borrowed by a member of this library
  
```

The mention of a Fine means we should say what it is. We might as well take the opportunity to

add in other administrative details about members:



An AccountItem represents a financial debt or credit to a Member. There are various kinds, of which Fines, Waivers and Payments are but three. Librarians sometimes make notes against account items, and may cross-refer from one to another — for example if a Waiver is for a specific Fine.

## Payment

use-case pay (member, amount)  
goal member.account += new Payment (amount, member)

It is often convenient to define the specification equivalent of a constructor. We have used them for Payments and Fines. Here are the definitions:

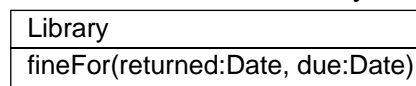
```

new Payment (amount, member) =
  let pay belong to new*Payment
  and pay.when = today and pay.amount = amount
  
```

```

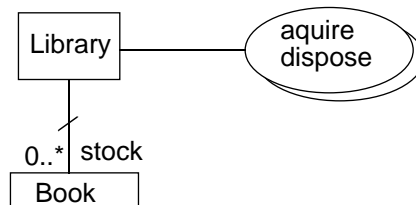
new Fine (loan) =
  let fine belong to new*Fine
  and fine.loan = loan and fine.when = today
  and fine.amount = loan.member.library.fineFor (today, loan.due)
  
```

Each Library has a function for working out a fine for a particular period. We defer until much later the details of that — it may be different for each library.



## Stock

The Library acquires books from time to time, adding books to the stock. “Stock” is the word for all the books that are on the shelves, held by the library’s members, or held for reservations at any one time. Books are also disposed of. These use-cases are interactions between the Library and some external agencies whose identities we know and care nothing of.

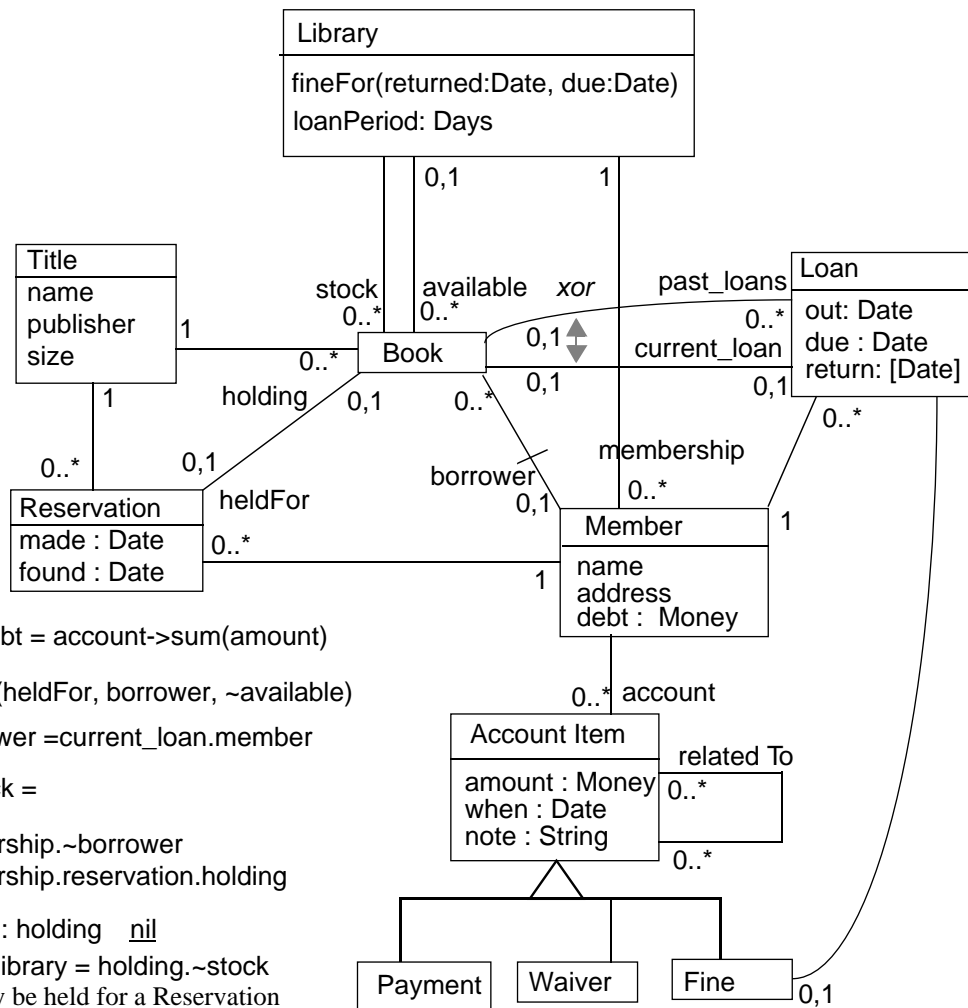


inv Library :: stock = available + membership.~borrower + membership.reservation.holding

use-case acquire (book, library)  
goal library.stock += book  
pre not (library.stock includes book)

## Business model Logical View summary

Bringing together the different pieces of type diagram that we have drawn in this model, we get the diagram below.



inv Member :: debt = account->sum(amount)

inv Book :: XOR (heldFor, borrower, ~available)

inv Book :: borrower =current\_loan.member

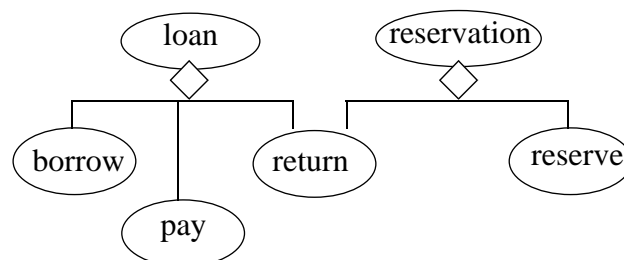
inv Library :: stock =  
 available  
 + membership.~borrower  
 + membership.reservation.holding

inv Reservation:: holding nil  
 => member.library = holding.~stock  
 -- a book can only be held for a Reservation  
 for the library to which the Book belongs

## Higher-level reflections on business model

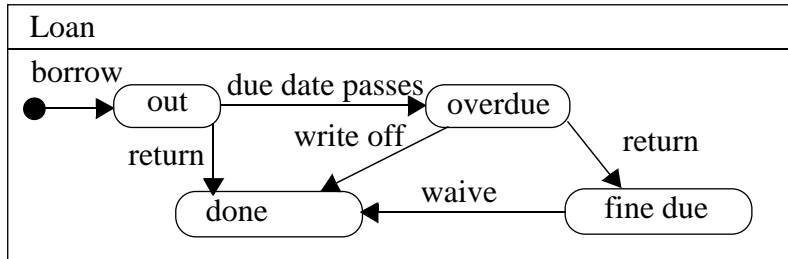
### Abstraction of use-cases

It might be useful to reflect for a moment on the nature of the Loan and Reserve classes. These seem to be reifications of higher-level use-cases — that is, records of the use-cases that have occurred, and showing what states they go through. We could draw this breakdown, showing how a more abstract view of the transactions between the library and its members is composed of occurrences of the more detailed actions we've just shown.



## Loan statechart

In connection with this, we can draw statecharts to show how the overall use-cases progress through their stages by means of the smaller steps.



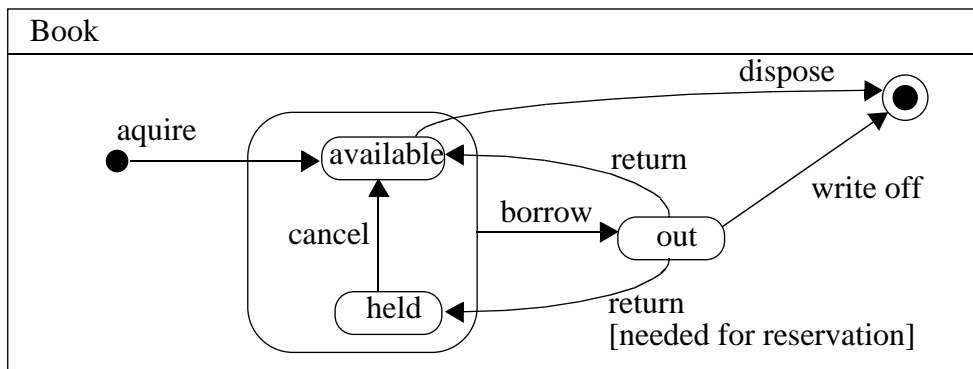
Statecharts often suggest possibilities we did not think of before. In this case, the idea of writing off a long-overdue return is suggested, and is something we should add to the use-case list.

As a cross-check, we should be able to define the states in terms of the model.

```
Loan :: out = (~current_loan nil and today <= due)
           — we are the current loan of a book, but not overdue
Loan :: overdue = (~current_loan nil and today > due )
           — we are the current loan of a book, and past date due
Loan :: done = (~current_loan = nil and ( fine = nil or member.debt <= 0 ))
Loan :: fine_due = (~current_loan = nil and (fine nil and member.debt > 0))
```

## Book statechart

We can write a similar statechart for Book:



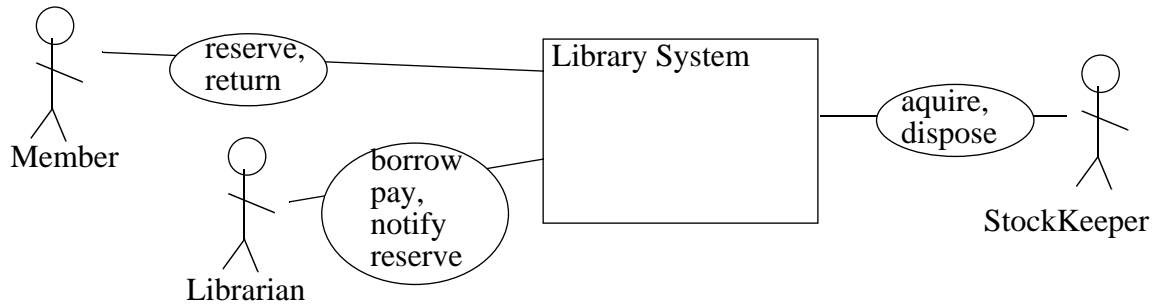
This makes it clear that books should only be disposed when available. It also raises the issue of canceling a reservation: if this happens when a book is being held, it can go back on the shelves. We should write specs for canceling a reservation.

Notice how some of the same use-cases come up: the postcondition of each has an effect on several objects.

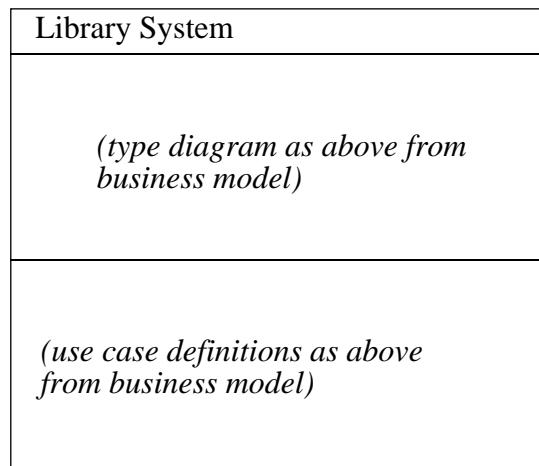
## Library system requirements

### System context

In this design, we shall get Members to make their own reservations (possibly over the internet); and they return books by putting them into a return bin, the entrance to which scans the bar codes on the books. The Librarians supervise loans and fine-payments.

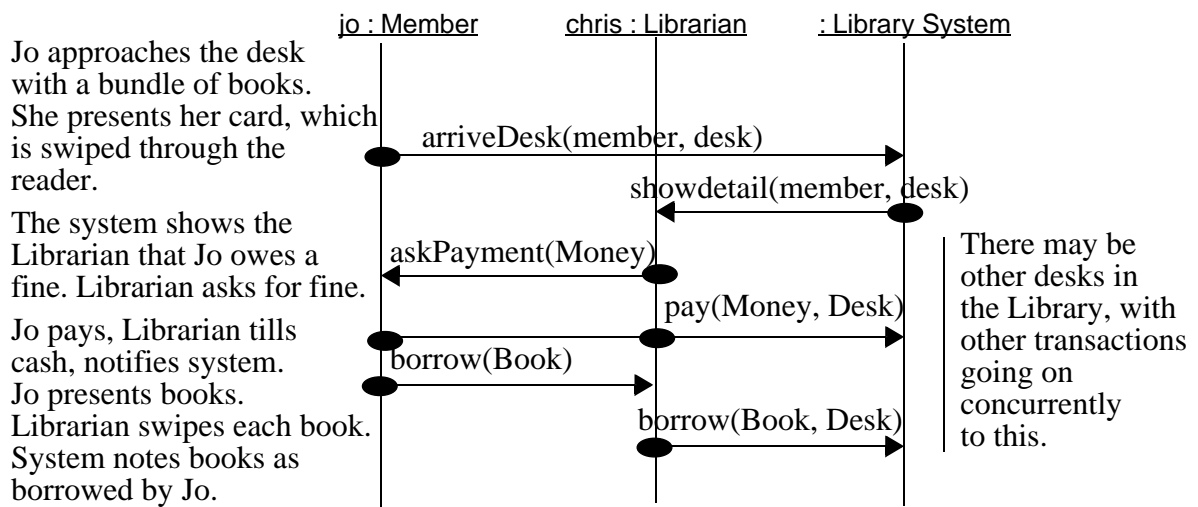


From the diagram, it can be seen that we are automating just about every use-case we discussed for the library as a whole. We will begin by adopting the business Logical View and use-case specs for our system model:

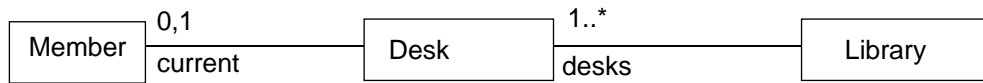


But a little thought suggests we should also look in a little more detail at how each use-case is achieved. Each of the use-cases has several parameters, which always raises the question “how are these parameters chosen?” For example, borrow, reserve, and pay all require a member to be identified: if several books are borrowed at a time, does this mean someone has to type in the member’s name for every book? If not, then what?

Here is a scenario involving these use cases: it is more or less what happens in a non-automated system. Note that the system we’re designing is one of the actors:



Notice that after the member has been identified to the system, subsequent steps identify the desk at which it is happening. This suggests that there is a notion of current member at each desk.



We can modify the relevant use-cases from the business model to use the idea of the member at the desk:

use-case borrow (desk, book)  
pre desk.library.available includes book — the book has to be available in this library  
and desk.current nil — there has to be someone at the Desk  
goal let loan belong new\*Loan  
and loan.out = today and loan.due = book.library.loanPeriod + loan.out  
and loan.member = desk.current and loan.book = book

etc.

And specify the arrival at the desk:

use-case arrive (desk, member)  
pre member.library = desk.library — only if the member is a member here  
goal desk.current = member