More on the Unified Modeling Language

Perdita Stevens, University of Edinburgh

July 2010

Agenda

- And the rest... deployment diagrams, component diagrams, object diagrams, timing diagrams, etc.
- OCL and alternatives (programming language, English, informal mathematics, formal specification lang.)
- Ways of modelling: agile modelling, executable modelling, model-driven development, DSMLs

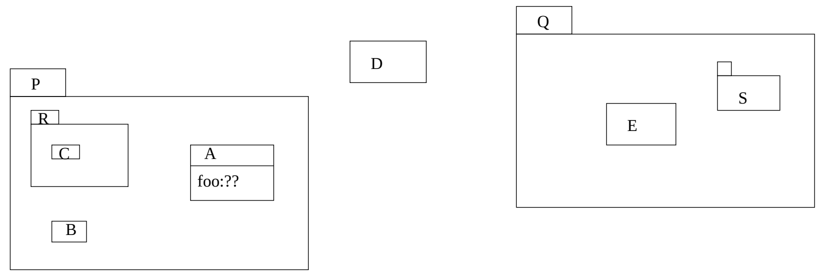
UML diagram types

Recall that we have covered the most basic elements of:

- Class diagrams (static structure)
- Use case diagrams (summary of requirements)
- Sequence diagrams (inter-object behaviour)
- State diagrams (intra-object behaviour)
- Activity diagrams (workflow; parallelism)

Now - extremely briefly! - for the rest.

Packages



Using packages in UML

Packages are basically namespaces, as in Java.

Main use: represent how Java (or other PL) packages will be used in the system. E.g.

1. a diagram of the packages can show hierarchy and dependencies of packages;
2. using packages in a class diagram can show which individual classes depend on other packages.

However: packages can appear on (almost) any diagram if convenient, and can enclose any “sensible” collection of model elements.

Unlike Java packages, UML packages give a genuinely hierarchical namespace.

Component diagrams

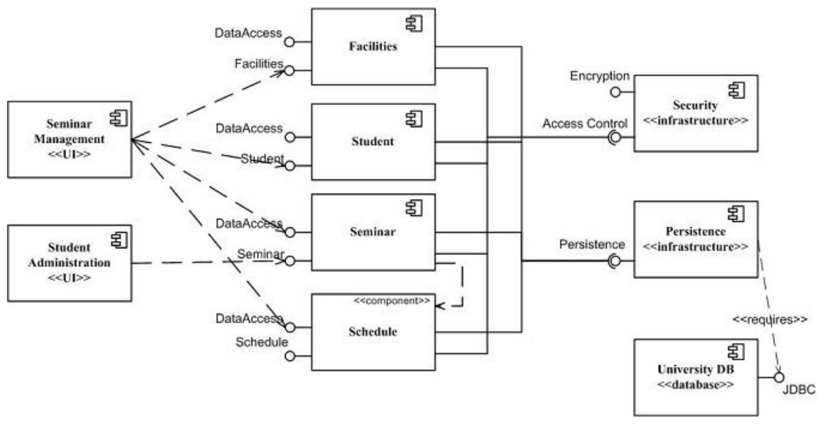
show dependencies between software components and how components are composed.

A component can be higher level than a class, and unlike a package can be instantiated.

Exactly what components are is technology-dependent – rather fruitless to make rules at the UML level.

Beware: UML’s treatment of components and deployment has changed radically since UML1.x.

A component diagram, from Scott Ambler



http://www.agilemodeling.com/artifacts/componentDiagram.htm

Physical view

So far we have ignore the hardware.

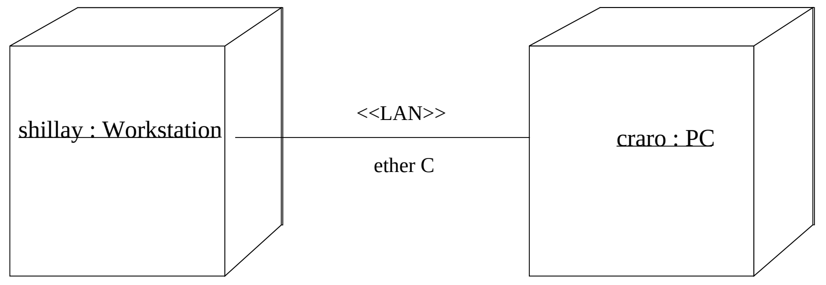
Somewhere early in the process you must have considered, for example, the processors needed and the network: achieving decent performance is otherwise impossible. This is outside the scope of the course, however: here we just glance at the UML facilities for recording such things.

Deployment diagram

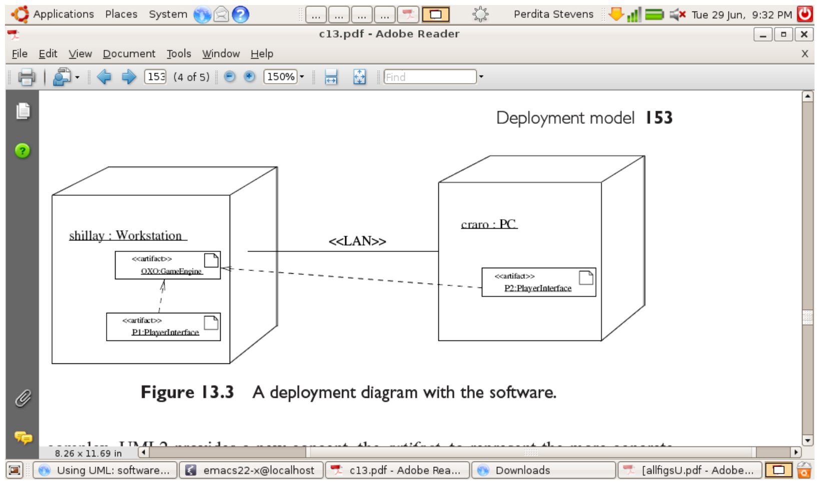
The deployment diagram shows the relationships between physical machines and artifacts – physically deployable manifestations of components – e.g. what runs where.

Boxes represent run-time processing elements, usually computers. Lines between boxes (associations) represent physical communication links.

Deployment diagram: hardware only



Deployment diagram showing software



Deployment model **153**

Figure 13.3 A deployment diagram with the software.

Things we are really not going to talk about

Interaction overview diagrams are a kind of hybrid of sequence diagrams with activity diagrams

Timing diagrams show the change in state of an object in response to stimuli over time.

Beyond diagrams

Diagrams are great, but...

...sometimes something you need to specify about a design is hard or impossible to express using diagrammatic UML.

Let's look at things we may want to express, before we consider how to express them.

Generally we'll talk about *constraints*, with *contracts* as an important special case.

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.

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.

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?)

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.

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.

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:

$\boxed{\text{Demand no more: promise no less}}$

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.

Languages for contracts

Writing contracts in English can be

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

– but nevertheless, careful English is very often the best language to use!

Formal languages for contracts

If English is not good enough, you could consider:

- the chosen programming language – e.g. write a Boolean expression in Java that should evaluate to true.
  + can be pasted into the implementation and checked at runtime
  − may be too low level, e.g. lack quantifiers
- "plain" mathematics and/or logic
  + don't need any special knowledge or facilities
  − can end up being the worst of all worlds, e.g. unfamiliar, lacking UML integration
- a formal specification language e.g. Z or VDM
  + can be truly unambiguous, some tool support exists
  − unfamiliar to most people, may be non-ASCII, need special UML-integrated dialect
- The Object Constraint Language, OCL

About OCL

OCL aimed for the sweet spot between formal specification languages and use of English. It tries to be formal but easy to learn and use.

Extensively used in the documentation of the UML language itself, and related standards.

Written in ASCII.

Had serious semantic problems, but these seem to be solved now.

Some tool support.

Worth knowing a bit about.

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.)

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.

OCL collection types

- Collection
- Set
- Bag
- Sequence

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

Reasonable facilities for manipulating collections. E.g.

```
context Company inv:
self.employee->select(age > 50)->notEmpty()
```

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`

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!)

Two-stage navigation

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

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

It's deemed to be short for

```
self.student->collect(module)
```

which is a *Bag* (not a Set) of all the modules taken by students linked to self.

Notice that putting such a constraint into a UML model creates a dependency of self on module, if there wasn't one already.

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 has an attribute isQuery which must be true for the operation to be legal in OCL.

Ways of modelling

Modelling languages are used in very different ways. All can be good provided there is consensus!

- agile modelling
- executable modelling
- model-driven development
- DSMLs