The Unified Modelling Language

Perdita Stevens

(Perdita.Stevens@dcs.ed.ac.uk, http://www.dcs.ed.ac.uk/home/pxs/)

Division of Informatics

University of Edinburgh

Slide 1

Objectives

At the end of today:

- you will know what UML is for and its history in brief;
- you will have seen examples of all the main features of UML (though not some of the more esoteric bits)
- you will be able to read and write simple UML models
- we will all understand more about the research agenda around UML.

Do ask questions and comment...

Slide 2

What is a modelling language?

A language for describing models of systems.

A model describes an *aspect* of the system *at a certain level of abstraction*: for example, the class model describes the classes and their (static) relationships, without being concerned with requirements.

Modelling languages are usually diagrammatic, because people seem to find this natural.

Software engineers are used to the idea that programming languages have formally definable syntax and semantics: a program may be legal or not and has a (fairly) certain meaning.

Modelling languages are no different!

Slide 3

Why do we model systems?

Two main reasons:

- To help talk about, think about and work with them before and as we build them: in this case the model is an abstraction of a larger amount of knowledge about the system;
- In order to be able to use them; in this case we want to be able to use the abstraction(s) without needing to know more.

The purposes are related especially in CBD: one design criterion for a good component is that people can understand how to use it.

Must avoid developing models that are not useful!

Slide 4

What is a good modelling language?

It should be:

1. Expressive enough: that is, we can express important aspects of the design, and we can meaningfully reflect changes in the design which we make during analysis and design as changes in the models
2. Easy enough to use, so that it aids clear thought rather than getting in the way
3. Widely used?
4. Supported by suitable tools?
5. Unambiguous

Slide 5

Models of a system

We will want to distinguish models on several axes. For example:

- A *static* model describes the elements of the system and their relationships
- A *dynamic* model describes the behaviour of the system over time

Again, we may take a

- *logical* view: which parts notionally belong together?
- *physical* view: which parts will run on the same computer?

We probably won't need to fill in all the squares...

Slide 6

4+1

Philippe Kruchten

- logical view: how does the system satisfy the functional requirements?
- process view: what are the threads of control?
- development view: how can the system sensibly be built?
- physical view: how will the software be deployed on hardware?

plus the use case view: what should the system achieve?

Slide 7

History of UML

1990s: many different OO development methods each with their own modelling language, including

- Booch's OOD
- Rumbaugh's OMT
- Jacobson's OOSE and Objectory

1994 Rumbaugh joined Booch's company Rational

1995 Jacobson joined Rational: announcement of Unified Method, soon replaced by Unified Modeling Language.

UML1.1 adopted by OMG November 1997, followed by UML 1.3 in June 1999. Many flaws, but obviously going to dominate.

Slide 8

By the way...

Notice significance of having UML instead of Unified Method:

UML tells you nothing about how to develop a system. UML is not a development method.

In the same sense, C++ tells you nothing about how to write programs. Certain strings of symbols are legal C++ programs; a certain C++ program has a (fairly) certain meaning; but the language does not tell you how to write the program.

Nevertheless there is a discipline of C++ programming. Some aspects are controversial, others more or less agreed.

Same with UML: we could discuss some agreed aspects, and some approaches to more controversial aspects...

Slide 9

Present and future of UML

As of 11/2/00, the UML Revision Task Force page lists 3316 issues, ranging from trivial misprints to fundamental flaws. Around 50 outstanding.

Current version of UML is 1.3 (June 1999). Two revision processes ongoing:

- Minor revision, 1.4
- Major revision, 2.0

(**Beware:** Booch Jacobson and Rumbaugh's UML User Guide says it's up to date with respect to 1.3 – this is based on a mistaken anticipation of when that would appear!)

Slide 10

Books and other resources

Vast range. Extremes:

- Using UML: textbook aimed at students, even inexperienced ones
- The official specification, which makes no concessions.

There are also huge numbers of books aimed at professionals. Two deserve special mention:

- Booch Jacobson Rumbaugh UML User Guide, Reference, and Process book.
- Fowler UML Distilled

Web sites: various, including OMG, Rational, UML RTF. See my book page (www.dcs.ed.ac.uk/home/pxs/Book/) for links.

Slide 11

What is an object?

Something you can do things to.

An object has state, behaviour and identity.

State can affect behaviour.

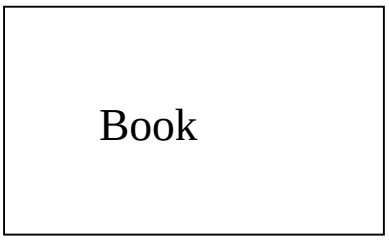
Behaviour can affect state.

Objects communicate by sending messages: the behaviour of an object on receipt of a message is "up to the object".

A class defines the structure and behaviour of similar objects. (That is, their *implementation*, not just the *interfaces* they provide.)

Slide 12

A class



A class as design entity is an example of a **model element**: the rectangle and text form an example of a corresponding **presentation element**.

UML explicitly separates concerns of actual symbols used vs meaning.

Slide 13

An object

This pattern generalises: always show an instance of a classifier using the same symbol as for the classifier, labelled instanceName : classifierName.

Slide 14

Classifiers and instances

An aspect of the UML metamodel (more anon) that it's helpful to understand up front.

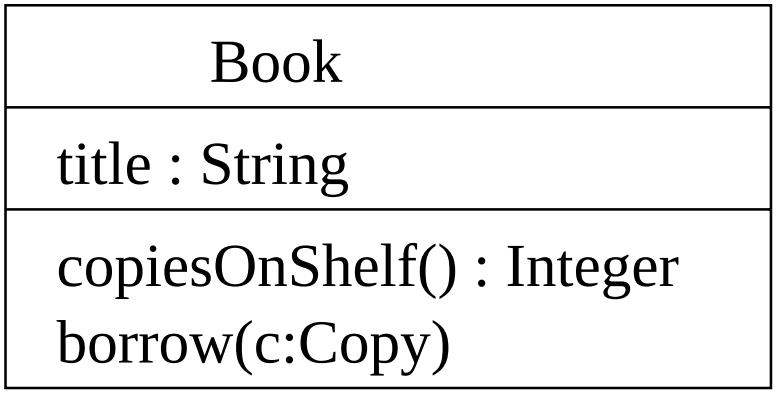
An **instance** is to a **classifier** as an object is to a class: instance and classifier are more general terms.

(In the metamodel, Class inherits from Classifier, Object inherits from Instance.)

We'll see many other examples of classifiers.

Slide 15

Showing attributes and operations

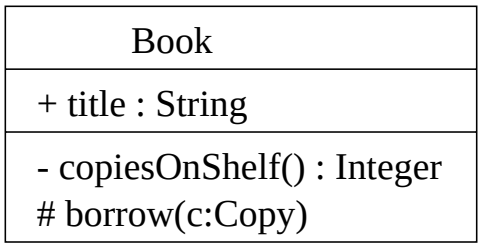


Notice how argument types and return types are shown (can be adapted for different programming languages.)

They may be omitted (together) – oddly though, the formal parameter name is compulsory when the argument list is given.

Slide 16

Visibility



Can show whether an attribute or operation is

- public (visible from everywhere) with +
- private (visible only from inside objects of this class) with −

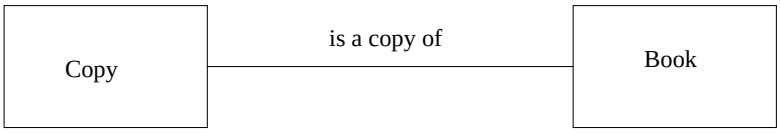
(Or protected, with hash, or other language dependent visibility.)

Can show *abstract* operation or class using italics for the name.

Can add further labelled compartments for other purposes (e.g. responsibilities.)

Slide 17

Association between classes



This generalises: association between classifiers is always shown using a plain line.

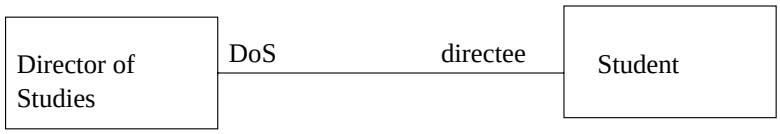
An instance of an association connects objects (e.g. Copy 3 of War and Peace with War and Peace).

An **object diagram** contains objects and links: occasionally useful.

(However in the metamodel an association is not a classifier...)

Slide 18

Rolenames on associations

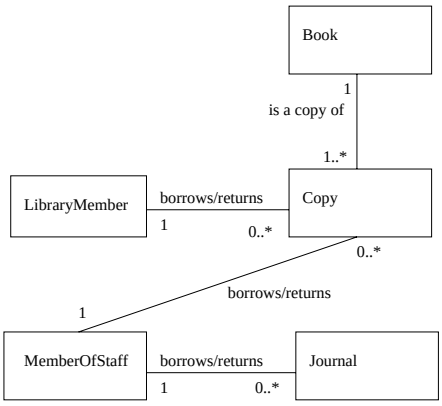


Can show the role that one object plays to the other.

Useful when documenting the class: e.g. a *class invariant* for DirectorOfStudies could refer to the associated Student objects as self.directee (a set, if there can be more than one).

Slide 19

Multiplicity of association

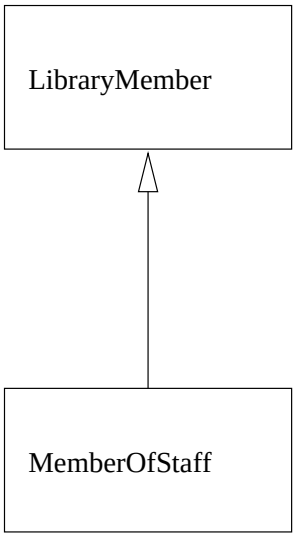


Commas for ranges, *two* dots for ranges, * for unknown number.

E.g. each Copy is a copy of exactly one Book; there must be at least one Copy of every Book.

Slide 20

Generalisation



This generalises: generalisation between classifiers is always shown using this arrow.

Slide 21

Appropriate inheritance

Powerful reuse mechanism but very dangerous because of tight coupling.

cf the *fragile base class problem*: altering the base class affects all its subclasses.

"Implementation inheritance" where a class inherits from another not because they are related by a conceptual generalisation relation but just for code reuse, is *usually* a bad idea.

Overuse of inheritance is probably the single major problem encountered by new OO developers.

Slide 22

Overusing generalisation

Suppose I'm developing a personal organiser application, and I want to implement an appointments diary using the class LinkedList.

I should *not* make the relationship between AppointmentsDiary and LinkedList be a generalisation!

Why not? Because this isn't conceptually a generalisation relationship: it isn't true that an appointments diary is a kind of linked list.

It's tempting, and I've succumbed, because it saves writing one-line wrapper methods for things like add and next – but when I've done this kind of thing I've always regretted it. One of the problems is that you have no option but to expose the whole of the linked list interface, even if (say) you didn't want to allow things in the system to access the number of appointments. This kind of additional dependency can make it hard to alter the implementation later.

Slide 23

Liskov substitutivity

It is surprisingly difficult to come up with a precise definition of when subclassing is safe. Liskov substitutivity is a popular attempt.

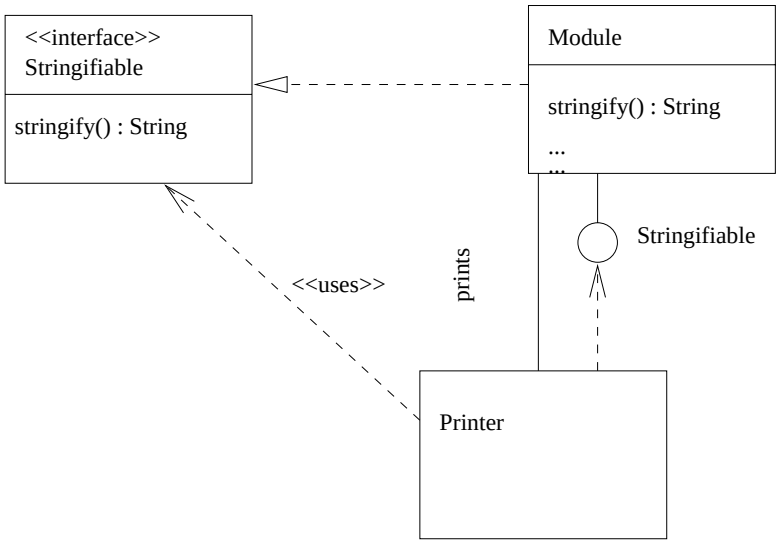
> Suppose some program expects to interact with an object of class C, and that instead it is given an object s of class S, a subclass of C.
>
> If Liskov substitutivity holds, there is some object c of class C which could be used instead of s without altering anything about the behaviour of the program.

Exercise: play with this idea.

Slide 24

Interfaces

In UML an interface is just a collection of operations.



Slide 25

Designed relationships between classes

So far we've dealt with what Fowler calls the *conceptual model*. We've identified the key domain abstractions and the conceptual relationships between them.

Here we look at more advanced features of class models, especially at how to record information pertaining to the design.

Slide 26

Dependencies

To make the system maintainable we want to minimise the dependencies between parts of the system. Not all "real world" connections are reflected in the system.

A is dependent on B if a change to B may force a change to A.

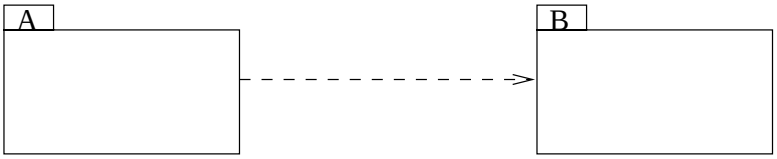
What counts as a change is context-dependent.

Aim to avoid complex dependencies especially circular ones.

Slide 27

Dependencies in UML

Dependencies are shown using a dotted arrow:

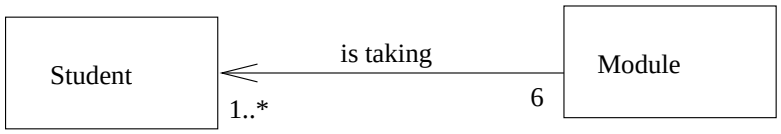


We've seen them between use cases and between a class and an interface: used generally for "relationship not otherwise specified".

Note that some dependencies are implied, and need not be repeated: for example any class depends on its superclasses.

Slide 28

Navigability



When should the navigability of an association be decided?

Some experts believe vehemently that you should never identify an association without deciding its navigability. Others disagree.

"As early as possible, but no earlier."

Slide 29

An aggregation relationship

Non-exclusive part relationship.

A common fault is identifying too many aggregations. If in doubt use plain association.

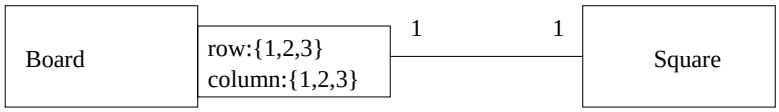
Slide 30

An composition relationship

Exclusive part relationship.

Slide 31

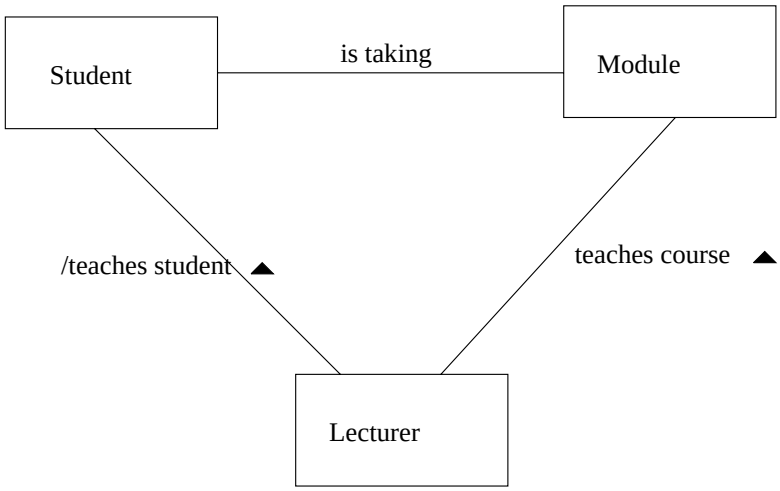
A qualified association



Given a Board and a row and a column, can identify a Square.

Slide 32

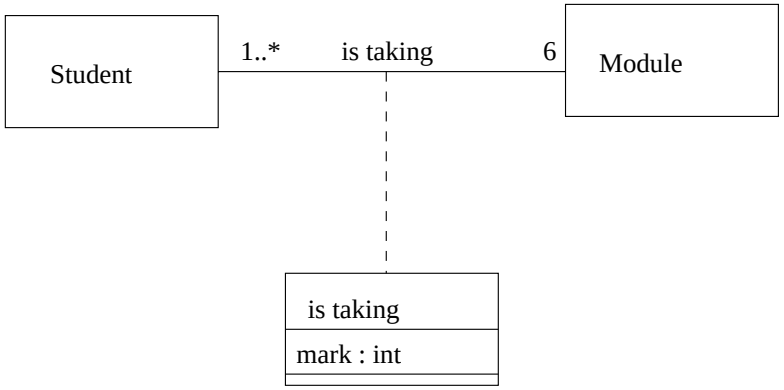
A derived association



Record the association, but also that it's not independent.

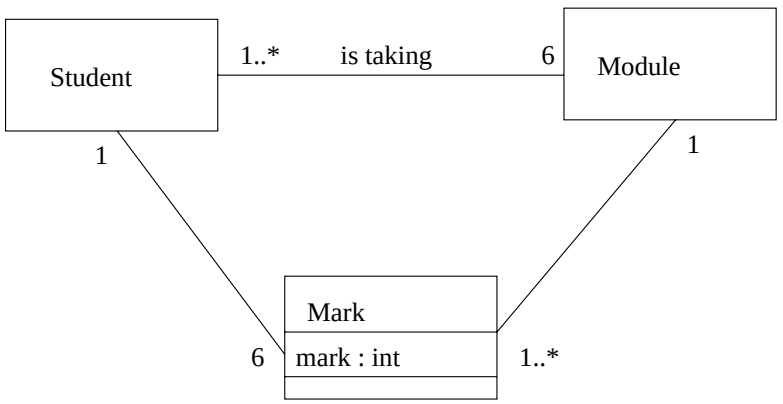
Slide 33

An association class



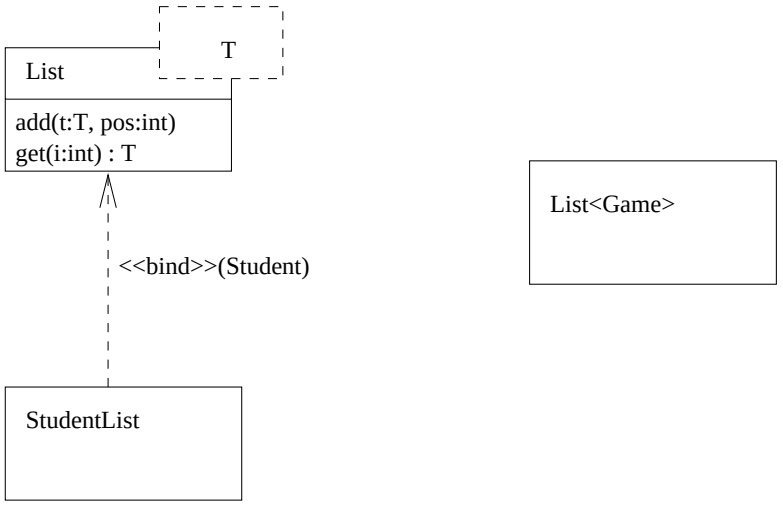
as opposed to

Slide 34



Slide 35

A parameterised class



Slide 36

Interlude...

Slide 37

Use cases

document the behaviour of the system *from the users' points of view.* They help with three of the most difficult aspects of development:

- capturing requirements
- planning iterations of development which are good for users
- meaningful system testing

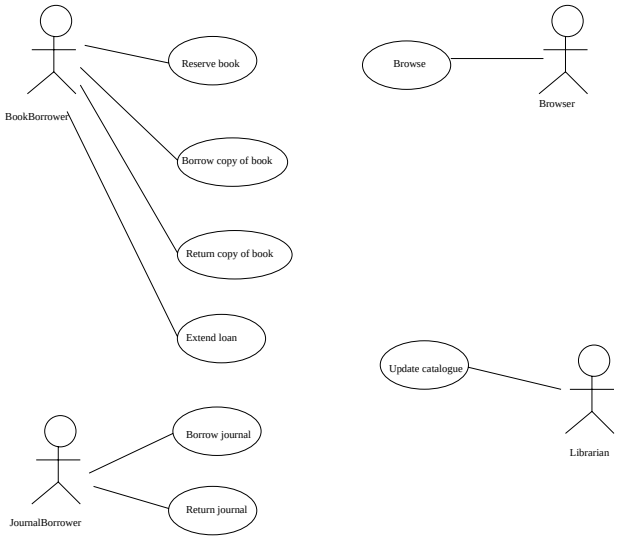
They were first introduced by Ivar Jacobson (early 90s), developing from *scenarios*.

They are independent of OO – strength or weakness??

Simple use case diagrams are easy to understand: can be useful for communication between customers and developers.

Slide 38

A simple use case diagram



Slide 39

Actors

An **actor** is shown in a use case diagram as a stick figure. An actor can be:

- a human user of the system *in a particular rôle*
- an external system, which *in some rôle* interacts with the system.

Or more specifically, a particular *kind* of user. For example, the bank has many customers, but we only show one Customer actor on the use case diagram.

The same human user or external system may interact with the system in more than one rôle: he/she/it will be (partly) represented by more than one actor. (e.g., an bank teller may happen also to be a customer of the bank).

Slide 40

What is a use case?

A **use case** is shown on a use case diagram as a named oval. The name describes some coherent work unit of the system which has value for an actor, e.g. `Borrow copy of book`.

The use case includes a (textual) description of the (a?) sequence of messages exchanged between the system and any actors, and actions performed by the system, in order to realise the functionality.

It may include logic to handle unusual or alternative courses, e.g. "if the `BookBorrower` has the maximum number of books on loan already, refuse this loan" *even though these may result in the actor being unsatisfied.*

A use case may be associated with other UML models which show how it is realised.

Slide 41

Requirements capture

Use cases can help with requirements capture by providing a structured way to go about it:

1. identify the actors
2. for each actor, find out
   - what they need from the system
   - any other interactions they expect to have with the system
   - which use cases have what priority for them

There may be aspects of system behaviour that don't show easily show up as use cases for actors.

Slide 42

Politics

If we capture requirements in terms of use cases, we should understand *what is important to whom.*

Make sure system delivers added value:

- soon
- to all the people who might scupper it
- in every iteration

Result: the project isn't cancelled. Supposedly...

Slide 43

Analysis vs design

Some actors are part of the requirements: usually the ones who derive benefit from a use case.

Others are part of the (business process) design: the ones who interact with the computer system to provide the benefit.

For example, consider a `FindBook` use case of a library, in which the user enters details of a book and wants to end up with a copy of it. Maybe the system will give the user directions to where the book is on the shelf. Maybe it will alert a librarian to go and fetch it. In the latter case, should the librarian be shown as actor? In some sense, the choice is a design decision.

Slide 44

Using use cases in development

Use cases are a good source of system tests: requirements documented as desired interactions, which translate easily into tests.

Earlier, they can help to validate a design. You can walk through how a design realises a use case, checking that the set of classes provides the needed functionality and that the interactions are as expected.

Use cases are not limited to documenting the whole system: they may also describe, e.g.

- subsystems
- classes
- COMPONENTS.

Slide 45

What use cases are not

Use cases document the requirements of a system: not the whole business process into which the system fits.

For example, UML does not permit associations between actors: you cannot legally use a use case diagram to show an interaction between two humans followed by one of them using a system. (E.g. can't legally show librarian and library member as separate actors in Borrow Book, if only the librarian interacts directly with the system.)

There are proposed extensions to UML to allow business process modelling, not considered here.

Slide 46

CBD at the very beginning

We've seen simple use cases. But how can we record which use cases have behaviour in common, or show reuse of components?

Note:

- UML's notation for relationships between use cases has recently changed.
- Even now there are black holes: the formal distinction between the two mechanisms we're about to cover is far from clear.
- Using this extra notation makes use case diagrams less immediately understandable.

Slide 47

Use cases as collections of scenarios

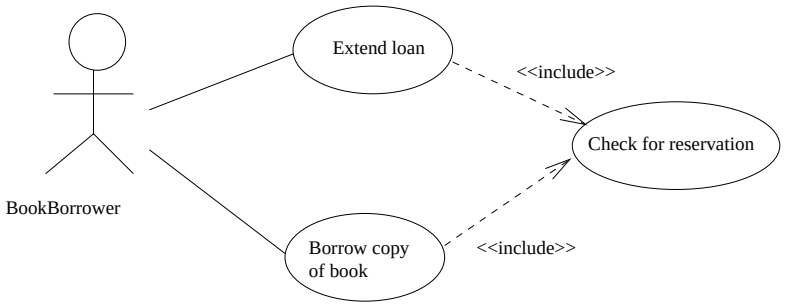
Recall that a use case may include qualitatively different scenarios, e.g. including reaction to error conditions.

Each scenario is viewed as a sequence of actions which are communications between the system and the actors.

(However, beyond that level UML's definition gets informal, even vague...)

Slide 48

Use case reuse: ≪include≫

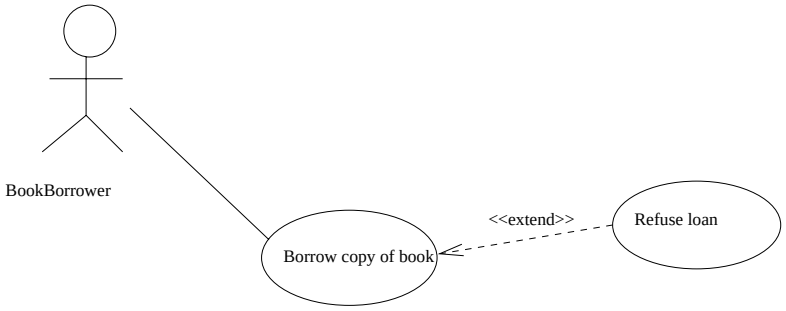


Purpose: to demonstrate commonality between use cases, or the use of an existing component.

A scenario in the base use case (e.g. Extend loan) gets to a certain point, then follows a scenario in the included use case, then returns to the original point and continues with the base use case scenario.

Slide 49

≪extend≫

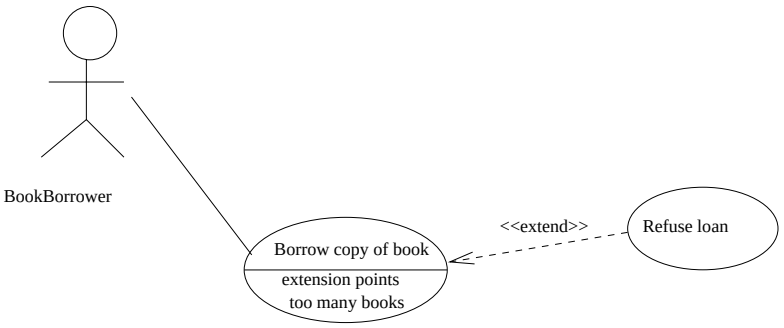


Purpose: to show special case behaviour.

Note direction of dependency arrow: the extending use case may depend on the main use case but not the other way round.

Slide 50

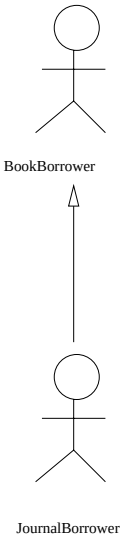
≪extend≫



Optionally, we can document the decision point.

Slide 51

Generalisation of actors



Slide 52

Use case driven development?

use case-driven In the context of the software development life cycle, a process in which use cases are used as a primary artefact for establishing the desired behaviour of the system, for verifying and validating the system's architecture, for testing, and for communicating among the stakeholders of the project.

from Booch Jacobson Rumbaugh UML User Guide

Traceability is mentioned in passing under RUP, but is a major benefit of this approach.

However, the use case model must be used in conjunction with the class model: cannot identify reuse from use case diagram alone!

Slide 53

Shortcomings

1. Danger of losing OO: plan focus on use cases may encourage a top-down functional view of the system
2. Danger of mistaking design for requirements
3. Danger of missing requirements

Use use cases to *guide* a disciplined OO development – don't let them *be* the development.

Slide 54

Extensibility of UML: stereotypes

We've now seen one mechanism by which UML is extensible: stereotypes.

≪include≫ and ≪extend≫ are predefined parts of UML, but you can define your own too.

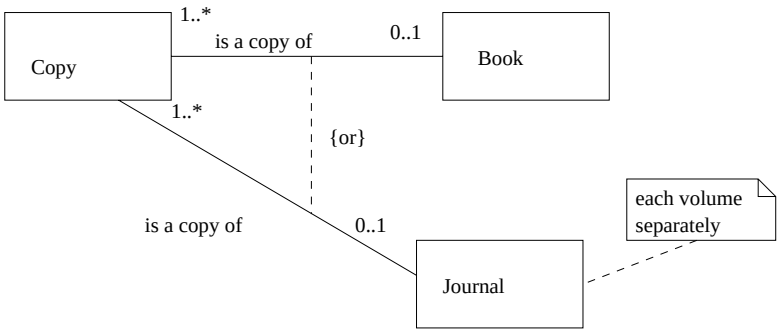
A stereotype makes the model element to which it is attached more specific in a user defined way. For example, if you decide you'd like to distinguish between actors who are beneficiaries and others, you could invent a stereotype ≪beneficiary≫ of actor.

Can define new graphical icons for the stereotyped model element.

Slide 55

Comments and constraints

Used to add to a UML model information not (easily) expressible in UML.



Write the constraint or comment in whatever language (natural or formal) is appropriate.

Slide 56

Constraints in a UML model

Constraints allow you to give more information about what will be considered a correct implementation of a system described in UML.

Specifically, they constrain one or more model elements, by giving conditions which they must satisfy.

They are written in an appropriate language, enclosed in set brackets and attached to the model in some visually clear way.

We'll consider OCL later.

Slide 57

CRC cards

Class, **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.

Slide 58

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 and returned	 `Book`

`Book`	
Responsibilities	**Collaborators**
Maintain data about one book Know whether there are borrowable copies	

Slide 59

Interaction diagrams

describe the *dynamic* interactions between objects in the system, i.e. the pattern of message-passing. They let you record how you wave CRC cards around!

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 *collaboration* diagrams – the differences are syntactic.

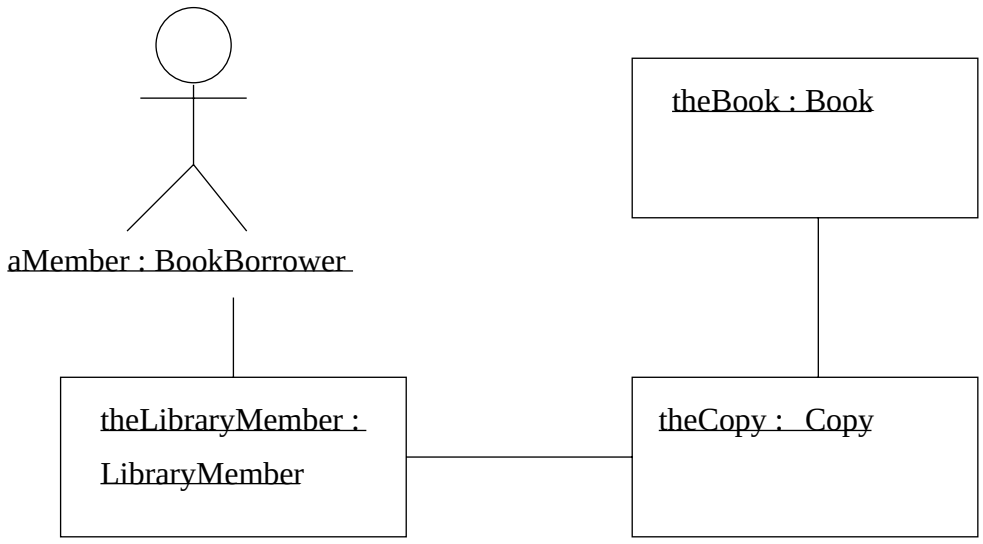
Slide 60

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 or collaboration diagram.

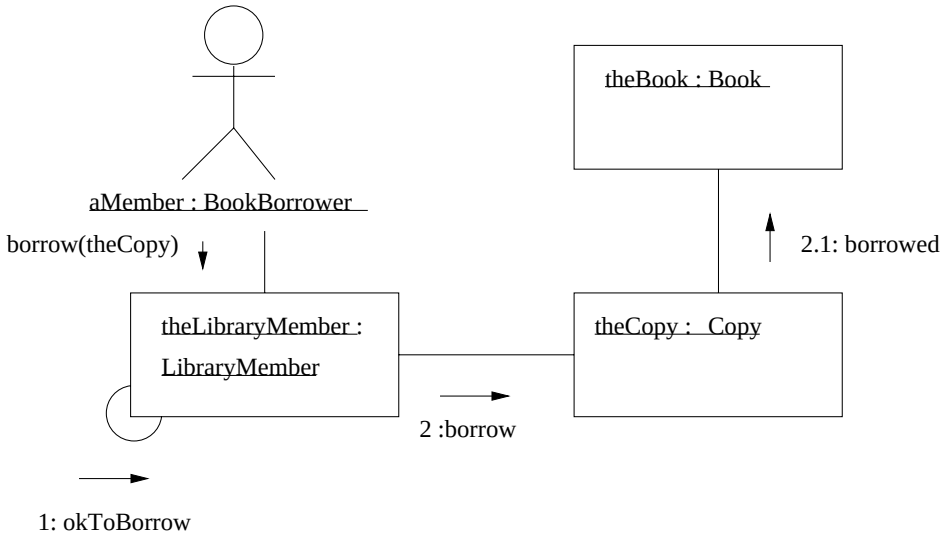
Slide 61

A collaboration



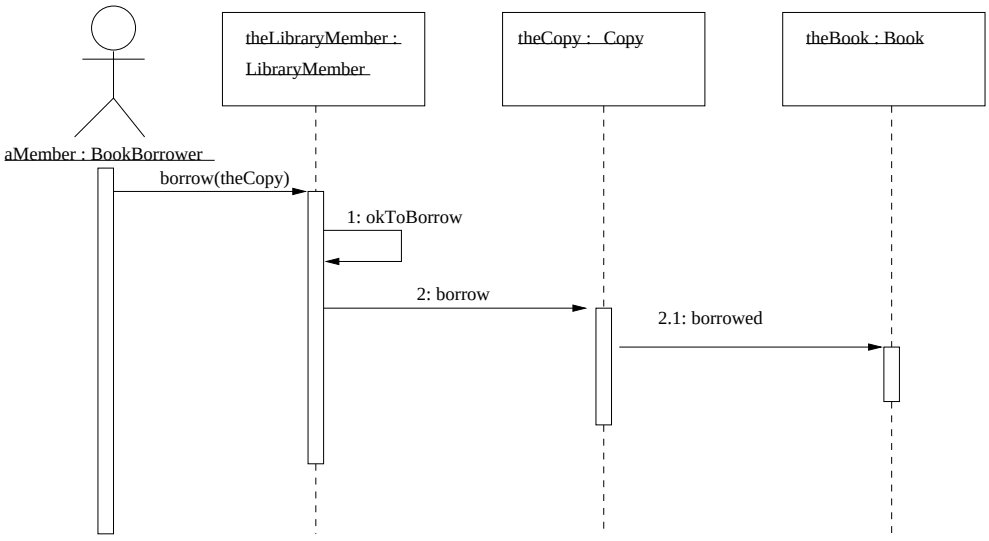
Slide 62

An interaction on a collaboration



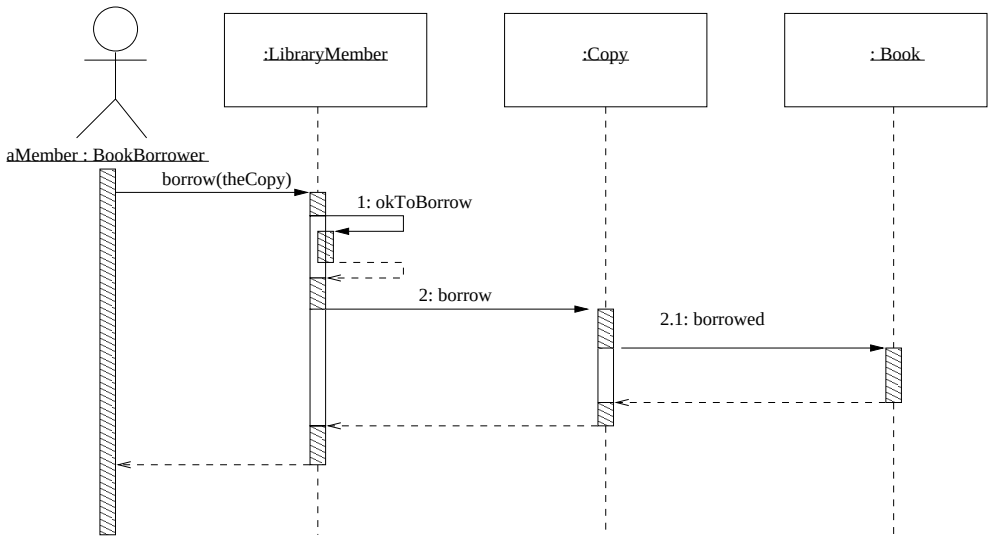
Slide 63

Sequence diagram of same interaction



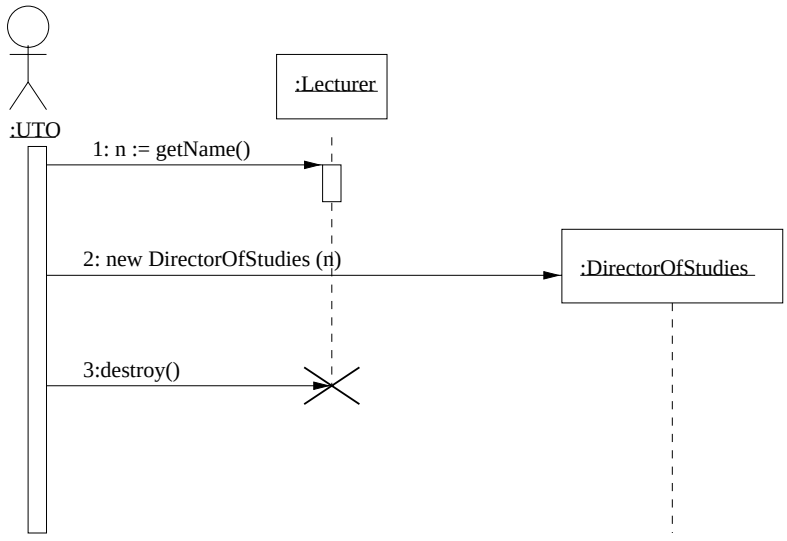
Slide 64

Showing more detail



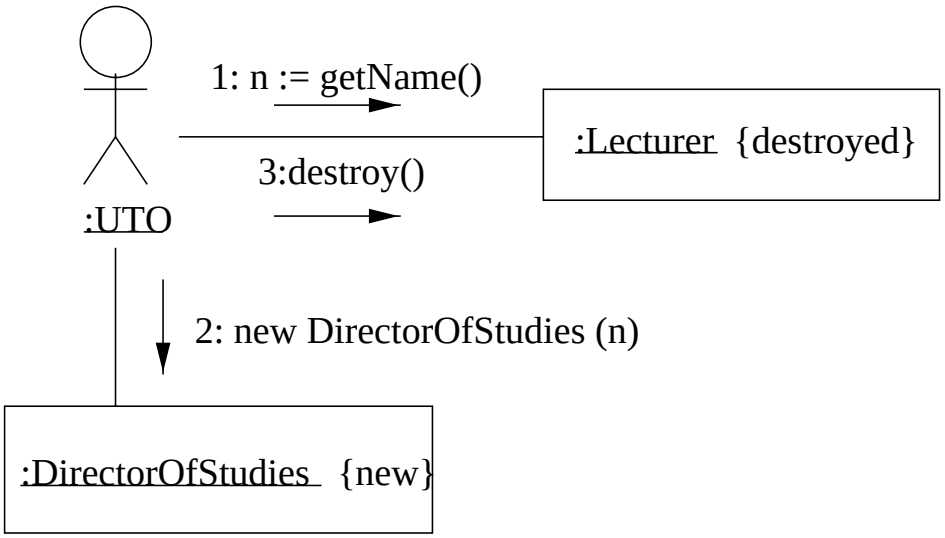
Slide 65

Creation/deletion of objects in sequence diagram



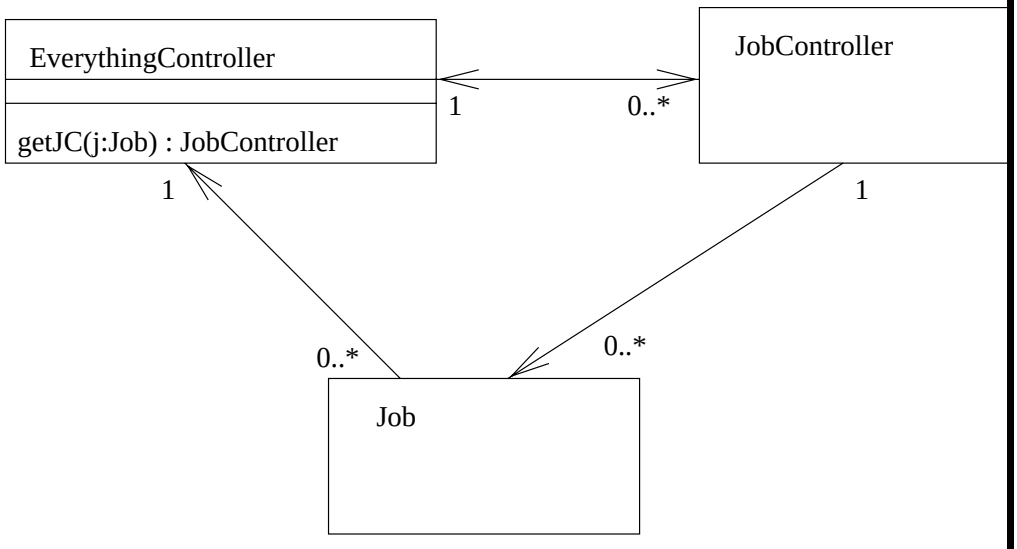
Slide 66

Creation/deletion of objects in collaboration diagram



Slide 67

Designing interactions



Problems?

Slide 68

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.

Slide 69

Collaboration diagram or sequence diagram?

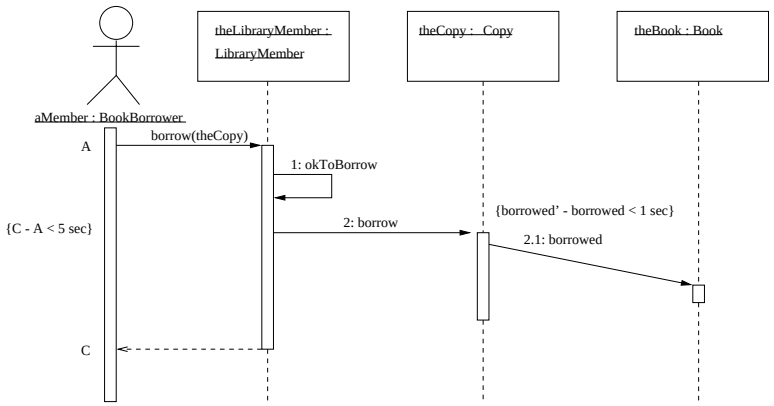
An interaction can be shown on a collaboration diagram or on a sequence diagram. They show almost the same information.

- Collaboration diagrams are better at showing the links between the objects.
- Sequence diagrams are better for seeing the ordered sequence of messages that passes.

We haven't (yet) talked about generic interaction diagrams which allow you to show many possibilities on one diagram: sequence diagrams support these much better than collaboration diagrams.

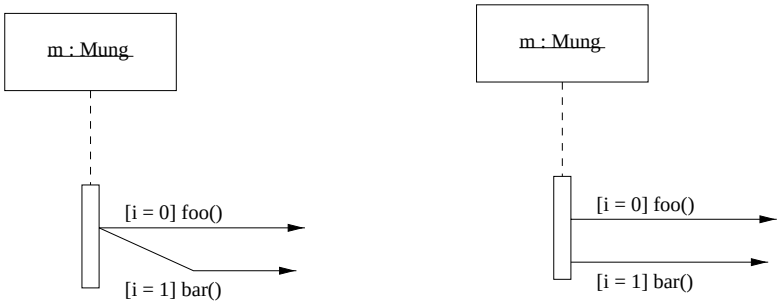
Slide 70

Showing timing constraints



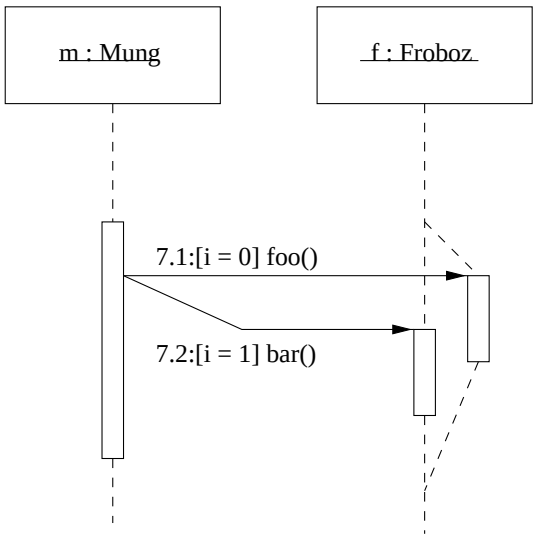
Slide 71

Conditional message send



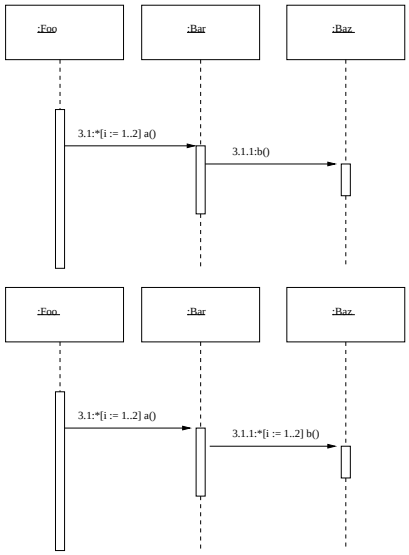
Slide 72

Parallel universes



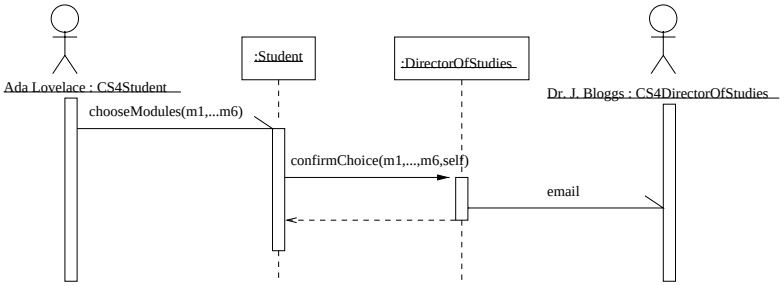
Slide 73

Iteration



Slide 74

Asynchronous messages



Slide 75

Interlude...

Slide 76

Effects of interactions on objects

So far we've seen how to model:

- the requirements of the system with use cases
- the structure of the system with a class model
- the interactions between objects with interaction diagrams

Interactions describe how an object reacts to an event that forms part of that particular interaction. ("What happens next?")

But what determines this? In particular, the same object may react to the same event in different ways, depending on its internal state.

We model this using state diagrams.

Slide 77

State diagrams

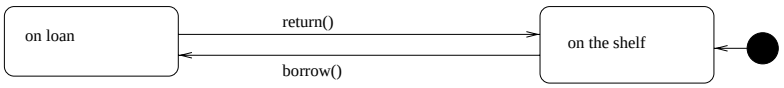
Useful for showing the way that an object of a given class changes state, if it has qualitatively different internal states. May include:

- states
- events that cause transitions between states
- guards that must be true for a transition to take place
- actions that are caused by a given transition
- activities that take place when in a certain state
- start and end markers

States may be nested, but most classes will not need statecharts drawn for them.

Slide 78

A simple state diagram



Slide 79

State diagrams as abstractions

If we could draw infinite[a] diagrams, we could represent each set of values for an object's attributes and links as a separate state and show exactly what happens when the object receives each message. We could include as much detail as the code.

In practice, a state represents an equivalence class of attribute (and link) values: objects which behave qualitatively the same way are in the same state.

That is, a real state diagram represents an abstraction of the "ideal" (and useless!) state diagram.

[a] OK, computers are finite...

Slide 80

So which abstraction?

Formally, could classify objects in different ways, getting different state diagrams for the same class. That is, we could choose different abstractions.

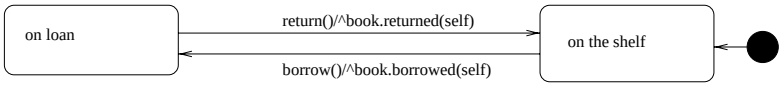
As always the right one is the one that answers the questions of someone using the diagram. This might be:

- the person who must code the class
- a client, writing code that will interact with the class

Differences in what they need to know?

Slide 81

State diagram showing actions



Slide 82

Transitions in more detail

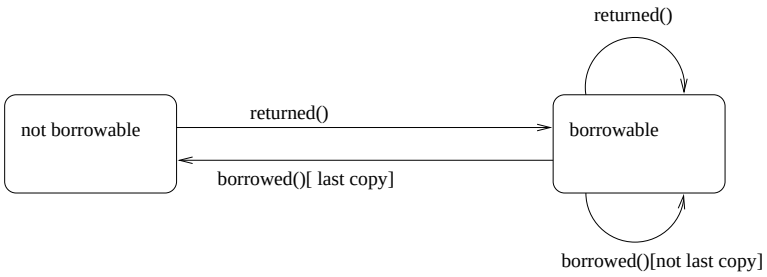
When an object passes from one state to another it does so as a result of an event, e.g. receiving a message. In addition to changing state, the object may react in some way e.g. by sending a message. Such (re)actions are shown after the slash: event/action.

Sometimes an event causes a state change only if a guard is satisfied. The guard is shown: event[guard] / action.

> An event is something done to the object:
> an action is something the object does.

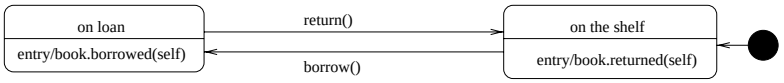
Slide 83

State diagram for Book, with guards



Slide 84

State diagram showing entry actions



Slide 85

Activity diagrams

Useful as an alternative to interaction (sequence or collaboration) diagrams for:

- detailing a use case
- explaining an an object's reaction to a message

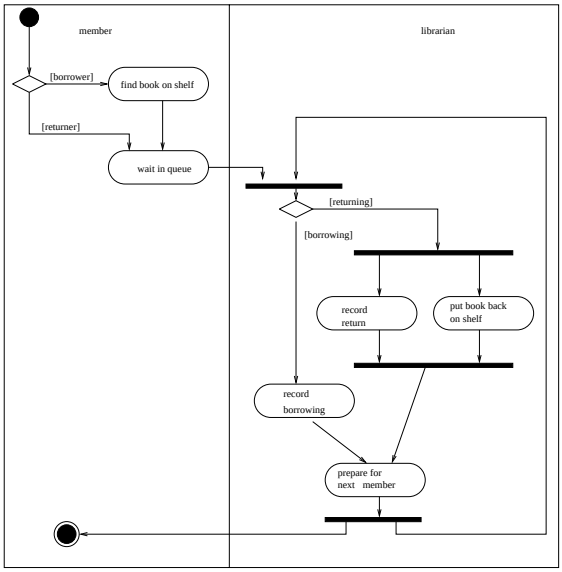
Also useful for showing the dependencies between use cases: e.g. *workflow* of an organisation.

Main advantage: can show parallel activities, so make dependencies and non-dependencies explicit.

Main disadvantage: correspondence with objects in the system may not automatically be clear. (Swimlanes, which for reasons of time we leave out, can help.)

Slide 86

Activity diagram



Slide 87

The development view

Recall the 4+1 view model:

- logical view: how does the system satisfy the functional requirements?
- process view: what are the threads of control?
- development view: how can the system sensibly be built?
- physical view: how will the software be deployed on hardware?

plus the use case view: what should the system achieve?

So far we've really only considered 2+1.

Slide 88

Component diagrams

show dependencies between software components.

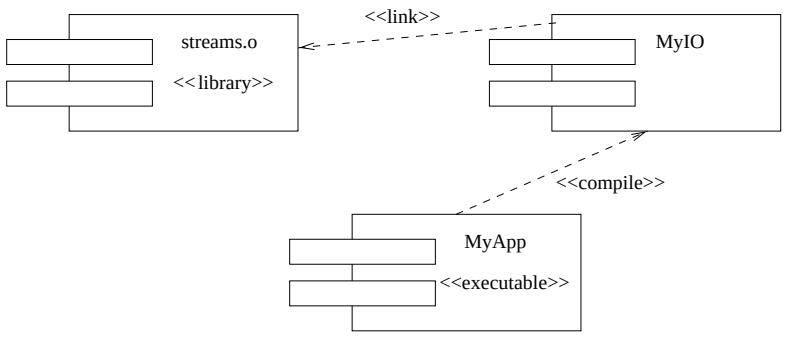
A component may be

- source code (some sensible implementation unit of it, e.g. the code for a given class, or perhaps a start-up shell script)
- binary code (e.g. a class library)
- an executable (e.g. a bought-in spreadsheet).

We'll only consider compilation dependencies (so nothing will depend on an executable, unless it's a compiler!).

Slide 89

A component diagram



Slide 90

Implementing components

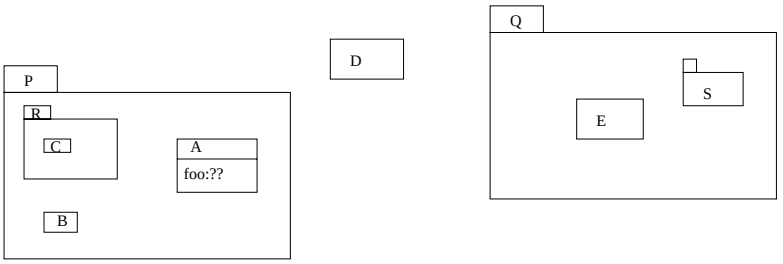
To build a component you need to specify and design it, and you want to have a clear boundary between this component and any other.

UML packages give separate namespaces for this purpose.

A UML subsystem is a kind of package which can have instances: it's particularly useful for modelling components.

Slide 91

Packages



Packages can appear on (almost) any diagram if convenient, and can enclose any "sensible" collection of model elements.

Slide 92

Physical view

It's too easy to forget that software runs on hardware!

Somewhere early in the process you must have considered, for example, the processors needed and the network: achieving decent performance is otherwise impossible. This is outside the scope of the course, however: here we just glance at the UML facilities for recording such things.

Slide 93

Deployment diagram

The deployment diagram shows the relationships between physical machines and processes, e.g. what runs where.

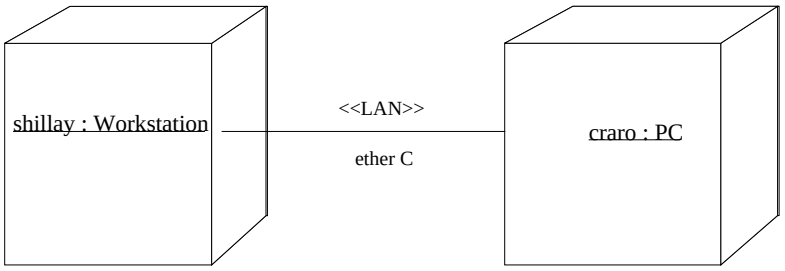
Boxes represent run-time processing elements, usually computers.

Lines between boxes (associations) represent physical communication links.

Components that have a run-time existence (i.e. that don't get compiled away) can be shown in the nodes.

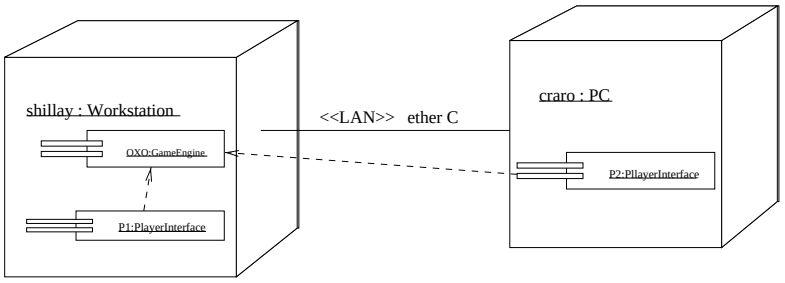
Slide 94

Deployment diagram: hardware only



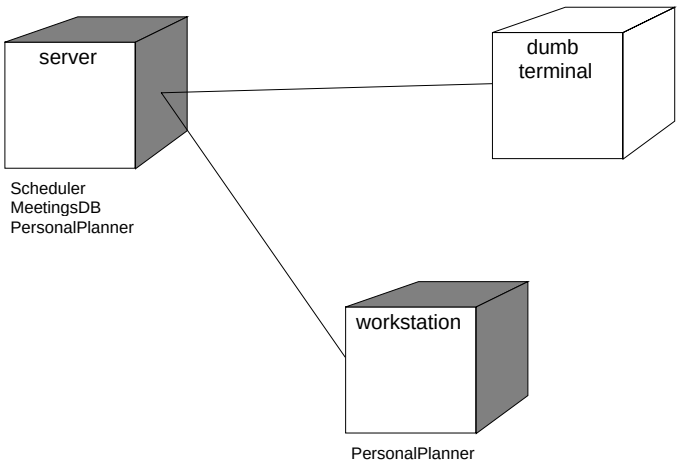
Slide 95

Deployment diagram showing software



Slide 96

A Rose-style deployment diagram



Slide 97

Summary of elements of UML

- Use case diagram
- Class diagram
- Behaviour diagrams:
  - Interaction diagrams:
    - sequence diagram
    - collaboration diagram
  - state diagram
  - activity diagram
- Implementation diagrams:
  - component diagram
  - deployment diagram

Slide 98

Summary

The industry standard modelling language **UML** provides a language in which to talk about designs. It doesn't say anything about how to get the designs.

(Compare English: it doesn't tell you what to say, but you do know whether a sentence is English or not, with some tolerance.)

If we'd had a week rather than a day we'd also have talked more about various aspects of the **development process**, which *is* concerned with how we get designs and programs:

- architecture-centric component based development
- example Methodology: Objectory, Catalysis
- use case analysis
- CRC cards
- design patterns

Slide 99

- achieving reuse

Slide 100

Interlude...

Slide 101

Research and UML

What is the research agenda concerning UML?

Obviously it depends whom you ask. But as I'm in charge here:

- UML provides an important opportunity for TCS to show its worth. At last we have a language which is
  - formal enough to provide ground to stand on;
  - very widely used: makes effort spent on language-specific issues worthwhile
- But there are significant dangers of blowing it. E.g. by:
  - trying to sell formalism for formalism's sake;
  - failing to appreciate that certain decisions have already been made whether we like them or not;
  - not doing things which are exciting enough;
  - doing things that won't ever help actual software engineering practice.

Slide 102

Familiarity with the standard is prerequisite so let's start there.

Slide 103

How UML is defined

Two main documents:

- Notation Guide : informal explanation of notation (concrete syntax) and its connection to abstract syntax.
- Semantics : semi-formal specification of abstract syntax, plus further explanation of semantics.

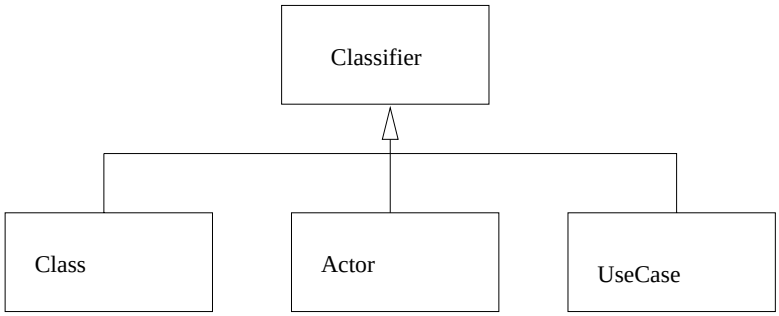
Semantics takes precedence over Notation Guide in cases of conflict – theoretically.

Plus: definition of the Object Constraint Language (OCL).

Slide 104

Reading the UML semantics document 1

The **abstract syntax** describes (in UML!) the relationships between kinds of UML model elements (the *metamodel*). For example, use cases, actors and classes are all said to be examples of classifiers:



Slide 105

Reading the UML semantics document 2

Well-formedness rules put further restrictions on what it means to be a correct UML model.

For example, a constraint on a Generalization states:

```
self.subtype.oclType = self.supertype.oclType
```

This expresses concisely that even though a Generalization can exist between classes or between actors, a class cannot be a generalization of an actor, etc.. (self refers to the Generalization; the abstract syntax defines that any Generalization is associated with two GeneralizableElements referred to as subtype and supertype; the oclType of any class is Class, etc.)

The section Semantics contains a variety of further explanation, in English.

Slide 106

Object Constraint Language

The well-formedness rules are given in a language called OCL.

Unfortunately on a formal level, OCL was developed (from a previously existing specification language) simultaneously with UML, and it too has unresolved problems!

OCL does not (yet) have a formal semantics. Thus though you will hear it described as a formal specification language, this is at best dubious.

Issues:

- does a given design model a given OCL statement?
- what are the proof rules for OCL?
- what is the type system for OCL?

and do they have the appropriate properties?

...

Slide 107

Semantics for UML(?)

Very controversial:

- does UML need a formal semantics at all?
- if so, of what kind?
- WHAT FOR?

There is a lot of work in the area.

NB need to read the semantics document!

But let's not get carried away: semantics for UML is only one possible research area...

Slide 108

Supporting Software Design

Present-day CASE tools are very limited.

Major issues of the moment include usability, being up to date with the standard, performance, price.

More interesting to us perhaps: power.

How can the practice of mainstream software design can be well supported by formal techniques?

Two possible approaches:

1. "Money no object": how can formality help raise the ceiling, increase the maximum dependability of systems? "Formal methods"
2. How can formality help *without* raising cost or requiring software engineers to be more mathematically oriented than they are now?

I'm personally most interested in 2.

Slide 109

Some open questions

1. How can what-if experiments be made easier? To what extent could a tool suggest scenarios which should be explored, provide support in exploring the dynamic implications of earlier decisions, etc.?
2. How can design by contract be supported?
   - Languages for appropriate contracts, which a tool can treat as formal statements, without requiring the designer to learn a formal language?
   - Unfinished design? Tool must support the identification of (in)dependent elements, and must not simply give up in the case of a possibly violated contract.
3. How can we ease the use and development of components and other chunks of designs, such as product-line architectures (PLAs) and frameworks?
   - complex interfaces (not just single operations)
   - restrictions on how choices are resolved (e.g. who chooses?)

Slide 110

- exploring, and perhaps changing or making explicit, the assumptions made about the environment.

Slide 111

Current state and progress

Some things it's clear tools could do now, without further theoretical progress:

- more ambitious sanity checking of models - provided can avoid ambiguities in semantics e.g. what associations mean...
- animation - in practice, how useful is this?
- any amount of model checking, provided the user agrees with your interpretation of the semantics
- more code generation - but it's easier to write code than UML models so maybe demand for this is limited

Challenge: do something useful enough.

I'm exploring how games may help in exploration of consequences of design decisions... watch this space.

For now, let's focus close to home on design by contract using OCL.

Slide 112

Design by Contract

The key is to avoid ambiguous situations in which something goes wrong but there are several views about whose faults it is.

By making explicit the contract between supplier of a service and the client, D by C

- contributes to avoiding misunderstandings and hard-to-track bugs;
- supports clear documentation of a module – clients should not feel the need to read the code!
- supports defensive programming;
- allows avoidance of double testing.

Slide 113

Example: class invariants

A class invariant restricts the legal objects by specifying a relationship between the attributes and/or the attributes of associated classes.

Simple example: the class invariant

{name is no longer than 32 characters}

could be applied to a class Student which has an attribute

name : String

to forbid certain values of that attribute.

Implementors of the class must ensure that the invariant is satisfied (when?)

Clients of the class may assume it.

Slide 114

Less simple example

Suppose our class Student is associated with classes DirectorOfStudies and also with Lecturer by tutor – a student has a DoS and a tutor.

Suppose it is forbidden for the student's DoS and tutor to be the same person.

We can represent this by a class invariant on Student, say

{ student's tutor and DoS are different }

(Is this sufficiently unambiguous?)

Slide 115

Constraining implementations of operations

We can constrain the behaviour of operations using pre and post conditions.

A pre condition must be true before the operation is invoked – it is the client's responsibility to ensure this.

A post condition must be true after the operation has been carried out – it is the class's implementor's responsibility to ensure this.

E.g.

Module::register(s : Student)

pre : s is not registered for the module

post : the set of students registered for the module is whatever it was before plus student s.

Slide 116

Subcontracting

When a subclass reimplements an operation it must fulfill the contract entered into by its base class – for substitutivity. A client must not get a nasty surprise because in fact a subclass did the job.

Rule of subcontracting:

Demand no more: promise no less

It's OK for a subclass to weaken the precondition, i.e. to work correctly in more situations... but not OK for it to strengthen it.

It's OK for a subclass to strengthen the postcondition, i.e. to promise more stringent conditions... but not OK for it to weaken it.

Slide 117

Languages for contracts

Writing contracts in English can be

- ambiguous
- long-winded
- hard to support with tools

Sometimes it's desirable to use a mathematically-based language – but such languages can be hard to learn.

OCL aims to be both formal and simple.

Slide 118

OCL basic types

- Boolean
- String
- Integer
- Real

With all the operations you'd expect.

Integer is considered a subtype of Real.

(Remark: OCL uses the terms class and type interchangeably, which is just about OK in this context, though normally a big mistake.)

Slide 119

Example: pre and post conditions

```
Stove::open()
-------------
pre : status = #off
post : status = #off and isOpen

ElectricStove::open()
---------------------
pre : temperature <= 100
post : isOpen
```

Do you have to re-specify the inherited precondition? Yes – not to do so is just too confusing. So instead include the `status = #off` everywhere.

Slide 120

OCL collection types

- Collection
- Set
- Bag
- Sequence

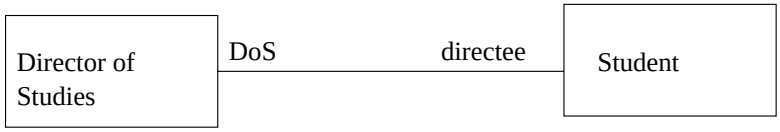
with the usual operations (size, includes, isEmpty, ...)

Things that would be collections of collections are "automatically flattened" – OCL cannot talk about a sequence of sets, etc.!!

Slide 121

Navigation

An OCL expression in the context of one class A may refer to an associated class B.



Single (? - 1) association: straightforward, since any object of class A determines just one object of class B:

- If there's a rolename use it, e.g. `self.DoS.name`
- If not may just use classname, e.g. `self.directorOfStudies.name`

Slide 122

More navigation

What if the association is not (? - 1)? E.g. consider the same association from the point of view of the DirectorOfStudies – a DoS may direct many Students.

For each DirectorOfStudies the rolename directee refers to a *set* of Students. Use OCL collection operations, e.g.

- `self.directee->forAll (regNo <= 200000)`
- `self.directee->notEmpty`

(If you use a collection operation on something that isn't a collection it gets interpreted as a set containing one element!)

Slide 123

The plot thickens

What happens if we take more than one "hop" round the class diagram?

e.g. what is self.student.module?

It should be a set of sets, but OCL doesn't have them!

So it's a bag.

Tempting to regard this as a bug, but it's very convenient in practice; what is the right compromise?

Slide 124

Using operations in OCL

Consider an operation `register(s:Student)` of `Module`. Should we be able to refer to this operation in an OCL expression?

Problem: it does something – alters the state of the Module. When should this happen, if at all?

Only good way round this is to allow in OCL *only* operations that guarantee not to alter the state of any object.

Such operations are known as queries – in UML an operation (in fact any BehavioralFeature) has an attribute isQuery which must be true for the operation to be legal in OCL.

(UML doesn't currently specify that such operations should guarantee to terminate, but presumably it should. Is there a problem with inheritance?)

Slide 125

Beyond OCL

Current activity in the UML RTF is focusing on dealing with OCL issues one at a time.

Personally I think something more radical is required, maybe in UML2.0...

As well as a better OCL, what more might be useful for design by contract?

Languages for expressing more than one can in OCL... with temporal features? with independence?

Challenge: keep these usable by developers and supportable by tools.

Slide 126

Conclusion

Now:

- you know what UML is for and its history in brief;
- you have seen examples of all the main features of UML (though not some of the more esoteric bits)
- you can read and write simple UML models
- we all understand more about the research agenda around UML!

Thank you for participating.

Slide 127