Software component interactions and sequence diagrams

Paul Jackson

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.

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 operation 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?

In this course, we only consider 1.

For 2, UML provides State diagrams, enhanced FSMs

Thinking about inter-object behaviour

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

1. Your classes should, as far as possible, correspond to domain concepts.
2. The data encapsulated in the classes is usually pretty easy to define using the real world as a model.
3. Then look at the scenarios in the use cases, and work out where to put what operations to get them done.

Can get hard when several objects have to collaborate and it isn't clear which should take overall responsibility.

CRC Cards can help.

Interaction diagrams

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

Good for showing how the system realises [part of] a use case

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 – we'll mainly talk about sequence diagrams.

OO message-passing terminology

```
class A {                    class B {
   f() {                       g();
      B b = ... ;            }
      ...
      b.g();
      ...
   }
}
```

Let `a` be some object of type `A`.

- Object `a` sends a (call) message `g` to object `b`
  - An invocation `a.f()` makes a call `b.g()`
- Object `b` sends a reply message to object `a`
  - An invocation `b.g()` finishes and control flow returns to `a.f()`

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 an interaction diagram.

A use case

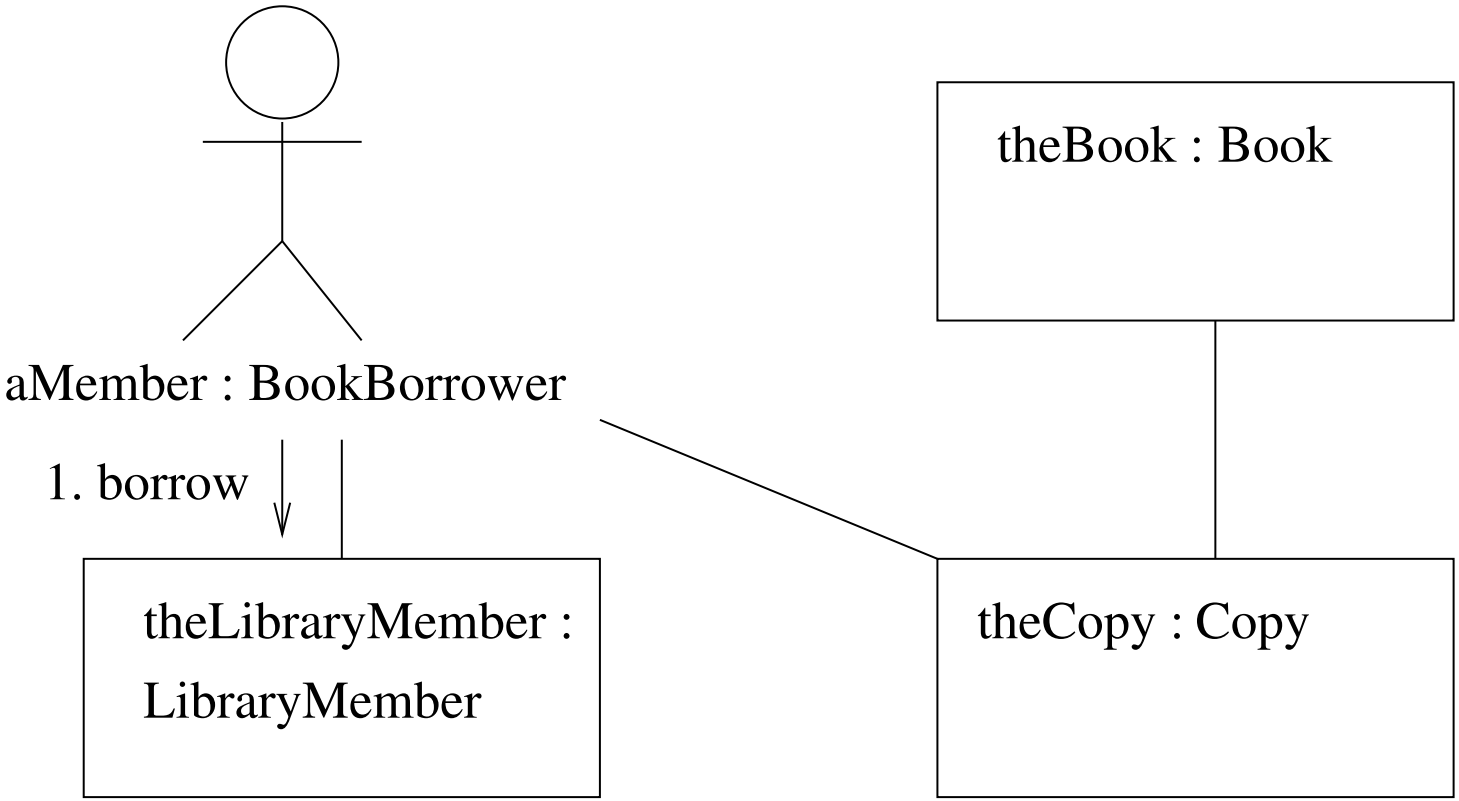
Scope: A system for keeping track of books owned a library and which books are checked out to whom.

Title: Borrow book

Primary Actor: a book borrower

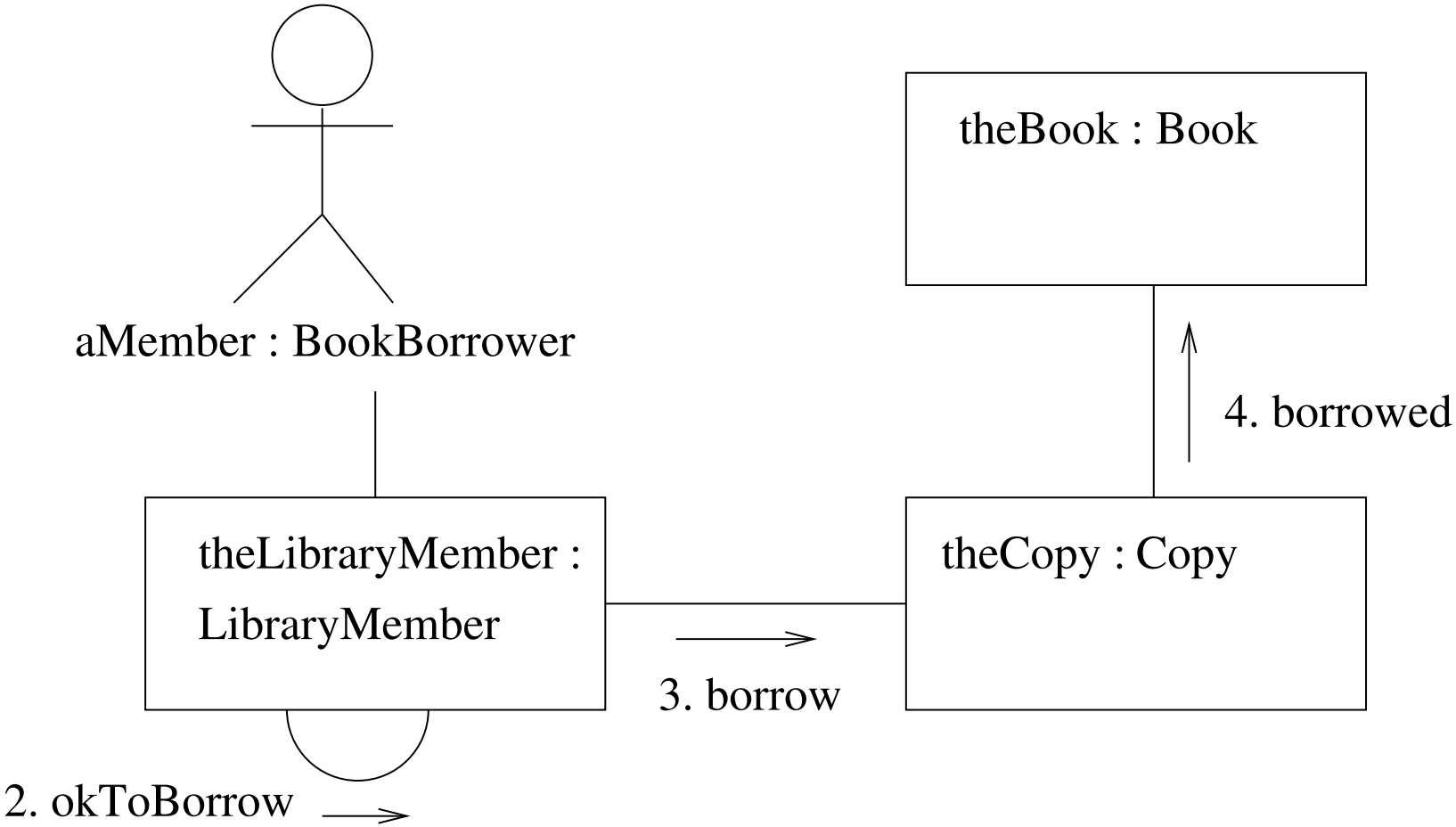
Description: A book borrower presents a copy of a book to the system along with some ID card. Assuming the borrower has not already checked out some maximum number of books, the system permits the loan, recording who the book is checked out to, and noting that the number of free copies of the book is reduced.

A collaboration I

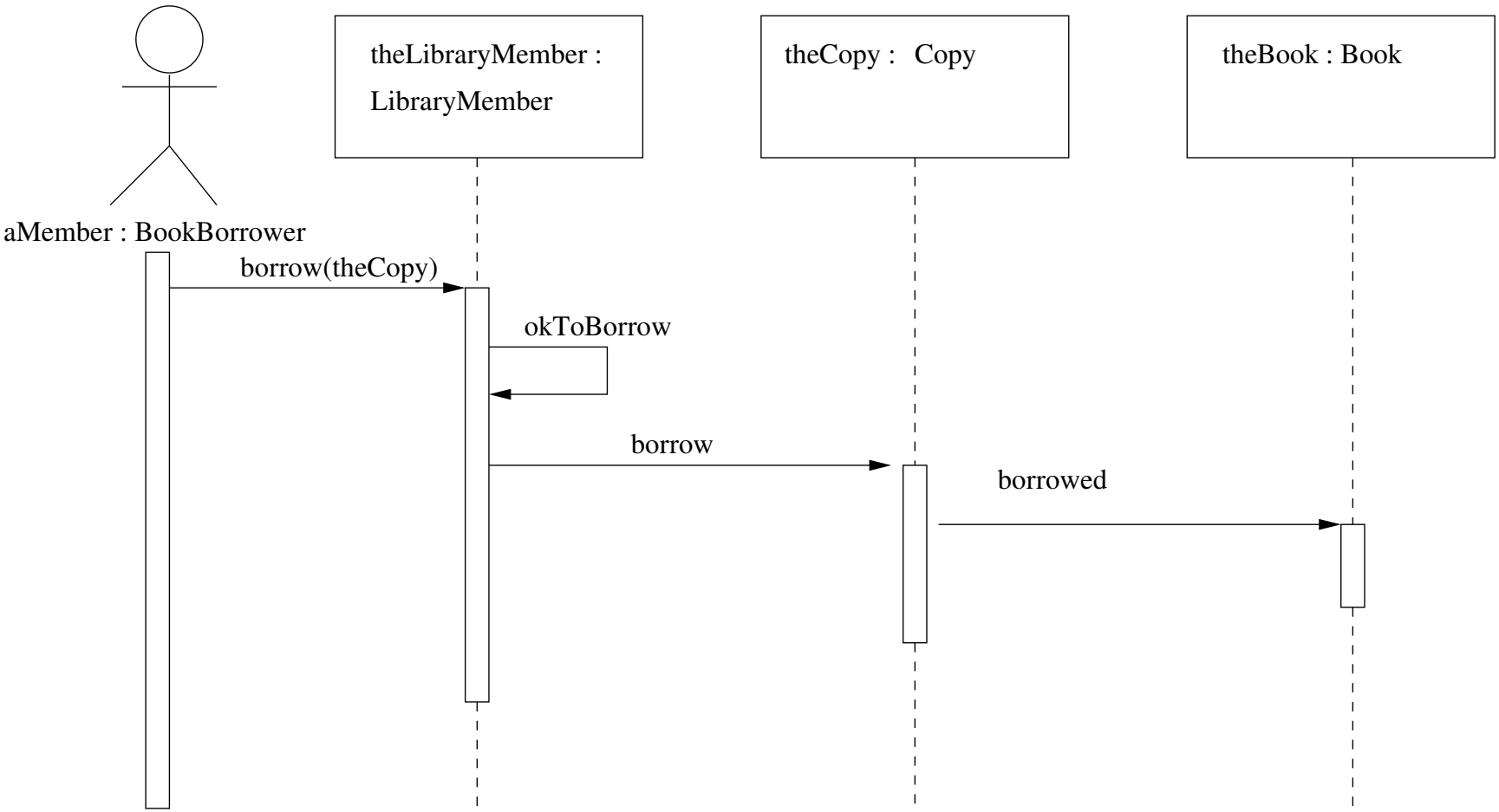


This is quite similar to a UML *communication* diagram

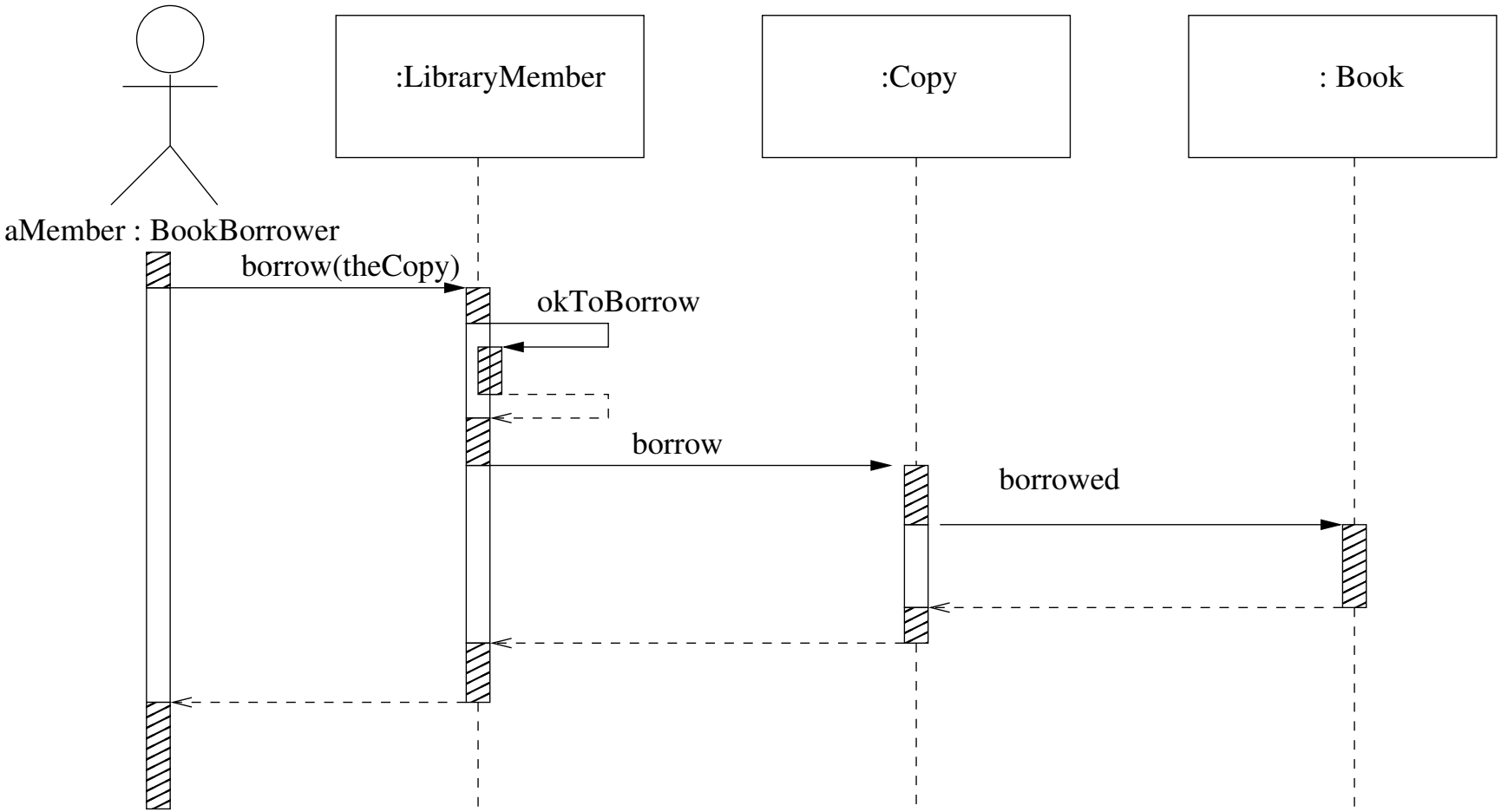
A collaboration II



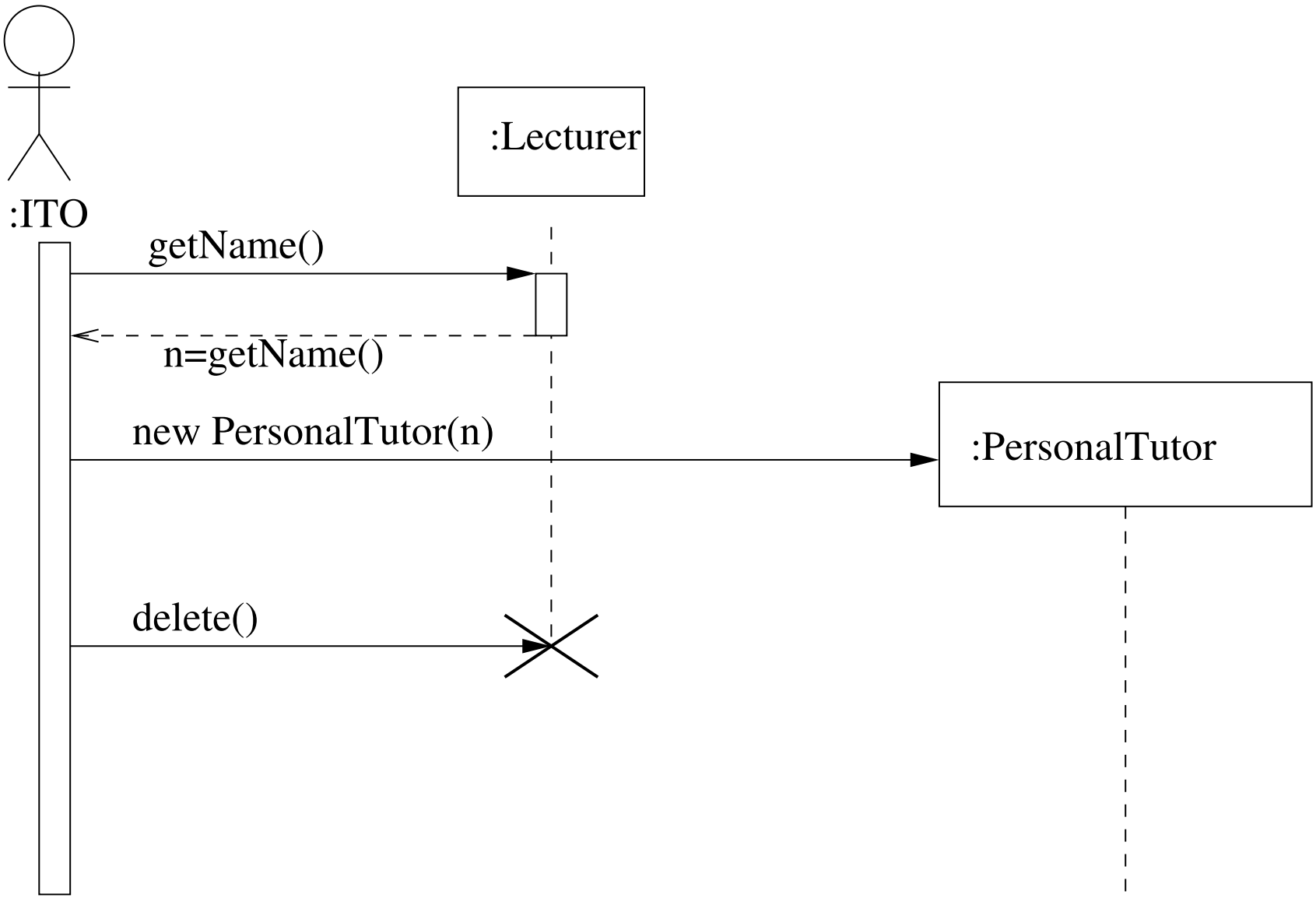
Sequence diagram



Showing more detail



Creation/deletion in sequence diagram



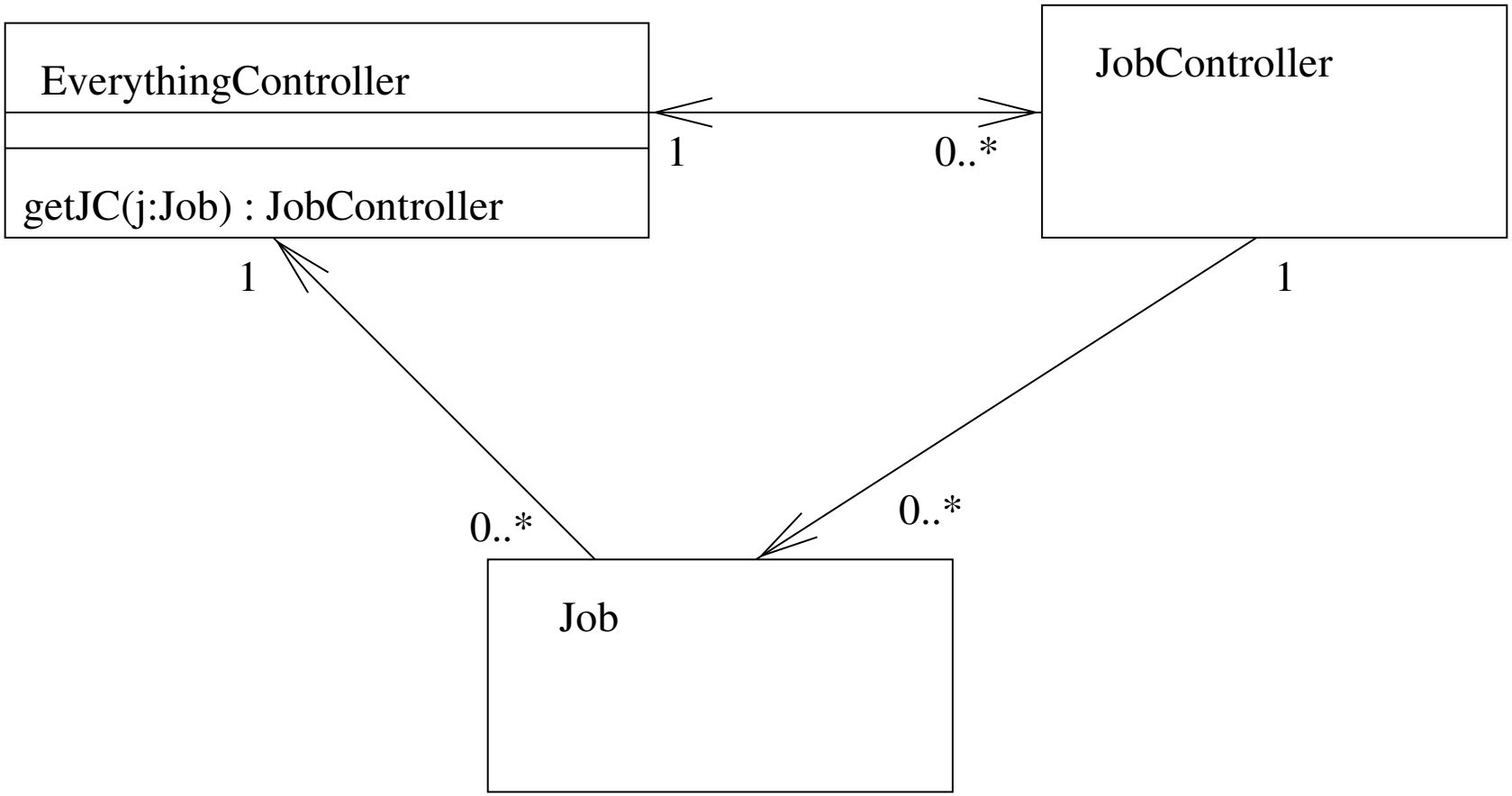
What is a good interaction pattern?

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



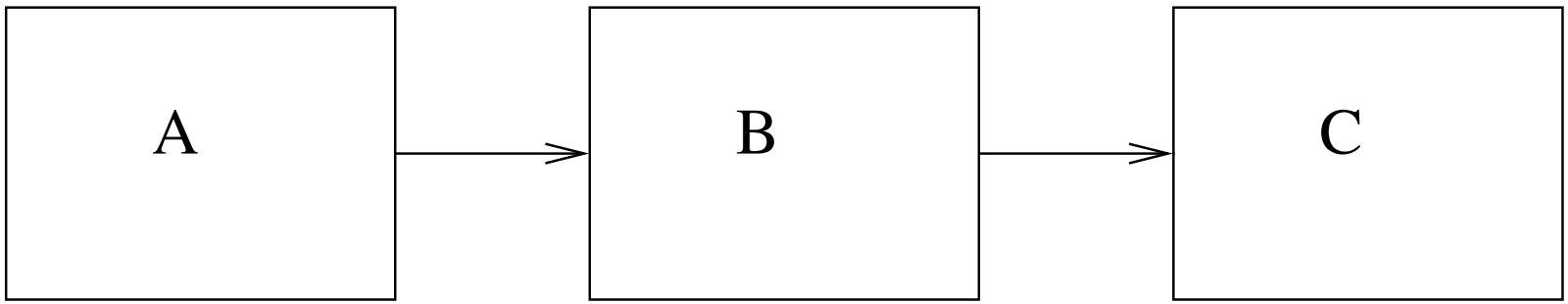
What cohesion and coupling issues are there here?

Reducing longer-range coupling I

The *Law of Demeter* design principle recommends that 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.

Reducing longer-range coupling II



Consider an A operation needing to access part of data in C object.

- We have longer range coupling if operation requests reference to C object from B object
- Better if A's operation calls operation in B designed just to pick out needed data from C

More complex sequence diagrams

We've only discussed very simple sequence diagrams.

UML provides further notation for e.g.

- conditional behaviour
- iterative behaviour
- concurrent behaviour
- Including one diagram in another

For this course, you should be familiar with first two.

Reading

Required: At least one of

- The original paper on CRC cards, a technique for designing interactions: *A Laboratory for Object-Oriented Thinking*, by Kent Beck and Ward Cunningham.
- Stevens, Section 5.6 on CRC cards

You should know the idea of the CRC cards technique, including the basics of what each letter in "CRC" refers to.

CRC cards are covered in detail in SDM.

Suggested: Stevens

- Ch 9: for basics of UML Sequence diagrams
- Ch 10: for conditional and iterative behaviour