

Refer to Chapter 3 of text on **Introductory Case Study**

- UML is for both ***specification*** and ***design*** of systems
- First, it is useful to have a rough glance at a fairly standard approach to specification and design, in terms of the tasks [*ovals*] done and the outputs [*boxes*] produced. This will provide a context, and a basis for comparison, as we introduce UML.

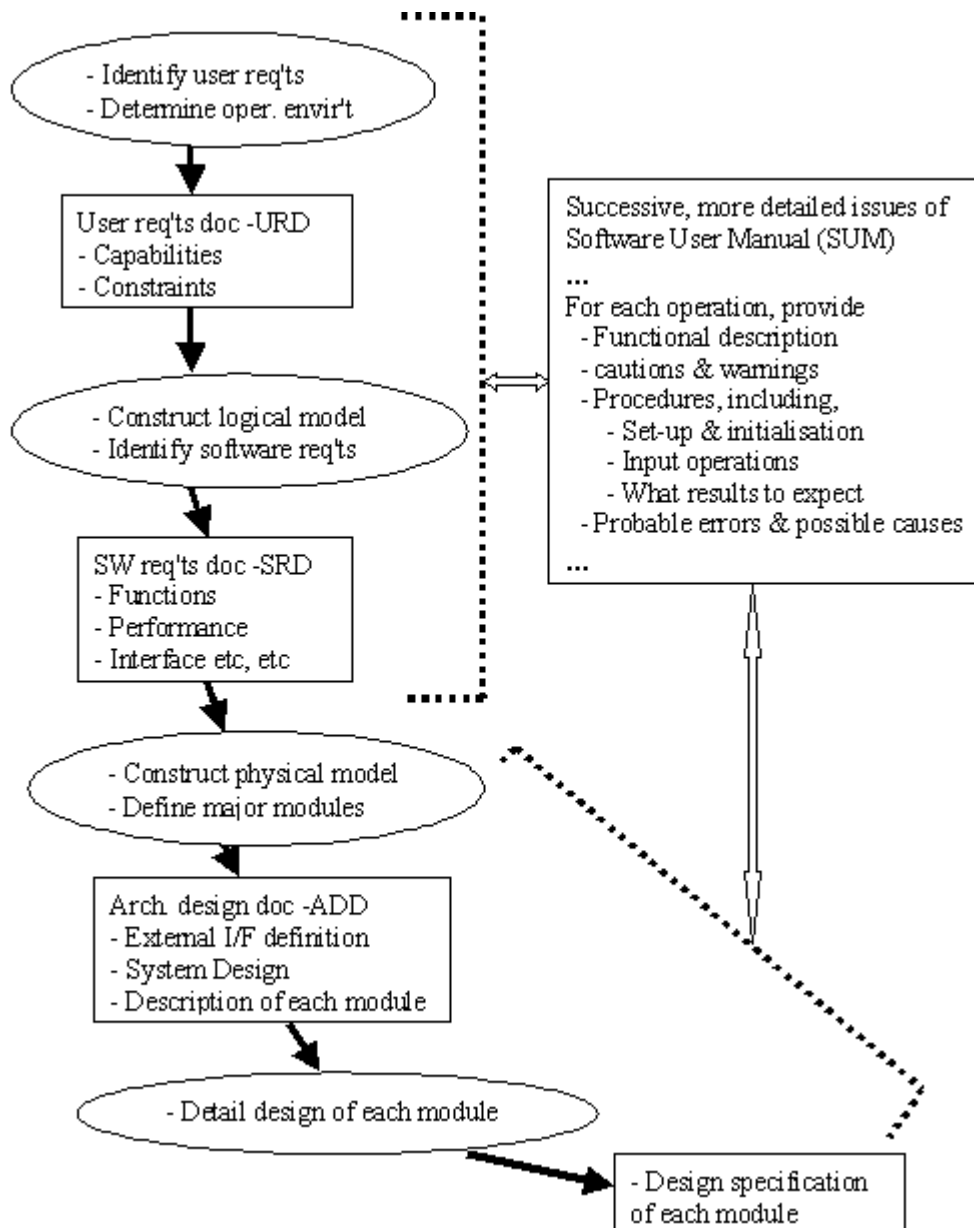


Figure 1: "Fairly standard" view of Specification & design

[[[An aside (from UML): Examples of 2 functional, a performance and a maintainability software requirement from the SRD of an actual project.

Note: (1) Illustration - not to be taken as only or best way to document requirements.

(2) Actual project was specialised with corresponding terminology. The abbreviations TM and TC stand for telemetry and telecommand, respectively. The OBDH (*on-board data handling*) is an external system - can be regarded as a user essentially (see textbook p29, paragraph 2)

Two functional software requirements:

S-1.1.7-2

The SW shall provide either a successful or unsuccessful acknowledgement for receipt of each TC. A successful acknowledgement shall be indicated by preparation of a **TM Successful TC acceptance** packet entry, while an unsuccessful acknowledgement shall be indicated by preparation of a **TM Unsuccessful TC acceptance** packet entry.

TRACE: CF3-1,MF5.2-2,#

S-1.1.7-3

If a TC is executed unsuccessfully then the SW shall prepare a **TM Unsuccessful TC execution** packet entry.

Remark: Criteria for execution reporting are particular to each TC.

TRACE: CF3-2(UNSUCCESSFUL),MF5.2-3,#

A performance software requirement:

S-2-4

The following time constraints for communicating of TM to the OBDH shall be satisfied,

(1) The ACC SW shall accomodate a maximum transfer rate of 9 complete TM packets per 4 second period.

(2) The next TM buffer, if available, shall be specified to the OBDH within 20mS of the occurrence of **interrupt 10** ...

TRACE:MP6-3(sentence 1), R4.2.2.3-2 of RD.3,#

A maintainability software requirement:

S-13-5

Procedures for building verified releases shall support eventual maintenance of these releases, i.e.:

- (semi)-automated procedures for performing compilation and linking should be developed and configured;
- actual source code used in the release shall be identified;
- copies of delivered files shall be retained.

TRACE: Untraced,#

End of Aside]]]

The following outlines how our text presents its introductory case study:

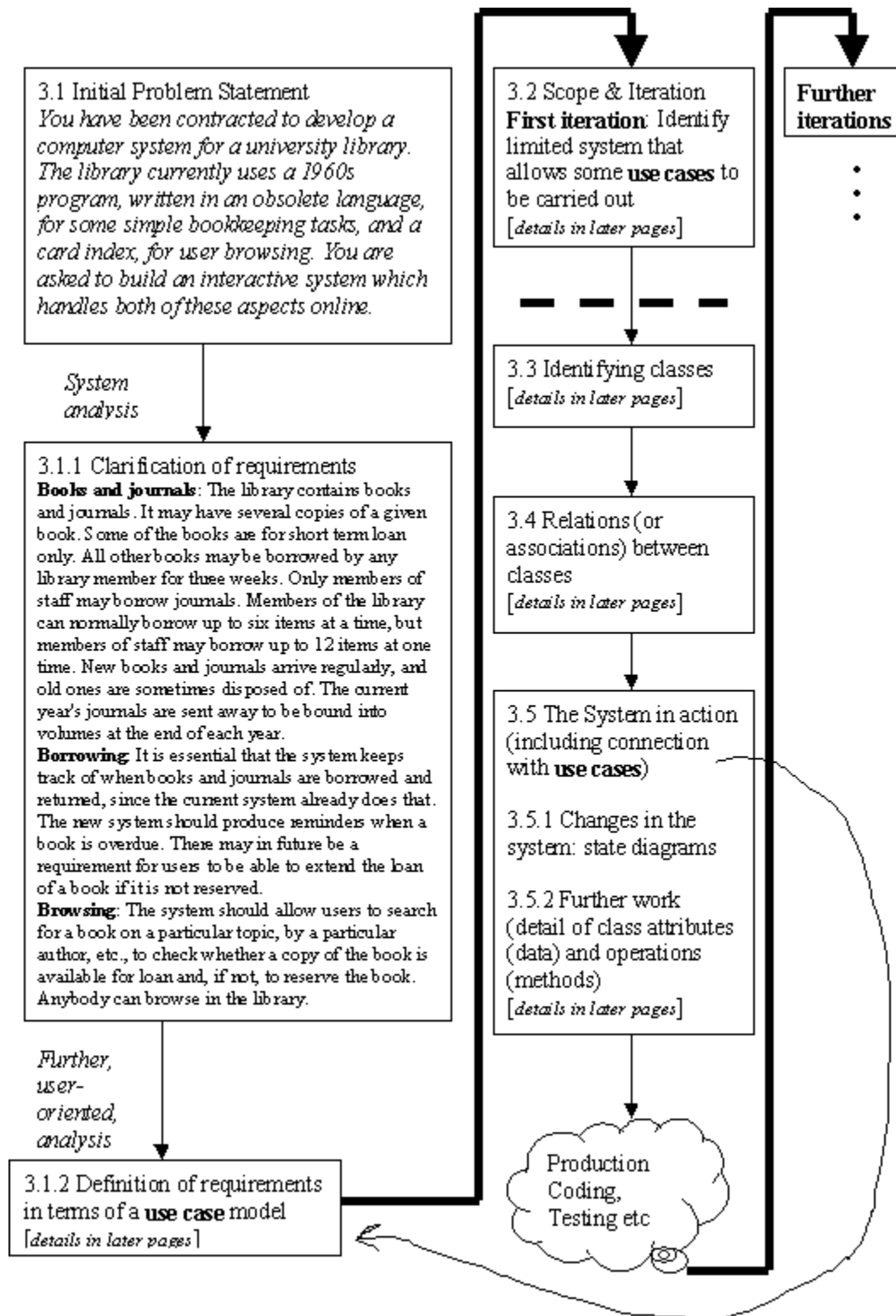


Figure 2: Overview of Introductory Case Study

Notes on different stages in the analysis and design of our introductory case study (refer to figure 2):

• 3.1 Initial Problem Statement

An initial, general statement of this kind is essential to set out the broad nature of the system desired. However, it is much too general and vague, and so must be analysed and refined in a dialogue between prospective users and software developers.

The points noted in the text (pp27, 28) are important:

- Different users have different priorities
- Users may not have clearly expressed views of what they want (& so should be engaged with to elucidate these views)
- A description alone of the system may miss vital aspects (in addition it may help to mock up some aspects, to look at similar systems, etc)
- Make sure that real users not just managers are included

The following extract on "Capture of user requirements" from the ESA SW standard is a good summary of the process:

"While user requirements originate in the spontaneous perception of need, user requirements should be clarified through the criticism and experience of existing software and prototypes. The widest possible agreement about the user requirements should be established through interviews and surveys. The knowledge and experience of the potential development organisations should be used to advise on implementation feasibility, and, perhaps, to build prototypes. User requirements definition is an iterative process, and requirements capture activities may have to be repeated a number of times before the URD is ready for review."

Remark: Notice that there are different "voices" in the above extract. It is mainly written perhaps from the point of view of the "acquirer" or customer. Other distinct "stakeholders" are the "users" and "developers".

- 3.1.1 Clarification of requirements (*refer to figure 2*)

During this first phase of analysis, emphasis is on understanding what the user problem is, in teasing out terminology, in making statements as unambiguous as possible, in defining terms clearly, trying to make sure terms are used consistently, and so on.

The outcome of the analysis (*see Figure 2 for full 'outcome'*)

<p>Books and journals: The library contains books and journals. ... Borrowing: It is essential that the system keeps track of ... Browsing: The system should allow users to</p>

is not yet a clear statement of requirements but several salient facts and constraints have been established. Also, functions that the system must provide have been defined and key "objects" that the system is concerned with have been identified. The problem now is to produce a clear *synthesis* of these "findings".

<p>The approach adopted in UML to produce this synthesis is "user-oriented", identifying</p>
--

- the users of the system

and

- the tasks these users must undertake with the system

Remark: This focus of UML is consistent with, for example, the ESA SW standard on "Determination of operational environment":

"Determining the operational environment should be the first step in defining the user requirements. A clear account should be developed of the real world in which the software is to operate. This narrative description may be supported by context diagrams, to summarise the interfaces with external systems (often called 'external interfaces'), and system block diagrams, to show the role of the software in a larger system."

3.1.2 Definition of requirements in terms of a **use case** model

(Fig. 2 gives context)

	UML terminology	
users of the system	actors	<i>persons or things external to the system, but which interact with it</i>
tasks users must undertake with the system	use cases	<i>tasks which an actor needs to perform with the help of the system</i>

Figure 3: Basic elements of 'use case' models

• Use cases:**Borrow copy of book**

A Bookborrower presents a book. The system checks that the potential borrower is a member of the library, and that s/he does not already have the maximum permitted number of books on loan. This maximum is 6 unless the member is a staff member, in which case it is 12. If both checks succeed, the system records that this library member has this copy of the book on loan. Otherwise it refuses the loan.

use case name

*use of third person
and active voice is
recommended*

Figure 4: Example of a use case description in "formal" English

Note: See text (page 29) for some remarks on "user interfaces" - not going to look at these in this module.

Remarks:

(1) Compare the above style of use case description with the style used in the illustrative software requirements (from a real project) presented earlier.

(2) When writing down software requirements, one should attempt to make sure that they are **consistent**, **complete** and **verifiable**. Use case descriptions should also have these three properties.

Question: Is the above use case verifiable? How would you verify it in the implemented system?

(3) CASE tools can be a big help in ensuring **consistency** and **completeness** of use cases (and of requirements). Such tools can range from fairly simple, in-house tools to large-scale tools from commercial suppliers.

N.B.: It is very important to check such tools thoroughly before use on an actual project and to document any limitations they may have.

• Use case diagrams:

As noted in text (p30) "use case diagrams" have been devised to represent use cases and such diagrams can be a very good aid, especially, in noticing *inconsistencies* between use cases. Also, they can form a good, understandable means of communication between users and developers; in particular, they can make it easier for "*incompletenesses*" (things missing) to be noticed.

Case tools are available to "make" use case diagrams, to share them and to check for consistency (to some extent at least).

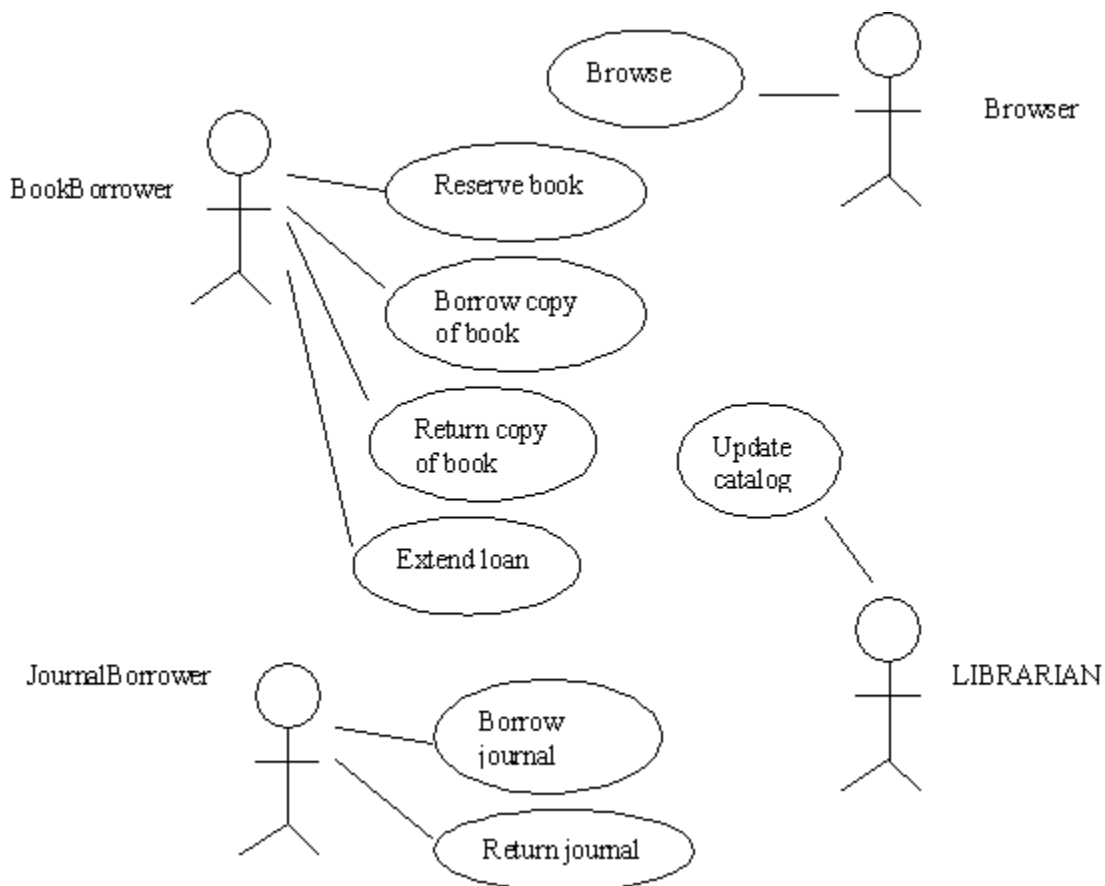
Example:

Figure 5 (Fig. 3.1 of text): Use case diagram for the library

- The notation is self-explanatory: stick figures represent actors, ovals represent use cases, and there is a line between an actor and a use case if the actor may take part in the use case.

Remark: We mentioned before that in software engineering in general it is advised that "*A clear account should be developed of the real world in which the software is to operate. This narrative description may be supported by context diagrams, to summarise the interfaces with external systems ...*". The above use case diagram (Figure 5) is an example of such a context diagram.

Note the following point made in text book re 'use case' diagrams:

- Beware of making diagrams very complex
- If becoming too complex can
 - either* -- Split Diagram
 - and/or* -- Use higher level of abstraction

For example, the text cites the use case "Update Catalog":

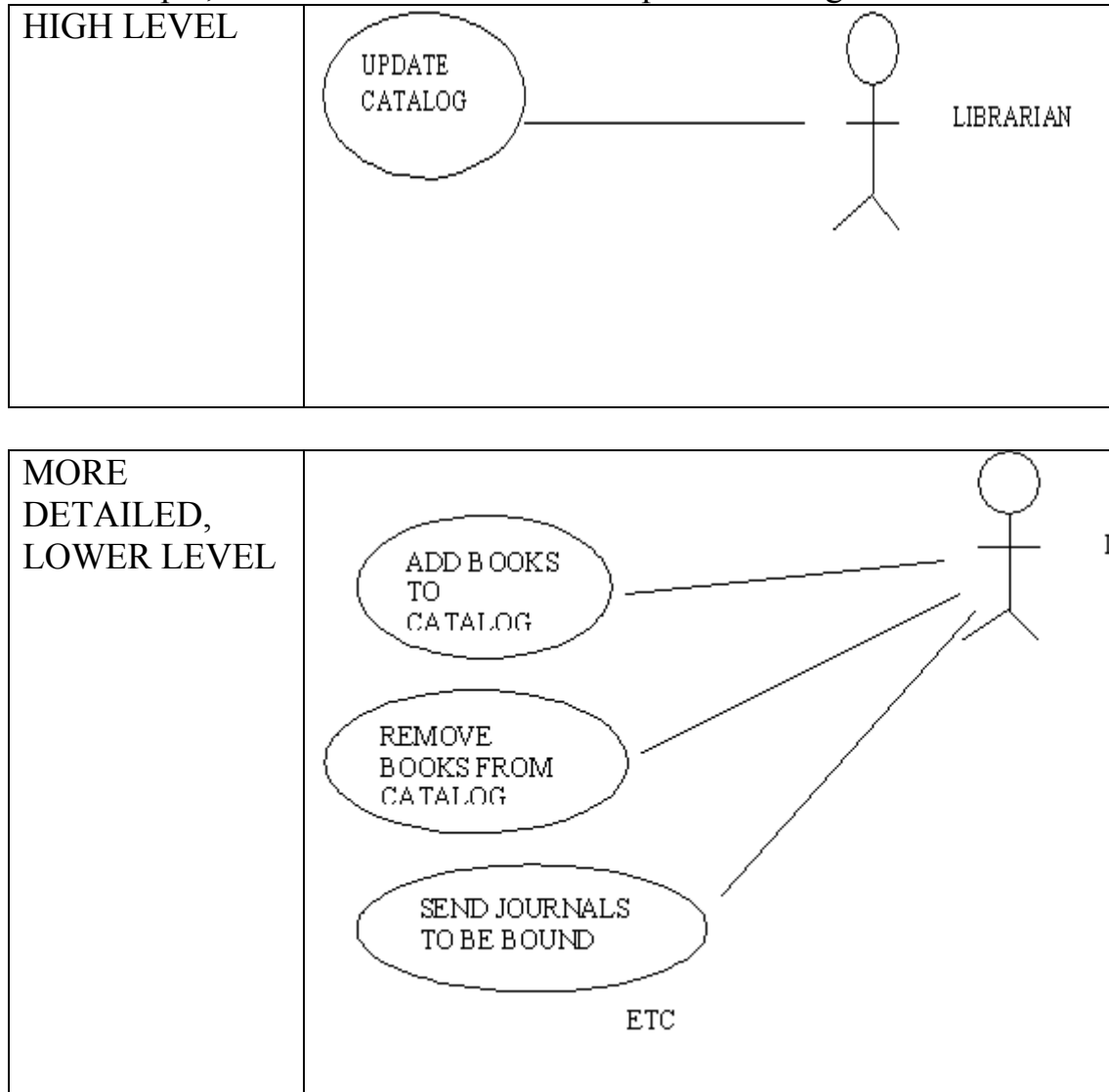


Figure 6: Example of avoidance of over complex use case diagram

- **Documentation of use cases:** We have one example "BORROW COPY OF BOOK". The other use cases identified should be described similarly.

- **Other points to note from text:**

- 1) In defining user requirements, including use cases in particular, focus on identifying *what the system should do* and not on how it should do it.

- 2) Do not invent requirements: Main point is to avoid confusing what the system *must* do (because the customer says so) and things that it might be nice to have.

- *In this regard, the text's suggestion of making lists of questions and possibilities for discussion with users (or with the customer) is a good one.*

- *Would expect to have questions anyway to resolve unknowns, lacks of clarity etc.*

- *Should bear in mind when suggesting "nice to haves" that it is up to the customer to pay for them - be wary of going ahead without customer's explicit agreement.*

3.2 SCOPE & ITERATION (*refer to figure 2*)

- See text for discussion of why delivering software in iterations rather than a "big-bang" is usually a good approach.

- (HOMEWORK - see Q19 of text p31): Draw up a table of advantages and disadvantages of iterated development:

Advantages of iterative approach	Disadvantages of iterative approach

- Selection of 'use cases' to be provided in first iteration: This is a good approach to describing the scope of each iteration.

- For comparison we note that, among the attributes that user requirements should have, the ESA standard includes *priority*:

"For incremental deliveries, each user requirement shall include a measure of priority so that the developer can decide the production schedule"

Example: Limited use case diagram selected for the first iteration:

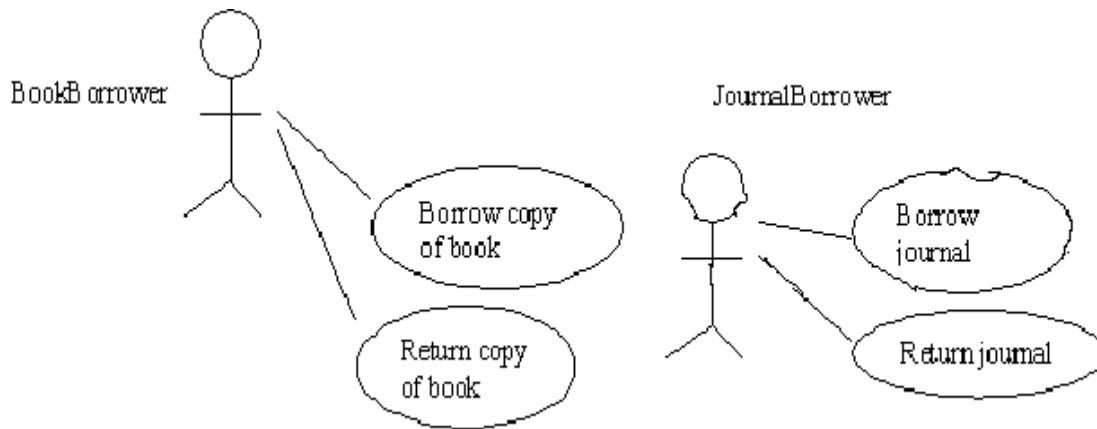


Figure 7 (Fig. 3.2 of text): Use case diagram for the first iteration

Complementing this limited diagram, text provides a re-statement of the requirements in which material irrelevant to the first iteration is omitted:-

Books and journals: The library contains books and journals. It may have several copies of a given book. Some of the books are for short term loan only. All other books may be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of staff may borrow up to 12 items at one time. Only members of staff may borrow journals.

Borrowing: The system must keep track of when books and journals are borrowed and returned, enforcing the rules described above.

Note: In a real project, the diagram and statement of requirements would probably be part of a document. It would be a project decision as to whether to produce a series of documents corresponding the series of iterations, or whether to have a single document covering all iterations. In the latter case, the particular requirements and use cases implemented in each iteration would have to be tracked in some way.

3.3 IDENTIFYING CLASSES (*refer to figure 2*)

Essentially, this is our first (very provisional) step in the (OO) design process. We have prepared the ground to some extent by our careful clarification of requirements and formulation of "use case" descriptions, including clear use of terms.

- Note specifically the process of identifying classes in text (pp32, 33):

DOMAIN = APPLICATION AREA (here the LIBRARY)

KEY DOMAIN Key aspects of the DOMAIN that are important

ABSTRACTIONS = to the proposed system

⇓ (*leads on to*)

CLASSES

- The method used in this chapter ("noun identification technique") is to take the clarified statement of requirements (or, alternatively, the use case descriptions) and to underline "nouns" and "noun phrases".

Books and journals: The library contains books and journals. It may have several copies of a given book. Some of the books are for short term loans only. All other books may be borrowed by any library member for three weeks. Members of the library can normally borrow up to six items at a time, but members of staff may borrow up to 12 items at one time. Only members of staff may borrow journals.

Borrowing: The system keeps track of when books and journals are borrowed and returned, enforcing the rules described above.

- This yields candidate classes: book, journal, copy (of book), library member, member of staff

- Notice the reasons given for discarding things:
 - outside scope (*library*)
 - event, not a thing (*loan*)
 - redundant (same as something else) (*Member of library*)
 - Too vague (*item*)
 - A measure, not a thing (*week*)
 - Not part of domain (*system, rules*)

[NB: *These are indications, not hard and fast rules*]

- Further notes on text (page 34):

- See later for CRC [Class Responsibility Collaboration] cards..

- We have some objects that represent users & need to decide what behaviour such objects will have. The technique in this example is to make the system objects representing actors responsible for carrying out actions on behalf of those actors.

Example given in text is of message "borrow(theCopy)" being sent to the "LibraryMember" object that represents the member wishing to borrow. Then, the "LibraryMember" object is responsible for carrying out [or for causing to be carried out] whatever is done to record (or deny) the loan.

- Identification of classes and objects is not an exact science - will provide more guidance later on.

Note: Authors state (p34) that we are not yet trying to design the system [*still mainly focussed on identifying the important real-world objects within the domain of the system*]. This means that we don't yet expect or need to get everything absolutely right.

3.4 RELATIONS BETWEEN CLASSES (*refer to figure 2*)

- We are thinking of *objects* as being implemented by *classes*
- Identify & name real-world relationships (associations) between classes in order to

<p><i>Clarify understanding of domain by describing how objects work together</i></p>	<p><i>Sanity-check coupling</i> in the end system - make sure good modularity principles are observed</p> <p>In particular, if one object is closely related to another then is probably OK for the class that implements one to depend on class that implements the other</p>	
	<p>1. May help maintainer to anticipate dependencies</p>	<p>2. May help re-use - an application that reuses one of the classes will probably reuse the other too</p>

- The structure of an OO system should reflect the structure of reality [e.g. "It is important that the system's model of the problem domain, and processes to be carried out, is compatible with the user's "model". ***This is because it is commonly found that domain objects change less frequently and dramatically than the exact functionality the user requires.***"]

As an instance of this reflection of reality, from our case study candidate classes,

book, journal, copy (of book), library member, member of staff we have

- a copy *is a copy* of a book
- a library member *borrow/returns* a copy
- a member of staff *borrow/returns* a copy
- a member of staff *borrow/returns* a journal

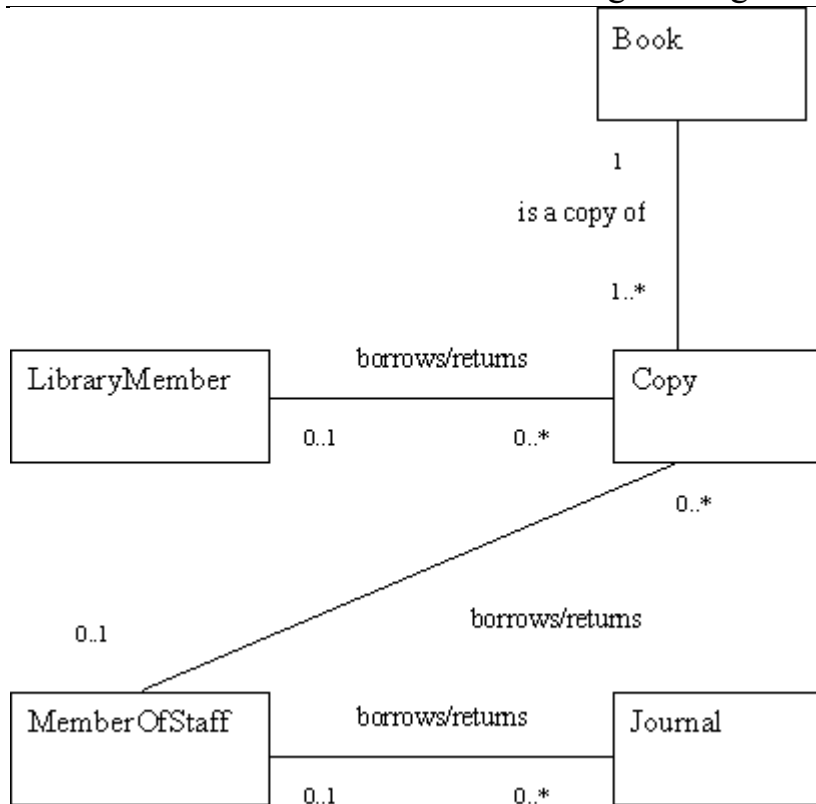


Figure 8 (Fig. 3.4 of text) Initial class model of the library

The above (UML class model) depicts the identified relations:

- It also shows **multiplicities** of the associations. For example, the fact that each copy is a copy of only one book is indicated by putting "1" at the book end of the relation "is a copy". On the other hand, the fact that there may be one or more copies of a given book is indicated by putting "1..*" at the copy end of the relation "is a copy". [*More details of this later*]

- Our diagram does not indicate which class depends upon (or "knows about") which. This is referred to the "navigability of the associations". The direction of navigability impacts on the degree of coupling.

- We can improve our diagram by reflecting the fact that a `MemberOfStaff` is a special kind of `LibraryMember`:

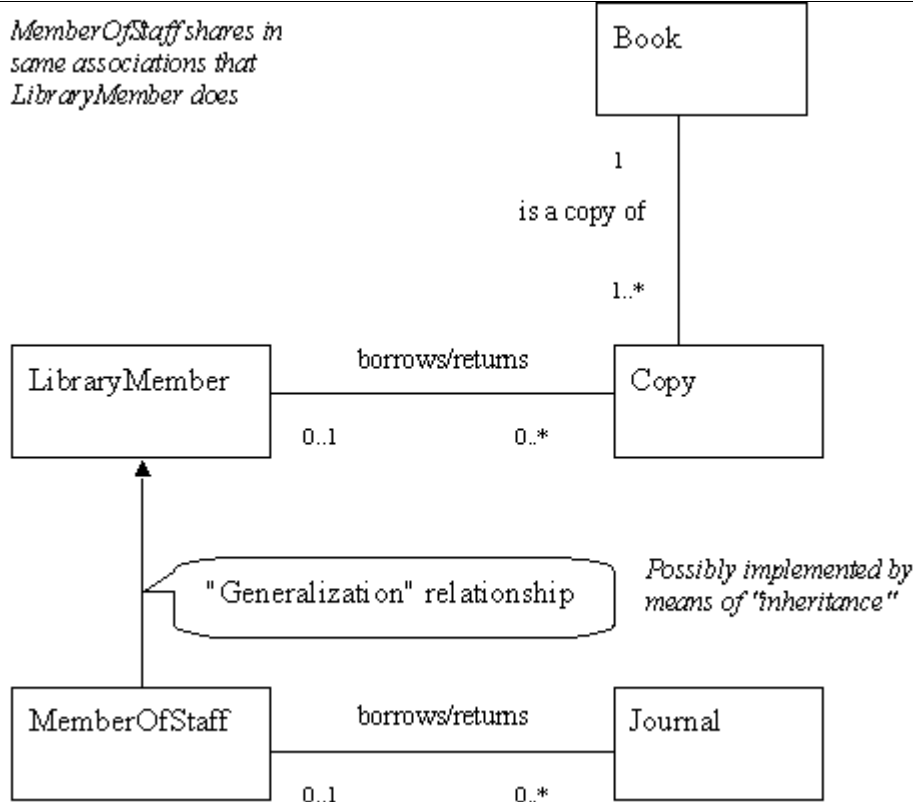


Figure 9 (Fig. 3.5 of text) Revised library class model

3.5 The System in action (*refer to figure 2*)

- We want next to make a connection between

Use Cases

we started with



Objects

we have decided make up the system

Sequence diagrams in UML can be used to show the interaction involved, i.e. how messages pass between objects of the system, in carrying out a task such as a use case. For example, for the use case "Borrow copy of book" we have

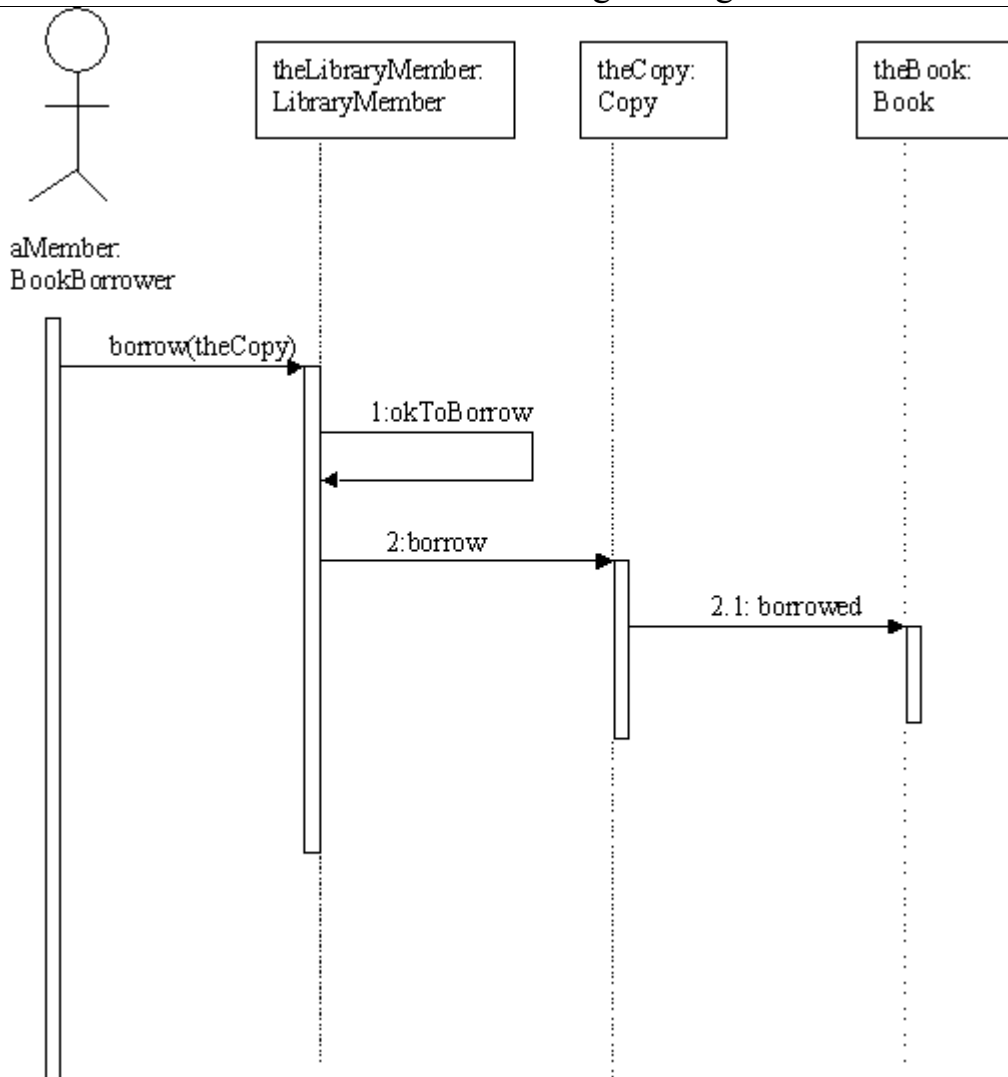


Figure 10 (Fig. 3.6 of text) Interaction shown on sequence diagram

- Diagram shows which messages are passed between objects and in what order they must occur - read the messages from top to bottom
- The diagram shows what happens when a library member borrows a copy of the book. It starts from the position of having a certain object of class "LibraryMember" called "theLibraryMember" and a certain object of class "Copy" called "theCopy" (corresponding to a person bringing a physical copy to the issue desk to borrow it)
- "theLibraryMember" acts (as decided earlier for this example) on behalf of the real library member so interaction begins with a message "borrow(theCopy)" passed to it
- Then, after a check if allowed to borrow, object "theLibraryMember" sends the message "borrow" to "theCopy". Finally, "theCopy" sends a message "borrowed" to "theBook" - we need to update system information on how many copies are on loan.

3.5.1 Changes in the system: state diagrams (*refer to Figure 2*)

• In our case study, if a *copy of a book* is borrowed, then the **state** of the *book* in question may change from "borrowable" to "not borrowable". A state (or state transition) diagram can be used to show this:

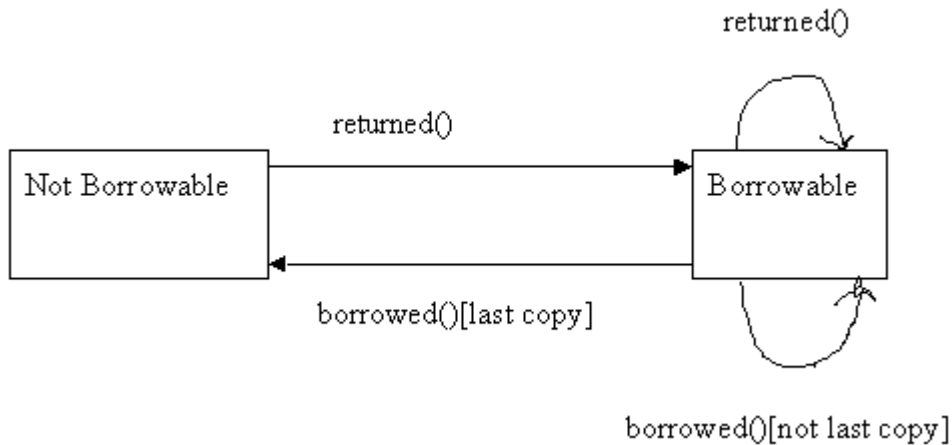


Figure 11 Example of a state diagram

- State transition depends on the number of copies in the library
- Concept of "state" is by no means unique to OO or UML
- Instead of a diagram we can use a state transition table (also called state transition matrix).

For example,

Destination Origin	NOT Borrowable	Borrowable
NOT Borrowable	-	Pre: - (<i>implicit?</i>) Action: returned() Post: Borrowed = Total -1
Borrowable	Pre: Borrowed = Total - 1 Action: borrowed() Post: Borrowed = Total	Pre: Borrowed < Total - 1 Action: borrowed() Post: Borrowed < Total
		Pre: - (<i>discuss!</i>) Action: returned() Post: Borrowed decreased by 1

where

Borrowed = No. of copies borrowed

Total = Total number of copies in library

- = Not applicable

Conclusion to case study (thus far):

Once we have identified how all the use cases are realised (as per Figure 3.6 of text - above), it is fairly straightforward to implement the classes. The result is the first iteration of the system, which can be verified by developers and validated by the users (misunderstandings? failings? etc). Thereafter, further iterations can be made, to get closer to the ideal.