# Software component interactions and sequence diagrams

Perdita Stevens

School of Informatics
University of Edinburgh

## What do we need to know? Recap

Recall that this is an *overview* of software engineering, dipping into some aspects. We've discussed:

- how to analyse requirements and summarise them in a use case diagram;
- how to tell good design from bad;
- how to record basics of the static structure of our designed system in a class diagram;
- how to get started with choosing an appropriate static structure.

We have not discussed dynamic aspects of design: what operations should our classes have, and what should they do?

## Dynamic aspects of design

Suppose that we have decided what classes should be in our system, provisionally. What next? Well, we have to meet the requirements...

In the end, we need to know what operations they have, and what each method should do.

Two ways of looking at this:

1. inter-object behaviour: who sends which messages to whom?
2. intra-object behaviour: what state changes does each object undergo as it receives messages, and how do they affect its behaviour?

Complementary: but in this course, we only consider 1. For 2, UML provides an enhanced variant on the FSMs you saw last year.

For more info, do SEOC next year, and/or read the recommended texts.

## Thinking about inter-object behaviour

There's no algorithm for constructing a good design. Create one that's good according to the design principles...

Your classes should, as far as possible, correspond to domain concepts.

The data encapsulated in the classes is usually pretty easy to define using the real world as a model.

Then look at the scenarios in the use cases, and work out where to put what operations to get them done.

Some of this is easy. Hard parts are usually when several objects have to collaborate and it isn't clear which should take overall responsibility.

## CRC cards

**C**lass, **R**esponsibilities, **C**ollaborations

Originally introduced by Kent Beck and Ward Cunningham as an aid to getting non-OO programmers to "think objects".

Also useful for validating the class model against the use case model.

We'll see how to record much of the information produced using CRC cards in UML *interaction diagrams*.

CRC cards are an aid to clear thought, not a formal part of the design process – though UML does permit you to record the information from them in the class model, if you wish.

## Examples

| `LibraryMember` | |
|---|---|
| **Responsibilities** | **Collaborators** |
| Maintain data about copies currently borrowed | |
| Meet requests to borrow and return copies | `Copy` |

| `Copy` | |
|---|---|
| **Responsibilities** | **Collaborators** |
| Maintain data about a particular copy of a book | |
| Inform corresponding `Book` when borrowed/returned | `Book` |

## C, R and C

Class: a well chosen name capturing the essence of the class

Responsibility: what services is this class supposed to provide? (Given at a more abstract level than operations; check for coherence and cohesion.)

Collaborators: what services does this class need in order to fulfill its responsibilities? (Again, at a more abstract level than message passing: may leave protocol undecided, but check for feasibility and coupling.)

## Refinements of CRC card use

Some people like to use more than the basic C, R, C, e.g. showing:

- sub- and super-classes under the class's name;
- emerging attributes and other parts on the back of the card;
- a concise definition of the concept represented by the class on the back of the card.

Yes, there are computer-based CRC card tools. But in fact there's value in using the physical cards.

## Interaction diagrams

describe the *dynamic* interactions between objects in the system, i.e. the pattern of message-passing.

Two main uses:

- Showing how the system realises [part of] a use case
- Showing how an object reacts to some message

Particularly useful where the flow of control is complicated, since this can't be deduced from the class model, which is static.

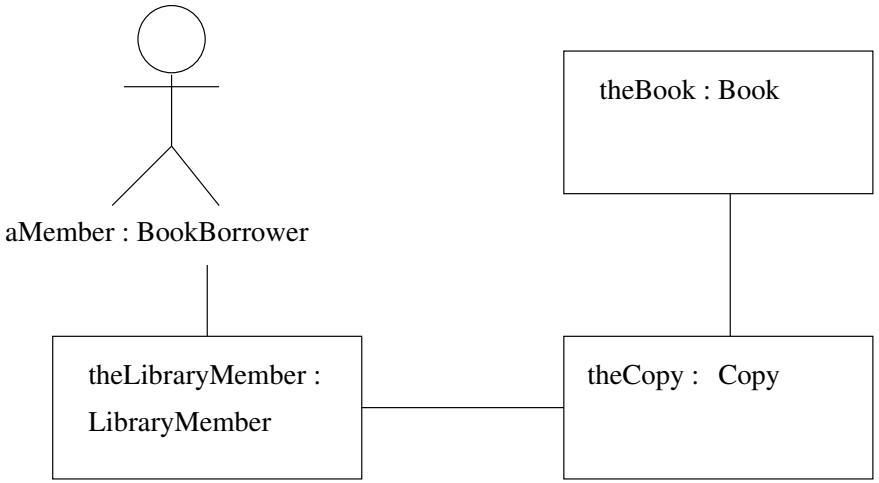
UML has two sorts, *sequence* and *communication* diagrams – the differences are subtle, and we'll only talk about sequence diagrams.

## Developing an interaction diagram

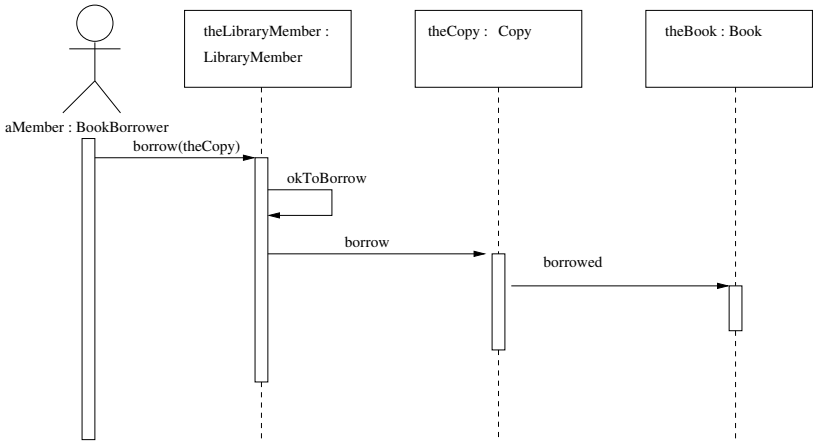
1. Decide exactly what behaviour to model.
2. Check that you know how the system provides the behaviour: are all the necessary classes and relationships in the class model?
3. Name the objects which are involved.
4. Identify the sequence of messages which the objects send to one another.
5. Record this in the syntax of a sequence diagram.

Simple :-)

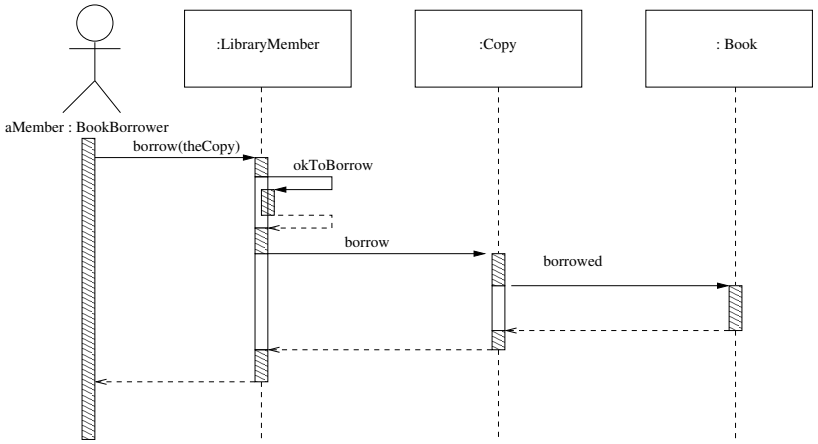
## A collaboration



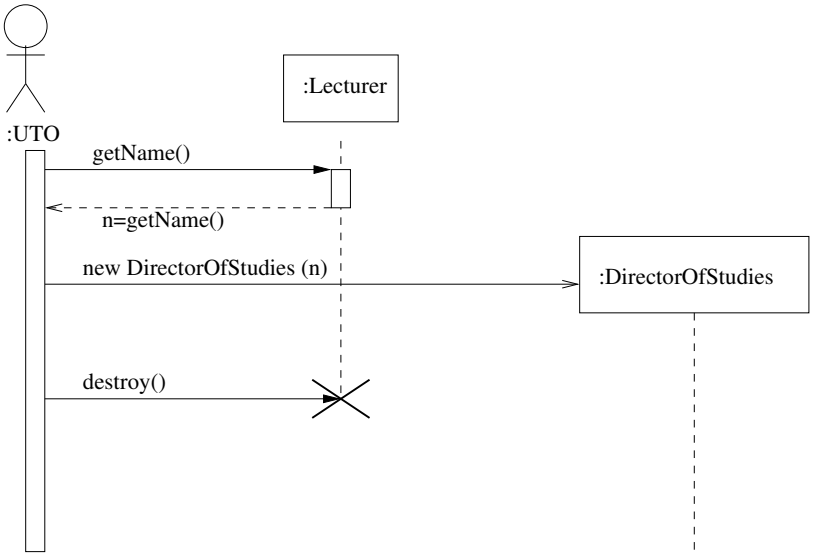

## Sequence diagram



## Showing more detail




## Creation/deletion in sequence diagram



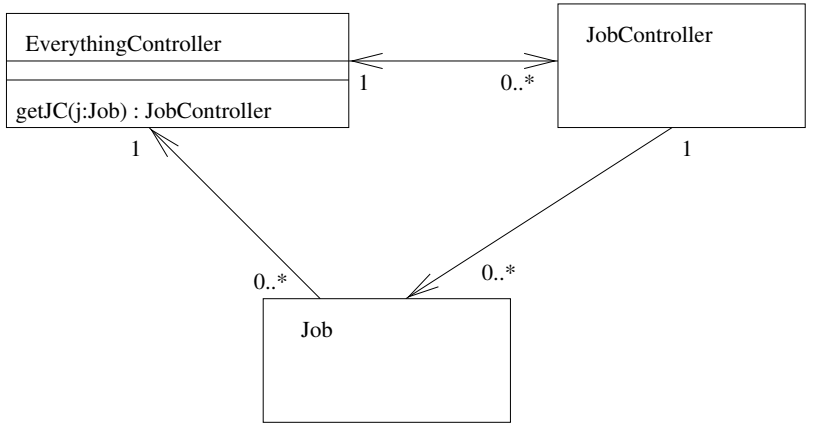

## What is a good interaction pattern?

In designing an interaction, your first aim is obviously to design *some* collection of operations that can work together to achieve the aim.

Next, consider:

- conceptual coherence: does it make sense for this class to have that operation?
- maintainability: which aspects might change, and how hard will it be to change the interaction accordingly?
- performance: is all the work being done necessary?

## Designing interactions




Problems?

## Law of Demeter

in response to a message $m$, an object $O$ should send messages *only* to the following objects:

1. $O$ itself
2. objects which are sent as arguments to the message $m$
3. objects which $O$ creates as part of its reaction to $m$
4. objects which are *directly* accessible from $O$, that is, using values of attributes of $O$.

## More complex sequence diagrams

We've only discussed very simple sequence diagrams. UML provides notation for reusing pieces of interactions, conditional or iterative behaviour, asynchronous messages, etc. etc.

## Reading

Required: The original paper on CRC cards: *A Laboratory for Object-Oriented Thinking*, by Kent Beck and Ward Cunningham. See web page.