

---

Presentation of Chapter 7 & 8 of text

**(Essentials of use case models)**

&

**(More on use case models)**

• **Recap from introductory case study:**

- We had a use case diagram supported by textual descriptions of each use case. We introduced the basic elements (actors and use cases).

- We saw [*and see below - §7.4*] that use case modeling helps in

capturing requirements and

planning iterations of development

The text also mentions that use case modeling helps in

Validating systems (see our mention before of *requirement verifiability*)

- Also, use case diagrams are fairly easy to understand & so they

enhance communication between user and developer

- Note a point of similarity with class diagrams (text pp94, 95):

-- An actor in a use case diagram represents *a rôle* that someone might play, rather than representing a particular individual

-- A “communicates relation” [*depicted as connecting line*] between an actor and a use case means that someone in that rôle *may* be involved in carrying out the use case task

Note: Terms *task* and *coherent unit of functionality* are used interchangeably in the text

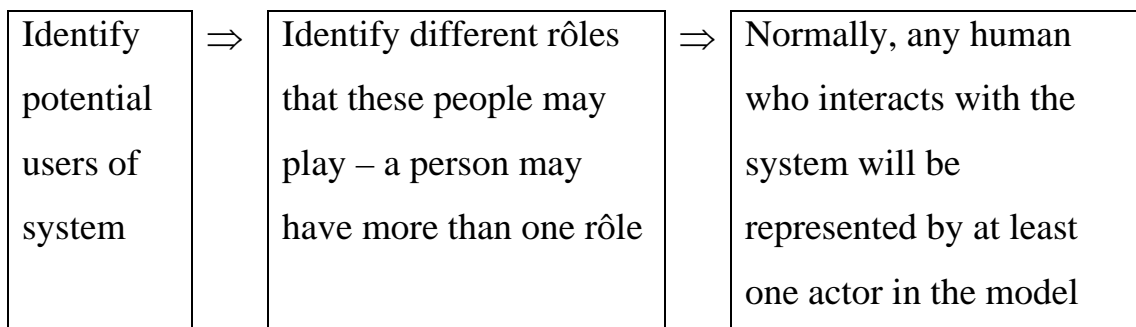
### • 7.1 Actors in detail

- Following are some guidelines on use case modeling

↑	Beneficiary of use case		NEEDS use case – Strong connection in some sense (e.g. BookBorrower & Borrow copy of book). Connection <i>remains throughout</i> development in some guise
ACTOR			
↓	Not beneficiary of use case		Involved in some way but does not need use case. Connection may not persist throughout the development (e.g. Librarian & Borrow copy of book). Implementation of use case may change so actor no longer involved

- There can be essential use cases *without* obvious beneficiaries (e.g. Send quarterly bill to customers in a utility company system)

- Suggested steps to **identify (human) actors**:



Identify non-human actors: Not as clear-cut & experts offer different advice. Best advice is to “do whatever is most useful”. For example, wouldn’t usually put in “keyboard” as an actor (too low-level, passive) but might represent by an actor a “clock”, that sends significant timing signals to the system

Note: Technically, an actor plays a different rôle in each use case and so can be regarded as a coherent set of rôles.

### • 7.2 Use cases in detail

- A more detailed description of a *scenario* is “a possible interaction between the system and some people or systems/devices (in their various rôles). The interaction is described as a sequence of messages.”

Example from text:

Book borrower Mary Smith borrows the library’s third copy of <i>War and Peace</i> , when she has no other book out on loan. The system is updated accordingly.	<i>Example instances of use case</i> Borrow copy of book
Book borrower Joe Smith tries to borrow the library’s first copy of <i>Anna Karenina</i> , but is refused because he already has six books out on loan, which is his maximum allowance.	- <i>The interactions differ</i> - <i>The outcomes differ</i>

---

- Recall the association of a **use case oval** [on a diagram] with corresponding text - the **use case description**. This text is where the detailed requirements are stated, including different possible “paths” through the use case. Usually text is a structured English (or other natural language) but one could use, as appropriate, a UML activity diagram, or, a pseudocode, some kind of formal language (see following **ASIDE**), mathematical notation, etc.

- We saw in the introductory case study how a “sequence diagram” could be used to show how a use case is realized by a system. We will return to this later.

**[START of ASIDE:**

As an example of the use of a formal language we 'turn aside' to an example of the use of the Z language to specify aspects of a library system that is similar, though not identical, to our initial case study.

**Fragment of a formal specification with "Z"**

(from *Formal specification using Z*, D. Lightfoot, 2001)

Notes:

- 1) There are some departures from usual Z notation, for type setting convenience (although special type setting is available - see list of software on school labs).
- 2) dom = domain, ran = range,  $\emptyset$  = empty set,  $\setminus$  = "less" (for sets)
- 3) Roughly, symbols  $\text{?}$  and  $\text{'}$  stand for "input" and "output", respectively.
- 4) A specification is presented as a mixture of English text and formal description.
- 5) "BOOK" as used below is analogous to "COPY" in our UML example

**1. OVERVIEW**

A library has a stock of books which may be taken out by its registered members.

**2. TYPES**

[BOOK]      The set of all possible uniquely identified books

[PERSON]    the set of all possible persons

**3. STATE**

LIB <sub>0</sub>
stock: $\bar{\bar{}} \text{ BOOK}$ members: $\bar{\bar{}} \text{ PERSON}$ outTo: $\text{BOOK} \rightarrow \text{PERSON}$
dom outTo $\subseteq$ stock ran outTo $\subseteq$ members

**4. INITIALISATION OPERATION**

Initially the library has no stock of books and no members and no books are recorded as out to members.

Init <sub>0</sub>
LIB <sub>0</sub> '
stock'= $\emptyset$ members'= $\emptyset$ outTo'= $\emptyset$

**5. SOME OTHER OPERATIONS****5.1 Acquire book**

The book must not already belong to the library's stock. It is added to the stock. The members remain unchanged.

Acquire <sub>0</sub>
$\Delta \text{LIB}_0$ b?: $\text{BOOK}$
b? $\notin$ stock stock'= $\text{stock} \cup \{b?\}$ members'=members outTo'=outTo

**5.2 Register member**

The person must not already be a member. The person becomes a member. The stock remains unchanged.

Register <sub>0</sub>
$\Delta LIB_0$ $p?: PERSON$
$p? \notin members$ $members' = members \cup \{p?\}$ $stock' = stock$ $outTo' = outTo$

**5.3 Take a book out**

The person must be a member. The book must belong to the library's stock and must not be out to anyone. The book becomes recorded as out to the member. The members and stock are unchanged.

TakeOut <sub>0</sub>
$\Delta LIB_0$ $p?: PERSON$ $b?: BOOK$
$p? \in members$ $B? \in stock \setminus \text{dom } outTo$ $outTo' = outTo \cup \{b? \rightarrow p?\}$ $members' = members$ $stock' = stock$

**5.x Similarly for other operations**

**END of ASIDE]**

- **7.3 System boundary**

A “box” can be placed around the use cases in a use case diagram to represent the system boundary

- Most useful in a big system to depict different subsystems with different boxes

**• 7.4 Using use cases**

Note: All material under this heading is guidance material, suggestions for good practice, problems to watch out for, etc.

**\* 7.4.1 Use cases for requirements capture**

- They provide a structured way to go about requirements capture (see §7.1 above esp. on identifying actors, what they need from the system, where they are beneficiaries and where they are (only) helpers)
  
- Use cases are useful in determining priorities in an iterative development
  
- A good piece of advice in text is that, when in discussion with users: “list actual people or systems associated with each actor” This detail would not appear in a UML diagram
  
- **WARNING**: *Some essential tasks* may not show as being of value to (needed by) any actors and so *may not appear as uses cases* ( see “bill customers” example earlier). So there is a risk of missing some requirements if one relies only on use cases for capturing requirements.

---

**\* 7.4.2 Use cases throughout the development**

PLANNING (Project Management tasks of estimating effort & scheduling tasks):

Prerequisite (?) for proper planning<sup>1</sup> is to have a list of all use cases, with

- A good idea of what each means
- An understanding of who wants each and how much
- Knowledge of which use cases carry most risk
- An estimate of how long it should take to implement each use case

Brings up some important points:

- If planned schedule turns out unreasonable, negotiate with customer about priorities
- Level of granularity of use cases: Maybe when broken down further some scope for re-use may emerge (due to shared functionality)
- Having decided on what functionality to deliver (and its “quality”) must decide on
  - order of delivery (scheduling work)
  - life cycle model – iterations – what to include per iterations – are iterations internal or external?

POLITICAL ASPECTS:

- Good idea, where feasible, “to demonstrate system doing something valuable to most influential people first” [*mitigates a potential major (non-technical) risk*]
- A possible outcome of the use case analysis is that it is decided not worthwhile to go ahead!

---

<sup>1</sup> if using UML!

More generally, the same points could be made about 'requirements' in whatever they are stated. Sometimes (quite often!) one must plan in the absence of detailed knowledge of requirements, for example in preparing a tender. Usually, planning is made more precise as analysis and design proceeds.

TECHNICAL ASPECTS:

- A good criterion *may* be to deliver the use cases with highest (technical) risk first. (But could alternatively mitigate by a feasibility study, prototyping, etc)
- Note point made in text of evolution of use case descriptions during a development, from initial focus of “WHAT” system is to achieve for each user to “HOW” it does this.

SYSTEM VALIDATION:

- Good technique (demonstrated during introductory case study) is to take each use case and check that the design allows (each scenario of it) to be carried out. (Referred to as “walking the use case”)
- Derive “system tests” for the use cases early on in the project (ideally beginning as soon as the descriptions are written). Related to “verifiability” of use cases, mentioned earlier. [*This could form the basis for an agreement on 'acceptance criteria' between developer and customer. For example, one 'extreme' could be that all the system tests are passed in all respects; less severe might be to agree on a key subset of tests that must be passed before acceptance*].

**• 7.5 Possible problems with use cases**

Use case modeling should be used with caution since

- (1) Danger of building a system that is not object oriented. Difficulty would be that if too much concentration were put on delivering particular use cases rather than on the overall design, one could end up with a top-down functional decomposition which [*it is claimed for OOD*] would probably not be as easily maintained. [*Of course, if you didn't have a maintenance contract you might be happy with this situation!*]

(2) Danger of mistaking requirements for design: A good illustration is given in text is that “*users are likely to describe the use case as a very concrete sequence of interactions with the system which is one way, not the only way, of achieving their real goal*”. Aim must be to identify what are real constraints, what are just unquestioned or unjustified assumptions & conventions.

(3) Danger of missing requirements: See earlier example (billing customers) where a requirement didn’t arise from an “actor – use case” relation.

• **Panel 7.1: “Use case driven development” (Buzz word!)**

- Idea is to keep focus on use cases throughout a development, not just in requirements capture. Use to keep attention where it should be, on the user requirements [This approach is reasonable]

- Some people have advocated basing identification of objects and classes on use cases primarily. This can lead to omissions, as mentioned before. The text book offers the sensible advice that one should use aspects of all three approaches “Use case driven”, “Data driven” and “Responsibility driven” to achieve a good OO development.

## 8. More on use case models

A GENERAL WARNING (text page 104): Chapter 8 introduces additional features to show “relationships between use cases” and “relationships between actors” on use case diagrams. However, such features make the diagrams more complex. This *may well be undesirable* in that one of the attractions of use case diagrams is their accessibility to “non-technical” stake-holders (users, customers, managers (?) etc).

### • 8.1 Relationships between use cases

We have

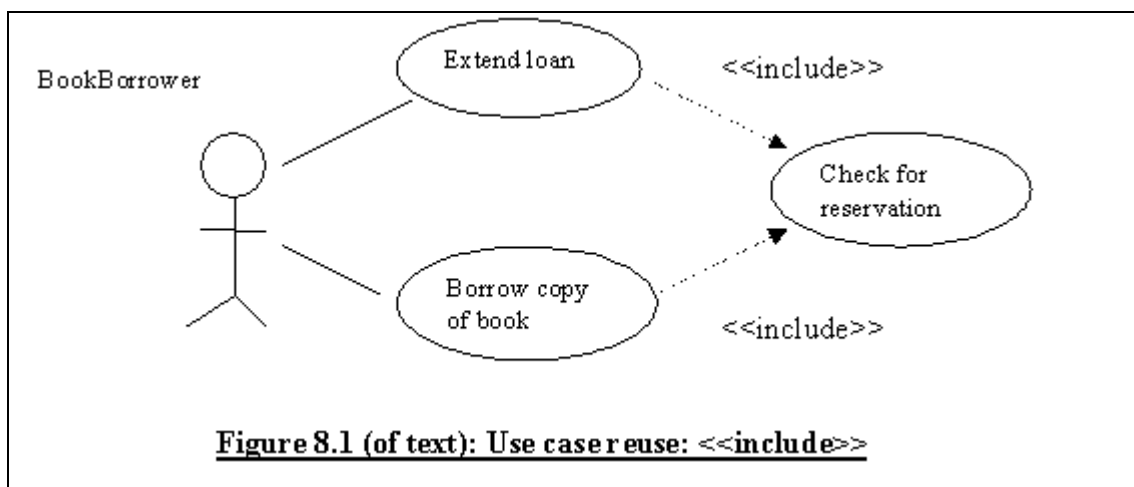
\* **8.1.1 Use cases for reuse:** <<include>>

\* **8.1.2 Components and use cases**

\* **8.1.3 Separating variant behaviour:** <<extend>>

\* **8.1.1 Use cases for reuse:** <<include>>

In our library example both the use cases Borrow copy of a book and Extend loan have a need to check whether there is an existing reservation on the book (to see whether the loan can be granted or extended, respectively). This could be depicted as follows:



- The *stereotype* [*we will come back to this term later on*]  
<<include>>.can be used when we want to factor out some common behaviour (as above). The ideal would be if the common behaviour were already available as a component.

- The diagram syntax is rather like that before to show a generalization in a class model (here a *dashed* arrow points to the item depended on; in other words from the *source* use case to the *target* use case)).

- The corresponding use case textual descriptions need to be changed also.

The text gives a number of examples of which following is one:

**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 checks whether there is a reservation on the book (use case Check for reservation); otherwise the system refuses to lend the book. If the book is reserved, the system refuses to lend it. Otherwise it records that this library member has this copy of the book on loan and prompts for the book to be stamped with the return date.*

*Italics mark the parts changed compared to earlier version, especially regarding the common Check for reservation use case*

[See text for 2 more examples of modified use case descriptions]

- FOR AND AGAINST documenting shared/reused functionality on a use case diagram:

<b>Advantages</b>	<b>Pitfalls</b>
<p>1. Convenience + avoids recording same information twice</p> <p>2. Properly done, should make use case descriptions easier to understand as well as shorter</p> <p>3. Identifying common functionality early on in use cases may lead to discovering possible component reuse later (a help in planning)</p>	<p>1. DANGER that looking for reuse at use case level will lead to a top down functional decomposition rather than an object oriented design.</p> <p><u>Guard</u> against this by constructing the class model in quasi-parallel.</p> <p>2. Inclusion of &lt;&lt;include&gt;&gt; makes diagram harder to understand for non-technical reader</p> <p><u>Only separate out significant chunks</u> of reusable functionality</p>

### \* 8.1.2 Components and use cases

#### **Impact of using a component on the use case model**

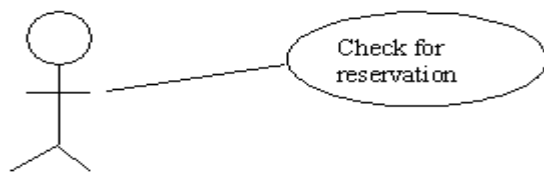
- Developer should resist offering infinite flexibility on requirements. If [standard] components are already developed attempt to adjust requirements so that some of them can be satisfied by existing components [*compare building industry*]. "Carrot" for customer is reduced price and schedule.

- Components not "made" by the software developer: Here focus is whether or not to show such components on our model diagrams (including use case diagrams). It probably depends on how significant the component functionality is.

Note: Other queries that can be raised about "*commercial off-the-shelf software*", include "is it of the same quality as the in-house software?"

#### **How a use case can help specify a component?**

- Suppose it's planned to develop a reusable component that includes some but not all the functionality of a given use case. For this situation, it makes sense to describe the proposed component by its own use case - the actors that interact with this use case may not be humans or external



Reservation Checker

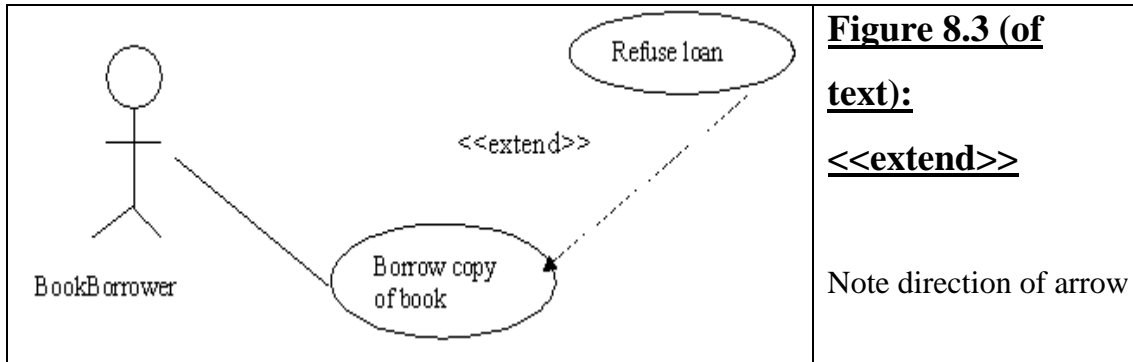
**Figure 8.2 (of text): A use case diagram describing a component**

systems/devices. From the perspective of the component, of course, the objects that interact with it look like external systems.

Remark: Normally, would expect <<include>> to appear as a later refinement of, rather than on the first attempt at a use case diagram.

\* **8.1.3 Separating variant behaviour:** <<extend>>

- Situation where a use case incorporates a number of significantly different scenarios & it's decided to ***depict these as a main case plus one or more subsidiary cases***. For example,



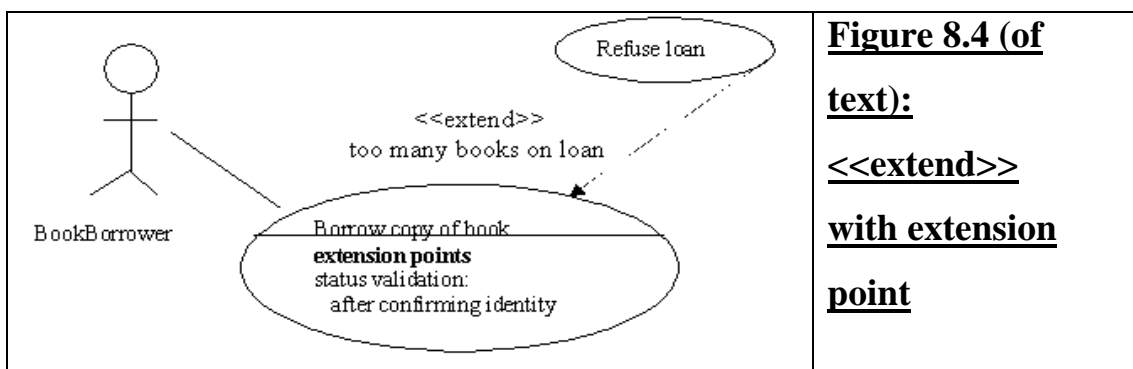
The alternative of showing this on a diagram is to do what we have been doing, namely define the different scenarios in the use case description.

If we are using <<extend>> then the use case description must be decomposed into separate descriptions for the main (normal) and subsidiary cases. In the normal case description we must show:

--- Condition(s) under which exceptional case(s) occur

--- **Extension point** where condition is tested & behaviour may diverge.

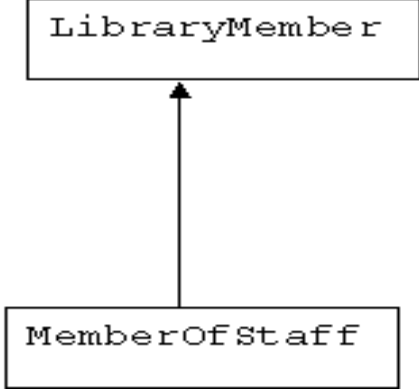
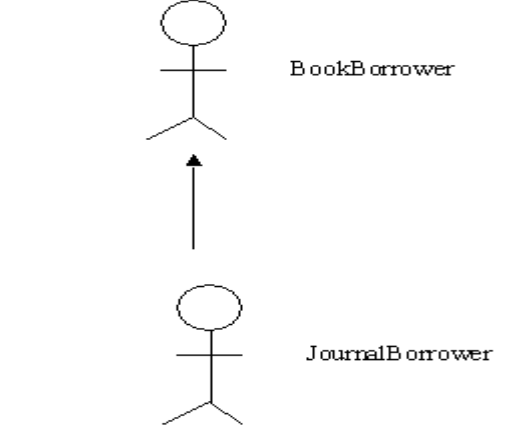
**If desired**, (a) the condition may be shown on the extension arrow and (b) the extension point recorded within the oval (or ellipse) of the normal case:



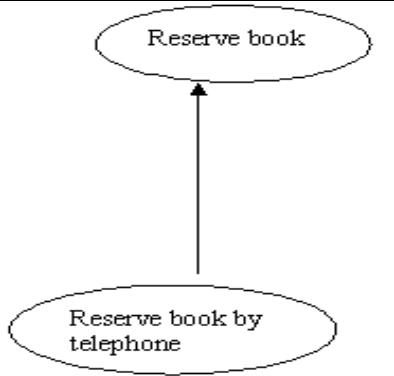
• **8.2 Generalizations**

**Note:** In UML, each of class, actor and use case is a *classifier*. Any classifier can be generalized.

- One *may choose* to depict actor generalizations:

Previous example of a class generalization	Example of an actor generalization ("is a")
 <pre> classDiagram     class LibraryMember     class MemberOfStaff     MemberOfStaff -- &gt; LibraryMember                     </pre>	 <p style="text-align: center;"><b>Figure 8.5 (of text): Generalization between actors</b></p>

- The same type of depiction may be used for use cases. For example,

 <pre> usecaseDiagram     usecase UC1 as Reserve book     usecase UC2 as Reserve book by telephone     UC2 -- &gt; UC1                     </pre>	<p>Could be useful if library system needed to behave differently for the specialization (e.g. member's card cannot be scanned).</p> <p>Similar to &lt;&lt;extend&gt;&gt; - arguable whether both should be in UML. Text suggests to use &lt;&lt;extend&gt;&gt; when documenting run-time conditions as opposed to a specialization of a whole task as here</p>
--	---

### • 8.3 Actors and Classes

Introductory case study: - 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.

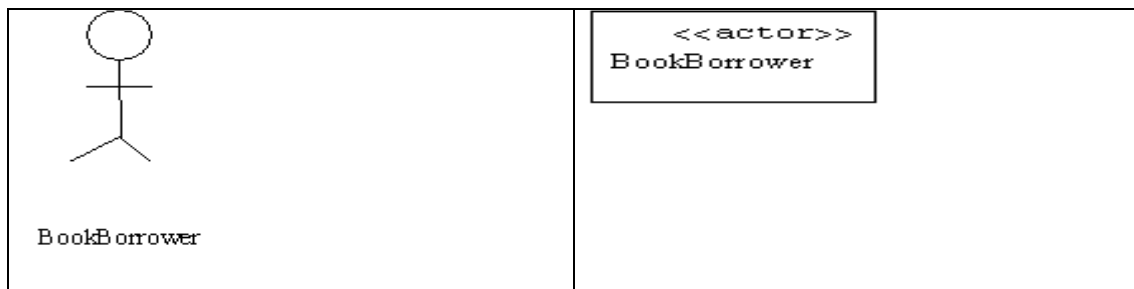
*Example given in text is of the message "borrow(theCopy)" being sent to the "LibraryMember" object that represents the member wishing to borrow. Then, the "LibraryMember" object is responsible for carrying out [or for causing to be carried out] whatever is done to record (or deny) the loan.*

- This is an example of the common situation where (a) a system interacts with an actor (instance) and (b) has an internal system object representing the actor instance. The text book distinguishes 2 main situations where this can happen - *quite useful for design purposes* -

<p><b>I: System may need to store data about an actor in a certain rôle (in order to carry out its use case)</b></p>	<p><b>II. System <u>wraps</u> an external system in order to provide a manageable way for parts of the system to access the external system &amp; vice versa</b></p>
<p>e.g. which people can borrow books, how many books have they out currently, etc</p> <p>Probably means that there is</p> <ul style="list-style-type: none"> <li>-- a set of real people who can take the rôle of a given system actor</li> <li>-- a set of system objects, one for each person, which store information about the people in that rôle.</li> </ul>	<p>Examples:</p> <ul style="list-style-type: none"> <li>- An external transaction monitor might be accessed by sending messages to an internal TPMonitor object, which in turn invokes the real functionality of the external system (cf OBDH, sensors, etc)</li> <li>- A separate user interface program might be represented inside the system by a UI object which in fact mediates between the external UI program and the main system (passing messages both ways)</li> </ul>

Design suggestion from text: "One of the easiest ways to provide a simple user interface to a system, which can handle the system needing to respond to different users in different ways, is to make the use cases which can be initiated by Jane Bloggs *available as methods* of a system object representing Jane Bloggs" (*see our borrow copy of a book example*)

(MINOR) Notational point re actors as classes: In UML, these symbols mean the same,



One may say that actors are classes, with the stereotype <<actor>>.

We will come back to the notion of stereotype later on.