Software Engineering

with

Objects and Components

Perdita Stevens

CS4 course, Autumn 1997
Department of Computer Science
University of Edinburgh

Contents

1 Defining terms

1.1 Software Engineering

A doctor, a civil engineer and a software engineer were arguing about what was the oldest profession in the world. The doctor remarked, "Well, in the Bible, it says that God created Eve from a rib taken out of Adam. This clearly required surgery, and so I can rightly claim that mine is the oldest profession in the world." The civil engineer interrupted, and said, "But even earlier in the book of Genesis, it says that God created the order of the heavens and the earth out of the chaos. This was the first and certainly the most spectacular application of civil engineering. Therefore, fair doctor, you are wrong: mine is the oldest profession in the world." The software engineer leaned back in her chair, smiled, and then said confidently, "Ah, but who do you think created the chaos?"

What is software engineering?

We make an analogy between the development of *systems* (actually, not necessarily exclusively software systems) and the development of, say, an engine or a bridge (hereinafter "engine" for short).

Specifically:

- use a defined *process*, with clear *phases* each of which has an *end-product* (system specification, test spec, whatever)
- concerned with meeting a clear set of requirements defined at the beginning of the project, so
- regard testing (and more generally, verification and validation) as just as essential as building the product itself
- use a store of relevant *knowledge* (and *components*?)
- make sensible use of tools

Slide 1

Why do software engineering?

So what's so great about an engineering approach to software? After all, software isn't a physical object, it doesn't have to be manufactured – it's pretty different from most engineered artifacts.

It's clear we have a problem. (Ariane 5, Taurus, Denver baggage handling system, ...)

But is an engineering approach a solution?

Discussion point: what are the similarities and differences between software and engines, and the way they're developed? What should they be?

Slide 2

1.1.1 The problem

We have a *software crisis* that doesn't seem to want to go away. There are numerous well-known examples, and you probably know others. For example:

- Ariane 5
- Taurus
- Denver baggage handling system
- CONFIRM travel booking system
- ...

According to Scientific American, September 94:

- On average, large projects take 50% longer than they were planned to do
- Three-quarters of large projects are operational failures
- One-quarter of large projects are cancelled

(My tertiary source for this: Larry Paulson's notes on software engineering, which towards the beginning have some good notes on several software disasters: see link on course home page.)

1.1.2 Similarities and differences between software and engines:

Here's a conglomeration of things people did say in the meeting and things I expected them to say about the similarities and differences between software and "other engineered artifacts".

Similarities

- Designed to solve a problem
- May have to recall faulty products
- May fail under unforeseen circumstances
- Expensive to develop
- Disastrous to get wrong: severe consequences in terms of
  - lives
  - money
  - business reputation (and hence survival)
  - jobs
  - time

Differences:

1. Defining the problem to be solved is usually *much* harder for software than for engines. Therefore: will get it wrong, and pay the price.
2. No (interesting) "manufacturing" (replication) stage – more like building a bridge than building an engine. What about mass-produced word-processors etc.? Why do I say this isn't really "manufacturing"?
3. Software is an "invisible medium" in the sense that unless we take positive steps to alter this, the customer and the product won't meet till very late in the development. But:
4. prototyping (building something as a demonstration) may actually be easier in software than hardware: use this to mitigate the previous point.
5. Software can be arbitrarily complex
6. Therefore people often consider software to be much more fluid than a physical artifact: customers may expect, and developers may promise, very high levels of flexibility, late changes, etc. Combined with (1), get
7. Defining interfaces may be harder for software than hardware, partly because of the (mistaken?) perception that it's easier to change software interfaces.
8. If software has a fault, it's there all the time in every copy: concepts like "1 in 10000 units has a fault" aren't meaningful.
9. Software failures are usually repeatable, and often catastrophic: "graceful degradation" is the exception rather than the rule.
10. Software has no theory of tolerances.
11. Software may be temporarily broken: sometimes restarting it can make it just as good (or bad) as it was before (compare and constrast kicking the television when its screen goes fuzzy).
12. Software doesn't wear out (though it may become obsolete when, e.g., operating systems change – but that's not the same, is it?)
13. Maybe software failure tends to be less noticeable than engine failure? Is this true? If true, is it just an effect of the cultural expectation that software will fail? People tend to be more tolerant of software failure?

14. Quality of raw materials – in particular, the idea of using different versions of a material of different qualities to match the desired quality of the end product – doesn't really have a counterpart in software. (But might it, if we really had a wide variety of components?)

15. The tools we use to develop software are more likely to be faulty themselves than the ones we use to develop other engineered artifacts.

16. Can't exhaustively test most software. (Think about the senses in which this is something you can't do for engines, etc., either. But you can take it through one representative of every equivalence class of situations where it should perform??)

1.2 Modules

Before we go on to questions about objects and components, let's revise a question you covered in PM:

What is a module?

A piece of code with a well-defined *interface* which *encapsulates* some information about the code. E.g.

- subroutine!
- library function
- ML module
- Class
- Component

But this is no use unless we have the *right* modules with the *right* interfaces.

Slide 3

Why use modules?

Already in 1965, Edsgar Dijkstra was explaining that software systems beyond a certain size are too complex to be completely understood by any one person.

So aim of modules is to increase

- the understandability of the code, and hence
- the probability of correctness, and
- the ease of maintenance.

To achieve this:

a module should be meaningful. More specifically, it should have high *cohesion* and low *coupling*.

Slide 4

Dissenting voices

Not for the last time, we've given one definition of a term when there are others.

Another widely used definition of *module* – supported by language concepts in e.g. Ada – is that it is a physical collection of code, even a unit of compilation.

But there is normally still an expectation of a clear interface, and we will take that as the defining feature.

On the third hand, Meyer says that a step to Object Oriented Happiness is to obey the maxim:

"Every non-simple type is a module, and every high-level module is a class."

Q: What does this mean?

Slide 5

Here are two more definitions:

1. From the glossary of Booch:

module A unit of code that serves as a building block for the physical structure of a system: a program unit that contains declarations, expressed in the vocabulary of a particular programming language, that form the physical realization of some or all of the classes and objects in the logical design of the system. A module typically has two parts: its interface and its implementation.

2. From Liskov, B. 1980, A Design Methodology for Reliable Software Systems, in *Tutorial on Software Design Techniques.* 3rd ed. New York, NY:IEEE Computer Society, p66 (quoted by Booch):

   modularization consists of dividing a program into modules which can be compiled separately, but which have connections with other modules. We will use the definition of Parnas: 'The connections between modules are the assumptions which the modules make about each other' "

Actually, we're probably happier with Parnas' definition than with Liskov's addition! It does raise an interesting question, though, which we will notice and then largely pass by:

What is an interface, anyway? In what sense do we expect it to document the assumptions that may be made about modules? Doesn't this sound like a real specification, rather than just the names and signatures of the functions? But the latter is all we normally get in present-day interfaces.

1.3 Objects

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 6

Let's look at that definition of an object a little more closely. It's drawn from Booch, and I'm going to use Booch's examples and definitions quite extensively here because I think they're the best I've seen.

So what's a thing, anyway? Clearly not just a physical thing, though many objects do represent physical things – and it's worth remembering that the first time objects arose explicitly was in Simula, where they did represent real world objects, the objects being simulated. The crucial property of an object is that it has well-defined boundaries. These are conceptual boundaries, not necessarily physical ones.

Q: Which of these might be objects: hammer, sphere, chemical process, love, fog, river, anger, cat, greyness, grey cat?

State: properties of an object together with values of those properties. For example; if we consider the property "number of orders placed" its value may be 3. Every property has a value. Some properties will have static values: that is, the value of the property will not vary during the lifetime of the object. Other properties will have dynamic values.

Behaviour: the way an object acts and reacts, in terms of its state changes and message passing.

Identity: (a slightly controversial aspect). That property of an object which distinguishes it from all other objects. That is, objects are not just values. An object has a continuing existence: if our application refers to two objects each of which has a string property with value "Edinburgh", it makes sense to ask whether they are the same object or not. (Whereas, for example, if the constant string "Edinburgh" occurs twice, it doesn't.[1] This is probably only controversial if you are a pure functional programmer; and there are some distinguished ones who say you can do OO in a purely functional way. To me, this seems pretty pointless, but maybe someday I'll be convinced...

Why use objects?

Things people hope to get from objects include:

- Reduced costs, especially
  - maintenance costs
  - costs of later similar systems, via reuse

  (More controversially: reduced cost of first system??)
- Reduced (average) time to market
- Increased reliability
- Match users' model of the world, so
  - capture requirements more easily and accurately
  - follow changes in user requirements better
  - allowing more naturally interacting systems: direct manipulation etc.

Slide 7

[1](Unless you're programming in Smalltalk, in which case you have to understand mutable constants!)

Next time

No meeting on Monday!

Instead, I'd like you to:

1. do some basic playing with Rose. Use the on-line help (which is the best place to start) and the UML tutorial.

   I'm not expecting you necessarily to finish this, or to end it knowing UML! I just want you to be confident using Rose from the start. Feel free to experiment. Any problems, ask (me or each other).

2. fill in the short Web questionnaire which will (by Monday) be on the course home page.

Next Thursday, back here for more advanced OO concepts.

Slide 8

See notes on course home page on using Rose.

Today's new references

Started with a version of a very old joke: this version translated from the American of Booch's heading to Chapter 1 of the very important book

Booch, G. Object Oriented Design with Applications, Benjamin/Cummings
– now (sadly?) replaced by various separate books by Booch – bu also, Objectory book by Booch, Rumbaugh and Jacobson coming soon: it may be worth not purchasing any book on OO just now.)

Dijkstra, E., *Programming considered as a human activity*, in Proceedings of the 1965 IFIP Congress, pub. North Holland.

Anything from Programming Methodology

Basic object concepts from Software System Design

Dilbert Sunday Strip, 28/9/97 (relevant to the questions we considered today).

The secret of OO

When OO achieves these aims, it is usually *not* because of any of the things that are specific to OO. It's because

- the object oriented approach takes modularity, encapsulation and abstraction as fundamental
- and OOPLs make them (comparatively!) easy, the obvious way.

OO is a religion: it's the *people* who become oriented towards objects.

So people who say you can do OO in any language (even in ML!) are right and wrong. You can't sensibly do all OO things: but you can think objects, which is the important part.

Slide 9

Encapsulation, abstraction

encapsulation The process of hiding all the details of an object that do not contribute to its essential characteristics; typically, the structure of an object is hidden, as well as the implementation of its methods. The terms *encapsulation* and *information hiding* are usually interchangeable.

abstraction The essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply-defined conceptual boundaries relative to the perspective of the viewer; the process of focussing upon the essential characteristics of an object.

(From Booch's glossary, again.) I will try not to use the term "data abstraction", which usually seems to mean the process of abstracting *away* from data, i.e. encapsulating it.

Slide 10

Possibly more down-to-earth definitions of encapsulation and abstraction: abstraction is the stuff you want in the interface, or the process of identifying it. Encapsulation is what you have when you don't have anything else in the interface.

Back to earth: encapsulation

Format of data is a problem! In OO it's the classic example of something that should normally be hidden.

(Opposite of old COBOL approach which attempts to control data by putting it centrally in one place.)

But there's more to it than this: once we hide data, we also hide knowledge about whether or not it exists! *information hiding*

E.g. suppose an application refers to a Point in 2D space, and can get both its Cartesian co-ordinates (x, y) and its polar co-ordinates (r, θ).

Do we store both sets, or do we store one and use that one to calculate the other on demand?

The outside world shouldn't know or care.

Slogan for C++ programmers: public data members are Bad.

Slide 11

That's all I want to say about object concepts for now: in particular, I have deliberately not mentioned inheritance or polymorphism. We'll come back at least to the question of inheritance later, when we consider class diagrams.

1.4 Components

What and why are components?

This is much more controversial. Basically in the *component-based engineering* approach of this course, what components really are is

an attempt to get round the fact that the *object*-based approach was not (always, usually) leading to desired levels of reuse, by choosing a new name meaning "thing we will be able to reuse" in the hope that this would make reuse happen.

So a component is a *unit of reuse*; object orientation is a (not the only) suitable method of implementing components. Not every component is an object; not every object is a component.

Somewhat dangerous analogy:

a component is to an object as a molecule is to an atom

Slide 12

Ivar Jacobson in OOSE says:

> By components we mean already implemented units that we use to enhance the programming language constructs. These are used during programming, and correspond to the components in the building industry (that is, windows, doors and so on). They must be powerful, have simple well-defined interfaces, be simple to find, understand and use, and have a wide area of application.

and later in Software Reuse:

> A component is a high-quality type, class or other UML workproduct, designed, documented and packaged to be reusable. A component is cohesive and has a stable interface. Components include interfaces; subsystems; use cases, types, actors, or classes; and attribute types. Components also include other workproducts, such as templates, or test case specifications.

In other words, if you're happy to reuse it, it's a component! Some, but not all, modules will be components; some, but not all classes will be components.

Note that this is different from the UML definition of the term: UML defines a component to be a physical unit of code. This course will take ideas like Jacobson's, *not* the UML version, as the primary meaning of "component".

2 Development methods

So far we've defined the terms in the title of the course, and given some idea of why we should be interested in each of them. But how do they fit together? We've briefly discussed the relationship between objects and components, and why components are desired by software engineers. We haven't discussed *how* software engineers can use (or create) components, nor what object oriented software engineering might be.

It's possible, and used to be common, to use object orientation as a programming technique within a conventional, data and functionality based, software engineering process.

More feasibly, one can use a conventional *structured analysis* method with object oriented design. (This is what BT used typically to do in the late 80s, for example: by the early 90s, when I was there, a concensus had pretty well formed that experience had shown that it was better to use an all-OO development.) However, the main benefits of object orientation come from a close and clear connection between system objects and problem domain objects; this is easiest to obtain if the whole development process focusses on objects. That is, rather than identifying separately the functionality of the system and the data on which it operates, we identify the key problem domain entities which have, naturally as it were, both state and behaviour. Moreover, using an iterative method requires a process with no watersheds to prevent you from revisiting and revising decisions made earlier.

2.1 History: The Three Amigos

The methods war As object orientation has become fashionable, and now has moved through from being fashionable into the mainstream of softwar development, a plethora of object oriented development methods has naturally been put forward, each with ideas and notations which overlap, but are not identical. Three of the most popular (my guess is, in this order) were:

1. Grady Booch's method, occasionally called OOD
2. James Rumbaugh's OMT
3. Ivar Jacobson's OOSE and Objectory

(Each had a major book which appeared in the early 90s). As you see, each method is championed by a single inventor, who in each case is a highly experienced and respected expert in the field of OO development. Like most people, I expected that we would for the foreseeable future have more or less this situation, the "methods war" with each method having its proponents.

The methods war is over: we won However, it isn't turning out that way. Booch was and is in charge of Rational Software (yes, the developers of Rose). In 1994 Rumbaugh also joined Rational, and the two of them declared that they intended to merge their methods. (Fowler's nice phrase is "achieve standardisation the Microsoft way".) Needless to say, the rest of the software industry wasn't too happy about this.

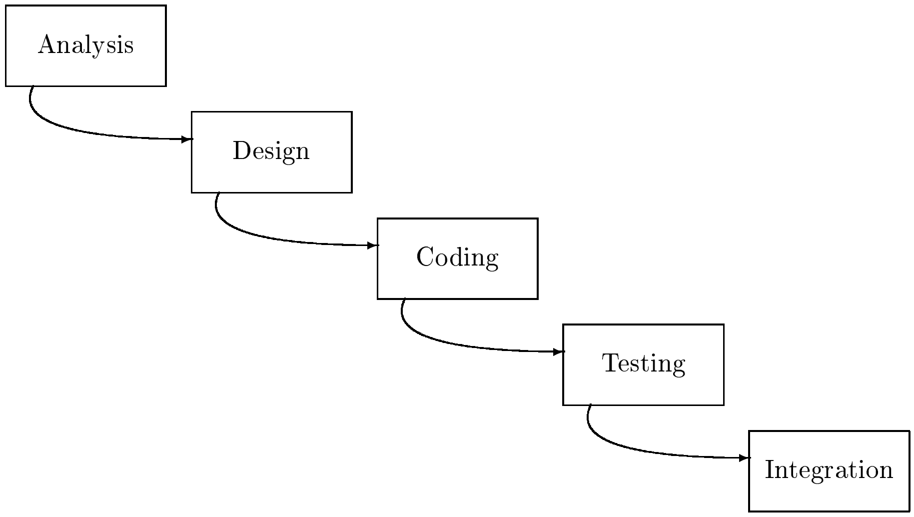
But resistance was futile. In 1995 Rational bought Ivar Jacobson's company Objectory, and it was all over bar the shouting. An early draft of documentation for the so-called *Unified Method* was produced (and volubly discussed on comp.object, of course, as well as elsewhere. Unfortunately by that time I was trying to get to grips with this strange new – to me – programming language, ML, and missed most of the fun!)

Later the focus was shifted away from developing a unified method towards developing a unified modelling language, and so we have UML, which has been submitted for ratification by the OMG. Interestingly, there is still a common process, called Objectory after Jacobson's method, but incorporating ideas from all three people. More on this in a moment, since we've raised the question of what a development method is, and what a modelling language is, and what the significance of the shift is.

2.2 What is a "development method"?

A development method prescribes a *process* for developing software, which implies that it prescribes a particular *life-cycle* for the product. You've probably seen at least two different life-cycles:

Waterfall model



Slide 13

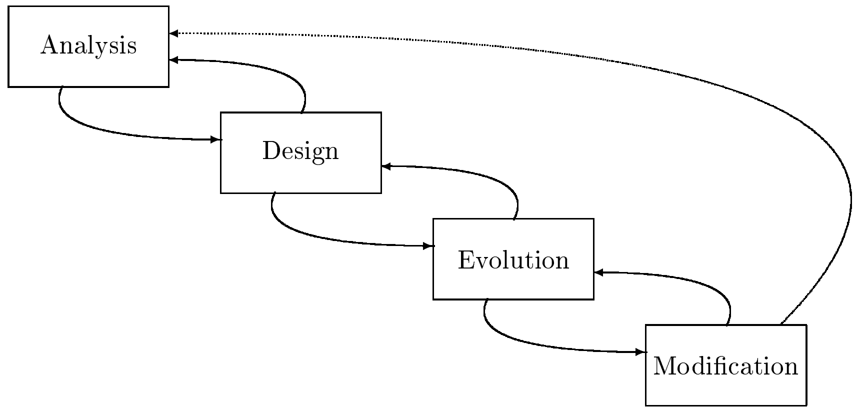
Problems with waterfall (Boehm 86)

- The waterfall model does not adequately address the concerns of developing program families and organizing software to accommodate change.
- The waterfall model assumes a relatively uniform progression of elaboration steps
- The waterfall model does not accommodate the sort of evolutionary development made possible by rapid prototyping capabilities and 4th generation languages
- The waterfall model does not address the possible future modes of software development associated with automatic programming capabilities, program transformation capabilities, and ‘knowledge based software assistant’ capabilities.

Slide 14

(See reference Boehm) Note that Boehm was writing more than ten years ago, and that “automatic programming capabilities” are no longer a “possible future mode of software development” – although for various practical reasons (cost of the tools, and the fact that with current techniques you need a very detailed design to take advantage of them) automatic programming capabilities, or code generation, is still not common, it is a reality. For example, it's part of what Rose can do.

Iterative development



Slide 15

However, this isn't the end of the story. This purely iterative approach works well for small, fast developments, but even there, there will be a stage before the project is definitely decided on, and a stage after it has been "finally" delivered, which it's sensible to separate out. It's also normally essential, for management and political reasons, to have a plan for what the iterations will be and what will be covered by each, rather than simply taking as many as a required. The Objectory process puts its main spiral inside its Construction phase:

Objectory process

Inception ends with commitment from project sponsor to go ahead: business case for the project and its basic feasibility and scope known.

Elaboration ends with

- basic architecture of the system in place,
- a plan for construction agreed
- all significant risks identified
- major risks understood enough not to be too worried

Construction (definitely iterative) ends with a beta-release system

Transition is the process of introducing the system to its users

Slide 16

These phases may differ wildly between projects in how long they take and what their deliverables are. For example, for a small or straightforward project (or one with a laid-back sponsor!) the inception phase might be a development manager having a short chat with the project sponsor. At the other extreme, it might require a series of high-level board meetings and a formally signed off business case. (Remark: Fowler seems to put less in the Inception phase than the Rational web site suggests.)

The book ("The Objectory Software Development Process") was due to be published this Autumn, like the UML user guide and reference manual: but I see that all have now been delayed to next Spring or Summer. In the meantime there is a high-level description of the process on the Rational web site. We will not say much more than this about any particular process in this course, although in the final section we will discuss (at the meta-level, if you like) the desirability of having a defined process (how defined, and by whom?) and the implications and problems.

I think that unification of process will be much harder and more interesting and less likely to happen in reality than the unification of modelling languages, because it will have to either expose, solve or fudge real differences of approach between the three people.

For example, something which is not really discussed in any of my favourite OO books (or any others I know) is the question of the level of abstraction at which each iteration happens. For example, both the Booch method and Jacobson's OOSE involve an iteration in which each iteration includes a complete analysis, design, implement, test cycle. The obvious thing is to assume that these methods agree on the point. However, this is not really so. Jacobson makes an important point of the idea that each iteration should deliver the functionality of at least one new use-case (see next lecture) – that is, in each iteration we change the system resulting from the previous iteration enough to add some new user-level functionality. His iterations are vertical slices through the eventual system, though he does recognise that the way in which a piece of functionality is delivered may change in later iterations.

Booch, on the other hand, is careful not to be rigidly prescriptive; indeed, although people talk about "the Booch method" it's arguable that this isn't a process so much as a discussion of how to go about design, together with an (extremely useful) collection of techniques. (It certainly does not have the scope to be, without alteration, a Methodology in the sense of the Heffalump Trap below, which we may consider an advantage!) However, his iterations are presented in the context of what he calls "round-trip gestalt design", capturing the idea that design neither top-down nor bottom-up. An iteration need not deliver new user-level functionality, indeed, will often not result in an implemented system at all. It may instead move you down to a lower level of abstraction, deepening rather than widening your understanding of the system.

Whichever process we choose, we need a language in which to express our designs, and this should almost certainly be diagram-based, since experience suggests that's how we naturally think about systems. It's inconceivable that a single diagram could capture everything about our design; and indeed that might not be desirable, since we probably want to abstract different aspects of interest in different contexts. So we will build different models of our design, reflecting different aspects, and will express each model using a diagram.

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 17

Crucial point: The diagram is not the design: the diagram is a *model* of the system designed – it captures the design in a form which can be discussed.

This implies, for example, that when you draw several diagrams to do with one design, there will be burdens of consistency checking: it is essential that all your diagrams really *are* models of one design, which will of course not be possible for arbitrary collections of diagrams! (For example, if your static structure diagram shows that Class Foo has no relationship with Class Bar, your dynamic diagram had better not show an object of class Foo sending a message to an object of class Bar.) Some of this consistency checking can be automated (see Check Model on Rose's Tools menu, for example).

There is an interesting question about the sense in which the notation can be given a rigorous or formal semantics, and this has been partly addressed by the developers of UML: but I'm not going to go into that here.

Is a unified modelling language, and/or a unified method, a good idea? You decide. I think the UML is a good idea, given that the language is big enough to encompass quite a wide range of models: I've been in the situation of finding it hard to talk to someone about their design because I didn't understand the notation in which they'd expressed it, and that seems silly. Probably more important even than communication between individuals:

some kind of modelling language is probably essential for describing large components. So if we want to do component-based software engineering, we don't want to be impeded in our search for suitable components by the fact that we don't understand their descriptions. So we'd better have a single modelling language for describing them. On the other hand the idea that there will be a single method (so that, say, you might not be able to claim to be doing a professional job of object oriented development unless you were doing it their way) I find a bit scary. It depends a lot on what the method looks like, and how prescriptive it is.

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 18

Fowler quotes as "an old joke":
Q: What's the difference between a methodologist and a terrorist?
A: You can negotiate with a terrorist.
Be aware that there are different, possibly conflicting sets of interests around.

1. The interests of the methodologists (in this case, Rational!) are in convincing you that their method will solve all your problems, and that if you follow their method you won't have to think about this. If your managers believe this, they will buy the Methodologists' tools and training, and the Methodologists will become rich
2. The interests of the users of the system are complex: on the one hand, they want a reliable cheap system quickly. On the other hand, they also want to be able to get their future systems...
3. The interests of the developers are to keep their jobs, and to keep their jobs interesting. Does this conflict with 1?
4. The interests of the managers, again, are complex. They need to satisfy their customers, but not at the expense of future customers.

We'll come back to these issues in the final part of the course when we consider such "people issues". In the mean time, you might like to have a look at Rational's Objectory pages, bearing in mind that they are methodologists.

Heffalump Trap 1

Don't be seduced by Methodology

There is a big difference between *methodology* and *Methodology*. A *methodology* is a basic approach one takes to get the job done, consisting of:

- a tailored plan (specific to the work at hand)
- a body of skills necessary to effect the plan

A *Methodology* is an attempt to centralize thinking. All meaningful decisions are taken by the *Methodology* builders, not by the staff assigned to do the work. The overt case for the *Methodology* includes standardization, documentary uniformity, managerial control and state-of-the-art techniques. The covert case is simpler and cruder: the idea that *project people aren't smart enough to do the thinking.*

(adapted from Peopleware p115)

Slide 19

New references

Peopleware: productive projects and teams

Webster, Bruce F. Pitfalls of Object-Oriented Development M and T Books

Boehm, B. *A spiral model of software development and enhancement* Aug 86 Software Engineering Notes, vol 11 (4) p23

3 Use cases

In this section we consider the uses and purpose of the most central of the UML diagrams.

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*.

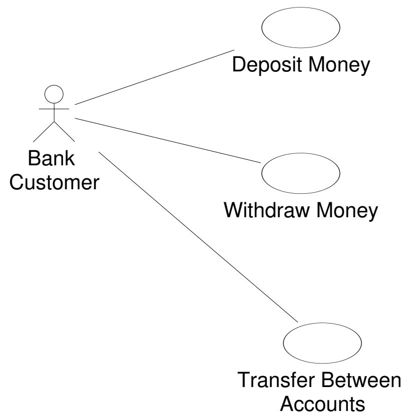
They are independent of OO – strength or weakness??

Slide 20

(Jacobson seems to me, admittedly on limited evidence, to be the maverick or wild child of the Three: he has some superlatively good ideas, but there's some evidence that he doesn't always maintain a proper connection with reality. See, for example, p116 of the Revised Fourth Printing of OOSE-AUCDA. Everything I've read by him has been very prescriptive: he seems to say "Here is my way to do it. It's right, absolutely right, all the time." I think most of what he says probably is right: but read with critical faculties alert. Use cases are his main thing. Enough said.)

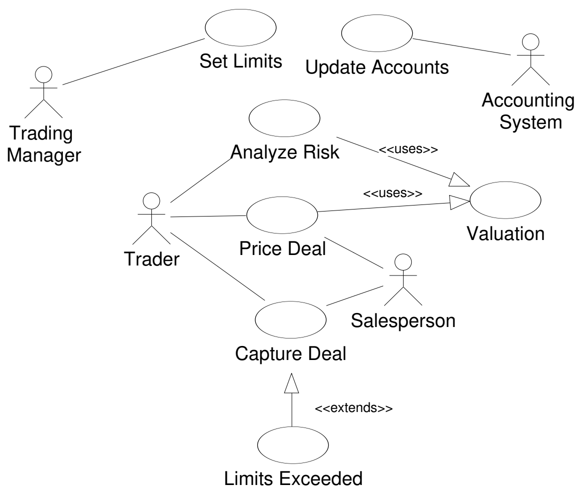
A simple use case diagram

(after Jacobson et al. Software Reuse fig 3.16)



Slide 21

More complex UC-diagram



(after Fowler UML Distilled Fig 3.1)

Slide 22

Note that these diagrams show, not a single use case, but all use cases for the given system: each oval represents a use case.

3.1 Actors

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 23

Remark on actors in Rose The Rose tutorial points out that, inside Rose, "Actors are treated as stereotypes of classes" which might give you the mistaken impression that for each actor there will be a corresponding class. But in fact the connection between actors and classes is for Rose's convenience only: Rose is presumably reusing some of its internal code for representing classes in order to represent actors. In general, there will *not* be a class in the system corresponding to an actor – the actors are normally outide the scope of the system itself.

"Stereotype" by the way is UML's way of attaching extra classifications to model items; it's one of the ways that UML is made extensible. We will not discuss them in this course: all you need to know is that if a Foo has a stereotype Bar it means it has been classified as being an instance of a particular kind, Bar of Foo *at the modelling language level.*

Actors tend to be comparatively easy to identify. (If they aren't, be very suspicious: you may possibly be being pushed into embarking on the surprisingly common task of building a system which somebody wants built but which nobody really wants to use.) They can be extremely important from a political point of view, but are unlikely to be very important from a technical point of view. Technically, the use cases themselves are more important. (For example: if two use cases start off involving the same actor, but later it turns out that two different actors are actually involved, it's most unlikely that our system will be affected. Unless, of course, this means that the system is now less valuable to its sponsor, in which case it may be cancelled.)

Even here, though, we encounter interesting questions. It's likely to be clear who are the human users of a system. It may be much less clear which systems should be represented as actors: do we really want, as I suggested in the slide, to show every system which may interact with ours? I think we probably do, but Fowler holds a different view. Here's his summary of the options (Fowler p46):

> 1. Some people feel that all interactions with remote systems need to be shown on the diagram. For example, if you need access to Reuters in order

to price a deal, you should show a link between the Price Deal use case and Reuters.

2. Some feel that you should only show external-interaction use cases when it is the other system that initiates the conteact. Using this rule, you would only show the use case for the Accounting System if that system invokes some process to tell the source system to do that.
3. Some feel that you should show system actors only when they are the ones who need the use case. So if the system generates a file every night that is subsequently picked up by the Accounting System, then Accounting System is the relevant actor because it is the one that needs the file produced.
4. Some feel that thinking of a system as an actor is the wrong focus. Instead, they deem that an actor is a use who wants something from the system (for example, a particular file). In the case of our example system, the actors would be the internal auditors of the company.

Fowler favours option 3. On the whole I prefer option 1: taking option 3 seems to be treating an actor differently depending on whether it's a human or a machine, since we certainly want to show all human users, whether they are getting benefit from the use case or (as in the case of a bank teller) helping to provide benefit to another actor. As Fowler says, this is a matter of "stylistic variation" – we will not, in this course, designate any of these options as right or wrong. (Though of course, following one blindly could lead to things that were definitely wrong. For example, blindly following option 4 could lead you to analyse in detail the workings of the Accounting system, which would probably be a waste of time; following option 1 blindly might lead to a diagram cluttered with irrelevancies; following option 2 or 3 blindly might lead to omitting major system interactions from your requirements capture process. You need to use your common sense and be flexible.)

3.2 Anatomy of a use case

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. Withdraw Money or Analyze Risk.

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.

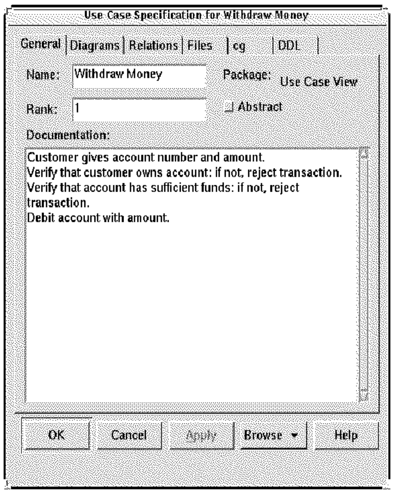
It may include logic to handle unusual or alternative courses, e.g. "if the customer's balance is too low, stop and say so" *even though these may result in the actor being unsatisfied.*

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

(Secretly, a use case is a winning strategy for a certain game, but that's another story...)

Slide 24

Textual part of a use case



Slide 25

If an actor takes part in a use case, either sending or receiving a signal or some data, the actor is connected with the use case by a line, which represents a Communicates relationship between the actor and the use case. You may see (e.g. in Jacobson) these lines shown with an arrow on one or both ends, showing the sender at the origin of the arrow and the receiver at the other – if the use case involves several interactions in different directions we'll have an arrow at both ends. Fowler uses lines without arrows, as prescribed by UML 1.1, and that's what we'll do here. Rose defaults to having a single arrowhead, but you can (and I recommend that you do) fix this by:

1. double clicking on the arrow to get its specification box
2. choosing the tab Role B Details
3. checking the Navigability box

Alternatively (I just found out), you can achieve the same aim more simply by selecting the arrow and then choosing Navigable from the short-cut menu, on the right mouse button. (I don't know how to make this be the default navigability for all Communicates associations in use case diagrams – I imagine you can, if only by defining a custom toolbar. If anyone finds out how, please let me know!)

Notice that the way in which the use case is described will vary through the development process. To begin with it's important that we identify *what* the system should achieve in each use case, not *how* it should achieve it. Later we'll choose an implementation. This may well involve changing the actors; probably not the one who wants the use case, but the ones who take part in a "helping" capacity. As an extreme example, you might start off assuming that to satisfy the bank's Customer who wants a Withdraw Money use case, you need to involve the actor Bank Teller, but later you might decide to implement this using an automatic teller machine, in which case the Withdraw Money use case will be changed to involve an ATM actor.

Later we'll see how to match up use cases with other UML diagrams, in order to demonstrate how the design realizes the use case.

Even in the definition of a use case, we need to be careful, for Jacobson/Griss/Johnson in Software Reuse (p66) say:

> A use case is a sequence of transactions performed by a system, which yields an observable result of value for a particular actor.

This is what UML (and others, e.g. Rumbaugh's OMT) calls a *scenario*: it describes a single path through the system with no alternative courses.

3.3 Use cases for requirements capture

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 26

We only connect an actor with a use case when the actor participates in a use case, not, more generally, when an actor is in any way concerned about the existence of a use case. So there may very well be cases when a work unit is essential, but not particularly valuable to any of the actors which participate in it. The classic example is when a utility company sends quarterly bills to each of its customers. The company as a whole gets benefit from it; but the only obvious actor in the use case is the customer, and possibly the company's mailing office, which doesn't itself particularly benefit from sending out the bill. I recommend that you do show such work units as use cases when you discover them, even though this involves relaxing the requirement that every use case is of benefit to one of the actors involved – but do be careful, in these cases, to ensure that the use case does represent a real requirement.

3.4 Example

3.4.1 A system for CS4 administration

The current situation Towards the end of each academic year, the Syllabus Committee determines which courses will be available to CS4 students in the following year.

At the end of each academic year, the Head of Department allocates duties to members of teaching staff and others; in particular, one person is assigned to lecture each of the courses which are supposed to be available in the following year. (We'll call these people lecturers for simplicity.)

Each lecturer updates the CS4 Course Handbook entry for his or her course. The CS4 co-ordinator updates other parts of the handbook, and checks the course entries produced by the lecturers.

Somebody in the Undergraduate Teaching Office (from now on we'll call any such person "the UTO") produces the paper version of the course handbook; the CS4 co-ordinator produces the HTML version by running latex2html on the latex source.

The CS3 Co-ordinator is supposed to give a list of the students entering CS4 from CS3 both to the CS4 co-ordinator, and to the UTO. The CS4 co-ordinator tells the UTO about any students entering CS4 other than from CS3, for example, non-graduating students. The UTO keeps the master list of all CS4 students, and updates the mailing list cs4.

CS4 students provisionally register for courses by filling in paper forms and handing them in to the Undergraduate Teaching Office. The UTO checks that every student who registers is listed as a CS4 student, and that every CS4 student is registered for a reasonable number for courses.

The UTO then produces for lecturers lists of the students taking their courses. These arrive not sooner than week 3, in particular, too late to be useful for letting lecturers know how many copies of things to make...

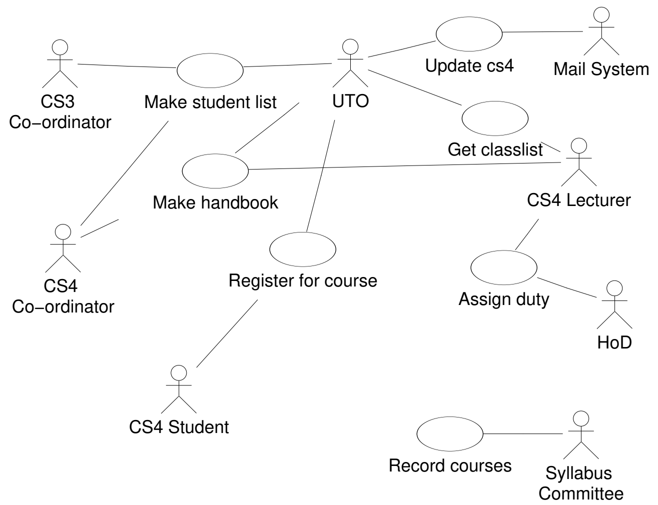
The investigation The Department has asked you to investigate the possibility of developing a system to automate parts of this process, because they hope it may be possible to:

- decrease the burden of routine work on all staff, especially the poor overworked CS4 co-ordinator
- allow students to register for courses on-line
- make it easy to obtain (from the UTO) up-to-date, reliable information;
- improve the traceability of such information;
- make information such as the course handbook and lists of students taking courses available sooner.

Draw a simple use case diagram for such a system.

Hint: not all the information you've been given will be relevant at this point.

One possible solution



Slide 27

New references

I've made heavy use of Fowler's UML Distilled again here. Also:

Jacobson, I. et al. *Object Oriented Software Engineering: a Use Case Driven Approach* Addison Wesley 1994.

Jacobson, I., Griss, M., and Jonsson, P. *Software Reuse: Architecture, Process and Organization for Business Success* Addison Wesley 1997.

3.5 Use cases through the development

Planning By the end of the Elaboration phase of Objectory, or by the end of analysis, if you prefer, we should (in an ideal world) have a complete list of all the use cases for our system, together with:

- a good idea of what each means
- an understanding of who wants each and how much
- knowledge of which use cases carry most risk, where risk may be:
  - requirements risk: higher if it's more likely that we haven't got the requirements quite right yet, or that they may change
  - technological risks: higher if we're using some unfamiliar component collection, system, compiler, database, combination of these, etc.
  - skills risk: closely related to technological risk; higher the more areas there are where developers aren't very experienced.

  (or others, e.g. maybe "safety risk" if you see what I mean, in a safety critical system).
- a plan for how long it should take to implement each use case (ha ha – they won't be independent, will they?)

Every use case will be implemented in some iteration of the system. Indeed, as I mentioned before, Objectory may go further and require the converse, that every iteration of the system delivers some use case. At any rate, by this stage we should have planned the iterations, and should know (at least provisionally) which use cases will appear in which iterations. So in what order do we tackle the use cases?

Political aspects Remember that 25% of systems that were never delivered? It wasn't because the developers decided to go on holiday instead: it was because the project got cancelled. That is, someone somewhere decided it wasn't worth going ahead with. How do we stop that happening to our project?

If we've captured requirements in terms of use cases and actors, we are likely to have a very good idea of which use cases – i.e. which aspects of the system's behaviour – are most important to which people. (And here I really do mean people, not actors.) So, other things being equal, we want to be able to demonstrate the system doing something valuable to the most influential people *first*. As soon as possible, we want to make sure that everyone who was the power to scupper the system (and remember this may include people who don't have formal power in the organisation!) has some good reason not to do so. This means that they must see that if the system is completed and delivered, they will get something they want, which they will lose if they get the project cancelled.

Comparing use cases which are important to a given person, of course other things being equal we should implement the highest priority ones first.

This may all sound very cynical, but I think you'll find it's sound.

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 28

Technical aspects Another criterion, which may conflict with the ones above, is that we should deliver the use cases associated with highest risk first, so as to tackle the greatest risks when we still have contingency to tackle them, and so that we don't get tied into a design which will not allow us to deal with the hardest (highest risk) use cases.

By the way, compare and contrast this with Booch's approving quotation of Heinlein: "When faced with a problem you do not understand, do any part of it you do understand, then look at it again".

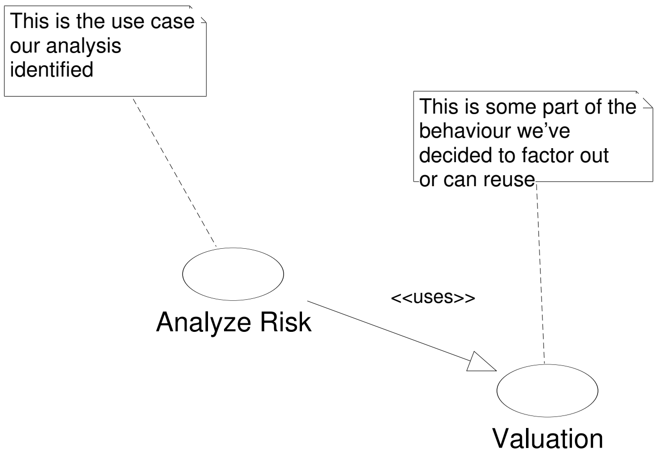
3.6 Relations between use cases

There are two kinds of situations in which we may want to document a relationship between two use cases. On the use case diagram, this is shown as an open-headed arrow between two use case ellipses. The two cases are distinguished by being given different *stereotypes*: the first is given the stereotype ≪uses≫, the second is given the stereotype ≪extends≫.

We call a use case in our use case model *abstract* when it does not interact with any actor in our diagram; in this case, it must be that the abstract use case is related to another use case in one of the ways we will discuss.

Notice that, even though Rose calls relations between use cases "generalisation relationships", this is very different from generalisation between classes. Do not confuse the two. Note also that, as we shall discuss, a use case may be associated with a component which is being reused. In this case it may well be that the use case will sometimes interact with an actor, whereas at other times it will be another use case that plays the part of the actor with respect to the component use case. That is, a use case is abstract with respect to the current use case model; this is not an absolute fact about the use case itself.

Use case reuse: ≪uses≫



Slide 29

3.6.1 Use cases for reuse: ≪uses≫

The most vital case is when we are able to factor out common behaviour from two or more of our original use cases, or (better still) when we see that we can implement part of one of our use cases by using a component (analysis, design or implemented). We document the shared or reused part of the work-unit as a use case in its own right, and document which other use cases use this new use case.

As in the example above, we show this by an arrow from the "user" use case to the "used" use case labelled (with the stereotype) ≪uses≫, to represent that the source use case makes use of the target use case.

The concept of use differs according to what stage of the development we're at. It may be a specification concept: these use cases have common functionality which we only want to describe once. Of course, we'd like to be able to follow this through by only implementing it (at most) once: but even if there are good reasons for this not being possible, it may still be helpful to show the common behaviour in early diagrams, separating them later. More commonly, we will discover common behaviour as we perform analysis and design.

If we are to be serious about doing component-based design, it's essential to think about using components as early as possible (certainly during the elaboration phase of Objectory). There are several reasons for this.

Firstly, we will certainly need to adapt the way we describe the use cases to match the available components; we may well need to negotiate small changes in the requirements themselves, in order to make good use of available components. (This may sound radical: but consider the parallel case in, say, architecture. You are unlikely to be asked exactly what size and shape you want your doors. More likely, you will be asked to choose from a range of available shapes and sizes, each with its own cost; if it's possible at all to specify that you want a round green door with a brass knob, you can expect to pay a much higher price for it, and to wait longer, than if you want something standard. It seems reasonable to say that if the software industry is to make the shift to component-based engineering, our approach to flexibility of requirements will similarly have to change. Or do you think that the inherent flexibility of software will be enough to mitigate this?)

Secondly, if we will be able to implement some reqirement using a component, we want to know that before we expend time and effort working out how to build it ourselves.

Jacobson describes the ≪uses≫ relationship as being a kind of inheritance. I think this is highly misleading: the only similarity is really that both inheritance of one class by another and use of one use case by another are forms of reuse.

We meet here for the first time a question which will recur: when we use a component which we don't have to develop, do we show it on our design diagrams? Or do we treat it as though it were part of the programming language, and just use it? Either is possible; I suspect we will want to do it both ways depending on what the component is. For example, suppose we use an OrderedCollection class from a library of collection classes, in order to order the items on a bank statement. I doubt that there's any value in showing "Order Things" as part of a use case. But if we use a more complex component – say, using "Transfer Financial Instrument" to implement "Transfer Money" for a bank customer – we probably do want to show it. Yet again, it's a judgement call.

Naturally if we are planning to develop a reusable component that will include some, but not all, of the functionality of a given use case, it will also make sense to describe the proposed component clearly by its own use case, in this way.

Note on ≪uses≫ and ≪extends≫ in Rose Suppose you have in a Rose use case diagram two use cases, shown as ellipses, and you want to related them by ≪uses≫ or ≪extends≫. Here's how:

1. Use the open-headed arrow tool from the Use Case diagram toolbar to get the right kind of arrow between the uses cases. (Make sure you have it going in the right direction!)

2. Get up the specification window for the new arrow. You can do this by double-clicking on the arrow, or by choosing Open Specification... from the arrow's "shortcut menu", the popup menu that appears when you right-click on the arrow.

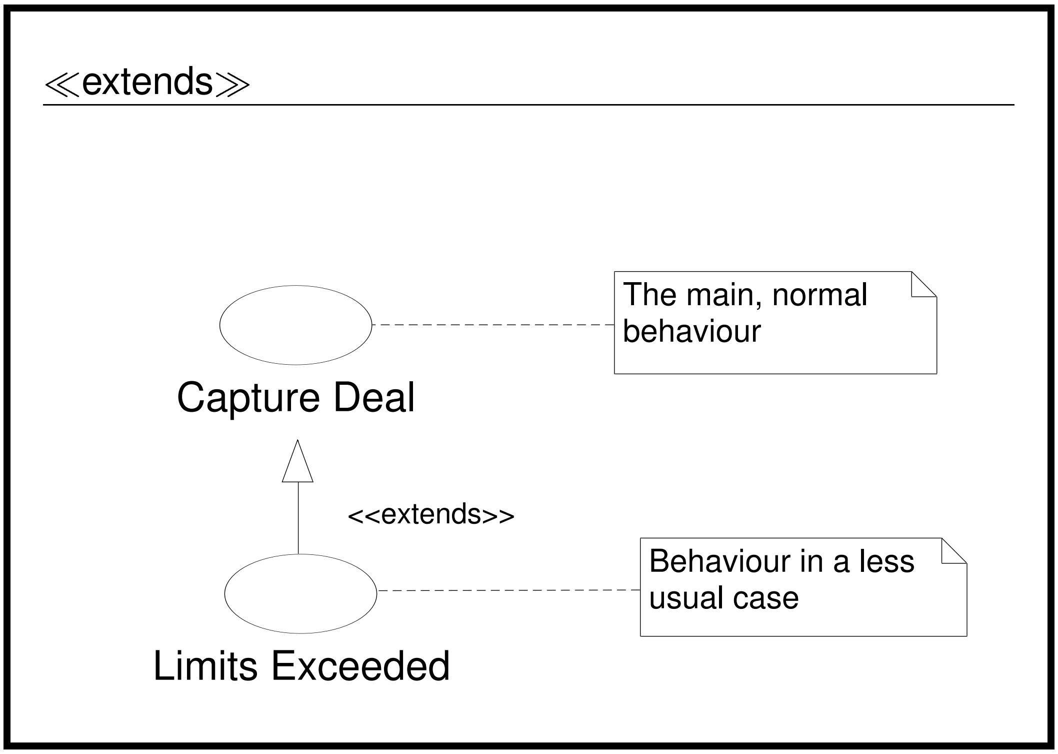
3. Record whether the relationship is a ≪uses≫ or a ≪extends≫ one, by entering "extends" or "uses" in the Stereotype box of the specification window of the relation. (You'll only have to type "uses" or "extends" once; after that these options will appear on the pull-down menu attached to the Stereotype box. Why do you have to type them even once, i.e. why aren't they in the menu by default? Search me.)

4. Click OK on the specification window to dismiss it

5. Get Rose to display the stereotype ≪uses≫ or ≪extends≫ on the diagram, by choosing Show Stereotype from the arrow's shortcut menu (right click on the arrow).

3.6.2 Separating variant behaviour: ≪extends≫

If a use case incorporates two or more scenarios – that is, several different things may happen depending on circumstances – we may decide that it would be clearer to show these as a main case and one or more subsidiary cases. When to do this is a matter of judgement, since we can always show variant cases in one use case. For example, the Capture Deal use case represents the straightforward case in which the deal can go ahead. Alternatively, it may turn out to be the case that the trader's limits have been exceeded (in the post Nick Leeson world), in which case the deal will not go ahead. We use the ≪extends≫ arrow from the less central case to the central case, as shown in the example above.



Slide 30

In UML, we may optionally document an *extension point* of the central use case, which is a formal or informal idea of the point in the description of the central use case at which it is decided whether or not to go into the extension use case. (For example, the extension point in the Capture Deal example might be called Check Limits.) UML permits the extension point to be recorded in the ellipse for the central use case. However, Rose does not appear to support this, so we won't use this feature in this course.

3.7 Possible problems with use cases

As we've seen, use cases can help with some of the most difficult aspects of system development, namely requirements capture, planning, iteration management and test planning. However, they do also have shortcomings.

Shortcomings

1. Danger of losing OO: plan focus on use cases may encourage a 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 31

In more detail:

1. Focussing on use cases may encourage developers to lose sight of the architecture of the system and of the static object structure, in the rush somehow to deliver the required use case. Later, if the functionality of use case has already been developed, it may be hard to justify the time to modify the design to maintain its integrity with respect to subsequent use cases. We may end up back were we started, developing a top-down function oriented unmaintainable inflexible system.
2. Capturing requirements by use cases may encourage developers to think too operationally: users are likely to describe the use case as a very concrete sequence of interactions with the system which is one way, not the only way, of achieving their real goal. It's important that developers distinguish between requirements and candidate designs.
3. It's a mistake to think that use case based requirement capture, examining the needs of each actor in turn, will automatically identify all requirements. Consider, for example, a utility company that needs to send out quarterly bills to its customers. How will you identify the use case "send out bills"? The customers aren't going to identify this as something they need! The billing department isn't involved in the process, since it's automatic every quarter. Is the Calendar the actor who needs it? Hmm...

4 The class model

Now we come to an element of UML which has a counterpart in more or less every OO modelling language, because it has a fundamental importance – the class diagrams. (Found in Rose under Logical Model.)

There is an important sense in which the notion of class is an engineering convenience added on to the more fundamental notion of object. So let's (re)visit that notion first.

4.1 Objects and messages

What I say here applies to standard modern class-based object oriented languages such as Java and C++. Most of it also applies to Smalltalk, Eiffel, CLOS, Perl5 and other object-oriented languages you may meet: but a detailed comparison of OOPLs is outside the scope of this course. The footnotes and paragraphs in brackets are intended to be optional reading!

An object *understands* certain *messages*, which is to say that it can receive the messages and act on them. An object usually also has some internal data, in the form of a collection of named *instance variables* or *attributes*. In each object, there is a value associated with each instance variable; the value may change over the lifetime of the object. The set of instance variables, though, and the set of messages the object understands, is fixed. The way in which an object reacts to a message may depend on the current values of its instance variables: in this way, even when (as usual) the outside world is not granted direct access to the instance variables, it may be indirectly affected by their values. That is, the values of the instance variables are the *state* of the object.

For example, consider an object which we'll call myClock, which understands the messages reportTime and resetTimeTo(07:43), resetTimeTo(12:30), indeed more generally resetTimeTo(newTime) for any sensible value of newTime. It may advertise this by giving an *interface* which says that it accepts messages of the form reportTime() and resetTimeTo(newTime: Time) where Time is a type or a class whose elements are the sensible values of newTime we mentioned.

How does it implement this functionality? The outside world doesn't need to know; but perhaps it has an instance variable time, the value of which is returned by the object in response to message reportTime, and which is reset to newTime when the object receives the message resetTimeTo(newTime), where the value newTime could take any appropriate value. Or perhaps it passes these messages on to some other object which it knows about and has them deal with them.

From this example we see some things about messages. A message includes a keyword (sometimes called a *selector*); here we've seen the selectors reportTime and resetTimeTo. A message may, but need not, include one or more arguments, which are values being given to the object just as values are passed into a function in a normal function call. In the example, 07:43 is an argument forming part of the message resetTimeTo(07:43). Usually, for a given selector there is a single "correct" number of arguments which should be present in a message which starts with that selector; we will not expect myClock to understand resetTimeTo with no argument, or resetTimeTo(4, 12:30) etc. The acceptable values of arguments are also laid down[1] in the interface of the object.

(Why did I say "Usually"? Because in many languages, such as C++ and Java, a message can be *overloaded* – that is, an object may understand messages with the same selector but with different collections of arguments. In our example, perhaps myClock might understand both resetTimeTo (newTime) and resetTimeTo with no argument; perhaps in the latter case it resets its time to midnight. This is a very useful technique, because the reader of the program will naturally assume that an object responds very similarly to messages with the same selector, which can speed understanding of your code *provided this assumption is justified!* You must be careful only to use overloaded messages to implement functionality which is semantically essentially the same, in some sense.)

This sounds just like the idea of a function signature: the difference is that when you send a message to an object, you do not in general know what code will be executed as a

[1]by giving the class, or sometimes just the interface, which the object passed as an argument should have, or its type if it should belong to a basic type like int or bool – the details here are language dependent.

result, because that information is encapsulated.

4.2 Visibility

Often, it isn't appropriate for every object in the system to be allowed to send any message to an object that the object could possibly understand. The object's public interface defines which messages it will accept. It may send itself other messages (defined in its private interface), but the outside world may only send the ones in the public interface. When I say "interface" in future I'll mean the public interface.

(We'll probably mention Java and C++'s "protected" visibility later.)

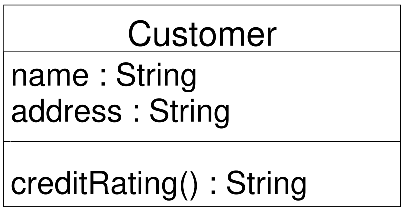
4.3 Classes

In class-based object oriented languages, every object belongs to a class, and the [most specific: see note on Terminology below] class of an object determines its interface. In fact more is true: even the way in which an object reacts to a message is determined by the object's class, together with the values of the object's instance variables. The set of instance variables which an object has is determined by its class, although of course the values taken by those instance variables are not determined by the class, but may vary. For example, perhaps myClock belongs to class Clock, which specifies that objects belonging to Clock have an instance variable called time and specifies how they react to the messages we described. We call the process of creating a new object belonging to class Clock as *instantiating* class Clock, and we call the resulting object an *instance* of class Clock – this, of course, is why the variables whose values belong to the object and may vary over its lifetime are called instance variables.

In UML, a class is shown in a class diagram as a rectangle giving its name, attributes (names of instance variables) and operations (messages for which our class defines some behaviour – rather than inheriting it: we'll come back to this). We can put symbols by each attribute or operation to show its visibility, but in this instance I haven't shown them. (In Rose you can control whether this is shown, and do other things like adding new attributes and operaions, using the class's Shortcut menu.)

This and later examples in this section are drawn from Fowler Fig 4.1.

A class Customer



Slide 32

4.3.1 Digression: Why have classes?

Why not just have objects, which have state, behaviour and identity as we require?

Classes in object oriented languages serve two purposes. First, they are a convenient way of describing a collection (a class) of objects which have the same properties. Object oriented programming languages can use classes essentially just as a convenience. For example, there is no need to store a copy of the code representing an object's behaviour in every object, even though conceptually we think of every object as encapsulating its own code. Instead, compilers create a single representation of a class, and allow objects of that class to access the class representation to get the code they need.

You can see that there are other ways of achieving this convenience; for example, we could define a prototypical object, and then define other objects as being "like that one, but with these different values". Again, we would only need to keep one copy of the code. This is roughly what *delegation based* object oriented languages such as Self do. However, it now seems clear (famous last words?) that delegation based languages will end up being a historical dead-end: classes achieve the same aims more conveniently.

Secondly, in most modern OO languages, classes are used in the same way that types are used in many other languages, to specify what values are acceptable in given contexts, and thus to allow the compiler – and the human reader – to check that none of a large class of errors occurs in the program.

Often people thing of classes and types as being the same thing – indeed I may slip into that way of talking myself; it's convenient, and not often misleading. However, it's wrong. Remember that a class defines not only what messages an object understands – which is really all you should need to know in order to check whether it is acceptable in some context. It would be good to go into this in more depth, and to talk about *interfaces* as used in Java, which separate out the specification of an object's interface from its implementation – but we almost certainly won't have time.

(

Subdigression on absence of barriers between objects of the same class Often, as in C++ and Java, encapsulation does not work between objects of the same class. In our example, suppose that myClock (and so, all other objects of class Clock) also understands the message synchroniseWith(aClock), where the argument aClock is another object of class Clock, and that they respond to this message by setting their own time to the time of aClock. We would not expect myClock to be able to access the private parts of aClock, but in languages like Java and C++ it can, simply because they happen to belong to the same class. That is, code implementing the behaviour of myClock can access the internals of every object of class Clock that it gets to know about, not just myClock's internals. There's a sense in which this is reasonable from a software engineering perspective; remember that encapsulation – which normally prevents code implementing an object's behaviour from accessing the internals of another object – has as its primary purpose preventing changes to one part of the system from causing changes to be made to other parts of the system. Now, if the internals of aClock change, it must be because of changes made to the code of the Clock class. But if code implementing myClock's behaviour takes advantage of knowledge about the internals of aClock, that taking advantage must also happen in Clock class code. If we're editing the code of the Clock class anyway, it isn't too much extra effort to change more of the code in the same class in order to fix the implementation of the synchronizeWith method – in particular, no extra recompilations will be required because of this absence of a barrier between myClock and aClock. However, taking advantage of this does potentially increase the number of places where changes may have to be made in such a case. You can always avoid taking advantage of it, simply by making more information about Clock public; but that has its own problems. Judge each case carefully on its merits.)

)

4.4 Relationships between classes

We want to capture the coupling in our design: that is, the places where one module (here, class) depends on another. There's a variety of ways in which this can arise.

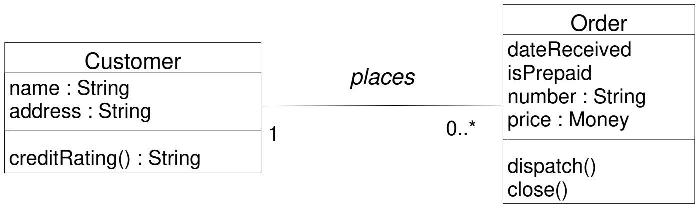
4.4.1 Association

The main one is when there's an association between two classes: this means roughly that some object of one of the classes needs to "know" about the existence of the other class. (And so in particular, might be affected by changes to that class.)

For example, this might be because some object of one class:

- sends a message to an object of the other class, or
- creates an object of the other class, or
- has the other class as the type of a message argument, or
- has the other class as the type of an instance variable

Association between classes



Slide 33

We can (and usually should) name the association; here I've called the association *places* because a customer places an order. UML 1.1 permits the name of an association to contain a small black triangle to indicate in which direction the associated is to be read, so I could have written *places*▷ (with the triangle filled in, can't off-hand find the LaTeXfor this!) to indicate that we mean "a customer places an order" rather than "an order places a customer". However, Rose doesn't seem to support this notation, so we won't use it. Alternatively, we can name the *roles* that each class plays in the association. If we do this (in Rose, set them in the assocation's specification window) the role played by the Customer in the association (say *placer*, perhaps) will appear on the arrow near the Customer end, and the role played by the order (*placee*?!) will appear near the Order end. Normally we will simply name the association, and the direction will be obvious from the context.

Multiplicity An association between classes exists whenever an association between two objects of the classes exists. Indeed, you can think of the classes as sets of objects, and of the association as a relation on those sets, if you like. We can sensibly ask: given an object in one class, how many objects in the other might be associated with it? (Or, what are the possible values of $|\{y : xRy\}|$ as x varies, and of $|\{x : xRy\}|$ as y varies?)

In the example, I showed a 1 at the Customer end of the association *places*, because every order (that is, every object of class Order) is associated by *places* with just one customer (object of class Customer), namely, the customer who placed that particular order. On the other hand, a customer may have placed any number, including 0, of the orders in our system (if 0, maybe they placed an order in the past, or asked for a catalogue, or otherwise got onto our system: depends on information I haven't specified). So the multiplicity on the Order end is $0\ldots*$ (UML's choice of notation. or at least Rose's. Horrible, but there it is.) As you see, we can specify an exact number by writing it, a range of numbers using $\ldots$. We can also specify a choice of multiplicities by giving a comma separated list of multiplicities: for example $3, 12\ldots15, 901\ldots*$ is an unlikely multiplicity saying that there can be exactly 3, or else between 12 and 15 (inclusive) or else at least 901, of whatever it is.

4.4.2 Generalisation

Another possible relationship between classes is generalisation. For example, Customer is a generalisation of Personal Customer because conceptually, every Personal Customer is a Customer. Everything that is true of every Customer is *a fortiori* true of every Personal Customer. So if some part of our system works on an arbitrary Customer, it ought to work on an arbitrary Personal Customer too, since every Personal Customer *is* a Customer. On the other hand, there may be things that don't make sense for every Customer, but only for Personal Customers. Personal Customer is more specialised than Customer; or Customer is a generalisation of Personal Customer.

In terms of interfaces, an object of class PersonalCustomer should conform to the interface given by Customer: if some message is acceptable to any Customer, it is of course fine to send that message to a PersonalCustomer. PersonalCustomer may understand other, specialised messages as well – that is, its interface may be strictly broader than Customer's.

As I've implied, to say that there's a generalisation relationship between classes is to make a strong, though informal, statement about the way their objects behave: an object of a specialised class is substitutable in any context which expects a(n arbitrary) member of the more general class. In particular, even though what an object does on receipt of a message is encapsulated, there had better be no conceptual gulf between what objects of the two classes do on receipt of the same message. Their behaviour doesn't have to be identical, but it must be similar enough not to make code that works with an object of a general class fail to work with an object of a specialised class.

Using English to check whether a generalisation exists Class Foo is probably a generalisation of class Bar if it's true that *every Foo is a Bar*. (Often we just talk about "is a" relationships – *a Foo is a Bar* – but then it's possible to come unstuck. For example (from Fowler p91), you might casually say "a Border Collie is a Breed" and conclude that Breed is a generalisation of BorderCollie, which is rubbish. Try out "every Border Collie is a Breed" which is more obvious rubbish. The problem is actually, I think, that the English "a Border Collie is a Breed" is wrong to start with: it should have been "Border Collie is a Breed". But putting in "every" may help. I don't personally find Fowler's suggestion that you consider "Border Collies are kinds of Breeds" helpful, but you may.

Implementing generalisation: inheritance One way (not the only way) to implement a generalisation is by *inheritance*. Whereas generalisation is a conceptual relationship between classes, inheritance is an implementation relationship. Instead of duplicating the definition of Customer, and possibly adding to it, to define PersonalCustomer – which would be possible, and there would still be a generalisation between Customer and PersonalCustomer – we often choose to use the facility provided by OOPLs of inheritance to define PersonalCustomer in terms of Customer.

If we define class PersonalCustomer by inheritance from class Customer, we only have to write code for the new parts of PersonalCustomer's interface; for example, we have to add operations to react to any messages which can be received by a PersonalCustomer but not by an arbitrary Customer.

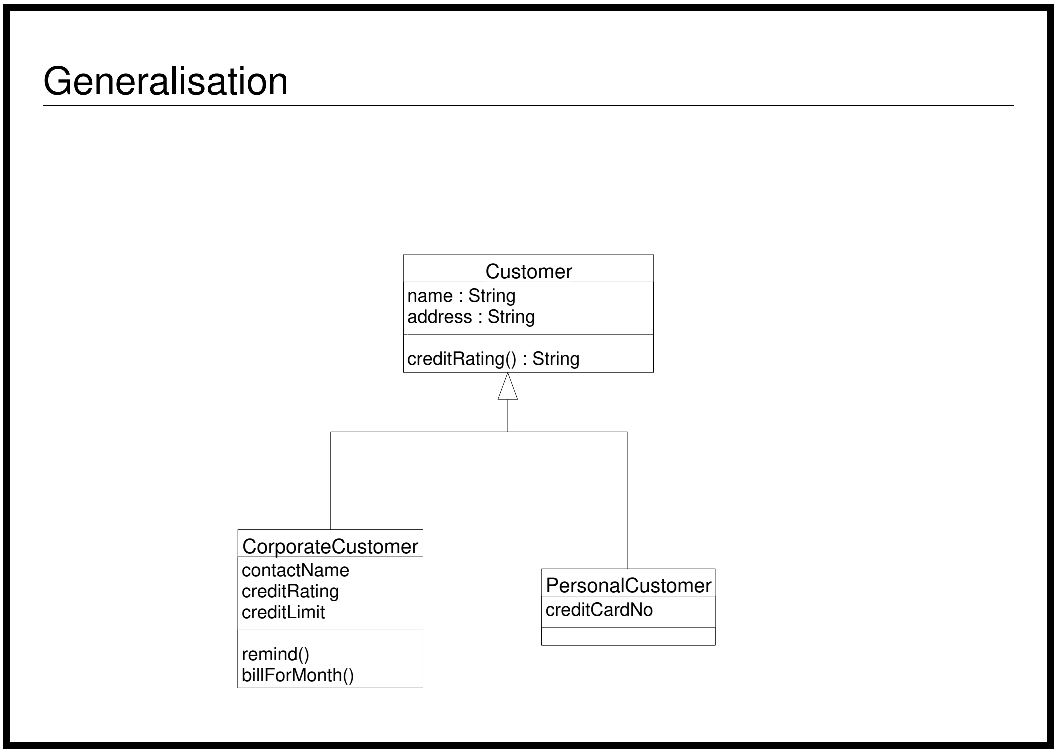
More interestingly, because in object orientation knowledge of which code an object executes when it receives a message is encapsulated, it is possible for class PersonalCustomer to implement different behaviour on receipt of some message from what Customer implements. That is, we can *override* some of Customer's behaviour in PersonalCustomer.

Terminology We say that

- PersonalCustomer *inherits from* Customer
- PersonalCustomer *is a subclass of* Customer
- Customer *is a superclass of* PersonalCustomer

Do we say that an object of class PersonalCustomer belongs to class Customer? We should, if we are thinking about classes as sets of objects. When I talk about "an object of class Customer" I will certainly include the objects which also belong to class PersonalCustomer. However, it's just too convenient to be able to talk about "the class of this object" as though the object only belonged to one class. When I say this, I will mean the *most specialised* class to which the object belongs. This means that it remains true that an object's behaviour is determined by its class!

In UML, we show generalisation like this:



Slide 34

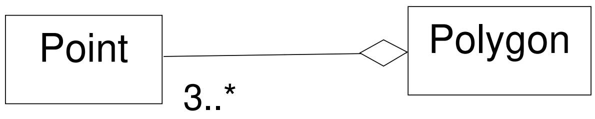
4.4.3 Aggregation: "is part of"

An *aggregation* relationship between two classes is a particular kind of association, which shows that an object of one class (B, say) *is a part of* an object of the other (say A). Usually, but not always, the object of class B lives and dies with the object of class A; if an object of class A is deleted, then its component part of class B must also be deleted.

(In fact, UML 1.1 also defines an even more specific term, *composition*, to represent the kind of aggregation in which the "Usually" above applies, but in this course we will not discuss that.)

Here's an example:

An aggregation relationship



Slide 35

This is part of an example from the UML 1.1 documentation, reproduced in Fowler; but even here, some designers might choose to show this as a plain association, rather than as an aggregation. The reason why it isn't totally clear that this should be aggregation is that a Point may play several roles in the system, and the "Usually" condition above will probably not be satisfied; for example, a point may be a vertex of several polygons.

In general, how do you tell whether aggregation is appropriate or whether you should just show an association? There's no hard and fast answer; different designers will make different decisions. Rough guidelines:

- If an object of class B must be created every time an object of class A is created, and must be deleted if and only if the object of class A is deleted, consider using aggregation.
- If class A is the only class in the system which has a relationship with class B – so that in particular, the existence of an object of class B is not going to be exposed in the interface of class A – consider using aggregation. If you find yourself using aggregation when this condition isn't satisfied, think very hard about whether you really mean it.

If in doubt, use plain association. A plain association will always be correct when an aggregation is: the purpose of aggregation is to be more specific about the kind of association involved.

We don't normally name an aggregation relationship, because it is assumed that the relationship is "is part of". Multiplicities may still be useful: in the example, I showed that each polygon must contain at least 3 points. I didn't specify the multiplicity at the Polygon end. If I had shown a 1 there, for example, it would have represented the assertion that each point object in the system is a vertext of exactly one polygon object in the system.

4.5 Development of class model

The class model is one of the first UML models you start to create, and it will be developed thoughout the project, changing its role as the development proceeds.

At the beginning, in the Inception and (early) Elaboration phase, or if you like in requirements capture and analysis, the class model records the understanding we gain about the problem domain. We try not to make premature design decisions. Therefore, for example,

- we do not consider in which direction(s) an association is navigable: that is, which object knows about (has a pointer to) which other object. We just record that there's some connection, association.
- when we record an attribute, say name on Customer, we do not imply that the class Customer will necessarily have an attribute called name, let alone a public one – we just mean to represent that customers have names
- if we identify aggregation and composition relationships at all (and some designers prefer not to, at this stage) we are describing a notional connection between the objects, which may or may not turn into an implementational relationship.

That is, in the initial class diagram is drawn at a conceptual level. As development proceeds and we make more design decisions, we make it more concrete and add more information, so that by the end, we have a diagram from which (some) code can be generated. We won't go into details here about this, but the process is important.

4.6 A simple example

Example

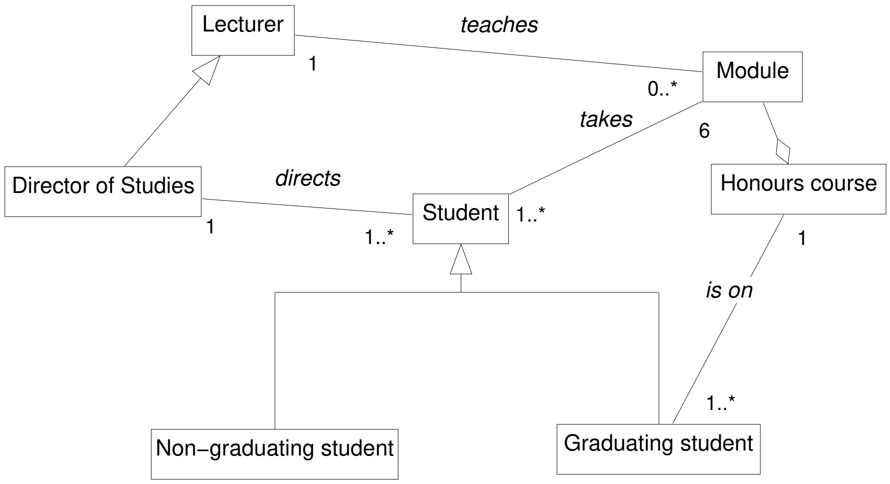
The CS4 administration system should be able to report on any student: for example, whether the student is graduating or non graduating, what modules the student is taking, what honours course a graduating student is registered for, which lecturer is the student's DoS.

It also acts as a repository of information on modules: who lectures them, what degree course they're part of, which students are taking them.

Draw a conceptual level class model to represent this information. Include multiplicities, but don't worry about attributes and operations.

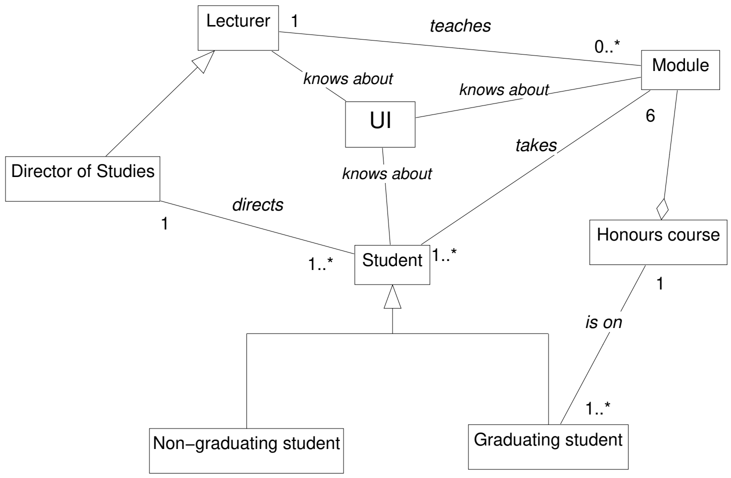
Slide 36

A possible solution



Slide 37

Another possible solution



Slide 38

4.7 Identifying objects and classes

One of the simplest and most effective techniques for identifying classes and operations is to take or write an English description of what the system must do, underline the noun phrases and verb phrases, and consider the noun phrases to be candidate classes and the verb phrases to be candidate operations on some class.

Of course it isn't as simple as this! You have to be careful to discard nouns which are inappropriate because they are:

- Redundant – maybe several noun phrases refer to things which are the same from the point of view of the system (Town and City in most applications, for example)
- Outside the scope of the system – for example, the actors in your use cases are likely to occur in the description, but probably don't need to be represented by classes
- Vague – e.g. "information", "concept", where unless there's some more concrete thing lurking behind the word it probably has no significance for the system
- Attributes – for example, "name" of a customer will occur, but name probably has the simple type string and no behaviour to speak of, and is better modelled as an attribute of class Customer than as a class in its own right
- Operations and events – e.g. the placing of an order. Sometimes you will need to model an event as a class, especially when the system has to keep a record of its having occurred, but this is the exception rather than the rule
- Implementation classes – e.g. List. By the time you reach an implementation-stage class model you may want to show these, but in the beginning they probably represent dubious assumptions about how the system must be implemented.

(This list is adapted from OU course M868, Object Oriented Software Technology.)

Classes and objects often derive from one of these sources (Schlaer-Mellor, paraphrased by Booch, adapted again by me)

- Tangible or "real world" things: courses, students, orders, pressure sensors
- Roles: DoS, mother, teacher, politicion
- Events: landing, interrupt, request
- Interactions: loan, meeting, intersection

These categories overlap, and the first two are much more common sources of objects and classes than the last two.

Other possible sources (these from Coad and Yourdan, again quoted in Booch) include: devices, locations, organisational units, other systems with which this system may interact. The last is particularly interesting: it's common, and good practice, to create a class to wrap such an external system – maybe a particular database or TP monitor – and let your system interact with that, rather than direct with the system. This limits the possible dependencies on the external system to what you put in the interface of the wrapper class, and makes it easier to replace the external system if necessary. The danger is that you can end up having to make the wrapper allow access to so much of the external system that actually it's inconceivable that another external system could be made to provide the same interface. If this happens, then you've done extra work of wrapping for no benefit, since if the external system changes, the wrapper will have to change too, and your system will be as hard to adapt as it would have been without the wrapper. Sometimes it's better to do without the wrapper and just accept that your system is dependent on the interface to another.

Identifying objects and classes

Simplest and best: look for noun phrases in the system description!

Then abandon things which are:

- redundant
- outside scope
- vague
- attributes
- operations and events
- implementation classes.

(May need to add some back later, especially implementation classes: point is to avoid incorporating premature design decisions into your conceptual level model.)

Similarly, can use verb phrases to identify operations.

Slide 39

Real world objects vs their system representation It's easy to get so hung up on the object way of looking at things that one forgets that objects are really things inside a computer program. When we talk about "students" and "courses" we really mean the representation of these things within our system. The consequences of this are that we must be careful

- not to record information that is definitely irrelevant to our system;
- to remember that the objects *are* the system!

The latter point is particularly interesting. In the CS4 administration exercise, many people made the error, which is a classic for people not yet steeped in OO, of inventing a class which was "the admin system". The whole caboodle is the admin system, that's the point! It's in order to avoid this error that I really prefer the solution that didn't even show the UI, as a conceptual model class diagram. You might think that I've just renamed "the admin system" to "the UI". I haven't, because the UI isn't actually *doing* anything bar providing a mechanism for the actors of the system to be able to send messages to the system: but it's a subtle distinction, and it's easy to drift into monolithic design in which there's one object that knows everything. This is bad because such designs are very hard to maintain: they tend to have assumptions built into them about how the system will be used.

I hope we'll have time later to talk a little about techniques for user interface design which help to avoid this kind of problem, especially the Model-View-Controller pattern.

Trap 2: 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 40

Trap 3: identifying too many classes

Particularly when considering generalisation relationships, it's easy to get carried away, and build complicated (and highly coupled) inheritance hierarchies that aren't necessary.

By all means identify lots of generalisations: but be careful that you don't automatically implement all generalisations using inheritance.

For example, we could have identified AIModule, CSModule etc. as subclasses of Module in the example. But what would be the benefit? Do they have any different behaviour from the system's point of view?

Better, here, to have an attribute of a module which records what kind of module it is. Inheritance is best left for when some subclass has extra behaviour or relationships that don't make sense for the superclass.

Slide 41

4.8 CRC cards

CRC stands for Class, Responsibilities, Collaborations. CRC cards were introduced (and described in a paper at the major OO conference OOPSLA'89) by Kent Beck and Ward Cunningham, as a technique to help programmers with experience in non-OO languages to "think objects".

On a small card, we record the name of a class (at the top), the responsibilities of the class (on the left hand side of the card) and the collaborators of the class on the right hand side of the card. Class B is listed as a collaborator of Class A if either some object of class A sends a message to an object of class B, or vice versa – so in particular, a class is in an association relationship (possibly aggregation) with each of its collaborators. The responsibilities of the class describe at a high level the purpose of the class's existence: they're connected with the operations the class provides, but are more general than that. For example, "maintain data about the model" is acceptable as a description of a responsibility, though there might be many associated operations, for getting the various bits of data concerned. A class should not normally have more than three or four responsibilities, and many classes will have only one or two. If a class does end up having more than 3 or 4 responsibilities, consider whether you can describe them more concisely, and if you can't consider whether it would be better to split the responsibilities and have two or more classes.

Too many responsibilities corresponds to low cohesion in your model: too many collaborators corresponds to high coupling. Using CRC cards can help to identify both of these faults in your model.

You can also use CRC cards to walk through use cases, working out how the class model provides the functionality required by the use cases, and where the missing bits are.

Beyond that it's up to you. Some people like to pile cards related by generalisation together, with the most abstract at the top, so that you use the specialised class only when the particular specialisation is relevant. Some people like to group cards which are more closely related close together, or to put a "controller" class at the top.

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 next week 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.

Slide 42

A CRC card

Class OrderLine represents a single line on an order; for example if an order is for 6 blue widgets and 10p each and 8 red ones at 11p each we have two order lines, one for each kind of widget.

Order	
See whether items in stock	OrderLine
Work out total price	
Check for valid payment	Customer
Dispatch to delivery address	

Not a toy example: the scale of this is about right. Crowded CRC cards mean either badly expressed responsibilities or non-cohesive classes.

Slide 43

4.9 CRC Example

Let us return to the CS4 administration system.

Using the class diagram and the use case diagram, and using my hint to each group about where you should start, build a (maybe partial) set of CRC cards for this system. You may well find that you have to alter the class model or the use case model as you proceed: this uncovering of deficiencies is of course one main purpose of using CRC cards! (You should be much happier about altering the class model than the use case model, of course, since the use case model is supposed to represent the users' requirements, which you can't cahnge unilaterally! However, it could happen that you find the use cases as recorded are too specific, for example: you could find that you're able to represent equivalent functionality in a different way. I doubt that it'll happen in this example, but in general, it might well.)

If you find it helpful to get concrete – in this small example, you might, although in a larger system you probably shouldn't at this stage – you may use the reverse side of the CRC card to record other information, e.g., the names of any operations you identify.

4.9.1 Discussion

Many of you seemed to find this exercise rather difficult and confusing. Probably I should have led up to it more carefully: will do so next year! However, between you I think you did identify all the key interactions.

I've added to the Rose model I've been maintaining for this system, and provided access to the file from the course home page: you might like to browse it in detail, though I am definitely not claiming it's definitive! The collaborations are represented as associations on the class model; I've listed the high-level responsibilities of each class in its documentation field in the class model. I've also added a very short note to each use case to say how it is implemented. Here for example is the specification of HonoursCourse.

Class name:

HonoursCourse

Category: Logical View

Documentation:

Responsibilities:
Maintains and provides:
collection of modules
students registered
Provides the core of the course handbook, i.e. descriptions of all modules.

Actor interactions:
Anybody should be able to use students() and moduleDescriptions() –– probably only the co–ordinator and the UTO should be able to add or delete students.

Remarks:
may we want an interface class MailSystem and a method here to update the mailing list? For now we assume that's done manually from the students attribute.

Export Control: Public
Cardinality: n
Hierarchy:
Superclasses: none
Associations:

<no rolename> : GraduatingStudent in association is on
<no rolename> : Module in association

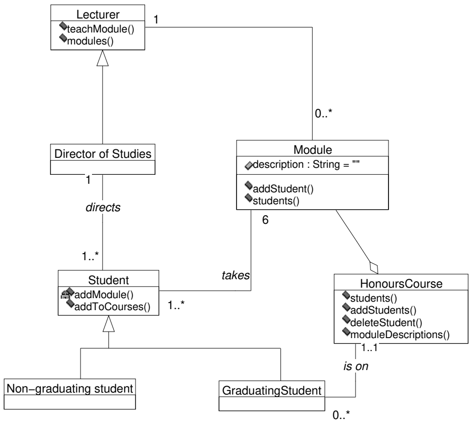
Public Interface:
Operations:

students
addStudents
deleteStudent
moduleDescriptions

State machine: No
Concurrency: Sequential
Persistence: Transient

Here's my revised class model, showing some operations needed to realise the responsibilities. The basic structure hasn't changed (much) from the previous version, only I messed up the layout so it looks as though it has changed more!

Revised class model



Slide 44

There are several issues that this brings up:

Very data-heavy system All our classes have encapsulating data as the major part of their responsibilities. This shouldn't surprise us, since the main purpose of the system is to maintain data about CS4! But it isn't really typical of OO systems – for example, the system you're dealing with for Coursework 1 has a lot more behaviour and less data, and is more typical. Classes which do nothing but encapsulate data are often, but not always, a sign that all the behaviour is somewhere else, maybe in a single Controller class, which is bad design because it tends to mean that the particular current use of the system (the current set of use cases) is being hard coded. So when your design has such classes you should check that you aren't missing some behaviour that goes with the data being encapsulated. In this case, I think what we have is reasonable: our system just doesn't *have* very much in the way of behaviour.

Maintaining associations Several of our operations (like addStudent on Module, addModule on Student) have the job of updating the association relationships which we already show on the diagram. Yet in other cases (like the association between Student and DirectorOfStudies) we don't provide operations to do this. Why, and how do you decide what to do?

I'm not sure that I can give a completely straight answer to this: I suspect it's an area where practice differs. I think the most useful thing is probably to show the relationship if it will be updated as the system carries out the behaviour you care about, but not if it's just set up when the objects are created and you don't expect it to change thereafter. For example, we do need to show how the particular Student objects get associated with particular Module objects, because this is the student registration process. But we haven't been asked to show how a student's DoS changes, so to show that would clutter the diagram. (Probably, the association is set up when the Student object is created – perhaps a DoS object is a parameter to the constructor of Student – and doesn't change thereafter.) As always, you have to do what you think is most helpful, and take the consequences in confusion if you get it wrong!

The user interface I haven't shown the user interface(s), i.e. how the actors get to use these facilities. On reflection, it became clear that a single UI object was not going to do: different actors need access to different parts of the system's functionality. I've commented on this in the Documentation field of each class on the class model, under Actor Interactions. (During development, you can use the documentation field for whatever seems useful – I've got a Remarks: section in many of mine, i.e. I'm using it as a convenient place to make notes about things we may want to think about later. By the end of the development, you probably want the class documentation field to contain only stuff that genuinely is documentation about the class, independent of the particular context in which the class is being used here. So the Responsibilities section would probably stay, but the rest would be deleted as the issues are dealt with.)

Probably, we would end up with a Model-View-Controller interface, in which each actor is provided with a different (but consistent) view onto the system. More design work needed!

Scope On writing this up, I noticed another feature that comes up fairly often: some uncertainty about *which* real-world objects we should be representing in our system. Pretty clearly we don't want random History students represented. Probably we just want students who take some CS4 module. But then those students may have DoSs not in CS and take modules not in CS, so we are of course representing lecturers and modules from other departments. We might want to consider to what extent we want to do this.

New references

- Schlaer, S. and Mellor, S *Object oriented system analysis* 1988 Yourdan Press
- OU M868 course material
- Coad,P. and Yourdan, E. *Object oriented analysis* 1990, Prentice Hall.

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 (Use Case View) – e.g. recording how you wave CRC cards around!
- Showing how an object reacts to some message (Logical View)

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 45

5 Interaction diagrams

The work we've just done with CRC cards highlighted an aspect of the design that we don't yet have a way of recording: the way that objects work dynamically together to achieve the goals of the system, e.g. as expressed in the use case model.

UML allows two major ways of doing this, and both are informally called interaction diagrams. The two ways are *sequence diagrams* and *collaboration diagrams*. You won't usually use both within the same project: it's more likely that that you'll pick one you prefer in the context and use that. If you're using Rose or a similar tool, you may not need to commit yourself: in Rose you can create one kind of interaction diagram, and Rose will create the other kind from it, and will then maintain the two in a consistent state, so that you can consult whichever seems more readable in a given context. As this implies, they record essentially the same information. (NB Rose isn't very good at laying out the diagram, so you may have to move the elements of a generated diagram around yourself to make it legible.) Most of what I want to say applies to both forms.

As the name suggests, the interaction model describes the interactions between objects. It does not describe the internal computation within an object. So, for example, an interaction diagram may show object aCustomer sending a message to aBankAccount, but it will not record how object aCustomer knows which message to send, or which object to send it to. If we need to record that kind of information, we'll use *state diagrams*.

The interaction model is in some sense less fundamental than either the use case model or the class model. Simple projects might well not develop them at all: if the interactions are sufficiently clear, you might go straight to code. Most projects will not develop interaction diagrams for every use case, but rather, will use interaction diagrams as a way of expressing the possible options in difficult cases.

Interaction diagrams can also be useful for documenting what an object should do when it receives a particular message, whether or not this corresponds obviously to some use case.

Generic and instance forms Interaction diagrams in UML come in two forms, a generic form and an instance form. (This is not explained well in Fowler: possibly it wasn't settled when that went to press.) In the generic form, you show all possible interactions between objects of the relevant classes: that is, you describe the whole use case, including all its variants, using special syntax to show things like constraints and iteration. In the instance form, you just describe one possible instance of this behaviour; that is, one scenario, without branch points or unspecified numbers of repetitions. If you use the instance form to describe how the system realises a use case that has several scenarios, you may well have to develop several interaction diagrams for the same use case. For example, in the case of the WithdrawMoney use case, you might have one instance form interaction diagram to show what happens if the customer doesn't have enough money in the account, and another to show what happens if s/he doesn't.

We'll deal with the instance form, since this is simpler, easier to use, and therefore arguably more useful. (And because it's what Rose supports!)

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.

(more detail in the notes)

Slide 46

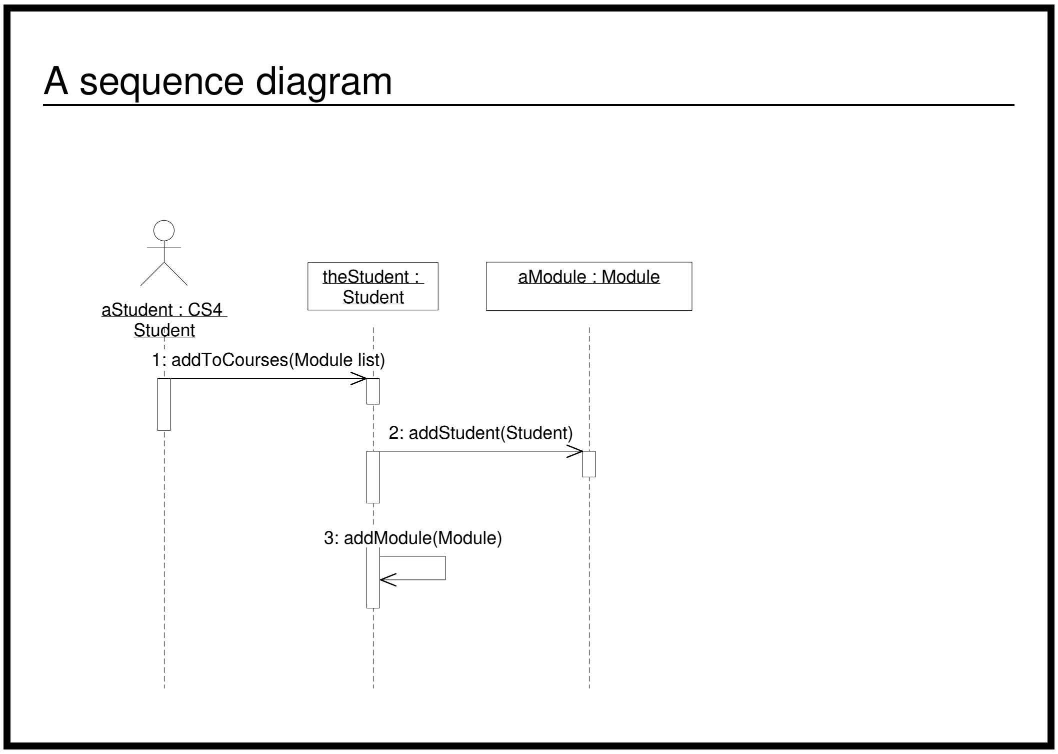
5.1 Developing an interaction diagram

In more detail:

1. That is, which use case, and if there are several scenarios in the use case, which scenario; or else, what message, and what assumptions if any you have to make about the arguments to the message in order to know exactly what messages will pass.
2. So you have to know how the system provides the functionality, maybe from your CRC work, maybe from some other technique. (If it's obvious, you probably shouldn't be developing the interaction diagram at all.)
3. By convention, if an interaction involves some object of class Foo, we often call the object aFoo. (Obviously you need another convention if an interaction involves several different objects of the same class.)

4. Remember that the message includes the selector as well as the specific arguments. UML allows you to give particular arguments to the messages that appear in an interaction diagram: for example, you could develop a diagram to show what happens when a bank customer withdraws £20. Rose, however, only allows you to specify the operation: you have to show any constraints using notes.

5. See specific sections below.



Slide 47

5.2 Sequence diagrams

A sequence diagram shows the objects at the top of dashed lines. The line represents time as seen by the objectL it is the object's *lifeline*.

The order in which you show the objects doesn't matter, though it will make for a more readable diagram if you put the objects which take part earliest furthest to the left, so that most messages flow left to right.

If object1 sends message selector(arg1,arg2) to object2, you show this by an arrow from object1's lifeline to object2's lifeline, labelled with the message selector(arg1,arg2).

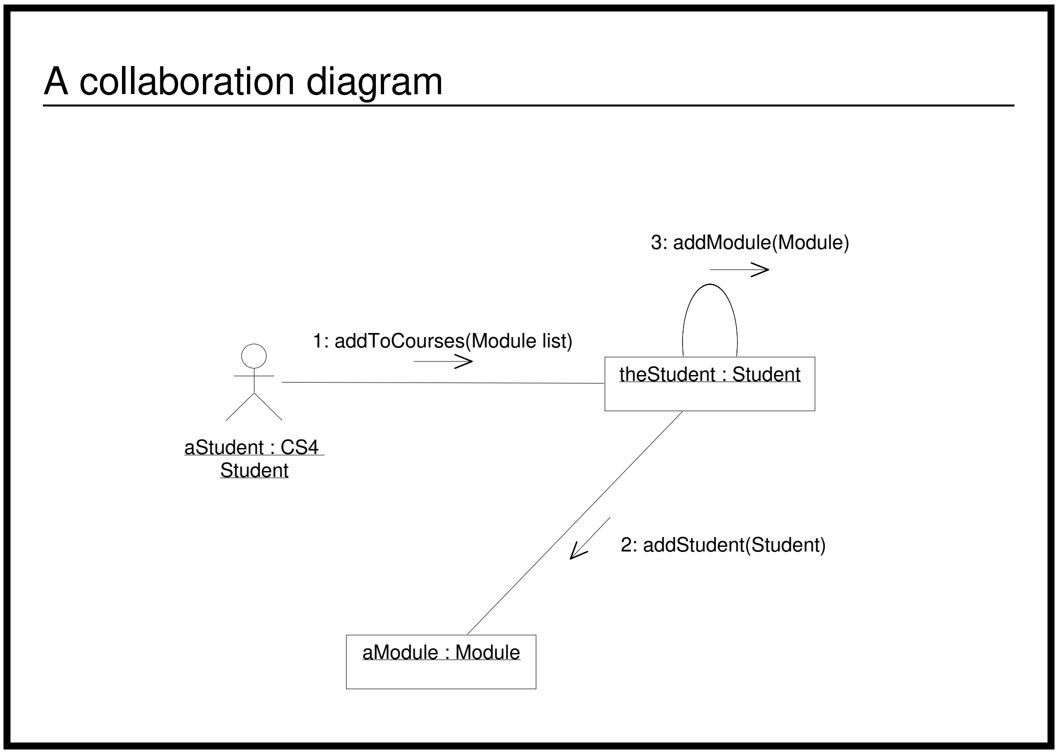
(You can show an object sending a message to itself, using an arrow from that object's timeline back to the same timeline a little further down, but in most cases you will probably decide that that counts as internal computation inside that one object. As mentioned, the main purpose of interaction diagrams is to show different objects interacting. It's a matter of taste, though: decide which is clearest in each case.)

The vertical position of an arrow represents the time at which the message is sent, so arrows appearing further down the page represent later messages. Messages are also (usually?) numbered in order, though the information is redundant.

UML permits actors, as well as objects, to appear at the top of lifelines. In the example, I show the actor aStudent sending a message to the object theStudent. Of course the actor – the flesh and blood student – can't do this directly! But at some stage we do have to bridge this human/machine gulf, if we want to have diagrams that show how they relate. Assume that when an interaction diagram shows an actor sending a message, it means that somehow or other the actor causes the message to be sent.

Here again you have to know that Rose treats actors as a special kind of class: to get an actor into a sequence diagram, you put an object there, and then in the specification of the object you specify an actor as the "Class" of the object. The shape of the object on the diagram will change from rectangular to actor-shaped.

In UML we can also show object creation and deletion, and a host of other features, but we won't discuss them here. (If you want more detail on this or any other aspect of UML, see the UML notation guide accessible from the course home page.)



Slide 48

5.3 Collaboration diagrams

The main difference between collaboration diagrams and sequence diagrams (at least, the simple forms we're considering) is the layout of the objects and messages. A collaboration diagram does not have object timelines, and you cannot deduce whether one message occurs later than another by looking at their relative positions on the page. In a collaboration diagram the numbering on messages is essential, not, as in sequence diagrams, a redundant repetition of information shown on the diagram. The advantage of this is that it gives you freedom to use the spatial relationships between objects for other purposes. We use a plain line between objects to represent an association between their classes, which should, of course, correspond to an association (maybe the special kind of association, aggregation) in the class model.

Collaboration diagrams in Rose In the "link specification" that you get by double-clicking (etc.) on a link between objects in a collaboration diagram, you can set the "association", and Rose gives you a menu to choose from which, if you've specified the objects' classes, will show the known association between those classes. Warning! Do not select this menu item! Rose will crash if you do. Type the name in instead. It's also a known bug that long labels on the messages sometimes display truncatedly. If on an interaction diagram you see only part of your message's name shown, clicking on the name and then on the background will normally fix it. (Any more Rose bugs, to me: I'll be sending them in at the end of term.))

Trap 4: allowing circular dependencies

going through more than one "sensible reuse unit".

E.g. (from Webster p228):

graphical rendering subsystem uses calls from text rendering subsystem uses calls from line layout subsystem uses calls from graphical rendering subsystem.

It's quite easy to get this by accident: one reason why we delay deciding in which direction our associations are navigable.

Problem: if this happens you can say goodbye to reuse. It's also very hard to understand.

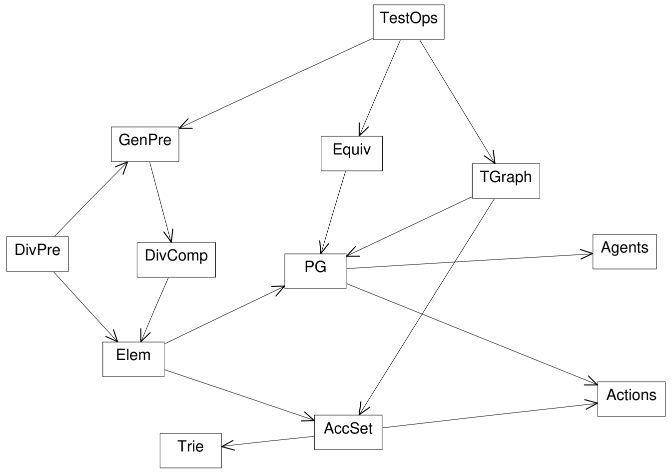
Small-scale case: if possible, avoid making associations navigable both ways. (But not if the only way to do it is to have a huge non-cohesive class including both "real" classes!)

Slide 49

It was correctly pointed out in the lecture that there isn't anything really special about circular dependencies (though this is languge dependent: in some cases it may be impossible to even compile such a system). The problem is with tightly coupled systems in general, and the thing about circular dependencies (with large circles) is just that they invariably are highly coupled. In particular, you can't reuse anything on the circle without reusing everything on the circle.

Here's a genuine example of a horrible dependency structure that doesn't include any circular dependencies – but it takes a while to satisfy oneself of this! When you add in the fact that PG represents an implementation of graphs which exposes every implementation detail to the outside world, you'll appreciate how hard it is to change this system, let alone reuse parts of it.

CWB dependencies (not designed by me!)



Slide 50

6 Design Patterns

6.1 What are patterns, and why?

Design Patterns

"Reuse of good ideas"

A pattern is a named, well understood good solution to a common problem.

Idea comes from architecture: e.g. **Window Place:** observe that people need comfortable places to sit, and like being near windows, so make a comfortable seating place at a window.

Similarly, there are many commonly arising technical problems in software design. Experienced designers recognise them and understand how to solve them. Without patterns, novices have to find solutions from first principles.

Patterns help novices to learn by example to behave more like experts.

Pattern catalogues: for easy reference, and to let designers talk shorthand.

Slide 51

The point is that you get to spend your effort on building the *best* solution, whereas if you have to start from scratch you may have to spend a significant amount of time hunting for any solution at all: and you may not spot disadvantages of a solution until you get bitten by them. Patterns let you stand on the shoulders of giants.

Patterns are almost by definition not new: they document, at a suitable level of abstraction, designs which are already tried, tested and well understood. Experienced designers probably already know most if not all of the standard ones, and if you read either of the recommended books there will be patterns that you've used, I'm sure.

Even if you already recognise the techniques used in a pattern and are confident you would essentially have used it in appropriate circumstances anyway, it's still useful to have a good, abstracted description. It makes it easier to spot improvements and variants: and the handle that a name with a commonly agreed meaning provides makes it easier to discuss designs.

Patterns exist at many levels: Buschmann et al distinguish between *architectural patterns* (such as layered architecture, and MVC which we'll look at) at the highest level, *design patterns* (such as Publisher-Subscriber, which we'll study) in the middle, and *idioms* at the programming level. Confusingly all three of these levels are included in what most of the patterns community calls *design patterns*.

We'll study one very simple pattern in detail, and then look at how it is incorporated in a more complicated framework, which we won't cover in detail.

6.2 Example: Publisher-Subscriber

One common problem

It often happens that when one object changes state, one or more other objects need to know about it so that their states remain consistent.

E.g. one or more elemements of a user interface need to show the correct version of some data item, part of the state of an object (say aCustomer) inside the system.

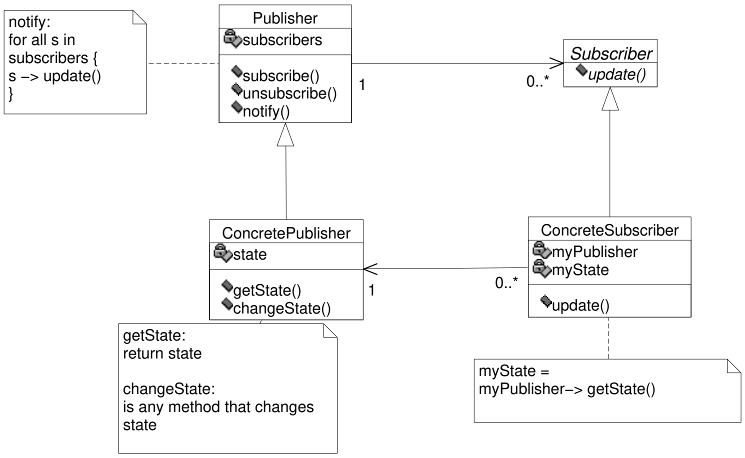
We could introduce explicit dependencies, and have the changing object notify its dependants by sending them messages: but this is inflexible and prevents reuse (e.g. of aCustomer). User interfaces are very prone to change, so internal objects shouldn't know what the user interface is like.

We could have the dependants poll the changing object to see when it changes: but this is very inefficient.

May well want to add or remove dependants at run time.

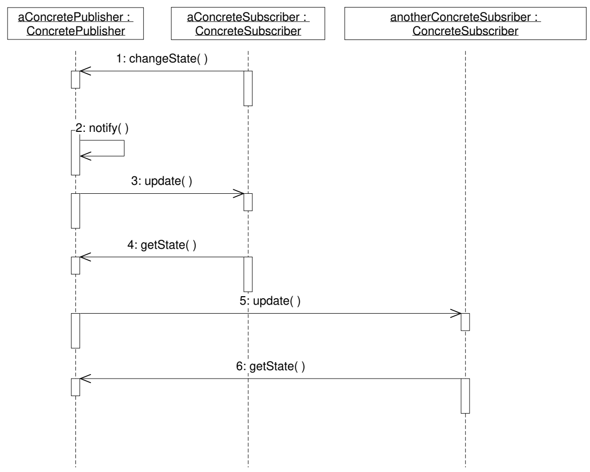
Slide 52

Class diagram



Slide 53

Sequence diagram, 2 subscribers



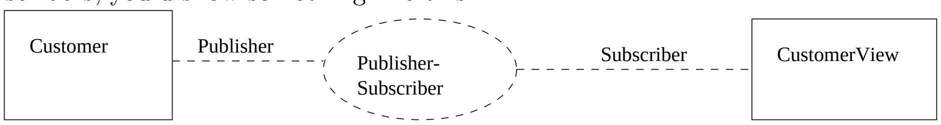
Slide 54

6.3 Pattern catalogues

Rather than giving general instructions about what constitutes good design, a *pattern catalogue* documents particular designs that are useful in certain contexts. One of the main challenges for the patter writer is to get the level of abstraction, at which the pattern is described, right: it must be abstract enough to be applicable to as large a family of situations as possible, but not so abstract as to be platitudinous. (Unfortunately even the experts don't always get this right: it's harder the larger the scale of the pattern.)

Pattern catalogues have to be easy and quick to refer to. A common format for describing patterns helps here. There's a role for UML here too: the existing patterns books each use their own notation, usually some form of OMT-like class diagram and sequence diagrams like UML's. In future, assuming UML really does become the standard modelling language, we may expect pattern catalogues to describe patterns using it, so that you will be able to expect not to have the language barrier imposed by the need to understand different notations in different books.

UML in fact includes the possibility of using patterns explicitly in class models: the pattern name is shown inside a dotted oval, with dotted lines joining the oval which represents the pattern to the class rectangles which represent the classes taking roles in the pattern. The lines are labelled by role names. So for example, if your design uses Publisher-Subscriber with objects of class Customer as the Publisher and objects of class CustomerView as the Subscribers, you'd show something like this:



As you see from my xfig'd diagram, Rose does not support this notation.

If you do use patterns, by name without explaining them, as part of your design, it is of course important to have an agreed "vocabulary" of patterns which can be used in this way! This might be agreed at the team level or the organisation level. Probably you would name one or two pattern catalogues whose patterns could be used without description: one would almost certainly be Gamma et al.; I've already seen their patterns referred to by other papers in the expectation that the readers will know them. Possibly an organisation might build up its own collection of patterns as well as agreeing on standard ones: but I have a suspicion that the need to do this might be a sign that the patterns are being described at too low a level of abstraction. I doubt that genuinely commonly occurring design problems are really domain-specific. This really remains to be seen.

Elements of a pattern

A pattern catalogue entry normally includes these elements:

- Name (e.g. Publisher-Subscriber)
- Aliases (e.g. Observer, Dependants)
- Context (in what circumstances can the problem arise?)
- Problem (see slide above)
- Solution (see slides above: normally a mixture of text and models)
- Consequences (good and bad things about what happens if you use the pattern.)

The elements aren't always named in this way. GoF lump context and problems together, Buschmann et al don't have a separate consequence section.

It may contain more, e.g. variations, CRC cards, example code.

Slide 55

6.4 Patterns in context: MVC

In this section we consider the Model-View-Controller user interface architecture. We do not go into full detail even of MVC, let alone of the motivations for using it or not. It would be quite reasonable to have an entire course on the software engineering of user interfaces, but this course isn't it!

There are two reasons for looking at MVC:

1. to show a more complex pattern
2. to show a technique for building a UI onto an OO system, and to make it clear "how the whole thing is started up", which some people were justificably confused about when we studied the CS4 admin system.

User interfaces: hard problems

Why? They're very visible: all errors matter. They're prone to changes in requirements. They often have to exist on different platforms. They're complicated. They're often not just text. And we haven't mentioned usability yet...

OO and GUIs have grown popular together: OO techniques for GUIs have a long history.

Building on Publisher-Subscriber, we'll *briefly* consider Model-View-Controller, a framework which originated in Smalltalk-80 (cf InterViews).

More (e.g. Presentation-Abstraction-Control) and less (e.g. Document-View, Model-Pane-Dispatcher) sophisticated OO techniques exist. So do non-OO techniques, e.g. Seeheim model: and one can build hybrids.

Slide 56

MVC in brief

The **Model** is the core of the application: e.g. the CS4 admin system. It is independent of the user interface.

Their is one **Controller** for each **View**; there may be many pairs for one Model. Between them Views and Controllers are the UI.

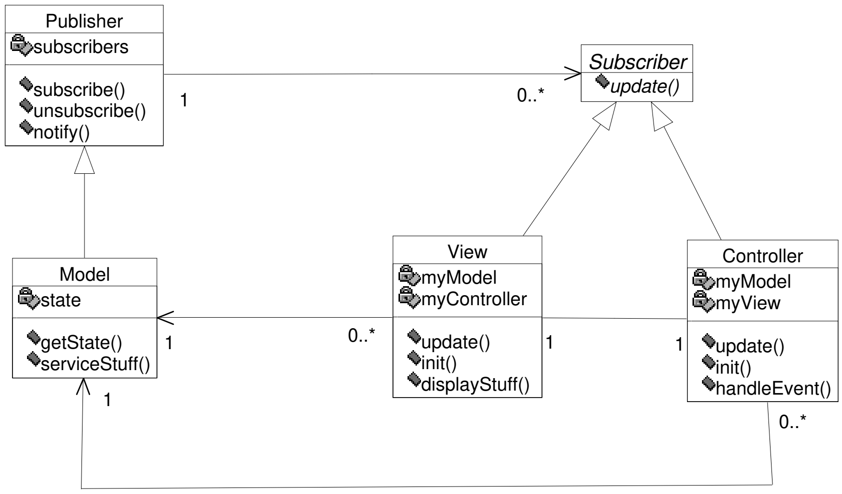
View components control how data from the model is presented to the user. A View normally creates and initialises its Controller.

Controller components accept user input (as events from the X event handling mechanism, or whatever) and do the right thing.

There may be compositional hierarchies of them (using the pattern Composition), but we'll ignore that for simplicity here.

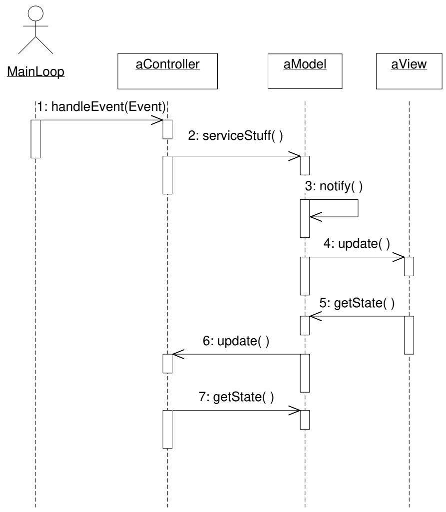
Slide 57

MVC class model



Slide 58

MVC handling an event



Slide 59

The controller normally reacts to user input by invoking public methods on the model. (For example, if the user presses a button or whatever that is supposed to mean "get me the course handbook material for this honours course", the controller reacts to the button press by sending message getDescriptions to the correct HonoursCourse object. Notice from this that of course the UI does have to know about the model! It's the reverse direction we're trying to avoid. Sometimes, however, the user input will be invoking functionality which is independent of the model – scrolling, for example – in which case the controller may instead send message directly to the view.

The view and its controller are quite tightly coupled, and for this reason some variants on MVC, like Document-View used in Visual C++, combine them into one class. On the other hand it is possible to swap the controller of a view for another, and sometimes useful: for example, to inactivate an interface component you might swap its controller for one that ignores all input.

There are also other patterns that help in building generic controllers e.g. to do the right thing with selection lists: when you use such a controller, you configure it with which message it should send to the model under what circumstances.

It's clear that Views have to be Subscribers in the Publisher-Subscriber pattern, with the Model as the Publisher. We've also shown Controllers being possible Subscribers to the Model. This is because certain kinds of interaction may only sometimes be available: for example, the state of the model may determine whether a menu item is greyed out, so then the controller needs to know when the state changes. (Yes, there's a grey area connected with exactly what's the responsibility of the View and what of the Controller.)

In the beginning

A small amount of setup code – outside even the model:

1. creates and initialises the model
2. creates and initialises the views, setting up the association with the model (e.g. by passing a reference to it). On initialisation:
   - The views create and initialise their controllers, which
   - register interest in whatever events they want
3. starts event processing, using `XtMainLoop()` or whatever.

From then on, the event processing loop feeds user input events to the controller... and it Just Works.

If our application has many controllers at different levels of abstraction, there is work to be done to get the event passed down to the Controller at the correct level of abstraction: but it's generic. Toolkits help.

Slide 60

New references

For links to patterns-related material, see my software engineering links page – there's a link to that from the course home page.

- Gamma, E., Help, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software* Addison-Wesley Professional Computing series, 1994.

  This is the Gang of Four book, the most famous and important pattern book to date.

- Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. *A System of Patterns: Pattern-Oriented Software Architecture* Wiley

Also good: more patterns in less detail.

- Fowler, M. *Analysis Patterns: Reusable Object Models* Addison-Wesley 1997

  I don't own this one, but it's by the same person who wrote UML Distilled. It's probably good!

7 Activity Diagrams

Activity diagrams can be useful in broadly the same ways as interaction diagrams:

- to show how an object reacts to a message being sent to it, or
- to show how a system provides the functionality required by a use case.

They can also be useful for describing how use cases depend on one another, for example, when it's the case that one must be carried out before another can be started. In this context, you might use an activity diagram during analysis to document the overall *workflow* of an area of the customer's activities.

Rose does not support activity diagrams. (In very simple cases, not involving synchronisation or decision diamonds, you might be able to draw them as though they were statecharts – see Relationship between activity diagrams and statecharts at the end of the next section.)

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.

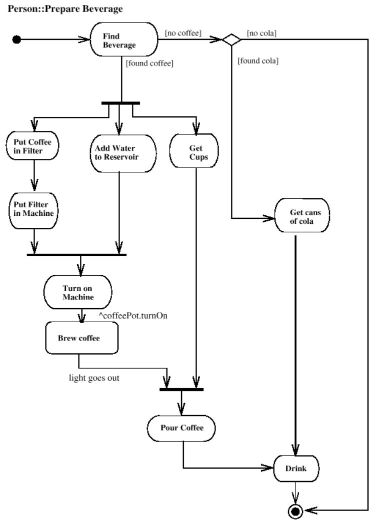
Main advantage: can show parallel activities, so make dependencies and non-dependencies explicit.

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 61

Activity diagram

Figure 52. Activity diagram



Slide 62

Elements of activity diagrams

activity: the rounded box showing something that happens

synchronisation bar: arrows out of here can't be taken until all the arrows into it have been taken

transition: from an activity to the next thing (an activity, a synchronisation bar or a decision diamond)

decision diamond: used to show nested decisions. How you use them is to some extent a matter of style: e.g. in the example we could have shown three arrows out of the Find Beverage activity, with guards [found coffee], [no coffee AND no cola] and [no coffee AND found cola], but it's probably clearer the way it is.

start marker: optional, but usual: as for state charts, see below.

end marker: ditto

guard: e.g. `[found coffee]`: shows under what circumstances the transition can be take. If it's omitted, the transition can always be taken.

event: e.g. `light goes out`. Something that happens to trigger the transition. Usually not shown, because the transition is just triggered by the completion of the preceding activity; a matter of style.

action: e.g. `ĉoffeePot.turnOn`. Something the system should do when the transition is taken, e.g. send a message (the message `turnOn` to the object `coffeePot`. Sometimes as here, the sending of a message is really all that an activity represents.)

Advantages of activity diagrams They aren't sequential. In the interaction diagrams, you describe a particular *sequence* of messages that get sent. Even if two messages could be sent in either order, you still have to specify one order or the other in the diagram. This represents our old enemy, the premature design decision. Even in in the end you will implement the system in single-threaded code, it's a good idea not to decide on the order of events until you must. Some orderings will be essential (e.g. check payment before deliver goods), others inessential (e.g. it doesn't matter whether we check payment before or after we check whether the goods ordered are available). In order to keep the design as flexible as possible, so as to have leeway to change things at a late stage if necessary, it's a good idea to maintain the distinction between essential and inessential orderings.

Of course, if your system will *not* be implemented by single-threaded code, your incentive to use a notation like activity diagrams, which can represent several things happening at once with specified synchronisation points, is overwhelming, particularly since multi-threaded applications are notoriously hard to understand just from code.

Disadvantage of activity diagrams Like use cases, they aren't object oriented. The items shown are pieces of functionality, not objects. A box in an activity diagram won't necessarily even be something that can be implemented by a single method. All the same caveats that we put on use case diagrams apply: relying too heavily on them might make you tend to ignore and compromise the object oriented structure of your system.

8 Statecharts

You've probably all seen state diagrams before. They are useful for describing individual objects in detail. You can describe the states an object can get into and the messages that cause it to move from one state into another. This is likely to be a useful thing to do when an object has complicated behaviour, and particularly when how it reacts to a message depends in a qualitative way on what state it was in.

What do we mean by qualitative in this context? Well, for example, perhaps a bank has a policy that customers can withdraw money, even if it makes them overdrawn, unless a stop has explicitly been put on the customer's account. Then perhaps objects of class Account have two states, Normal and Stopped, and reacts differently to a message withdrawMoney(amount) depending on the state it's in. On the other hand, it will send back different answers to the message getBalance depending on what the balance is, but this isn't different behaviour in the sense we want to consider. There's no hard and fast distinction, because ultimately all the object's data is its state, so in the extreme you can argue that an object changes state every time any part of its data changes at all. A state diagram should show the interesting ways in which an object's behaviour (including which messages it makes sense to send it) depends on its state, and you decide what's interesting in the current context. (In particular, the number of states that there are in an object's state diagram, even if we specify the object completely, is usually somewhat subjective, because two designers can legitimately make different decisions about what's interesting.)

More formally, you can think of states as shown in a state diagram as equivalence classes of "real" states of the object, where two states are equivalent if the object "behaves qualitatively the same way" in both states. I haven't managed to come up with a formal definition of "qualitatively" that isn't obviously useless! Let me know if you think you have one.

Statecharts

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.

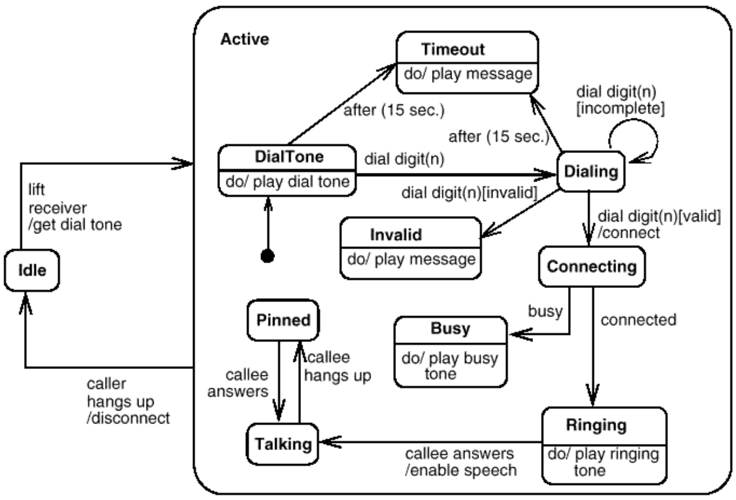
Most classes will not need statecharts drawn for them.

Slide 63

Most classes will, and should, have boring state transition diagrams (e.g. with one state) that aren't worth drawing. One reason why they should is because an object which reacts in a qualitatively different way to a message depending on its state is harder for client code to interact with correctly than one that does not: there is a new class of possible errors, namely assuming the object's in one state when it's in another. Sometimes a complex state transition diagram is essential, but other times it's a sign of not having found the right abstractions yet – maybe a class is trying to do the work of two, or maybe you have one operation where it would be better to have more than one.

Statechart

Figure 41. State diagram



Slide 64

Modelling the state changes of an object is the most common use of a UML statechart, but of course you can also use statecharts in other situations if it seems useful: e.g., to model the changes of local state internal to some complex method.

Elements of a statechart

Start: shown as a black disc. This is optional, but should usually be included. If I'd drawn the example diagram, I'd probably have put a start blob with an arrow to the Idle state, meaning that the initial state of the system is idle.

State shown as a named round-cornered box, e.g. the one named Idle. States may be nested: the example shows that states DialTone, etc., are substates of Active. Notice that a statechart nested inside a state, like the one nested inside the Active state, can have all the usual features of a statechart; here the nested state chart has its own start blob.

State compartment entry, e.g. do/play dial tone. This is an instance of a compartment entry with form do/*activity*, representing some activity that should be done when the object is in the associated state. The *activity* could be the name of another statechart.

Transition shown as a labelled arrow between states. The label has three parts, any of which can be missed out if irrelevant. (But if all three are irrelevant, there's probably something wrong – are the states really different?). The full form is *Event [Guard] / Action*. For example, in `dial digit(n)[valid]/connect` we have:

- an event, `dial digit(n)`, which is what prompts the transition between two states.
- a guard, `[valid]`, which says that this transition can only be taken if the condition `[valid]` is fulfilled – presumably in this case this means that the new digit n is a valid continuation of the number so far dialled. Notice that if the condition is not fulfilled, there is another transition with the opposite guard which can be

taken. This condition – that if an action is possible from a given state at all, the guards on the transitions with that action OR up to true, as in this case (valid OR invalid) is equivalent to true – is very common, but UML doesn't insist on it. That is, if we deleted the transition `dial digit(n)[invalid]` from the example we'd still have a legal statechart. However, it might not be clear what it meant – you should probably add an explanatory comment to a diagram in which an action is only sometimes possible from a state. It may be that the situations in which the action is possible actually correspond to something that ought for clarity to be shown as a state in its own right.

- an action `connect` which should be done when the transition happens.

Notice the difference between an event and an action. Basically, an event corresponds to something that happens to the object (probably the object receives a message) and an action corresponds to something the object does (probably sending a message).

End shown like a start state but with a circle drawn round it. Optional, and less important than the start symbol since any state with no outgoing transitions is obviously an end state. Use it if you think it makes the diagram clearer, perhaps to show finishing in an expected and orderly way.

Statecharts in Rose To create a statechart for a class, select the class in the class model, then choose State Diagram from the Browse menu – it's greyed out unless you have a class selected, which seems like a bad user interface feature to me, but there you are! If you want to show an activity (or an entry, exit or upon action, which we haven't discussed) in a state, you have to

1. get the state's specification window up (e.g. double click on the state)
2. go to the detail tab
3. pick Insert from the shortcut menu up (right mouse button)
4. specify the activity in the Action Specification window

There may well be a shortcut that I haven't found (let me know if you find it!)

Terminology State diagram, statechart, state transition diagram, and probably others, are all synonyms.

Relationship between activity diagrams and statecharts The UML documentation states that an activity diagram is a special case of a state diagram, namely the state diagram of a procedure. It says that the activities are the states of the procedure and the triggers are transitions.

At first sight this is very strange – if two activities are shown as happening in parallel, this interpretation says that the the procedure is in two states at once – scarcely intuitive, unless you're into quantum mechanics, and even then the analogy is flawed!

Actually when you look into it more, you see that (although I haven't told you about this feature) UML allows statecharts to describe multi-threaded behaviour – so even in a statechart, an object can be in more than one state at once, and this is to be interpreted as parts of the system being in the specified states. This feature could be useful if you're describing a multi-threaded system, but is unlikely to be useful otherwise. However, it does mean that UML activity diagrams and statecharts aren't as different as one might imagine without knowing about the extra features of UML statecharts: the concept of being in more than one "state" at once exists in both.

Accordingly the notation used by UML for statecharts and for activity diagrams is closely related. (So much so, that it's a bit surprising Rose doesn't support activity diagrams – you'd expect them to be able to do this easily by reusing some components :-)

9 Implementation model

UML defines two models which describe how your system is implemented. The *component model* shows the dependencies between parts of the code, and the *deployment model* shows the structure of the run-time system: which bits run on which processors.

9.1 Component model

Component diagrams

show dependencies between software components.

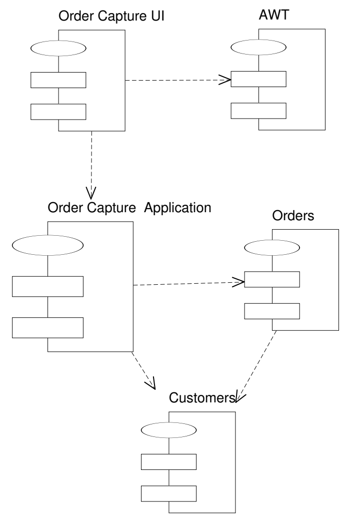
A component may be

- 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 65

A component diagram



Slide 66

Note This is one of the less stable (or at least, most recently stabilised) areas of UML. You'll get 3 different views if you read Fowler's book, the UML1.1 documentation, and the Rose on-line help. I've tried to stick to the common core! The main question is whether a package is the same as a component (more or less Fowler's view) or a more general thing that can contain components (UML1.1 and Rose). Here I take the latter view, and then don't talk about packages, but only about components!

Fowler's book doesn't describe component diagrams as such. He uses *package diagrams* (a special case of a class model, using the fact – which we didn't cover in the course – that you can group classes into packages and show dependencies between them) to show dependencies between subsystems.

9.2 Deployment model

Deployment diagram

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

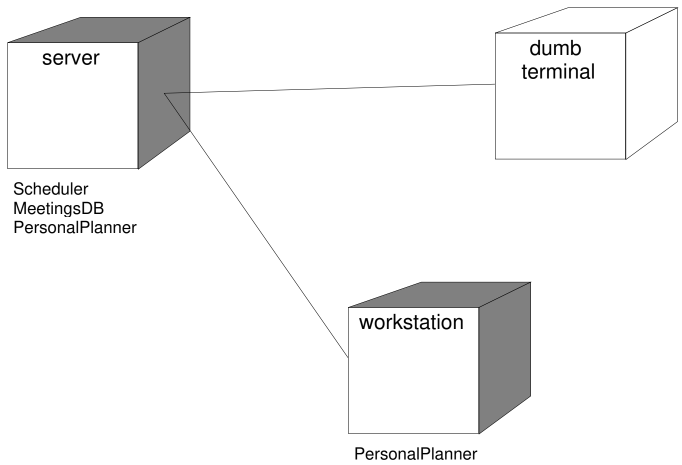
Dark boxes represent machines with processing power, light ones represent devices with no processing power. (Grey areas: Xterms?)

Lines between boxes represent physical connections between machines.

UML's view and Rose's are somewhat different – UML allows more detail about what's where than just processes, but doesn't distinguish devices from processors.

Slide 67

A Rose-style deployment diagram



Slide 68

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 69

What was it all for?

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.)

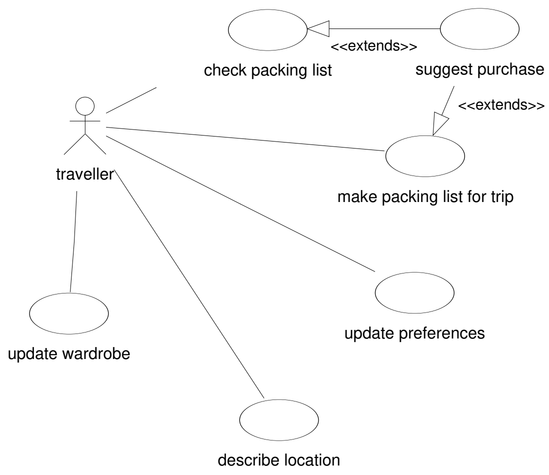
We've also talked a little about the **development process**, which *is* concerned with how we get designs and programs:

- example Methodology: Objectory
- CRC cards
- design patterns
- module dependencies

The rest of the course is about aspects of the development process.

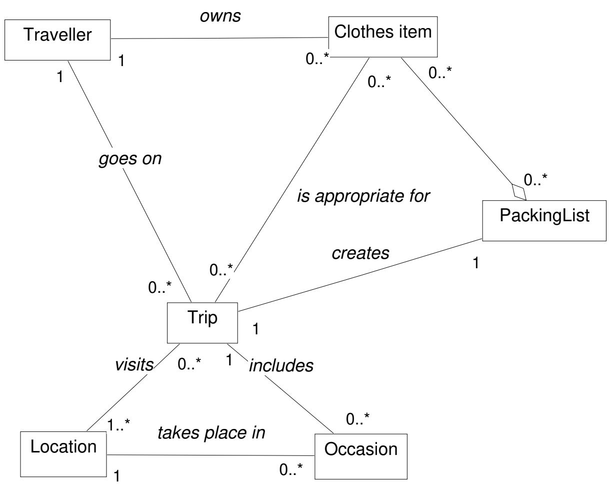
Slide 70

A use case model for Coursework 1



Slide 71

A class model for Coursework 1



Slide 72

10 Quality

Reading, assessment

Various papers are part of the material for this course. Some are required reading. This means they are examinable! Others are merely recommended.

You should not be aiming to memorise details from the papers: what I'm after is an understanding of the main issues, questions, problems, solutions to do with quality and things that affect quality, especially human aspects of them.

One way of revising them: write your own one page summary, bullet point style, of a paper. Then get together with a few other people and discuss your summaries: work towards a concensus on what's important.

Slide 73

Required reading on quality: Two short articles: How to Avoid Software Quality, and What is a High Quality Software System.

11 Testing

Testing

You should remember (from PM) the process aspects of testing.

Ways of testing: black box (specification-based) and white box (structural).

Different testing purposes:

- Module (or unit) testing
- Integration testing
- System testing
- Acceptance testing
- Stress testing
- Performance testing

and maybe some techniques, like equivalence partitioning?

Slide 74

Why test?

Testing has three main purposes:

- to help you find the bugs
- to convince the customer that there are no bugs
- to provide information that will help with system evolution.

The first one is the hard one!

A successful test is one that finds a bug.

Only once you can't find bugs does it make sense to try to convince the customer that there aren't any. (Honest!)

Slide 75

Test what?

From the customer's point of view:

- Testing a system's capabilities is more important than testing its components.
- Testing old capabilities is more important than testing new capabilities.
- Testing typical situations is more important than testing boundary value cases.

Petschenik, H.N., *Practical priorities in system testing*, IEEE Computer, 2(5) 18-23.

Often overlooked: customer documentation testing.

Slide 76

How to test

Need to test old functionality: regression test. Therefore however tedious it is, tests must be:

- repeatable
- documented (both the tests and the results)
- precise
- done on configuration controlled software

Ideally, test spec should be written at the same time as the requirements spec: this helps to ensure that the requirements are highly testable. It may seem backwards to consider testability when writing requirements but it's now standard.

E.g. may write requirements in numbered sentences and keep a tally of which test(s) tests which requirement(s). Use cases may help structure tests.

Slide 77

Why is testing so often done badly?

It's *very very* **boring**, especially the documention of the whole process. Automation may help.

It's very expensive, in time and hence money. (May exceed 50% of project time: 30% is about normal.) The alternative may be worse, but the cost may not show up until later...

It's often planned mostly for the end of the project, and gets squeezed by deadline pressure.

Customers may pressure developers to deliver on time and untested rather than late and tested. (What should you do?)

Slide 78

Special problems of OO

Two main extra problems at the unit testing level:

1. The unit is the class, which is harder to test than a function. It has state, on which its behaviour may depend – so can't test methods as though they were independent.

   Moral: try to avoid designing classes with complex statecharts.

2. Polymorphism. When an object of class A interacts with an object of class B, it may in fact be interacting with an object of any subclass of B – which may not even have been invented when class A is tested!

   (This problem is to my mind the most convincing justification for work on formal methods...)

   Moral: remember this problem, and keep inheritance for when its advantages outweigh the disadvantages.

Of course, because the state dependency is in objects, we hope to save effort at integration and system testing, and to win overall.

Slide 79

But that's not the whole story

Problem: it's very hard to describe complex tests properly. How to handle bugs that only show up if you do a complex sequence of things?

- write well-modularized and module-tested software so that this is less likely to happen in the first place!
- have someone testing software with the aim of breaking it, however sneakishly.

(E.g. particular problem of testing GUIs: if you can come up with a really good solution to this one, patent it!)

That is, precise short documented tests are essential, but not sufficient.

The Black Team.

Slide 80

Reading

Software Quality, Software Process and Software Testing

by Dick Hamlet.

Optional – it's revision of this course's prerequisite, PM, so omit iff you're confident you remember this stuff – sections 1,2,3,7 are relevant.

Required reading:

Improving the Software Testing Process in NASA's Software Engineering Laboratory

by Sharon Waligora and Richard Coon

Slide 81

12 Social structures

i.e. Management, teams, leadership, organisations

12.1 Management

Slide 82

One view: management as enabling activity

> No project can succeed without management support. The best sort of management support is the kind in which management doesn't find out about the project until it's a market success. If management notices a project too soon it'll support it in the following ways:
>
> - Demand frequent status reports to explain why the team doesn't have enough time to meet deadlines.
> - Demand explanations of how the project differs from all the projects that have similar acronyms.
> - Ask the team what it could do if it had only half as much funding.
> - Appoint an Oversight Committee whose members are always on trips.
>
> To put it another way, managers understand that their role is to remove obstacles from the project team. They probably could do that, with the help of Dr Kevorkian, but most managers are not such good sports. Therefore, coincidentally, the biggest obstacle to the success of any project is management itself.

(from The Dilbert Principle, p233)

This is of course a caricature: but the kernel of fairly uncontroversial truth here is that an important function of the manager of a task is to keep mindless bureaucracy away from the people who are doing the task, maybe by making the necessary arguments to make it go away, and if necessary by doing the dirty work him or herself.

Another view: management as financial control Here's an extreme expression of a financial view of management (from Drucker, P.F. The Practice of Management):

> Management must always, in every decision and action, put economic performance first. It can only justify its existence and its authority by the economic results it produces. There may be great non-economic results: the happiness of the members of the enterprise, the contribution to the welfare or culture of the community, etc. Yet management has failed if it fails to produce economic results. It has failed if it does not improve or at least maintain the wealth producing capacity of the economic resources entrusted to it... The first definition of management is therefore that it is an economic organ of an industrial society. Every act, every decision, every deliberation of management has as its first dimension an economic dimension.

Of course, we may need to take a very wide view of economics to make this view defensible: the manager of a safety critical software project cannot ignore the safety of human beings, but maybe that can be accounted for by the effect of a failure on the financial position of the company. On the whole I tend to think not...

Project management

A project manager arranges for the following functions to be fulfilled:

- Planning
- Organizing – e.g. defining lines of communication, assigning tasks among team members
- Staffing – e.g. making sure appropriate people are selected for the project
- Monitoring – e.g. measuring the progress of the project
- Controlling – e.g. telling people what to do, keeping the project on track
- Innovating – e.g. making sure appropriate tools are used
- Representing – e.g. liaising with the customer or with other parts of the organisation.

Software project management

A project manager arranges for the following functions to be fulfilled:

- Planning
- Organizing
- Staffing
- Monitoring
- Controlling
- Innovating
- Representing

Slide 83

12.1.1 People management

Normally each employee will have a personal manager (line manager). This may be the same person as the project manager, but more often will not be. Whereas you may have a different project manager for each project, a personal manager (sometimes called line manager) will have a long-term involvement. His/her job is to make sure your career is progressing, that you are getting appropriate training, that your performance is satisfactory, etc. Probably also has a role in deciding what projects to assign you to.

Reasons to have two separate people: there's a conflict of interests, since the project manager has the immediate project's best interests at heart, whereas the personal manager is aiming to further the long-term interests of the company by making best use of people and enabling them to develop their skills. Reasons not to: it's extra overhead, you may end up with a personal manager who doesn't know the employee or their work very well.

Line management

As a software engineer, you probably at any one time have both

- a project manager (determined by what project you're on)
- a line manager (determined by who you are)

Your line manager oversees your career progression, training, overall performance, and probably has a role in assigning you to projects.

These managers may or may not (esp. *matrix management* be the same person. Should they be?

Conflict of interests (so no) ?

Communication problems (so yes) ?

Slide 84

What are the characteristics of a good manager?

Teams

What makes a successful team? Hard to describe, but certainly includes:

- balance of appropriate personal characteristics. For example, the team needs people (probably different people!) who are good at
  - seeing the customer's point of view
  - coming up with ideas
  - finding the flaws in ideas
  - helping the team stay focussed on the task
  - making sure everyone's voice is heard
- balance of appropriate skills for the task
- people who get on with one another
- being part of a basically happy and successful organisation

Success builds on success. For example, the more successful a team the more likely it is to have low turnover, which is likely to lead to further success...

At the lowest level teams normally have between 3 and 8 people: small enough to keep communication problems under control, though of course the size depends on the project: each team should have a fairly clearly defined task, otherwise the burdens of communicating with other teams will be overwhelming. There will usually be a team manager: this may be a member of the team so designated for one project only, or it may be a separate job title for a managerial person. The team manager will have the main responsibility for managing interactions with other teams. On a large project there might be a hierarchy with groups of teams forming sub-projects under a sub-project manager and so on. Managing communications in such a project is very hard, and failure to do so is one of the main reasons why projects fail. There are various approaches to tackling this. At one extreme, a large project may try to keep all interaction between people on different teams to managed pathways: you talk to your team manager who talks to your sub-project manager who talks to another sub-project manager who talks to a team manager on that sub-project who... At the other extreme, a project may try to keep all possible channels of communication open, for example by having regular project meetings to which everyone involved in the project is invited. If we knew that one of these worked better than the other, we might have less of a software crisis than we do!

Individual roles within a team may be well-defined or not...

There's also a wider concept of teamwork involving a whole company "pulling together" – this is more nebulous.

Successful teams

What makes a successful team?

- balance of appropriate personal and interpersonal skills
- balance of appropriate technical skills for the task
- people who get on with one another
- being part of a basically happy and successful organisation
- appropriate size (3 - 8 ?)
- good team manager
- low turnover
- success!!

Slide 85

12.2 Leadership

Leadership is often confused with management (not least by managers!) but the functions are essentially different, and need not be taken by the same people; indeed it's possible to argue that they shouldn't be. (I've several times observed apparently successful situations where the X is a leader whilst the Deputy X is a manager, and perhaps this is an organizational pattern?)

So what's the difference? It's hard to characterise, but the best I've come up with is that a manager is a smoother-down and a leader is a shaker-up.

A leader is a person who knows where they're going and has ideas about how to get there: a vision person. They may concentrate on the most important goals to such an extent that they lose sight of more peripheral issues altogether. They aren't easily sidetracked, or indeed diverted from their course in any way. As you see, any of these characteristics can be strengths or weaknesses, and which one is in any circumstance may well depend on whether the observer agrees with the leader or not!

Management skills can (to some extent at least) be learned, and someone who's a good manager will probably be competent at managing any task which they understand well enough. With leadership skills it's not so clear that they can be learnt – and it's certainly possible for someone to be a good leader for one task, and quite unable to lead another. On the other hand, it's not essential for leadership to come always from the same person, even within a task. Since a large part of good management is about understanding lines of communication, I don't think it's possible to distribute management in the same way. So one might argue (I think I would) that in making appointments into a hierarchy one should look primarily for good management skills, and let leadership arise where it will.

Leadership

A leader is

- one who makes things happen
- a shaker-up
- a vision person
- someone who sees what's important
- not easily diverted from his/her course

Can someone simultaneously lead and manage?

Slide 86

What makes a good leader?
Do you think you are more likely to make a good manager, or a good leader?

12.3 Organisations

There are probably as many types of organisations as there are organisations! Moreover successful organisations vary so wildly that it doesn't seem sensible to lay down rules here about what's right. The only common factors I can think of are the rather obvious:

- a successful organisation will make its customers happy
- a successful organisation will make its employees happy

I guess it has to make its shareholders happy too, but maybe that's a consequence!

Here are some things that differ, in question form since they come from a piece of coursework last year that asked people to consider what kind of organisation they'd like to work for:

- Some organisations tend to have separate "analysts", "designers" and "programmers", so that you would be working primarily on one part of the life-cycle. Others tend to employ people (often called "software engineers" or "analyst/programmers") who will work on various stages of the life-cycle, often following a project through from an early stage to completion.

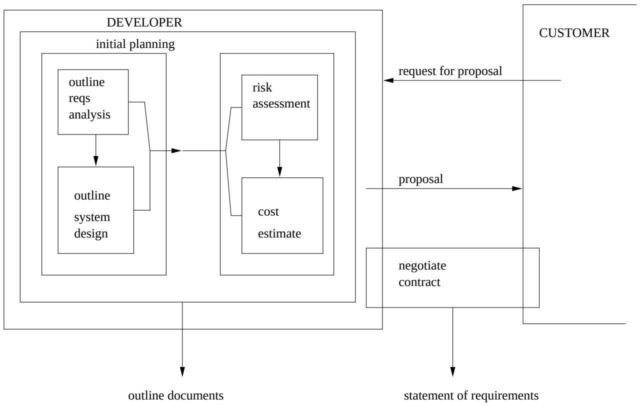
- Would you prefer to work in a situation where you will have a great deal of autonomy and responsibility for dealing with customers directly; or would you prefer to concentrate on solving well-defined technical problems, leaving interfacing with the customer to others?
- Promotion: in some organisations being promoted necessarily involves taking on more management responsibilities. In others, there is a "technical track" such that it is possible to be promoted into a position as "technical expert" available for consultation in your area of expertise, without moving into management.
- Most management structures will include the idea that a given team has a "team leader". In some cases, the team leader will be someone with a different job title from the rest of the team and particular experience or expertise in management; that is, you have to have a formal promotion before you can be a team leader, and from then on you will (almost) always lead the team on which you work. In other organisations, anyone, however recently appointed, can act as team leader for a project. In the latter case the decision might be made on the basis of such factors as the person's experience in the technical or business field. Which do you think is best, or which would you prefer?
- Would you prefer to work on a large project or a small one?
- Would you prefer to be involved in development of new products or in maintenance of existing products?
- Should your company frequently employ contractors to balance out peaks and troughs of work? What about specialist consultants? What should its attitude be towards subcontracting out parts of its work? Which parts? (e.g. should it contract out the boring routine parts, or should it contract out the difficult parts to specialists?)
- Would you prefer to be involved with products which are well-understood, where the challenges are to do with producing exactly the right variant of a basically well-understood thing? Or would you prefer to work on something technically innovative, where the challenge is to make the thing work at all, and where this may prove impossible?
- Would you prefer to work in an organisation which has a certification such as BS5750/ISO9001, or in one which does not?
- Would you prefer to work in the IT division of a larger organisation, satisfying the IT needs of that organisation? Or would you prefer to work for a specialist software company that bids for contracts from outside? Or for one whose business is selling "off the shelf" software?

13 Procedural structures

So far we've discussed what quality is – what we're trying to achieve – and what effects the social structures of the organisation may have on it. What about the more tangible aspects of the process the organisation follows to complete a project? And what are the interactions between process and society?

We've considered, briefly, how requirements analysis, analysis, design, construction and testing are done, mostly from a technical viewpoint. What about the management activities that surround this technical process?

Planning/estimation/bidding



Aim: convince the customer that the work is in safe hands with you. Must also be useful to you, so shouldn't be fiction!

Slide 87

Here's one possible outline of project plan, from OU M355 Unit 13. The company's QMS (Quality Management System) defines what they actually do.

1. The Project Introduction
   - project plan outline
   - any assumptions made when plan was produced
   - project schedule
2. The Project Description
   - project products
   - acceptance criteria for products
   - customer involvement in development tasks
3. The Phase Plan
   - project phases and associated tasks
   - project milestones and associated dates
4. The Organization Plan
   - staff and their responsibilities

5. The Support Requirements Specification
   - software development facilities required
   - hardware development facilities required
   - support required for the software development team
6. The Development Process Specification
   - development techniques and standards to be used
7. The Change Control Plan
   - method for handling changes during the project
8. The Documentation Plan
   - methods of document production
   - document format
9. Appendices
   - test plan
   - quality plan
   - example development notations

Notice that the project plan is a customer-readable document. This has several implications.

- It must be comprehensible by the customer. (Hence, for example, the possible inclusion of example development notations in an appendix. The developers are presumably familiar with the notations they use, but the customer will probably not be.)
- It must show the developers in a good light: any problems you highlight will come with suggested solutions, unless you can argue that none exist. It will not mention risks such as "we don't have the expertise to do this"!
- It must not include business-sensitive information you don't want the customer to know, e.g. cost estimate, as distinct from price estimate – you don't tell the customer your profit margin...

Different criteria might apply if you were planning an in-house project – but increasingly IT divisions bid against outside suppliers, so the gap is not as clear as it was.

Scheduling

In PM you've seen such things as:

- a phase plan
- a work breakdown structure
- a PERT network
- what a critical path is

and some information on how one makes time estimates for a project.

Slide 88

Project tracking

As the project goes on, estimates have to be revised in the light of progress, unforeseen problems etc. Typically a large project will replan once a week.

Depending on the circumstances slippages may mean

- reallocating resource (but see The Mythical Man Month)
- cutting functionality
- asking the customer for more money
- losing profit

Tell them the most important metric is what proportion of the project staff's time is spent explaining to the customer why the project is late Anonymous software engineer.

But why do projects always slip?

And how do we track the project?

Slide 89

Why do projects always slip?

Required reading:

The Rational Planning of (Software) Projects

by Mark C. Paulk.

This paper discusses the effects of three features of human nature:

- people tend to be risk-averse when there is a potential of loss
- people are unduly optimistic in their plans and forecasts
- people prefer to use intuitive judgement rather than (quantitative) models

It goes on to discuss how a framework like the CMM can help.

Slide 90

Use of metrics

You've encountered metrics before. But how do we decide which metrics are useful? A metric may be a:

- predictor – use one measurement (e.g. number of functions) to predict another (e.g. cost of project). The thing used is a predictor metric.
- result – if a measurement was predicted by a predictor metric, measure its actual value at the end of a project (or phase). Feed the discrepancy back into process improvement and future estimation.

Slide 91

Useful metrics must be...

- Measurable – e.g. not someone's opinion of how complex something was
- Independent – i.e. not something that can be altered without altering the quality of the software
- Accountable – i.e. managers should be able to justify both the use of this metric and the value given to it.
- Precise – i.e. the minimal accuracy of the number must be known.
- MEANINGFUL!! – there must be some reason to believe that numbers for the metric have something to do with software quality!

Slide 92

13.0.1 Independent

You may think that there are no such things as truly independent metrics and, indeed, they are rare. The reason that you seldom encounter independent metrics has to do with *Heisenberg's uncertainty principle (slightly revised).*
Measuring any project parameter and attaching evident significance to it will affect the usefulness of that parameter.
The keyword here is *evident.* If the project members know that you can count the number of smiles on Monday mornings and use that as an indication of project morale, then when they want you to believe that morale is high, they will all start smiling on Monday mornings. If on the other hand you keep to yourself the metrics you are using and the significance you attact to them, their usefulness (if any) will not degrade over time.

DeMarco, T. (1982), *Controlling Software Projects*, Yourdan Press, New York

13.0.2 Accountable

To justify the metric, will have to store:

- exactly what was measured
- by whom
- when
- what values were for other projects

It is essential, and can be hard, to measure the things you want to know. However software metric expert Tom Gilb propounded what Tom DeMarco dubs Gilb's law:

Anything you need to quantify can be measured in some way which is superior to not measuring it at all

Is there a contradiction here?

14 Quality Assurance

Recall that quality, in software, means the conjunction of any good properties a piece of software or its delivery should have in order to please the customer! For example:

- it should meet functional and non-functional requirements (stated or not!)
- it should be delivered on time and to budget
- it should continue to be useful in the face of changing situations and requirements, to the extent that this is necessary. For example, if it is still in use in the year 2000 it should not have the year 2000 problem...

Quality assurance

is the process of

(a) convincing oneself and one's customer that any product delivered will have high quality.

(b) providing a basis for continuous improvement

This normally means examining and monitoring the *process* of developing software to check that the process is likely to produce high quality software.

Various standards exist (ISO9001, TickIt, CMM level n). In some contexts adherence to particular standards or particular certification is required, either legally or contractually.

Slide 93

Quality assurance is the process of convincing oneself and one's customer that any product delivered will have high quality – or at least a guaranteed acceptable level of quality. This normally means examining and monitoring the *process* of developing software to check that the process is likely to produce high quality software. One way of increasing confidence that this is so is to acquire a quality certification such as BS5750/ISO9001 or TickIt. In some contexts adherence to particular standards or particular certification is required, either legally or contractually.

The most important aspect of quality assurance, however, is that by documenting and measuring what is done and how well it works, it provides a basis for continuous improvement of the process of producing software, with the aim of achieving a continuously improving level of quality in the product. If you're aware of what was done differently in a particularly successful project, and you have a way of rolling out changes to a whole organisation, you can spread successful innovations throughout an organisation. Unfortunately, many quality systems actually work against this aim, by making it difficult and bureaucratic to change processes, even where people agree that the change is for the better.

International and US companies often classify the process they follow by claiming to be at "CMM level n" for n between 1 and 5. This refers to the *Capability Maturity Model* developed at CMU's SEI. The stages of CMM are described in today's required reading, namely *The Rational Planning of Software Projects* (p6). In brief:

CMM

The idea is to:

- documenting the way work is performed
- measuring its performance
- using data for controlling the performance
- planning based on historical performance
- training people

Slide 94

CMM levels are:

- Initial
- Repeatable
- Defined
- Managed
- Optimizing

So quality and quality assurance are different but related things. Of course they are essentially bound up with all sorts of other properties of an organisation and the software it produces,

14.0.3 Questions to think about

Q: What difference does the method of QA (or none!) that a company uses make?

- To the customers?
- To the products?
- To the employees?
- To the shareholders?
- To the ability of the company to adapt to changing situations and markets?

Q: What are the positive effects of QA?
Q: What are the negative effects of QA?

15 Reuse

Finally, let's come back to the title of the course, specifically to *components* and review what we've done and what the effect may be. Recall that a component is, essentially, something which it's possible to reuse.

The term reuse covers any situation in which work done for one software project is used to help another. Interpreted broadly this may even include any use of any software library!

Reuse

1. What can we reuse?
2. What are the dangers of ill-thought-out reuse?
3. Why is it hard to achieve high levels of reuse?
4. What are the characteristics of genuinely reusable components?
5. When can we do the extra work that making reusable components involves?
6. What difference does object orientation make?
7. What are the motivations for achieving high levels of reuse?
8. What are the interactions between quality culture and reuse?

Slide 95

What can we reuse?

- Most obviously, code: classes, functions
- Designs, even sections of analysis: remember design patterns
- Documentation, e.g. standard bits of user manuals?
- Tests, e.g. what happens if we run out of memory/disk
- (expertise, tools – may or may not count these as reuse)

Slide 96

Reuse can be achieved in a variety of different ways. Most simply, we can *cut and paste*. Code reuse is just about the only area in which anything else is regularly and consciously done (though Design Patterns may change this).

Code is often split into chunks intended to be self-contained enough for easy reuse and placed in a library – in OO, typically a *class library*. The library may be a commercial product, or an internal resource of varying degrees of organisation and formality.

Note that in OO reusing a component may not mean putting up with its functionality just as it is: a class can be reused by using it as the base class of a more specialised class. (See Open/Closed principle below).

What are the dangers of ill-thought-out reuse?

- It may not work – and if you didn't write it, it's easy for it to be harder to debug it than it would have been to write it in the first place
- It may not do exactly what you want, and you may end up twisting the requirements to fit what's available (but beware: this is the classic excuse for never doing any reuse at all!)
- You may end up with many slightly incompatible versions of the same thing, which can be horribly confusing.
- It may work, but not be properly documented, and you may misunderstand it.
- You may have to import things you don't need along with the functionality you want to reuse, getting software bloat.

Slide 97

Why is it hard to achieve high levels of reuse?

- It's harder to reuse components the bigger they are
- Change may make "reusable" components outdated so they never get reused: any extra effort involved was wasted
- Avoid duplicating OS or library facilities: the easiest "reusable" things to build are useless because the functionality is already conveniently available (buying is almost always better than building)
- Finding the right thing, even when it's there! Once component libraries get bigger than a few hundred things the searching problem is very serious
- NIH syndrome – not invented here! (Justifiable? Whose head is on the block if it doesn't work?)

Slide 98

In the extreme, aiming for high reuse can be a sink for arbitrary amounts of effort, caricatured by Julia Vowler in Computer Weekly (11/1/96):

And if all this doesn't keep you busy this year, then you could always launch a corporate object-oriented strategy, and spend the next five years defining every single business object in the organisation, and coding it into C++. By the time you finish they will all be available from third parties for sixpence each off the Internet.

So, most importantly, high levels of reuse can be achieved in the context of component based development: building things to reuse is a higher risk activity than buying things that somebody else built.

What are the characteristics of genuinely reusable components?

- sufficiently general, in the right way!
- properly documented with specification
- thoroughly tested

Not only does it cost, but it's hard, and if you get it wrong the cost is wasted.

Slide 99

So there is some extra work to be done to make things reusable: e.g., a component has to be documented much more thoroughly if it is to be used later by people outside the project in which it was built, than if the documentation is only intended to be read by people who know a lot about the context and the current project.

When can we do the extra work that making reusable components involves?

Possibly the single main reason why reuse efforts fail is that the answer is "we can't". Here are some options (classification of Henderson-Sellars and Pant, Object World, 93):

- End of project: hard to find resources?
- G-C1 model: before delivery, so customer pays
- Two-library model: as time permits (will people remember?)
- Alternative core-technology model: permanent team to do reuse work (but are they the best people to do it, do they understand the components?)

Slide 100

What difference does object orientation make?

- OO encourages the high cohesion/low coupling style that is anyway good practice
- Object oriented analysis concentrates on problem domain objects, which are more stable than the functionality of a particular application.
- Aside from stability, problem domain objects are more likely to occur in several different contexts than functionality on them. A company can develop a collection of *business objects* reflecting the common entities in their own business.
- Inheritance allows developers to use the *Open-Closed* rule – develop classes that are both closed (usable now) and open (extensible/modifiable later). Maintenance done on a base class can automatically affect subclasses.
- Working in terms of objects makes missing but "obviously sensible" functionality more obvious.

Slide 101

One of the main claims for object orientation is that it enables higher levels of reuse than traditional software development methods. People do quote reuse percentages – sometimes as high as 70 percent – but beware, these are practically meaningless without a good definition of what's being measured!

What are the motivations for achieving high levels of reuse?

- saves money ... $$$$ ...
- improved reliability, by cumulative debugging of reused components
- reduced market cycle time (quicker development)
- reduced effort (ditto)
- encourages modularity

[Christmas tree unreproducable here, sorry]

Slide 102

What are the interactions between quality culture and reuse?

- Can reuse help us to achieve high quality software? Perhaps, by improving reliability and encouraging good modular design (see above)
- Can reuse prevent us from achieving high quality software? Perhaps, by encouraging us to bend the requirements to fit the available components.
- Can encouraging high quality (how?) enable reuse? High quality components are more likely to be reusable, and if things are routinely well documented and tested the gap between normal and "reusable" components may be less.
- Can encouraging high quality (how?) make reuse difficult? Well, a too-rigid quality assurance program can make reuse difficult to achieve, e.g. because a reused component may not have been built according to the current QA regs: is it OK to reuse it?

Slide 103

Major challenge for quality assurance: allow for component based design in a way which is flexible enough to get the benefits of reuse, but without compromising the quality of the product (too much).