

CA314 – Object Oriented Analysis & Design - 2

File name: CA314_Section_02_Ver01

Author: W.G. Tuohey

No. of pages: 23

Table of Contents

2. Introductory Case Study3
2.1 Some context.....3
 2.1.1 A traditional view of Requirement Specification & Design Processes3
 2.1.2 Software Requirement Specifications – A few examples.....4
2.2 Overview of the Introductory Case Study5
2.3 Initial Problem Statement (3.1 of reference 1).....7
2.4 Clarification of requirements (3.1.1 of reference 1)7
2.5 Definition of requirements via a **use case** model (3.1.2 of reference 1).....8
2.6 Scope & Iteration (3.2 of reference 1)12
 2.6.1 Selection of 'use cases' for first iteration of the case study13
2.7 Identifying Classes (3.3 of reference 1).....13
2.8 Relations between Classes (3.4 of reference 1)15
2.9 The System in Action (3.5 of reference 1).....17
2.10 Changes in System: State Diagrams (3.5.1 of reference 1)19
2.11 Conclusion to case study (thus far)20
2.12 Exercises20

2. Introductory Case Study

2.1 Some context

2.1.1 A traditional view of Requirement Specification & Design Processes

To begin with, it is useful to have a glance at a fairly standard, traditional approach to specification and design, in terms of the tasks done and the outputs produced. This will provide a context, and a basis for comparison, as we introduce UML. In the following Figure 2.1-1 tasks and outputs are represented by ovals and boxes, respectively; this is not UML notation. You should look back at Section 1.4 for comparison.

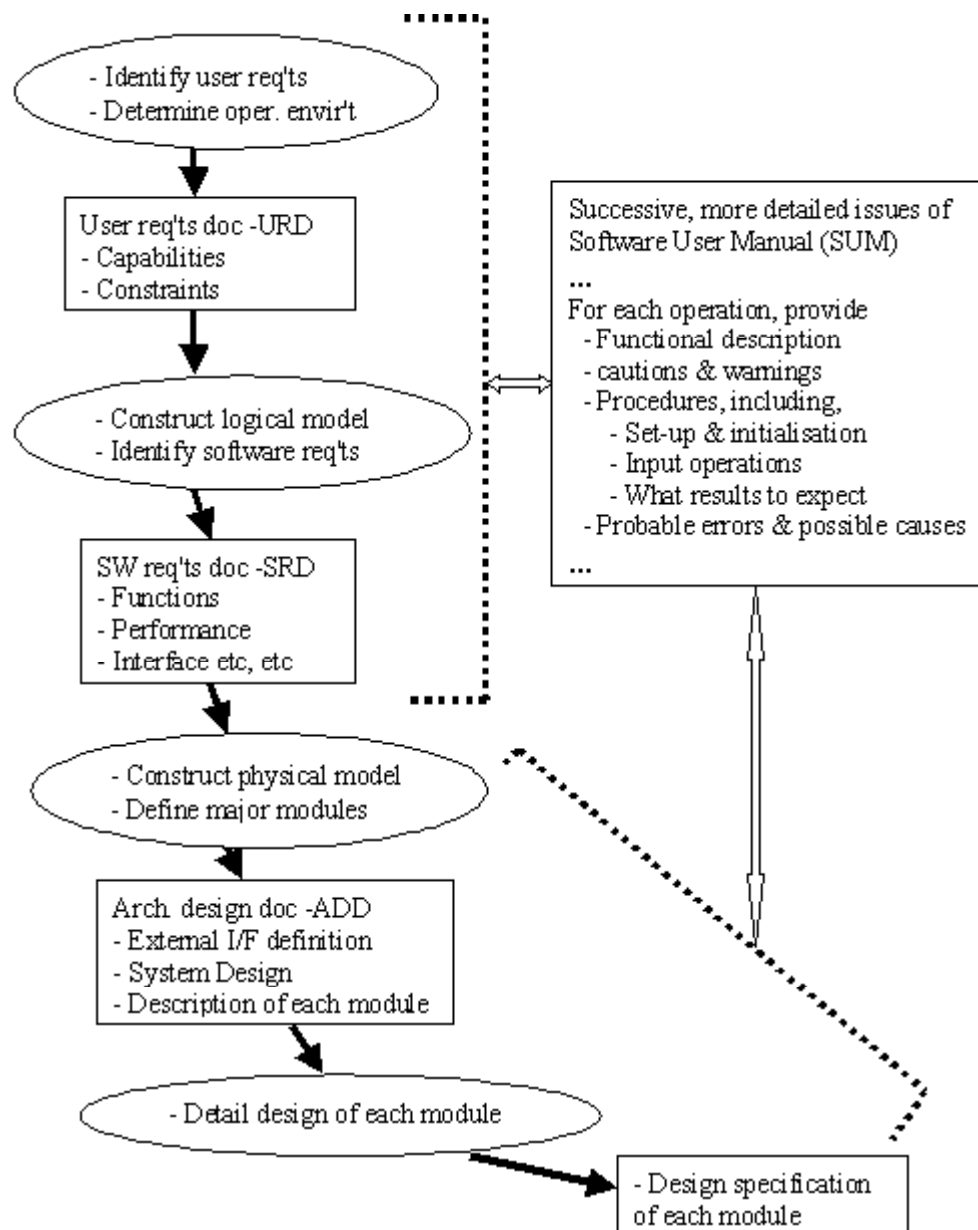


Figure 2.1-1: A "standard" view of Specification & Design processes

2.1.2 Software Requirement Specifications – A few examples

Example 1: 2 functional, 1 performance and 1 maintainability requirement

These requirements are taken from the “Software Requirement Document” of a real project. This style of specification is intended as an illustration and should not be taken to be the only or best way of writing requirements. Because it is from a real project, there is some specialised terminology; for our purposes it is enough to note that TM and TC stand for telemetry and telecommand, respectively, and that the OBDH (*on-board data handling*) is an external system to the software being specified. In a sense, the OBDH is a user of the system (called the ACC) being specified.

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(unsuccesful),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,#

Example 2: From Wiegers (<http://www.processimpact.com/articles/qualreqs.html>) Karl E. Wiegers presents some very good examples including the following one. The actual requirement is in italics and is then commented on. Then an improved statement is suggested.

“

"The HTML Parser shall produce an HTML markup error report which allows quick resolution of errors when used by HTML novices."

The word "quick" is ambiguous. The lack of definition of what goes into the error report is a sign of incompleteness. I'm not sure how you would verify this requirement. Find someone who calls herself an HTML novice and see if she can resolve errors quickly enough using the report?

Try this instead:

"The HTML Parser shall produce an error report that contains the line number and text of any HTML errors found in the parsed file and a description of each error found. If no errors are found, the error report shall not be produced."

Now we know what needs to go into the error report, but we've left it up to the designer to decide what the report should look like. We have also specified an exception condition: if there aren't any errors, don't generate a report.

”

Note: If you search on the internet for examples of software requirement specifications you will often find that use cases are involved. This reflects the fact that use cases are often used as a means of specifying at least the functional requirements of systems.

2.2 Overview of the Introductory Case Study

This case study follows the presentation of reference 1 (see section 1.3) very closely. To make correspondence easy, the section numbering of that reference is retained in the following text. First, the following Figure 2.2-1 outlines how reference 1 presents the case study.

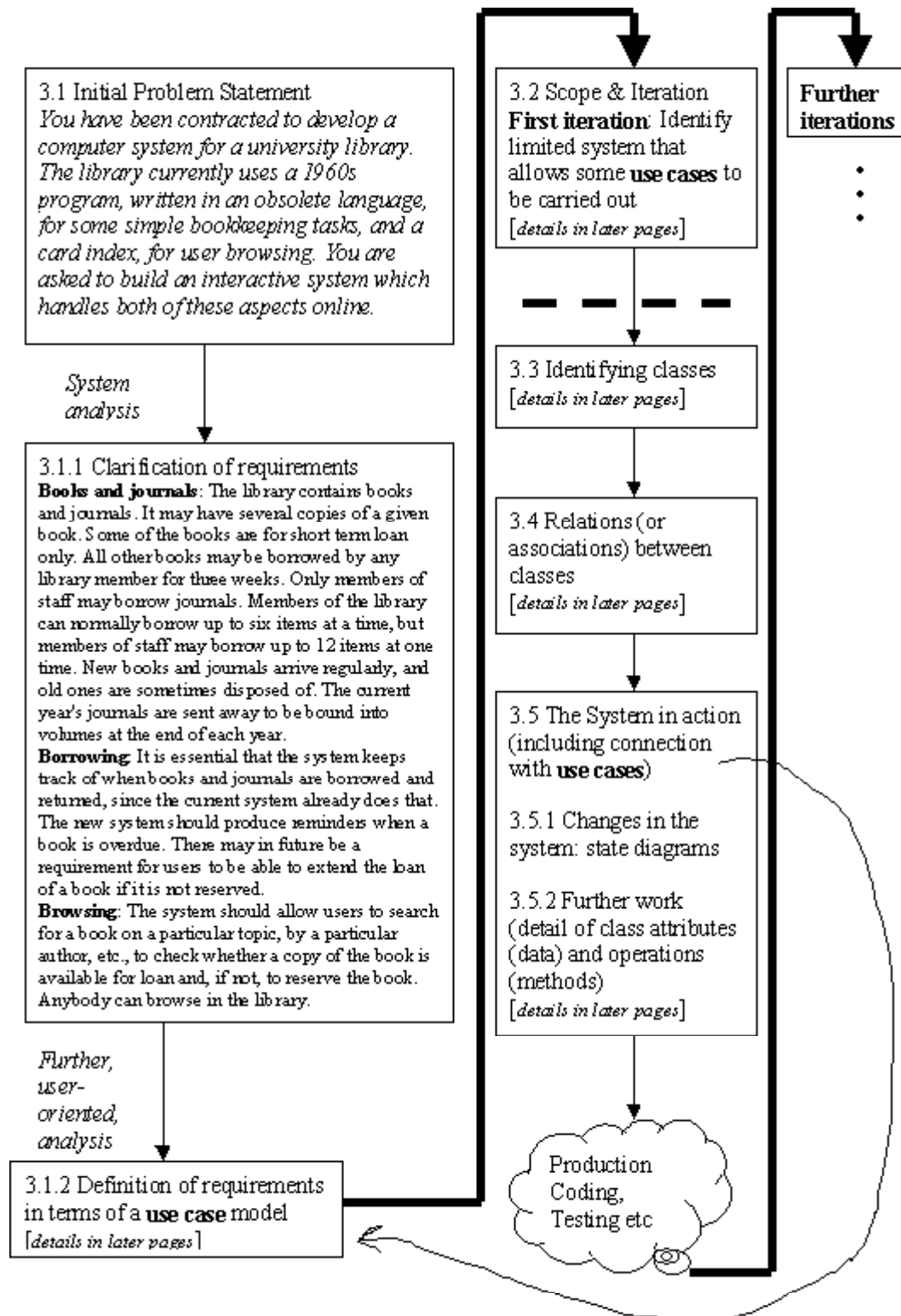


Figure 2.2-1: Overview of Introductory Case Study

The following sections provide notes on the different stages in the analysis and design as identified in this figure.

2.3 Initial Problem Statement (3.1 of reference 1)

An initial, general statement of the kind in Figure 2.2-1 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 reference 1 (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 an 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".

2.4 Clarification of requirements (3.1.1 of reference 1)

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 2.2-1 for the detail)

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

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".

The approach adopted to produce this synthesis, when using UML, is "user-oriented", identifying

- 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."

2.5 Definition of requirements via a **use case** model (3.1.2 of reference 1)

Refer to Figure 2.2-1 for the context.

The basic elements of a use case model are depicted in Figure 2.5-1:

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

Figure 2.5-1: Basic elements of 'use case' models

Figure 2.5-2 provides an example of a use case and its description (from reference 1):

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 2.5-2: Example of a use case description in "formal" English**

Remarks:

- (1) See reference 1 (page 29) for some remarks on "user interfaces"; largely outside the scope of this module.
- (2) Compare the above style of use case description with the style used in the illustrative software requirements (from a real project) presented earlier (section 2.1.2).
- (3) 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 for home work: Is the above use case verifiable? How would you verify it in the implemented system?

- (4) CASE (computer aided software engineering) 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.

As well as looking at individual use cases, we need to have a view of all (or at least related groups of) the use cases. As noted in reference 1 (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). Figure 2.5-3 is an example of a use case diagram depicting all the use cases and actors identified in reference 1 for our case study.

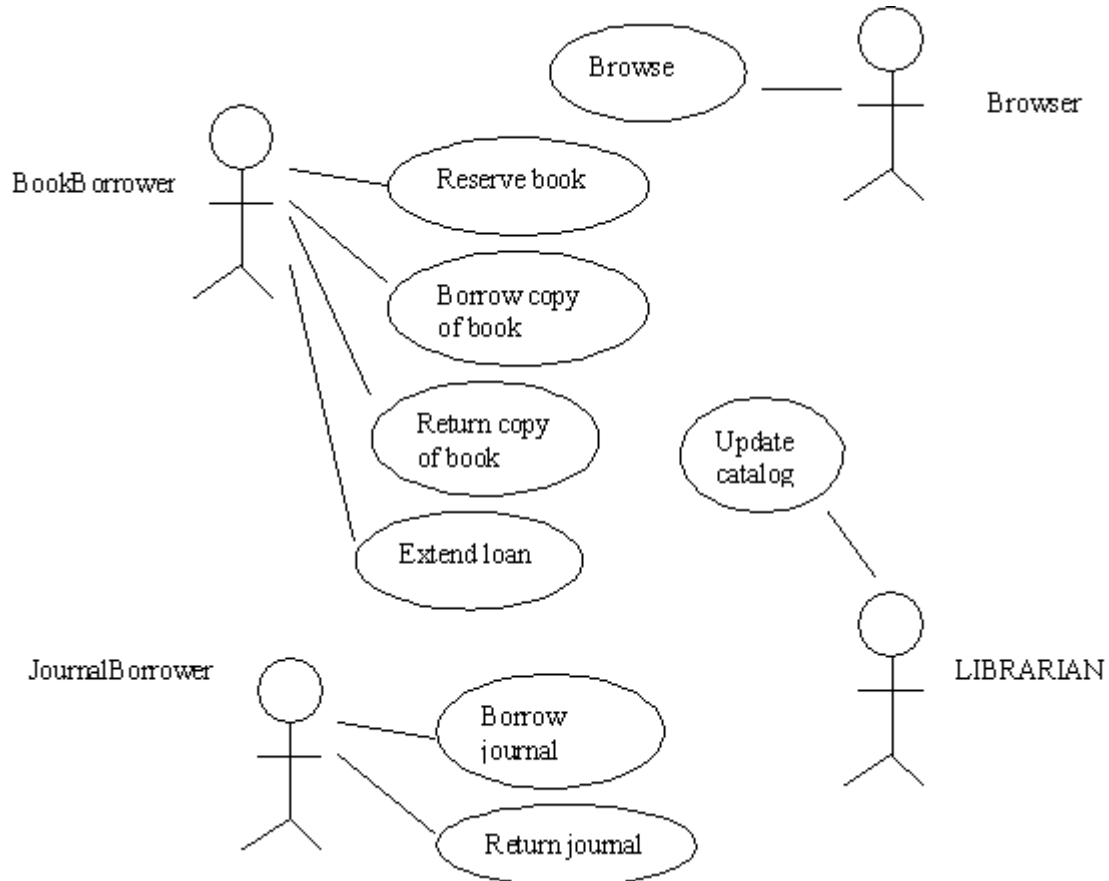


Figure 2.5-3: Use case diagram for the library (Fig 3.1 of reference 1)

- 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 (section 2.4) 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 is an example of such a context diagram.

Note: The following important points are made in reference 1 regarding 'use case' diagrams:

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

For example, reference 1 cites the use case "Update Catalog":

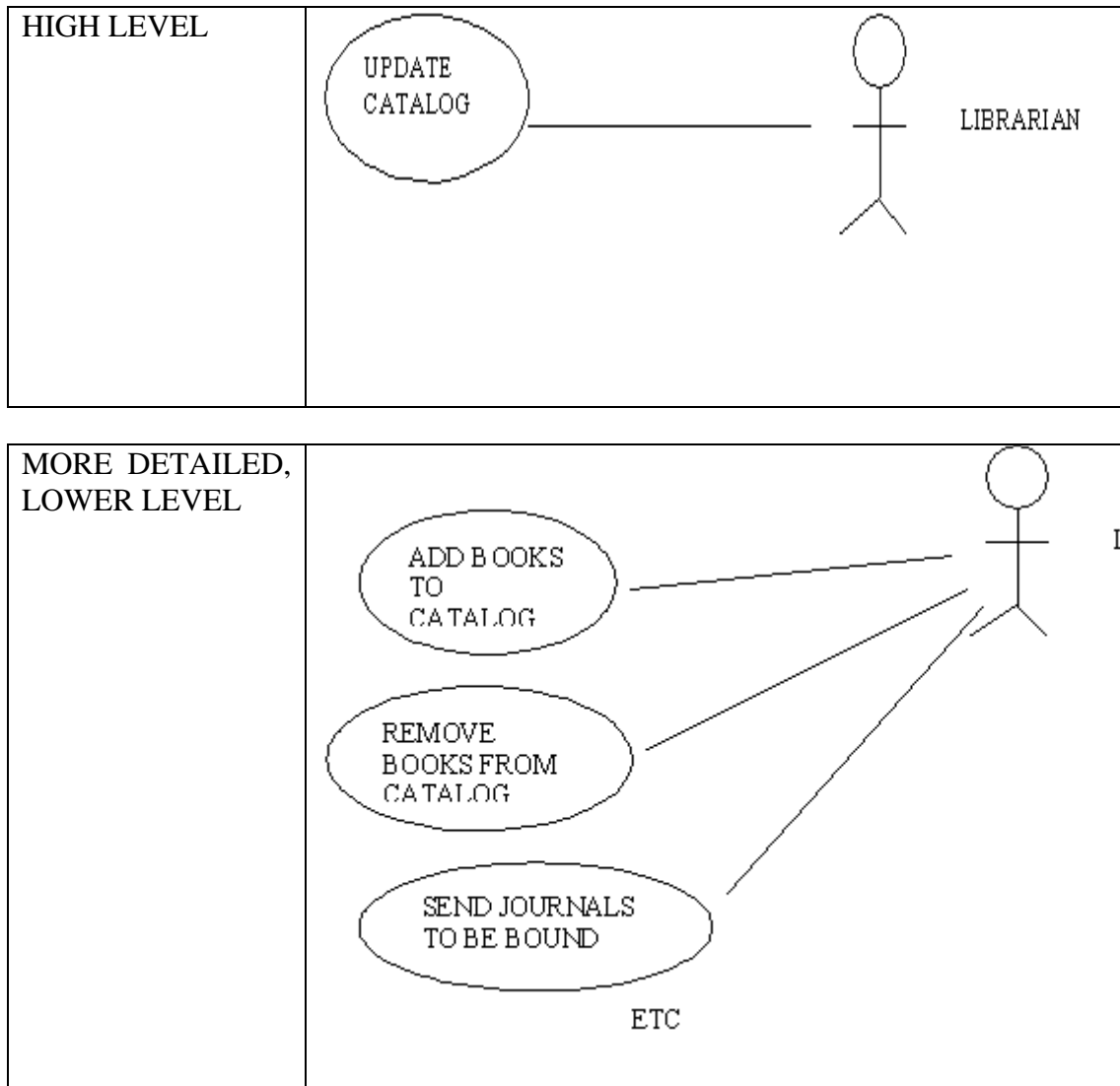


Figure 2.5-4: Example of avoidance of an over complex use case diagram

We have had one example, "BORROW COPY OF BOOK", of a use case description (Figure 2.5-2). The other use cases identified in the use case diagram must be described similarly. Then, the use case diagram(s) together with all the use case descriptions make up our documentation for the use cases. We will see later than, sometimes, use case descriptions are more formalized but there is no hard and fast rule.

The following points of guidance should also be noted:

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: The 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, reference 1's suggestion of making lists of questions and possibilities for discussion with users (or with the customer) is a good one.

- One would expect to have questions anyway to resolve unknowns, lacks of clarity etc.

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

2.6 Scope & Iteration (3.2 of reference 1)

Very often, delivering software in iterations rather than as a single "big-bang" is a good approach (see reference 1 for a discussion). If the use case approach is being used, then an obvious way of defining each iteration is to specify the corresponding use cases.

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"

In the same way, if use cases are being used in specifying requirements then one can assign a priority to each use case.

2.6.1 Selection of 'use cases' for first iteration of the case study

The following “subset” of Figure 2.5-3, incorporating the essential functionality of the system, seems appropriate as a first iteration:

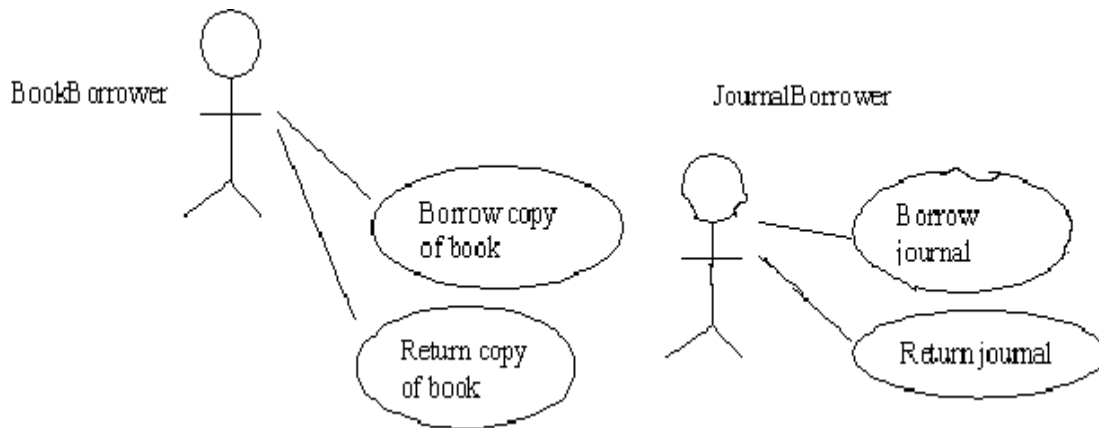


Figure 2.6-1: Use case diagram for first iteration (Fig. 3.2 of reference 1)

Complementing this limited diagram, reference 1 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.

Figure 2.6-2: Re-statement of a limited Requirement Spec. for 1st Iteration

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 to 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.

2.7 Identifying Classes (3.3 of reference 1)

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.

The following figure depicts the process of identifying “domain-level” classes (reference 1, pp32, 33):

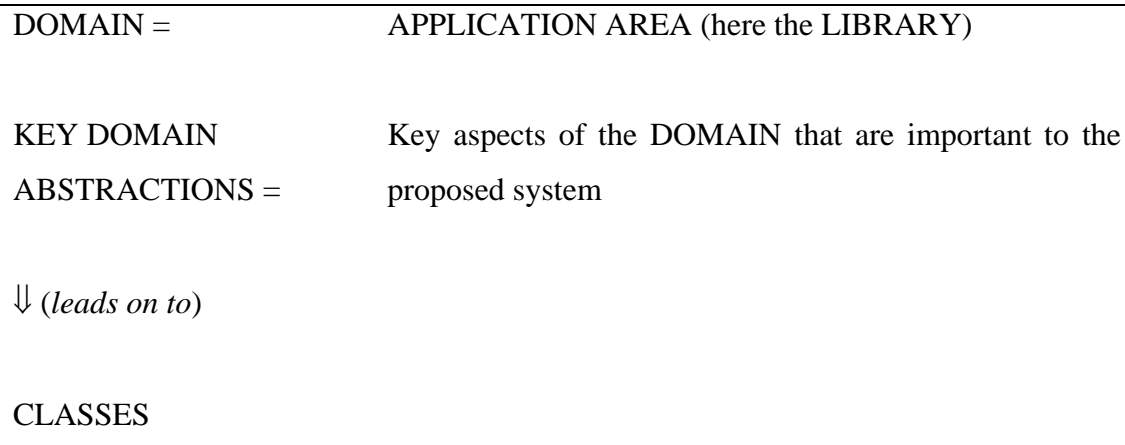


Figure 2.7-1: Identifying domain-level classes

There are a few specific methods and guidelines available to help in actually identifying candidate classes. The method used in the following, called the "noun identification technique", is to take the clarified statement of requirements Figure 2.6-2 (or, alternatively or in addition, 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.

Figure 2.7-2: "noun identification" – nouns and noun phrases underlined

This yields candidate classes: book, journal, copy (of book), library member, member of staff. However, some suggested items are discarded, as follows:

- 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*)

Thus, it should be understood that this method, including the discarding criteria, is an indicative guideline rather than a hard and fast rule.

A few other points mentioned in reference 1 (page 34) are worth noting:

(a) Class Responsibility Collaboration (CRC) cards: These cards provide another means of identifying classes, or of clarifying them if already identified. We will introduce them later and see how they can be used in this case study.

(b) Behaviour of Objects that represent Users (Actors): This note may be regarded as a design guideline or suggestion. We have some objects that represent users & *need to decide what behaviour such objects will have*; for example, “library member” and “member of staff” in the case study. In this case study, the decision is to *make the system objects representing actors responsible for carrying out actions on behalf of those actors*:

Example: The message "borrow(theCopy)" is sent to the "LibraryMember" object that represents the member wishing to borrow. Thus, the “LibraryMemory” class has an operation called “borrow()”. Then, the specific "LibraryMember" object is responsible for carrying out [or for causing to be carried out] whatever is done to record (or deny) the loan.

(c) Identification of classes and objects is not an exact science - more guidance will be provided later on.

(d) At this stage, the main focus is 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.

2.8 Relations between Classes (3.4 of reference 1)

At this stage, we are thinking of real-world *objects* as being implemented (or represented) by *classes*. Therefore, it is natural to identify relationships (associations) between these classes that arise from the real world.. This is a good approach for the reasons summarised in Figure 2.8-1:

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

Figure 2.8-1: Identifying real-world relationships between classes

Thus, the structure of an OO system should reflect the structure of reality. It is important that the system's model of the problem domain, and of the 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.*

Therefore, in our case study, we can identify expected relationships between the candidate classes, book, journal, copy (of book), library member, member of staff:

- 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

A UML class model, with its corresponding class diagram, is used to depict the identified relations:

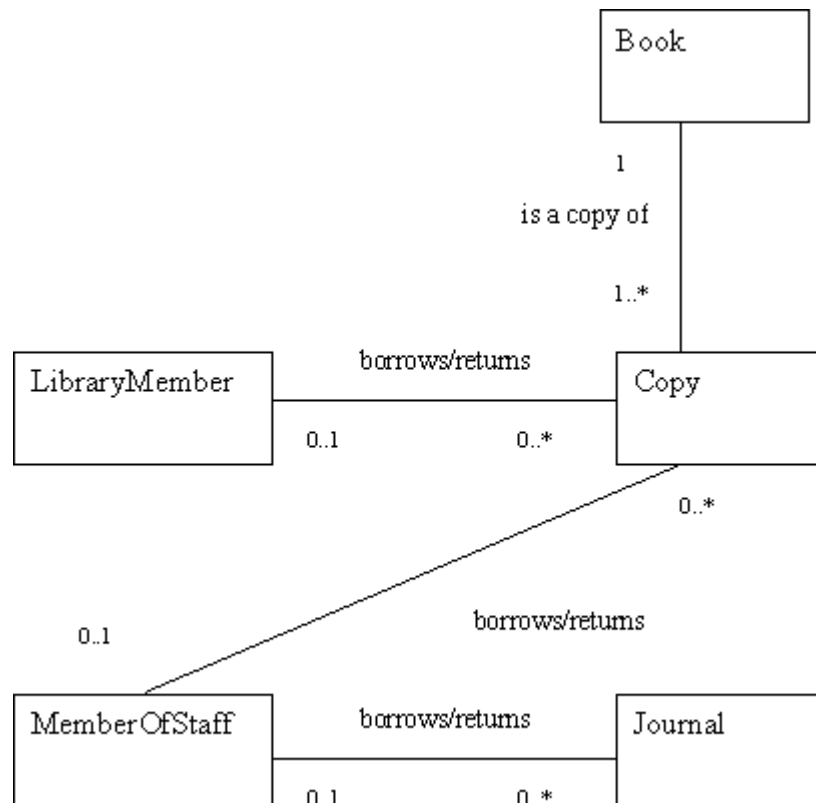


Figure 2.8-2: Initial class model of the library (Figure 3.4 of reference 1)

This 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. It will be decided on later.

Note: Figure 2.8-2 depicts **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 complete details of this syntax will be provided later.

The class diagram may be improved, or at least refined, by reflecting the fact that a MemberOfStaff is a special kind of LibraryMember:

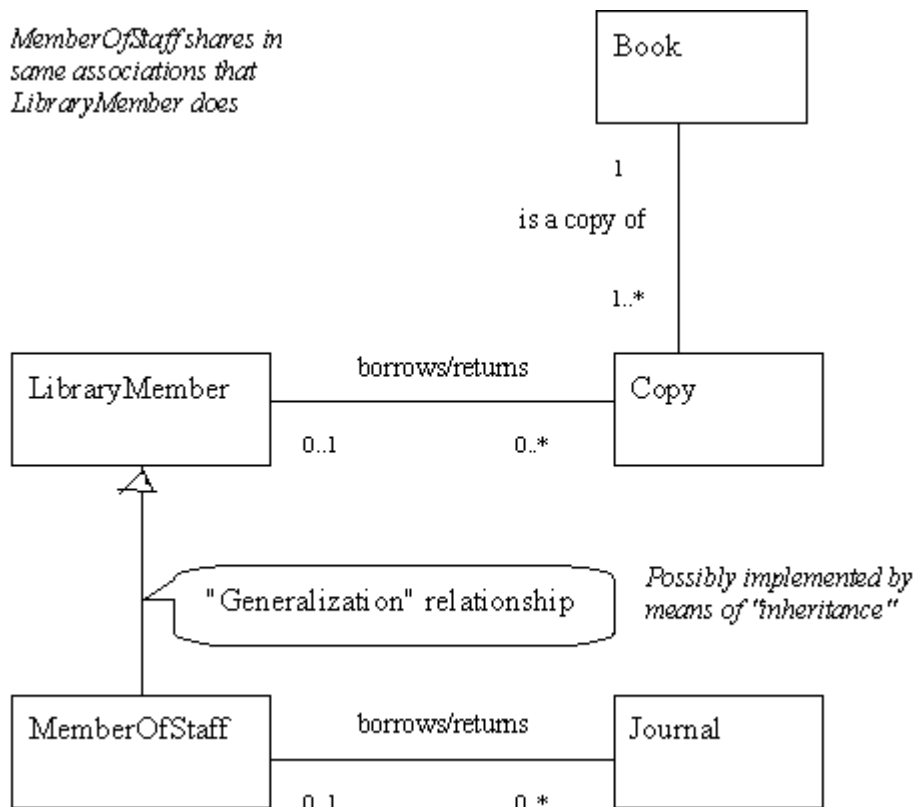


Figure 2.8-3: Revised library class model (Figure 3.5 of reference 1)

2.9 The System in Action (3.5 of reference 1)

Having developed both a use case model and an (initial) class model, we want to further develop our system description by making a connection between

Use Cases	Objects
we started with	we have decided make up the system

Figure 2.9-1: Linking Use Cases and Objects

In UML, sequence diagrams, or more generally interaction diagrams, can be used to show the interaction involved. (See Figure 1.4-1 "Convert use cases to sequence diagrams"). Specifically, we want to depict how messages pass between objects of the system in order to carry out each use case.

This means that we are forced to consider what messages are involved and how they should be named. Also, which objects send and receive which messages. Normally, in terms of implementation, an object receives a message when one of its operations is called. Similarly, an object normally sends a message when it calls an operation belonging to another object.

For example, for the use case "Borrow copy of book" we might have

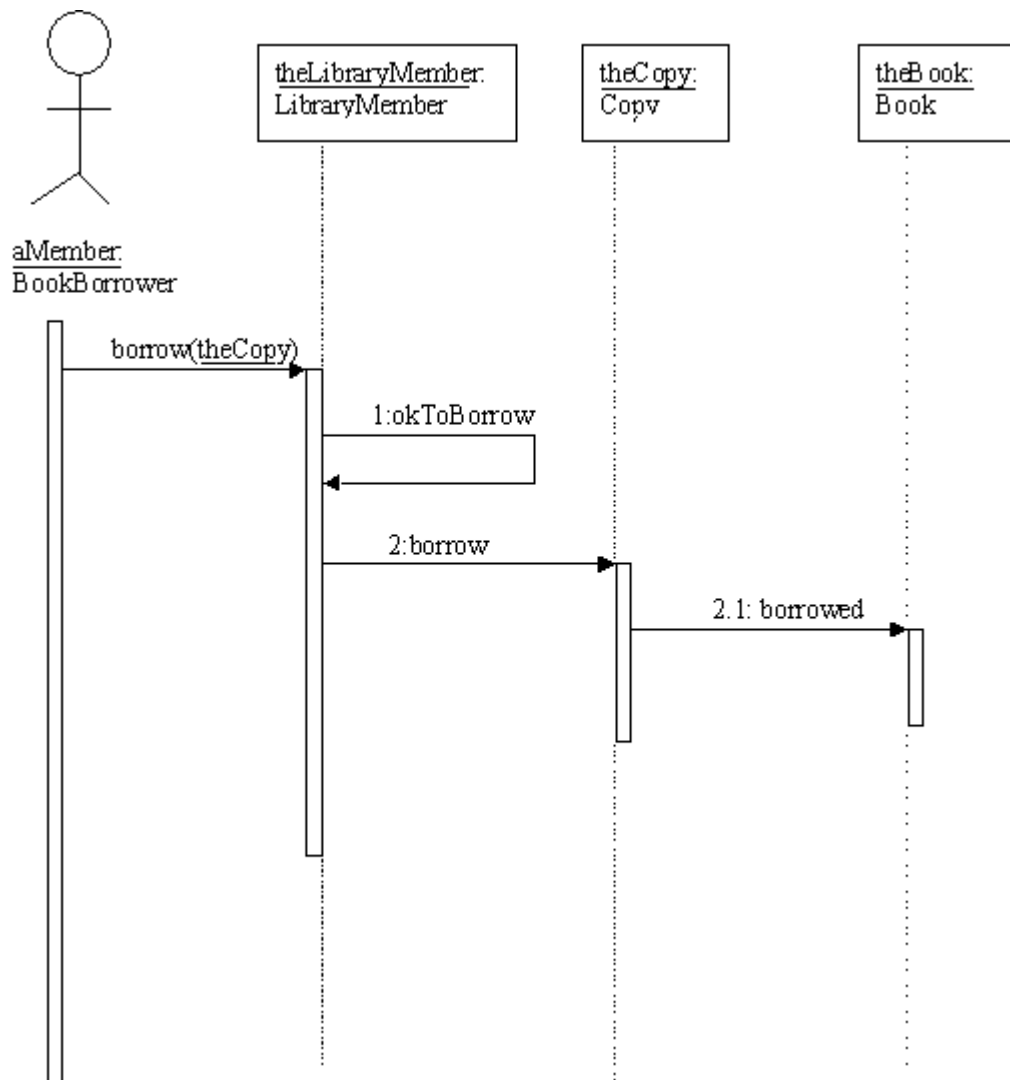


Figure 2.9-2: Interaction shown on sequence diagram (Figure 3.6 of text)

Figure 2.9-2 shows which messages are passed between objects and in what order they must occur – in sequence diagrams the order of message passing is from top to bottom.

This diagram presupposes that class LibraryMember contains the 2 operations “borrow(theCopy)” and “okToBorrow”, that class Copy contains an operation (also) called “borrow” and that class Book contains an operation called “borrowed”. We

have perhaps moved a bit quickly here to get to actual operations. We will re-visit the process for this example when discussing CRC cards.

Figure 2.9-2 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). Thus, "theLibraryMember" acts (see note (b) after Figure 2.7-2) 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" since we need to update system information on how many copies are on loan.

2.10 Changes in System: State Diagrams (3.5.1 of reference 1)

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 (or state chart) can be used to show this:

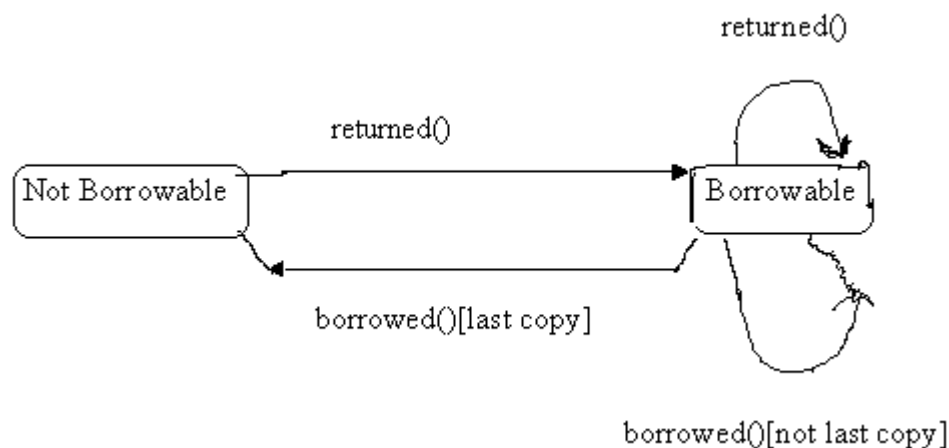


Figure 2.10-1: Example of a state diagram (for class Book)

Clearly, whether or not there is a transition to a different state depends on the number of copies in the library.

The concept of "state" is, of course, by no means unique to OO or UML. Sometimes the term "mode" is used in contexts where UML uses "state".

Though not usual for UML, it can sometimes be helpful to form a state transition table (or matrix) corresponding to a given state diagram. Though not as visually appealing, such a table forces one to make explicit aspects that are not made clear in the diagram.

For example, corresponding to Figure 2.10-1 we may form

Destination → Origin ↓	NOT Borrowable	Borrowable
NOT Borrowable	-	Pre: Borrowed = Total 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: Borrowed < Total - 1 Action: returned() Post: Borrowed decreased by 1

Table 2.10-1: Example of a state transition table (class Book)

where, say,

Borrowed = No. of copies of Book borrowed

Total = Total number of copies of Book in library

- = Not applicable

and “Pre” and “Post” refer to conditions applying before and after a transition.

2.11 Conclusion to case study (thus far)

Once we have identified how all the use cases are realised (as per Figure 2.9-2), it should be fairly straightforward to implement the classes. There is, of course, still quite a bit of work to be done, before going on to writing actual source code, in terms of precisely defining operations (e.g. methods in Java), attributes etc.

The result will be the first iteration of the system. This can be verified by the developers and validated by the users (with the aim for example, of identifying misunderstandings, gaps, etc).

Thereafter, further iterations can be made, to eventually achieve a satisfactory complete system (see Figure 2.2-1).

2.12 Exercises

1. Write use case descriptions for all the remaining use cases of Figure 2.5-3, in the style used for “Borrow copy of book” in Figure 2.5-2.

2. (see Q19 of reference 1, p31) With reference to section 2.6, draw up a table of advantages and disadvantages of iterated development:

Advantages of iterative approach	Disadvantages of iterative approach

3. In an accident management system, field officers, such as police officers or fire fighters, have access to a wireless computer that enables them to communicate with a dispatcher. In particular, they can report emergencies and can receive instructions.

On receipt of an emergency report, the dispatcher 'opens an incident'. The dispatcher can visualise the current status of all system resources, such as police cars or trucks, on a computer screen and can dispatch a resource by issuing commands from a workstation. Also, the dispatcher is aware of hospitals 'on-call' and can communicate with them. [*Object-Oriented Software Engineering*, Bruegge & Dutoit]

(a) Identify four Use Cases for this problem and provide a short (informal) description for each case. Also, present a corresponding Use Case diagram.

(b) Based on the given system description, and perhaps on your use case descriptions, identify candidate classes for the accident management system. Indicate very briefly the intended responsibility of each candidate class. Develop a corresponding Class Diagram, showing key associations between classes.

4. A simplified automatic teller machine (ATM) offers the following services (from *UML in Practice*, P. Roques):

(i) Distribution of money to every holder of a smartcard via a card reader and a cash dispenser.

(ii) Consultation of account balance, cash and cheque deposit facilities for bank customers who hold a smartcard from their bank.

Also,

(iii) All transactions are made secure.

(iv) It is sometimes necessary to refill the dispenser, etc.

(a) Identify four actors and six use cases of this ATM system.

(b) Construct a use case diagram for the system. Write summary textual descriptions for each of your use cases, noting any significant alternative scenarios that can arise.

5. A number of dentists work together co-operatively, and wish to acquire a software system for their group practice. They envisage that the system will support both their clinical work and management of the business. Thus, it will hold dental records for each of their patients enabling a dentist to review the dental history of a patient and to plan future interventions and procedures. On the other hand, it will maintain an inventory of the practice's equipment and material so that new purchases can be made

in good time and out of date stock can be removed. It will also keep track of financial records, particularly in terms of patients' accounts whether paid personally or through Government support mechanisms. Thus, it is foreseen that the system will be used in different ways by dentists, secretarial staff and accountants. The system must be highly reliable and, especially, must guarantee safe management of dental records because of the potential impact on patients' health.

(a) Identify four use cases for this Dental Practice System and draw the corresponding Use Case diagram. Write a clear description for each of the use cases.

(b) On the basis of the above problem statement and your use case descriptions, identify candidate domain classes for the system. Draw the corresponding class diagram indicating any significant associations between classes.

(c) For each of the use cases described in part (a), present an interaction diagram (sequence or collaboration) showing how the use case is implemented (or realized) by a collaboration of some or all of the classes of part (b).

6. An urban authority is planning to acquire a traffic management system (TMS). The purpose of the TMS is to facilitate improved monitoring and management of traffic flow within the urban area. It is envisaged that there will be traffic counting devices located at each junction that will transmit traffic counts at regular intervals to a central "traffic analyzer" system. It will be possible for the central system to transmit certain control messages to each counting device (e.g. to change the transmission interval from 10 to 5 minutes). It will also be possible, in the case of junctions controlled by traffic lights, for the central system to adjust timing of light changes remotely.

The primary users of the system will be dedicated traffic analysts who will use the system to monitor traffic and make adjustments. Guidance on what adjustments to make will be provided by the system in terms of traffic models and particularly by an extensive historical record of past traffic data. The system will include a graphical interface to support visualization of the current traffic situation.

The system will also be available to the police so that they can monitor the traffic situation. It is envisaged that the police and traffic analysts will liaise, for example in deciding whether police officers should be sent to take manual control of particular junctions. Finally, the system should be able to provide summary traffic data to broadcasting services for public information purposes.

(a) Identify four use cases for this system and draw the corresponding Use Case diagram. Write clear, precise descriptions for all of the use cases.

(b) On the basis of the above problem statement and your use case descriptions, identify at least four candidate domain classes for the system and summarise the responsibilities you envisage for each of them. Draw the corresponding class diagram indicating any significant associations between classes.

(c) For each of the use cases described in (a), present an interaction diagram (sequence or collaboration) showing how the use case is implemented (or realized) by a collaboration of some or all of the classes of part (b).