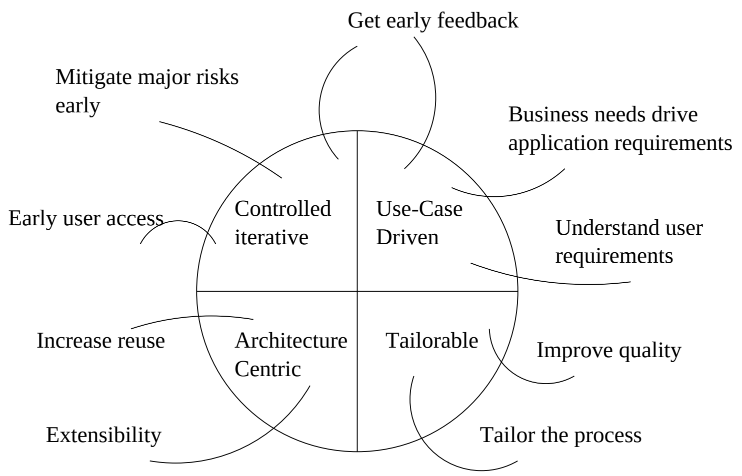
Development methodologies: the Unified Process

In the next section of the course (2 - 3 lectures) we'll consider 2 (or 3) modern development processes, or methodologies, that can be used with UML. The two main ones are the Unified Process (variants are known as Objectory and as the Rational Unified Process) and Extreme Programming; if progress permits we may also consider Catalysis.

Keep in mind the difference between big M Methodology, a way to centralise thinking, versus small m methodology, a tailored body of tools and techniques for solving the problem. The use of design patterns and refactoring fits into the context of each of the processes we'll consider, in different ways: considering how is quite instructive in exposing the differences between the methodologies.

The Unified Process, developed by the main authors of UML deliberately to be usable with UML, is a *use case driven, architecture-centric* methodology making use of *controlled iteration*. In the context of considering the UP we'll cover more of UML: specifically, the aspects of the use case notation that help in use case driven development (Chapter 8 of Using UML) and the further aspects that are required for describing architecture (Chapters 13 and 14).

Characteristics of the Unified Process



(adapted from Rational slide)

Slide 10

Phases of RUP

Inception ends with commitment from the 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.

Each phase may be iterative. Each ends with review.

Slide 11

Controlled iteration

Controlled in the sense that each iteration is a mini-waterfall.

\+ Mitigate risk. E.g. check user requirements, try out technology, practice new techniques in an early iteration to catch errors before main cost of project starts.

− Cost: e.g., repeated testing and documentation. A few projects are so low risk that iteration isn't cost-effective.

In practice, need for rework: essential to allow time for refactoring.

Alternatives:

- waterfall, on projects where risks are well understood, e.g. because it's very similar to a previous project (low risk).
- less controlled iteration, where correctness isn't crucial and requirements are flexible (low risk). (E.g. Linux).

Slide 12

At this point you need to read **Chapter 4 of Using UML** for a little more detail on UP and some (mostly revision) info about development processes in general and how UML fits into them. Moving on from that base:

In UP, use case drive:

- requirements capture
- analysis and design
- testing
- planning: estimation, iteration planning
- traceability
- architecture development

Slide 13

In a bit more detail, what does it mean for UP to be *use case driven*?

- requirements capture: recall use case analysis from SEOC1. (The alternative approaches being...?)
- analysis and design: which focus on how the designed classifiers (subsystems, classes,...) *realise* the use cases. In particular, distinguish use cases from "abuse cases" and "useless cases" to avoid unwanted or harmful functionality and over-design ("feature creep").

  Remember the basic but vital fact that there is no one-one correspondence between use cases and classes! Use cases are tasks; classes are normally based on conceptual concepts in the domain. The normal case is that objects of several classes collaborate to realise one use case, and that objects of one class are involved in several use cases. Working out how objects realise use cases can be seen as *the* main thing that analysis and design must do, in UP. It's not easy: techniques such as CRC cards (SEOC1; Chapter 5 of Using UML) help to work out what should happen, and interaction diagrams help to record these decisions.
- testing: use cases give acceptance/system tests (depending on the level of development of the use cases concerned). Recall the V model: this idea is not new in UP though you might think so to read Rumbaugh's book!
- planning: the development tasks are defined in terms of use cases ("design class roles for this use case", "test this use case" etc.). Estimate also on this basis. (Problems that might result?) Plan iterations to mitigate major risks early and to deliver tangible benefit to the customer in each customer iteration.
- traceability: each decision made is ultimately justified by some sentence in a use case description. Collaborations realise use cases (interactions on the collaboration realise scenarios in the use case).

- architecture development: in the early (elaboration) iterations, a few "architecturally significant" use cases are chosen and an architecture developed that supports them.

At this point you need to read **Chapter 8 of Using UML** for more advanced notation that help you to use use cases for planning without double-counting the same work etc. But pay heed to the warnings about how useful this notation is or isn't: it's one of the most controversial aspects of UML.

Architecture-centric

What is architecture? See Using UML ch1, 4 for starters. (This is a very large question: we are only skimming the surface here.) Quick reminder:

So what is architecture?

Many things to many people.

The way that components work together

More precisely, an architectural decision is a decision which affects how components work together.

Includes decisions about the high level structure of the system – what you probably first think of as "architecture".

Also includes things like,

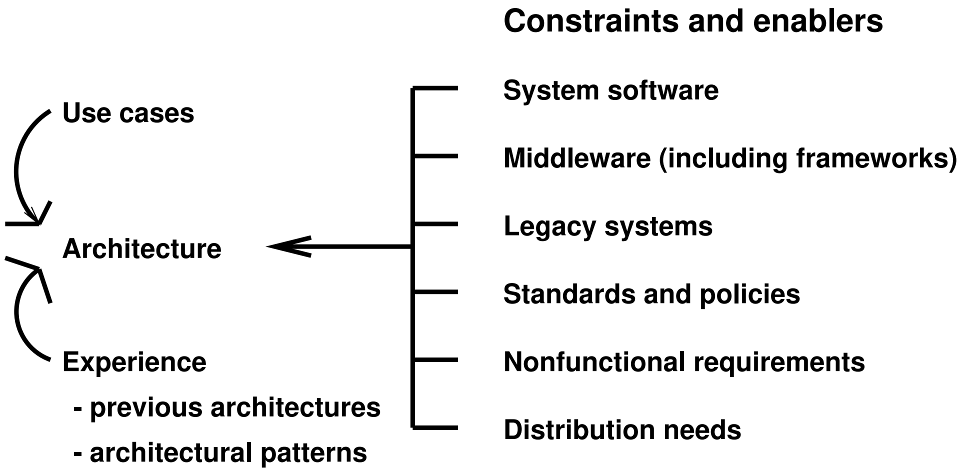
- what language and/or component standard are we using? (C++, Java, CORBA, DCOM, JavaBeans...)
- what conventions do components have about error handling?

Pervasive, hence hard to change. Indeed an alternative definition is "what stays the same" as the system develops and between related systems (Stuart Anderson).

Slide 14

What does it mean to say that UP is *architecture centric*? One thing is that we think about architecture up front and explicitly. A major purpose of the UP's elaboration phase is to identify a good, stable architecture for the product. This normally involves building a "small skinny" system with the basic functionality of several "architecturally significant" use cases, in order to validate the basic architecture of the system. This system is known as the architecture baseline.

Influences on architecture in UP



(diagram adapted from Jacobson et al., The Unified Process)

Slide 15

Next we will very fleetingly consider some examples of architectural patterns (or architectural styles: terminologies differ). Deciding which one or more of these to use is one of the things that will be done in UP's elaboration phase in order to build the architecture baseline. Note that several might be used together: for example, it would be quite usual to have a system with an overall 3-layer architecture (using Layers) where one of those layers was itself a largish subsystem built using Pipes and Filters.

Architectural patterns: "from mud to structure"

Layers

Here is an extremely abbreviated summary of Buschmann et al. p31–51.

Abstract

The Layers architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

Example

Networking protocols: OSI 7-layer model is the classic example. (Well known, but not typical: most real examples have fewer layers. Three is classic.)

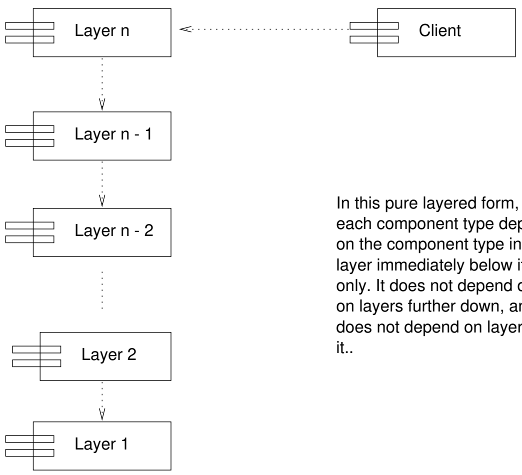
OSI stack

Application	Layer 7	Provides miscellaneous protocols for common activities
Presentation	Layer 6	Structures information and attaches semantics
Session	Layer 5	Provides dialogue control and synchronisation facilities
Transport	Layer 4	Breaks messages into packets and guarantees delivery
Network	Layer 3	Selects a route from sender to receiver
Data Link	Layer 2	Detects and corrects errors in sequence
Physical	Layer 1	Transmits bits: velocity, bit-code, connection etc.

(after Buschmann et al p31)

Slide 16

"Pure" Layers



In this pure layered form, each component type depends on the component type in the layer immediately below it only. It does not depend directly on layers further down, and it does not depend on layers above it..

Slide 17

Forces to balance

- Want to limit effect of late changes: isolate to small no. of components
- Interfaces should be stable (standardisable)
- Parts of the system should be exchangeable
- Lower level issues may recur: allow reuse?
- Want to group similar responsibilities to help understandability and hence maintainability
- Will have to decide granularity of components: no obvious "standard"
- Complex components need further decomposition
- Performance issues: crossing component boundaries may cost
- Team development needs clear boundaries

Slide 18

Good and bad consequences of using layers

$\surd$ Reuse of layers

$\surd$ Support for standardisation

$\surd$ Dependencies are kept local

$\surd$ Exchangeability

X Cascades of changing behaviour

X Lower efficiency

X Unnecessary work

X Difficulty of establishing granularity of layers

Slide 19

(Exercise: in reading the paper on the dependency inversion principle, which talks about layers, see how what it says help achieve the good and avoid the bad consequences.)

Mentions of other architectural patterns

- Pipes and Filters

  The Pipes and Filters architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent fulters. Recombining filters allows you to build families of related systems. One of the decisions to be taken is where the control over the dataflow resides: does the receiver call a method of the sender to get its data (pull model), does the sender call a method with void return type (aka a procedure) of the receiver to pass it its data (push model) or is there an active pipe? Pipes and filters is more flexible than Layers in the sense that you can recombine the filters in a different order, add and remove filters easily. Recombining layers of a layered architecture is not normally possible. However, error handling is much easier in a layered system than in a pipe and filter system.

  You might like to consider how error handling can be done in a pipes and filters system. For a start, what architectural decision is shared between Unix programs which may be combined using Unix pipes, to support error handling?

- Blackboard

  Used especially in AI systems. Useful when several components (e.g. implementing different techniques to solve the same problem or part of the same problem) must share information. Information is controlled by the Blackboard component itself; a scheduler controls the invocation of the various knowledge sources that may use and add to the information. It's often useful for the knowledge sources to run in parallel.

- Microkernel

  A technique used especially in operating systems (Mach, Windows NT) in which a microkernel controls resources and communication between bigger internal servers which provide most of the functionality.

You now need to read **Chapters 13 and 14 of Using UML** for some more notation which is useful in describing architectures.

Development methodologies 2: Extreme Programming

And now as they say for something completely different...

Extreme Programming

Extreme Programming (XP) is

“a humanistic discipline of software development, based on values of communication, simplicity, feedback and courage”

People: Kent Beck, Ward Cunningham, Ron Jeffries, Martin Fowler, Erich Gamma...

More info: `www.xprogramming.com`,
Beck “Extreme Programming Explained: Embrace Change”

Slide 20

Risk: The Basic Problem

- schedule slips
- project cancelled
- system goes sour
- defect rate
- business misunderstood
- false feature rich
- staff turnover

“Use XP when requirements are vague or changing”

Slide 21

An underlying idea of XP is that, whereas the traditional heavy-weight “do most of your thinking up-front” methodologies are that way because of the assumption that the cost of coping with a requirements change or correcting a defect rises exponentially through the development lifecycle, it needn't necessarily always be so. *If* it is possible to make the cost of reacting to change stay relatively constant through the lifecycle, *then* it is possible to be much more flexible.

Keeping that cost down is partly luck (i.e. being in the kind of project where it's possible to do so) and partly judgement (e.g., following those of XP's practices, like refactoring, which help to make it so).

The scope of situations in which XP is appropriate is somewhat controversial. At the extremes:

- there are documentated cases where it has worked well for development in-house of custom software for a given organisation (e.g. Chrysler).
- it seems clear to me that it wouldn't work for Microsoft: big releases are an essential part of the business; even the frequency of updates they do use annoys people.

XP classification of software development activities

- coding
- testing
- listening
- designing

Illuminating exercise: map these onto "standard" activities and contemplate implications of differences.

Slide 22

XP Practices

The Planning Game
Small releases
Metaphor
Simple design
Testing
Refactoring
Pair programming
Collective ownership
Continuous integration
40-hour week
On-site customer
Coding standards

Slide 23

Going into just a little more detail about the key ideas of each of the practices that makes up XP:

The Planning Game Defined responsibilities: e.g. "business" to make decisions about scope, priority, composition of releases, dates of release; "technical" to make decisions about estimates, consequences of the business decisions, process, detailed scheduling. Planning is done jointly but part of the idea is to avoid common problems like

- the customer being allowed to say "you MUST deliver by this date and it MUST include all these features" when it's impossible,
- the developers deciding that such and such a feature would be so much better if it did this extra stuff.

The "game" captures ideas like the fact that you can't make a total release in less than the sum of the times it's going to take to do all the bits: that's against the rules.

Small releases Release as frequently as is possible whilst still adding some business value in each release. This ensures that you get feedback as soon as possible and lets the customer have the most essential functionality asap. (May be talking about every week to every month – outside XP each 6 months would be more usual even in an iterative project, longer not uncommon.)

Metaphor Is basically XP's word for part of what other people call architecture – it avoids the word architecture to emphasise that it doesn't *just* mean the overall structure of the system. "Metaphor" is intended to suggest an overarching coherence, easily communicated.

Simple design Motto: *do the simplest thing that could possibly work.* Don't design for tomorrow: you might not need it.

Testing Test everything that could break. Programmers write unit tests using a good automated testing framework (e.g. JUnit) to minimise the effort of writing running and checking tests. Customers, with developer help, write functional tests.

Refactoring As we discussed before: but here refactoring is especially vital because of the way XP dives almost straight into coding. Later redesign is vital. A maxim for not getting buried in refactoring is "Three strikes and you refactor": For example, consider removing code duplication.

1. The first time you need some piece of code you just write it.
2. The second time, you curse but probably duplicate it anyway.
3. The third time, you refactor and use the shared code.

i.e. do refactorings that you *know* are beneficial

(NB you have to know about the duplication and have "permission" to fix it... ownership in common)

Pair programming One of the most distinctive things about XP is its insistence that all production code is written by two people at one machine. You pair with different people on the team and take each role at different times.

> There are two roles in each pair. One partner, the one with the keyboard and the mouse, is thinking about the best way to implement this method right here. The other partner is thinking more strategically about:
>
> - Is this whole approach going to work?
> - What are some other test cases that might not work yet?
> - Is there some way to simplify the whole system so the current problem just disappears?

The idea, of course, is that the advantages of getting things done better outweighs the cost of having two people doing it. This is as aspect of XP that many organisations find hard to stomach, but the XP experts insist that it's vital.

Collective ownership i.e. you don't have "your modules" which no-one else is allowed to touch. If any pair sees a way to improve the design of the whole system they don't need anyone else's permission to go ahead and make all the necessary changes. Of course a good configuration management tool is vital.

Continuous integration Code is integrated and tested at most a few hours or one day after being written. E.g. when a pair wants to checkpoint they go to an integration machine, integrate and fix any bugs against the latest full build, add their changes to the central CM database.

40 hour week i.e. not 60! People need to be rested to work effectively in the way XP prescribes. There might be a week coming up to deadlines when people had to work more than this, but there shouldn't be two consecutive such weeks.

On-site customer This is another common sticking point. The idea is that a customer – someone capable of making the business's decisions in the planning game – sits with the development team (maybe doing their normal work when not needed to interact with the development team), always ready to clarify, write functional tests, make small-scale priority and scope decisions.

Coding standards The whole team adheres to a single set of conventions about how code is written (in order to make pair programming and collective ownership work).