UML & Object Modelling - Case study II

CSCU9P5 – Software Engineering I

UML & Object Modelling

- Definition of the problem: Library management
  1. Identifying classes
  2. Use case: Borrow a copy of a publication
     a. sequence diagram, boundary and class diagram
     b. design oriented diagram
  3. Use case: Look up a catalogue to see if there is a copy on the shelves that we can borrow
     a. control class and other use cases

Case study: Library

- A **library** holds details of the **publications** it owns together with information on its **members**.
- There are two kinds of **publication**: books and journals.
- The library may contain several **copies** of each book or journal.
- Information held on each **book** includes: title, author(s), library catalogue number, copies held.
- Information held on each **journal** includes: title, volume number, library catalogue number, copies held.
- Information held about each copy includes whether or not it is currently on loan and, if so, the borrower.

Case study: Library

- The following information is to be held about **members** of the library:
  - Name, library number, copies of publications currently borrowed.
- A member can borrow up to **six** publications.
- Each member of the library has a library **card** giving their library number in a machine readable form.

Case study: Library

- Some copies of books can be borrowed for **two weeks**, **RBR** (Reserve Book Room) books only **overnight**. Copies of journals can be borrowed for **three days**.
- Each *copy* of a publication has a **label** giving the publication's library catalogue number and the *copy number* in a machine readable form.
- **Borrowing** a book involves the system reading a member's library card and a copy's label.

Identify classes

- Let us start by trying to identify some of the classes
- The following would seem fairly obvious:

```
Publication, Book, Journal, Copy, Member
```

- Class diagram ? Not obvious yet…
- We will defer the class diagram until after a little more analysis…

Identify classes

- We identify classes by dealing in terms of real world entities. (Quite natural for OO! - with exceptions, e.g. control classes).
- We could therefore regard the classes/class diagram as modelling the real world.
- However, our aim is to develop a software system
  - *An information processing system*
- Hence, we *change focus* and regard the **class diagram** as **modelling** the ***information that the system will hold*** about `Publication` etc.
  (Abstraction process!)

Associations

- We are told that members "knows about" the copies they have borrowed and that each copy "knows about" its borrower. Also publications "know about" their copies.
  - We can therefore draw **associations**.
- Naturally, `Book` and `Journal` should be **subclasses** of `Publication`.

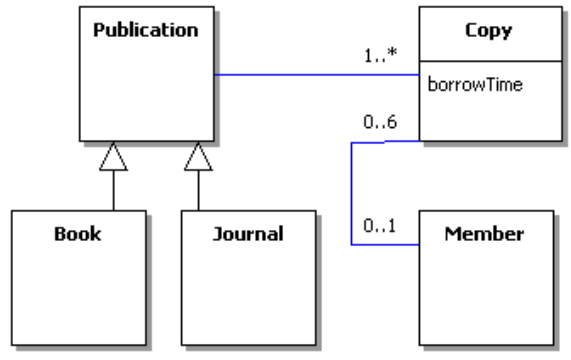
Copies

- What about **Copy**? Are we going to have different kinds of **Copy** object?
  We will try to have just one kind.
- In the class diagram, we have only one kind of **Copy**. It has the number of days it can be borrowed as an attribute.
- That way, we can move a copy of a book between the RBR to the normal shelves just by changing the value of the attribute.
- That is, of course, not the only answer and a designer could decide to change that decision later.

Identify classes

- Let us start by trying to identify some of the classes and to create an initial class diagram. The following would seem fairly obvious:

Publication, Book, Journal, Copy, Member

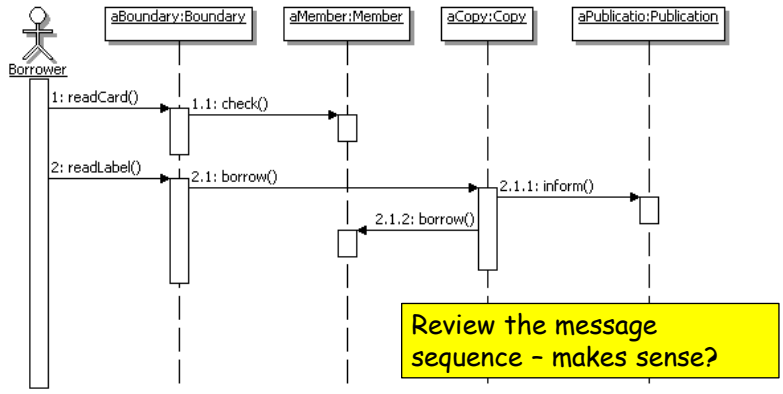


Use cases

- To proceed further we need to consider use cases. We can construct several use cases to give us information about how the system is to be used.
- Here is one: **Borrow a copy of a publication**
  - A person presents a library card and a copy of a publication.
  - The **system** reads the **card** and **checks** that it is valid and that the maximum allowed number of publications has not already been borrowed.
  - If the check succeeds, the **system scans** the **label** on the copy and records that the copy of the publication has been borrowed.
  - Otherwise, the loan is refused.

Sequence diagram: Borrow

- We then construct sequence diagrams to determine the object interactions required *to realise each use case*. The following sequence diagram shows the scenario where we successfully borrow a publication.

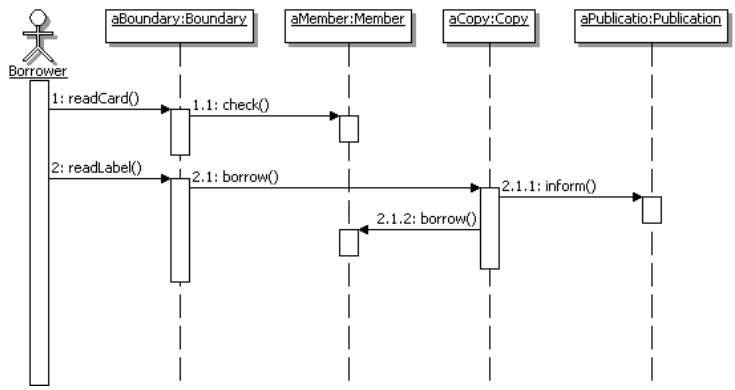


Sequence diagram: Borrow

- The environment (Borrower) sends a **readCard** message to the **Boundary** object giving the identity of the borrowing member.
- The **Boundary** object then sends a **check** message to the appropriate **Member** object to check that borrowing is allowed (no outstanding fines, less than 6 publications currently borrowed).
- The **Boundary** object displays the result to the Borrower.
- The environment (Borrower) then sends a **readLabel** message identifying the **Copy** being borrowed.
- The **Boundary** object sends a **borrow** message to the appropriate **Copy** object with the **Member** object reference as a parameter.
- The **Copy** object updates its attributes and sends messages to the **Member** object and to its **Publication** object for them to update their attributes. **Boundary** displays the result to the Borrower.

Sequence diagram: Borrow

- We have introduced a **Borrower** actor and a **Boundary** object into the sequence diagram.
- The operations have not been given any parameters as our aim was to get the structure right.

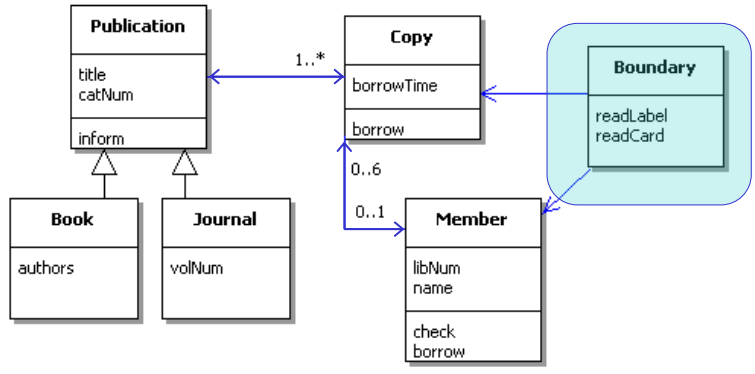


Boundary classes

- The final system will have **one or more boundary classes** written in some programming language. They control how information is passed between the environment (the users) and the system.
- The precise structure of the boundary will be **language dependent** as it very much depends on the level of abstraction supported by the language.
- Initially we can just create a single class in our analysis model knowing that it is likely to be divided into a set of classes when we come to the low-level design.
- Note that as we are dealing with software development, we model in terms of boundary classes rather than try to model
  - the actual hardware devices (barcode scanner, screen, mouse) with which the users actually interact
  - or the event handling

Class diagram

- From the previous scenario we can expand our class diagram showing **operations** and giving **navigability** information.
- We have also added the boundary class and **attributes** from the problem statement to the following class diagram.

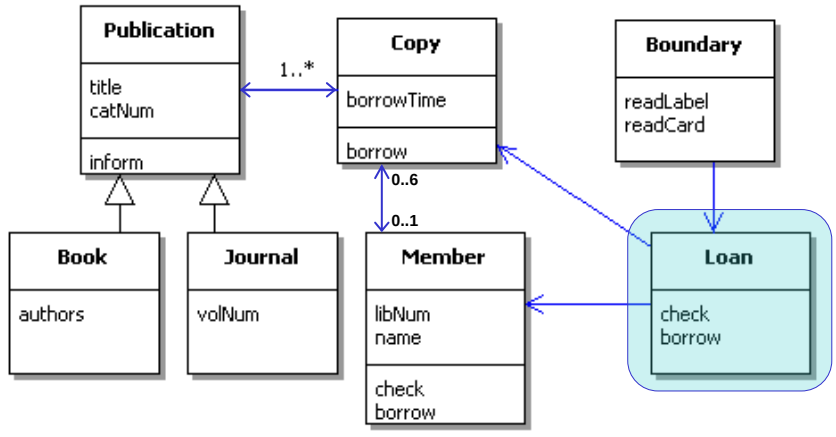


Class diagram: Control classes

- The class diagram gives us a reasonable analysis model.
- In a **design model**, it can be useful to add **control classes** *to coordinate each activity.*
- For example, we could introduce a `Loan` class that deals with the borrow and return of a copy.
- *A* `Loan` object is created when a copy is borrowed.
- The `Member` and `Copy` classes will have associations with the `Loan` class rather than with the `Boundary`.
- When the copy is returned, the `Loan` object **will be terminated**.
- In implementation: A control class allows separation of user interaction code (boundary class) from "business logic" (control class)

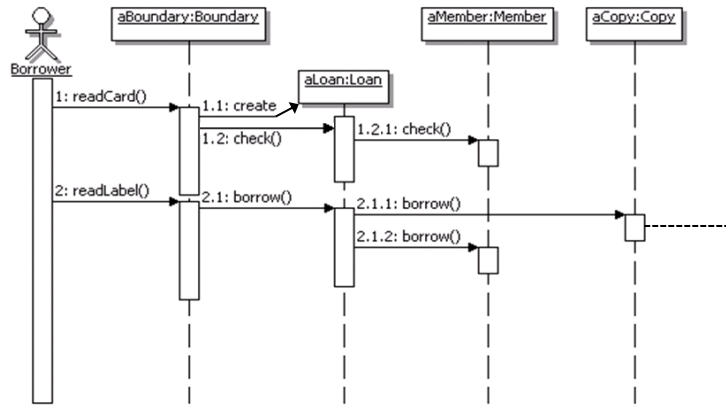
Class diagram: control class

- That gives the following class diagram.



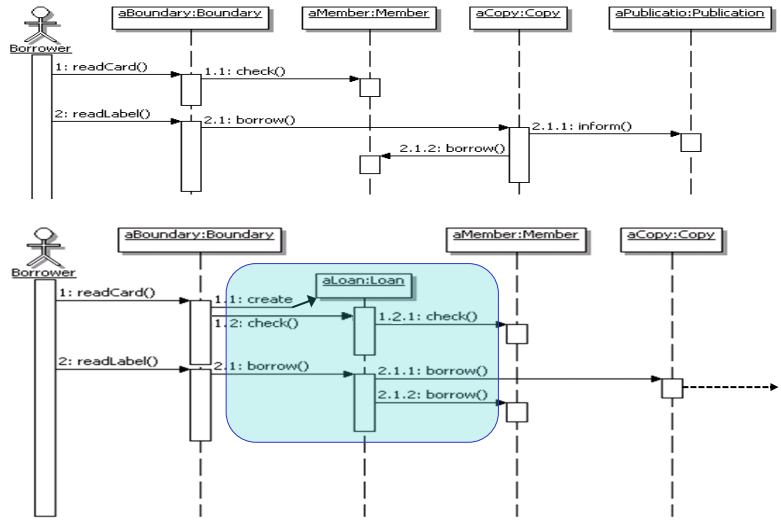
Sequence diagram with control class

- The sequence diagram for the **`borrow`** operation could become:



Sequence diagrams

- Compare the two cases:



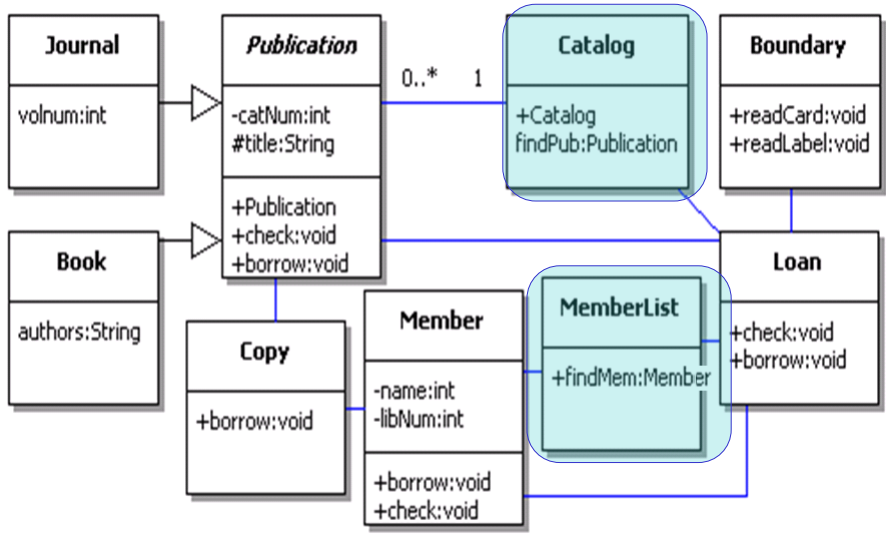
Design oriented diagrams

- The models seen deal with **entities** in the **problem.**
- When we move to a **design phase**, other issues arise.
- For example, where do we hold all the member information so that we can access information on a particular member?
- We can have a `MemberList` object which will contain all the `Member` objects. The **operations** offered by `MemberList` will be **list** operations such as `findMem`.
- In practice, this list information might be **held as a database**, but when we are modelling, we can regard it **as an object**.
- (Likely to be implemented as a Java class that will give access to the database.)

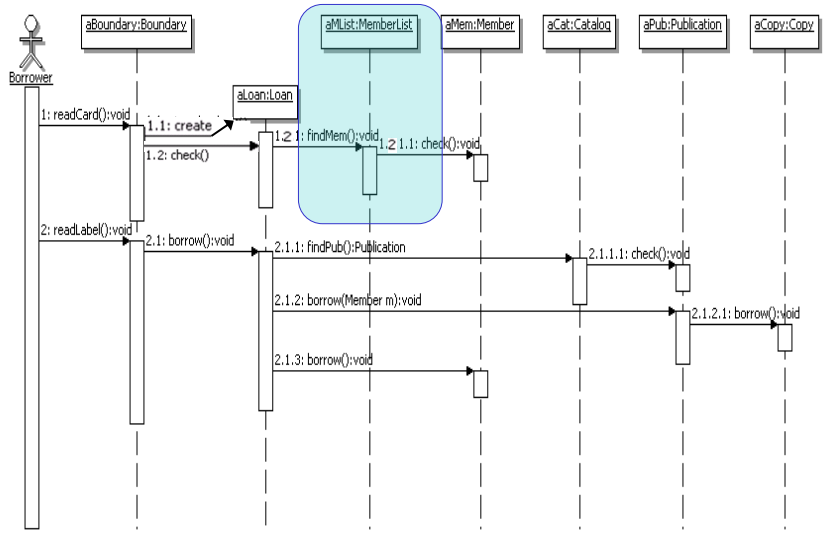
Design oriented diagrams

- **Important**: Objects are no longer just the problem domain entities.
- Hence our next version of the **class diagram** has **list classes**.
- We can consider our **library catalogue** to be a **publication list**.
- The **sequence diagrams** will now be more complex as we will explicitly look up lists in order to find items.
- The revised **class diagram** is shown on the next slide.

Class Diagram with lists

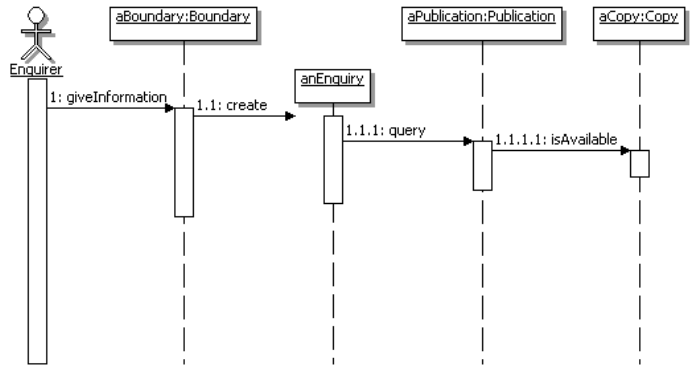


Sequence Diagram, with lists



Library case study

- Let us now consider the use case
  - **look up a catalogue to see if there is a**
  - **copy on the shelves that we can borrow.**
- We could create a new **control class**; let us call it **`Enquiry`**



Library case study

- This would seem to suggest a drawback of this approach; **each time** we have a new use case, do we need **a new control class**?
- Perhaps not always:
  - Might not be necessary for simple use cases
  - May be able to re-use for related use cases

Other use cases

- Some other use cases to be considered:
  - Return copy of publication.
  - Add new publication to library.
  - Add new copy of existing publication to library.
  - Add new member to library.
  - Letter to member who has overdue copy.

- **Exercise !**

End of case study