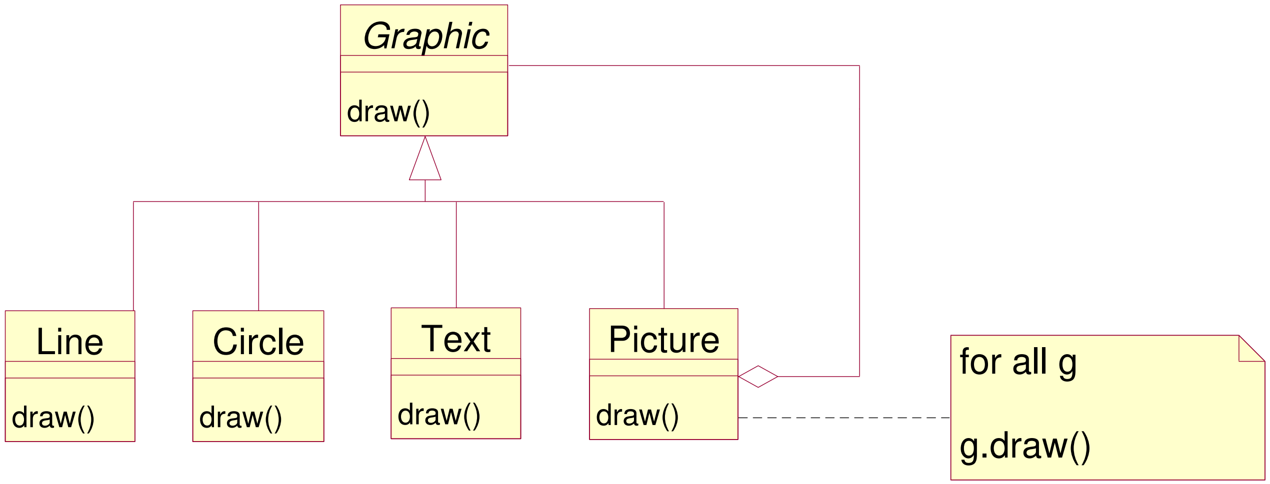
Familiar way to do this kind of task in ML

```
datatype graphicsElement = Line
                         | Text
                         | Circle
                         | Picture of graphicsElement list

fun draw Line = (* whatever *)
  | draw Text = (* whatever *)
  | draw Circle = (* whatever *)
  | draw Picture l = (map draw l; ())
```

Example: a Composite Graphics class



Reading and revision

Revision:

I will assume that you are familiar with the material of Chapters 1-4,5,7,9 and 11 of Using UML, from ProgMeth and SEOC1. If there's anything major there that wasn't covered in those courses, please let me know asap! They are currently available from

http://www.dcs.ed.ac.uk/home/pxs/Book/

but this source will go away when the book is published, so if you want to print copies for revision do it this month.

New stuff: (from *Using UML*)

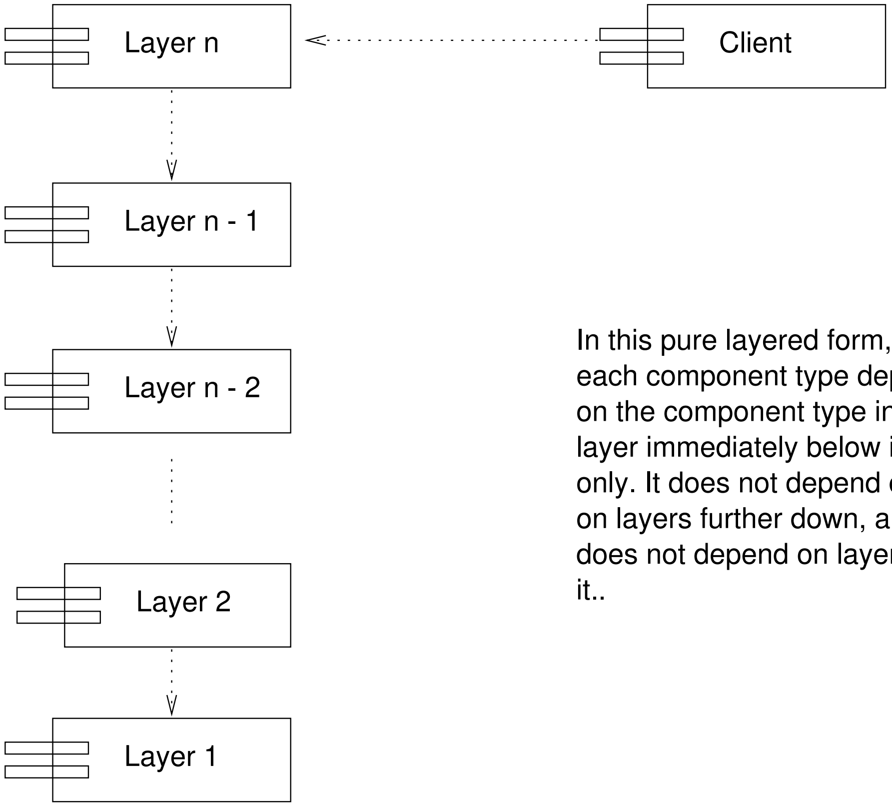
Book reading:
Section 18.2. Draft pages: 295–301. Published pages: 219–223

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)

“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..

Forces to balance

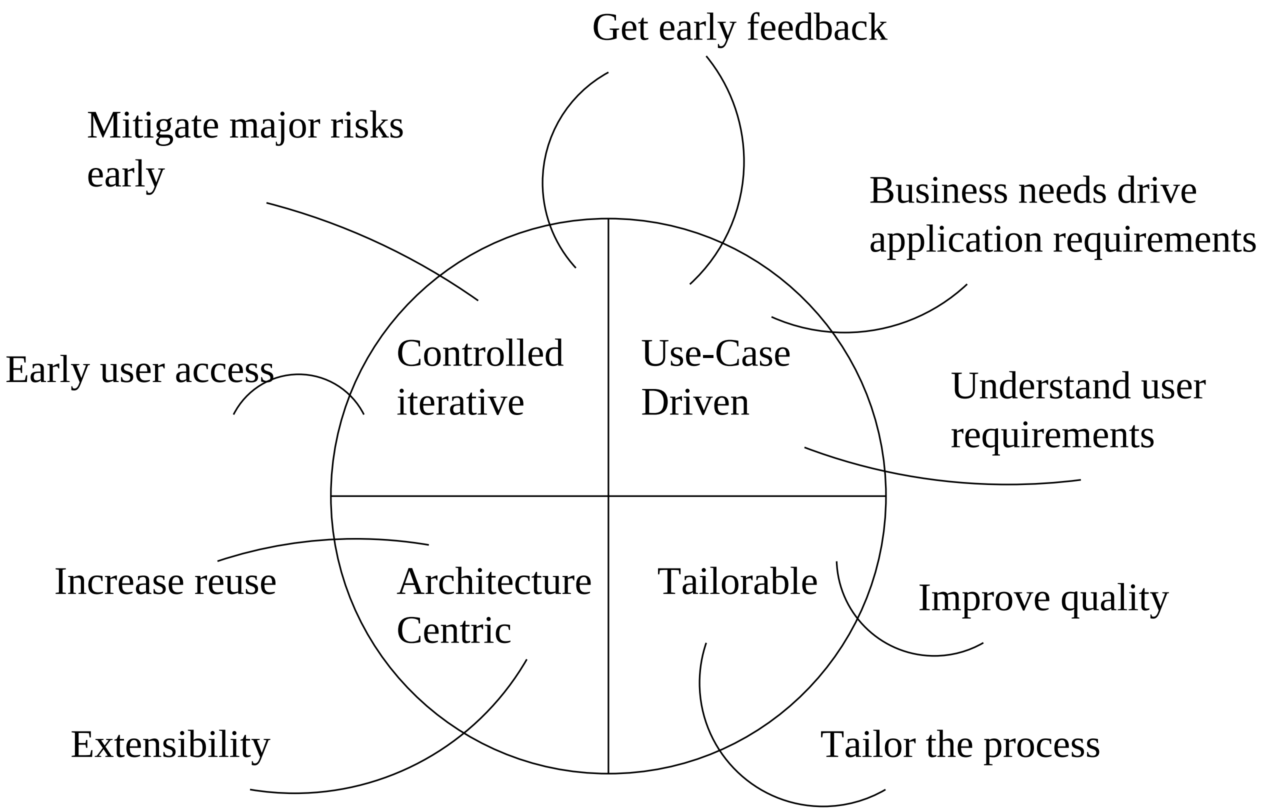
- Want to limit effect of late changes: isolate to small no. of components
- Interfaces should be stable (standardisable)
- Parts of the system should be exchangeable
- Lower level issues may recur: allow reuse?
- Want to group similar responsibilities to help understandability and hence maintainability
- 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

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

Characteristics of Objectory



(adapted from Rational slide)

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).

How can you get (re)use of components?

Today we'll consider some technical factors and various types of components. For example, CBD requires a clear understanding of architecture; and by some definitions parts of the architecture can *be* components.

However many of the most important factors are to do with culture, quality process etc. We'll consider those later in the course.

What are software components?

(We exclude for now other reusable things that in general we might call components.)

Crucial element: PLUGGABILITY

Covers both use and reuse. There is little (but perhaps not no) point in building a system from pluggable parts if you have to build all the parts first and never use them again.

- Want to have components available when we need them
- Want to be able to build big systems with few components
- Want as little as possible to do before and after the plugging: architecture-compatible components, late composition.

Low level interface and communication issues

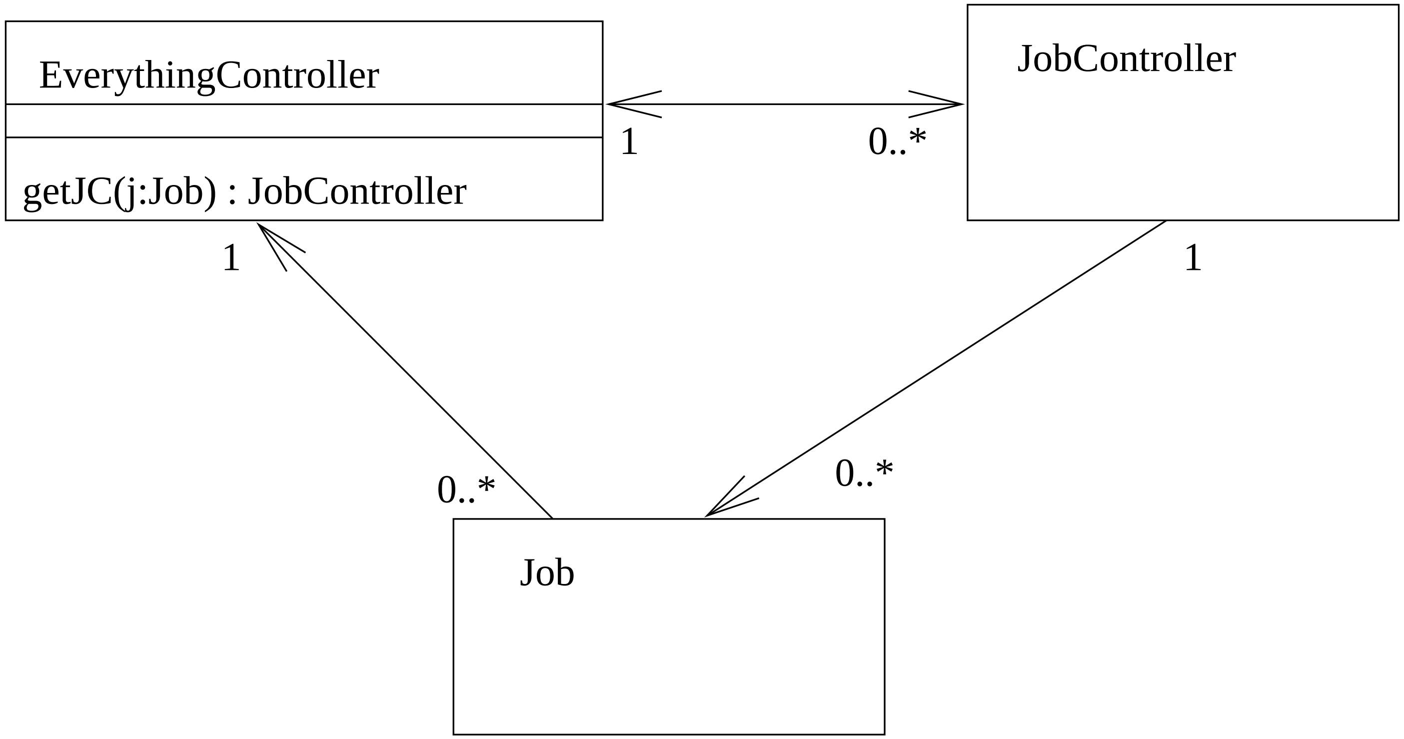
Obviously components have to

- be loosely coupled
- be highly cohesive
- have well-defined interfaces

Interfaces should limit the coupling in the system, too...

©Perdita Stevens 1999

A possible design



Law of Demeter

In response to a message m, an object O should send messages *only* to the following objects:

1. O itself
2. objects which are sent as arguments to the message m
3. objects which O creates as part of its reaction to m
4. objects which are *directly* accessible from O, that is, using values of attributes of O.

The point is, in particular, that it should *not* send messages to objects that it receives as replies to earlier messages.

Frameworks

or, why reuse one component when you can reuse many plus the architecture?

Two main kinds (various terminology exists, we'll use none):

- reuse by inheritance: you are provided with a bunch of classes that work together in a standard way, and you specialise the behaviour of the classes as necessary
- reuse by composition: you are provided with a bunch of classes that work together usefully but normally in a less rigid way, and you use them by composition.

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.

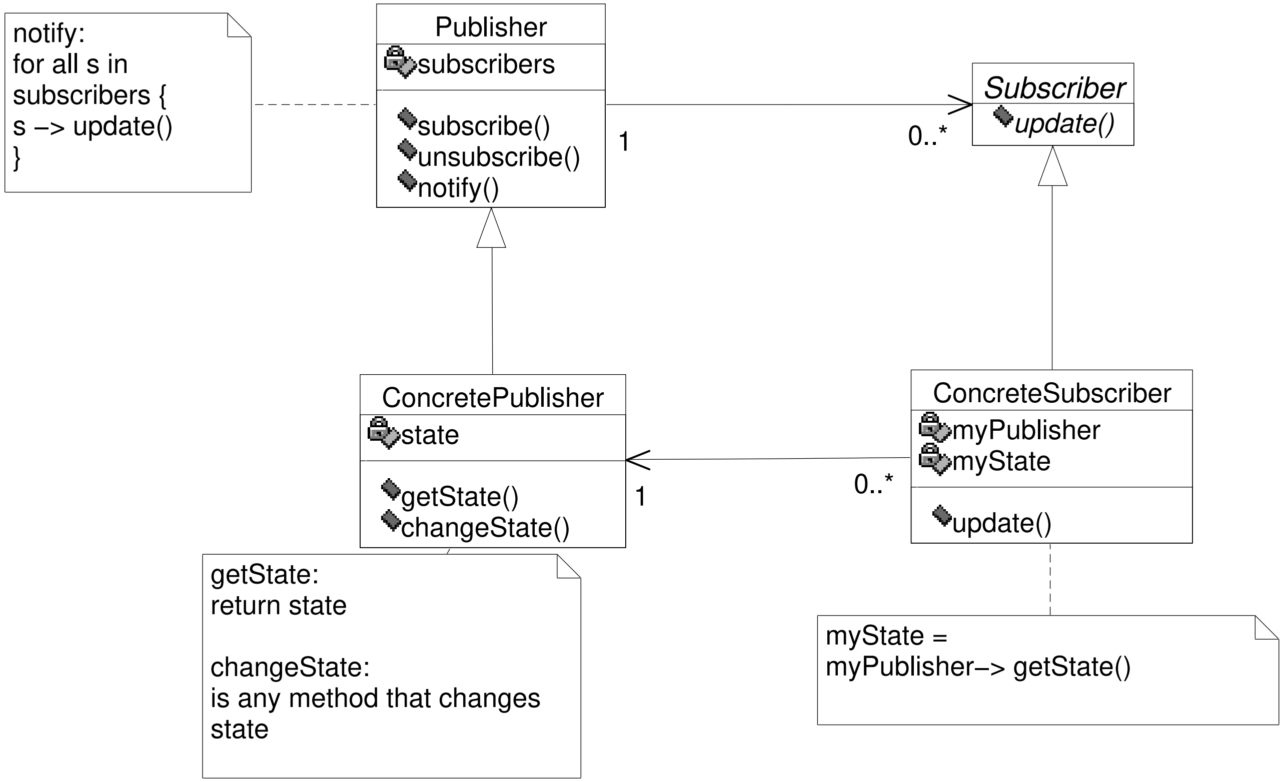
We could introduce explicit dependencies, and have the changing object notify its dependants by sending them messages: but this is inflexible and prevents reuse (e.g. of aCustomer). User interfaces are very prone to change, so internal objects shouldn't know what the user interface is like.

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

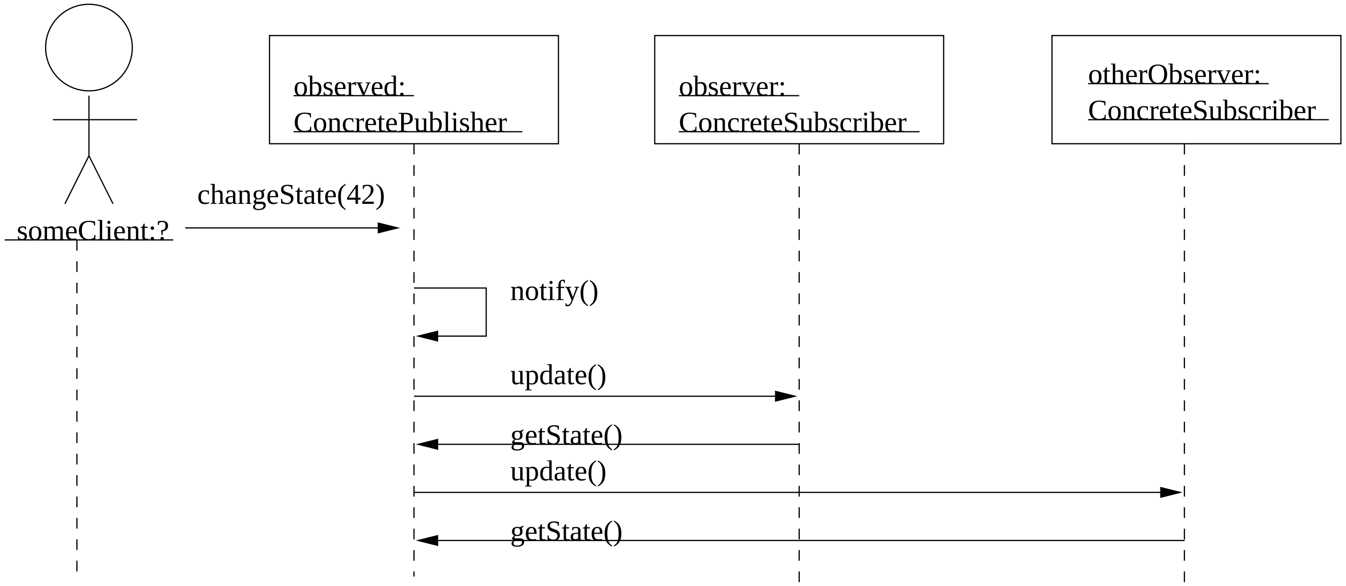
May well want to add or remove dependants at run time.

©Perdita Stevens 1999

Class diagram



Publisher/subscriber interaction



This sequence diagram does not show activations or return arrows.

Exercise (highly recommended): check that you understand the interaction by adding them in.

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.

Quality – if time permits

Key quotes from DeMarco and Lister:

The Flight from Excellence – the market will often accept a level of quality which is so low that achieving only that demoralises the producers.

Quality, far beyond that required by the end user, is a means to higher productivity.

Quality is Free, but... Philip Crosby made essentially this point in his book *Quality is Free* (1979) but this is often misunderstood. In fact:

Quality is free, but only to those who are willing to pay heavily for it.

Power of veto.

©Perdita Stevens 1999

Quality assurance

How can we go about ensuring that products and projects have the necessary quality attributes?

Two basic approaches:

- focus on the product itself (inspect, test, verify, ...)
- focus on the process of producing the product

Each focus has advantages and disadvantages: an effective quality policy needs aspects of both.

©Perdita Stevens 1999

Process focus

Advantages:

- potential to improve *all* products
- possibility of certifying a whole organisation
- some important things, especially planning (and thus time-to-market, cost etc.), are hard to approach in any other way.

Disadvantages:

- done badly, can easily prove very costly with low benefits: easy to spend time and money without improving *any* product

(more later)

Centers of process-focused QA

Possibly directions of influence, each with own philosophy:

Organisation $\longleftrightarrow$ Project $\longleftrightarrow$ Individual

1. Organisation's management decrees, influencing projects whose managers direct individuals into desired behaviour.

   (Underlying philosphy: people try to get out of work and need to be controlled.)

2. Individuals introduce improvements which are rolled up and out to the rest of the project and the rest of the organisation.

   (Underlying philosophy: people want to do good work and need their efforts to be noticed and enabled.)

The DeMarco/Lister passage I read last week assumed the latter philosophy: most QA today seems to be based on the former.

Controlling QA

We'll look at two examples with different aims:

- CMM, the Capability Maturity Model (from Carnegie Mellon's Software Engineering Institute).

  Crucial idea here is that maturity increases: quality planning, control and *improvement* framework.

- ISO9000

  Less emphasis on improvement: quality *control* framework.

CMM

(I think this was mentioned in ProgMeth last year in the context of planning but you didn't discuss it more generally?)

CMM defines 5 levels of capability maturity, plus *key process areas* that have to be addressed to progress and *key practices* that organisations should be carrying out.

No certification as such, though respected consultants etc. may express opinions about the maturity level of an organisation they've audited.

More info from ricis.cl.uh.edu/process_maturity/CMM/TR24/tr24.html

Brief definitions of CMM levels

Initial The software process is characterized as ad hoc, and occasionally even chaotic. Few processes are defined, and success depends on individual effort.

Repeatable Basic project management processes are established to track cost, schedule, and functionality. The necessary process discipline is in place to repeat earlier successes on projects with similar applications.

Defined The software process for both management and engineering activities is documented, standardized, and integrated into a standard software process for the organization. All projects use an approved, tailored version of the organization's standard software process for developing and maintaining software.

Managed Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.

Optimizing Continuous process improvement is enabled by quantitative feedback from the process and from piloting innovative ideas and technologies.

ISO9000

ISO9000 is a series of standards produced by the International Organisation for Standardisation. ISO9001, one of the series, is what you'll most often see cited for software quality. It covers quality assurance of design, development, production, installation and servicing.

Does not tell you exactly what your quality policy should be, e.g. what should be in your Quality Management System. Defines requirements for your own context-sensitive decisions.

It exists under other names too: BS5750 in the UK, AS/Q(?) 9000 in the US.

Certification/registration is done by 3rd party auditors, not by ISO: you choose.

More info: the Web, e.g. www.connect.ab.ca/~praxiom/9001.htm

E.g., to be ISO9001 registered management must:

- Define a quality policy. Your policy should describe your organisation's attitude towards quality.
- Define the organizational structure that you will need in order to manage your quality system.
  - Define quality system responsibilities, give quality system personnel the authority to carry out these responsibilities, and ensure that the interactions between these personnel are clearly specified. Also, make sure that all of this is well documented.
  - Identify and provide the resources that people will need to manage, perform, and verify quality system work.
  - Appoint a senior executive to manage your quality system and give him or her the necessary authority.
- Define a procedure that your senior managers can use to review the effectiveness of your quality system.

Total Quality Management

TQM embodies the idea that improving quality is everyone's job – not just that of the QA department.

Quote from SQM article "TQM OK: BS5750 maybe" by Barry Faith:

TQM is a management philosophy which can be defined as: 'continuously meeting agreed customer requirements, at the lowest cost, by releasing the potential of everyone in your team'.

This definition highlights a little known fact: every member of your team comes, not only with two arms and two legs, but also a free brain! One of the primary tasks of executives and managers is to find ways of releasing the latent potential that everyone in their team has. We all have gold mines between our ears. TQM helps us dig!

TQM is enabled by two complementary actions [in] our organisation:

- a programme of culture change.
- implementing a QM System that specifies and records how your organisation goes about its business.

Crucial ingredients

- Quality planning – how will you ensure that this project delivers a high quality product?
- Quality metrics – what measurements must you make in order to tell whether what you're doing is making the difference you intend?

You covered these in some detail in ProgMeth: good idea to revise that note.

Crucial extra ingredient:

- Quality improvement – what can you learn from this project to help you plan and run the next one better?

Even very simple techniques like routinely having a "post mortem" meeting can help a lot.

©Perdita Stevens 1999

What is configuration management?

CM (or SCM) is:

- not just version control of source files.
- one of the most important aspects of process which a QMS may prescribe.
- essential to traceability, problem-tracking, making changes reliably: answering questions like:
  - which customers are affected by that bug?
  - if this team makes this change, who needs to be consulted or notified?
  - after this change which system tests do I have to rerun?

  (Note overlap with requirements management.)
- particularly important when there are multiple versions of a product (i.e. nearly always) and/or multiple teams

“Looking after what you’ve got so far” (Geoff Cozens)

Version control

Basic ingredient of configuration management: important even on tiny projects and to individuals.

"Check in" changes with comments on what changed.

Tools such as SCCS, RCS, CVS, ... allow you to revisit previous versions, check what changes have been made, view change history, etc.

Extremely useful for backtracking when something goes wrong; means you can try something out knowing you can easily go back.

Emacs vc-mode makes this very easy: see VC submenu of Tools menu.

Strongly recommend you use VC in CS4 projects, for report, code, tests...

Personally I version control just about everything that might change – source code, tests, test results, papers, book chapters, CV, .emacs, ... and it has saved me considerable hassle many times.

Low overhead, high value.

Beyond versions of individual files

make and similar tools specify many practicalities of how to build a system out of its components.

However common problem is incompatible versions: part of CM's job is to record what works with what.

NB versions of compilers, linkers, system libraries etc. are just as crucial as versions of source, documents etc.

Common supporting practice: a component development folder collecting requirements, design notes, source, unit tests, test results, notes, problem reports, etc. pertaining to one module.

CM and QA

QA requires CM:

- CM practices have to be "defined" to reach CMM Level 3.
- ISO9000 doesn't say much explicitly about CM but does require e.g. being able to identify what versions of what components appear in what product – and in practice you won't be able to provide the proof of following good process without it.

CM supports QA:

- enables efficient review and audit: routine tracing work can have automated support
- if you can prove that something (e.g. a test, a review) has been done before – using CM – don't need to duplicate it.

Implementing CM: organisational issues

Tool support is essential to get full benefit – but many companies still use nothing more than version control (or not even that!)

No point in trying to introduce CM unless VC is already in place – walk before running.

Buy or build? issue.

Difficulty in identifying requirements: levels in the company have diverse requirements.

Organisational culture interacts of course...essential not to lose sight of aim to *reduce* effort overall and to get buy-in.

Evaluate tools: try out on pilot project: roll out to organisation: REPEAT.

Daily Build and Smoke Test

Good configuration management is one way of controlling *integration risk* – the risk that the components they don't work together properly when you integrate them – because it helps track the way in which things change and make sure the communications happen.

One way of verifying that certain such errors are absent is to *look for them early and often:* in the extreme, some projects especially at Microsoft every day:

- build (get out of CM, compile, link)
- "smoke test": run a suite of simple tests looking for serious errors.

Advantages of daily build and smoke test

- minimise integration risk
- reduce risk of low quality
- support easy defect diagnosis
- improve morale
- ... ?

Risks of daily build and smoke test

Cost: as usual, managing a risk has an associated cost and whether it's worthwhile depends on the magnitude of the risk in this project's case

Confusion: developers are trying to "follow a moving target"; they must update their code to a changed interface immediately.

Complacency: may mistake passing the DB&ST for being bug-free, since if something is hard to achieve it's difficult to remember that it may not be everything (cf ML "if it compiles it must be right")

Difficulty of making major changes: if the norm is to have a working version every day, there's a strong disincentive to consider major architectural changes. (But this is ambivalent.)

Coplien's pattern Named Stable Bases

See www.bell-labs.com/ cope/Patterns/Process/ for details.

Problem is how often to integrate.

Forces are basically the advantages and disadvantages of DB&ST: they pull you towards and against integrating daily, so what do you do?

Solution "Stabilise system interfaces – the architecture – no more frequently than once a week. Other software can be changed (and even integrated) more frequently."

May want to combine approaches: e.g. from an ATT project: nightly build (only), weekly build with sanity check, biweekly more thoroughly tested build.

Defect tracking

When a problem is reported (from a customer or from an internal customer) essential to make sure it gets dealt with in a timely fashion even when many people and stages are involved.

Bug reports will be categorised (how severe? really a bug, a misunderstanding, a change request?), analysed, assigned to someone to act on, acted on. The bug reporter will need to be informed.

You need to keep the information to avoid doing all the work again next time someone reports the same bug (even to a different person).

Tools exist: probably best overall is GNU's GNATS which also has the advantage of being free.

We've recently started using GNATS in the department to track defects in our systems (starting with mail to faults).

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?

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!)

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.

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.

©Perdita Stevens 1999

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?)

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.

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.

What is a formal technical review?

A review or inspection is a meeting which more or less systematically attempts to find problems with some product of the development process, e.g. code, design.

More systematically seems to work better – don't get sidetracked.

Thus FTR will normally have:

- members who are peers
- designated roles: Moderator, Scribe, Author
- a time limit
- a pre-defined agenda
- a set way to record defects
- no discussion on how to fix defects
- well-defined followup procedure with time limit.

Controversy 1

Some (notably Tom Gilb with Inspection Method) emphasise

- measurement, e.g. of how much time everyone spends, how many defects are found;
- formality, e.g. with written rules defining what a defect is.
- entry and exit criteria: what must be true before an artefact can be reviewed, and what must be true after the review is finished?
- author not allowed to defend own work: only there th answer questions.

Reference: Tob Gilb and Dorothy Graham's book "Software Inspections", Addison Wesley 1993

Controversy 2

But this can take *enormous* amounts of time, not always good value. Others use a more stream-lined approach.

Harmon at IBM, 1991 considered their formal inspections over-attended and over-documented; "interfering with the natural pace of programming". He made inspections leaner, e.g. got programmers to present their own code.

Success: systems produced on time and still with few defects.

But at the time IBM was under contract to the FAA, which held IBM in breach of contract because they'd changed the agreed process.

IBM apologised and Harmon was reprimanded.

(Story from "Software Runaways", Robert Glass, pp75 -77)

Controversy 3

In the middle, many companies use some ideas from FTRs informally: e.g.

> [we do reviews] with whiteboard, for the really difficult bits, together with report forms and signoff. All new code and code fixes get reviewed by someone else - usually a peer - before commitment. [...]
>
> Takes ages, and is usually worth it.

(personal communication from s/w eng in generalist IT consultancy)

As the Moderator of an inspection, how can you influence its success?

Apply rules sensibly: understand their interpretations beforehand.

If it is your job to choose the participants – it often is – choose a mixture of people with appropriate skills and non-conflicting personalitites. Prepare them properly: make sure they understand how the review works and that they should review the document beforehand.

Keep the meeting on track – cut off discussion when necessary.

What might suitable entry criteria for a design document be? And for code?

For example:

Design: should be complete with whatever the QMS requires to be present; for example, this might include UML models of the proposed design. Might require certain sanity checks to have been passed, e.g. the kind of checks Rose can do that all classes mentioned are defined.

Code: should compile without error (and maybe without warnings, depending on language and QMS details), should run (but probably won't be fully tested at this stage).

What advantages do code reviews have over code testing? Disadvantages?

Advantages: can assess aspects of quality of code that aren't testable, e.g. adherence to coding standards, quality of comments. May be able to catch certain errors that are hard to catch by testing; e.g. things that go wrong under wierd circumstances, memory leaks.

Disadvantages: not as repeatable as testing – may be hard to make sure that defects which are found stay fixed. Expensive. May be easier to find trivial violations of standards than actual wrong behaviour.

Why/when might you (not) want a customer representative at a design review?

Reasons for wanting it: defects in design often correspond to misunderstandings of requirements which may require customer input.

Reasons for not wanting it: design isn't the customer's expertise or concern – requirements should be well understood before the design review. You might be more likely to want a customer in a review of a requirements document.

It may well depend on the type of review: formal reviews require participants to be trained in the format of the meeting, and having a customer in such a thing would be likely to be disruptive.

Management

Two types of management are relevant to software:

- people management
- project management

Both types are best seen as enabling activities: a good manager doesn't do the actual work of building software but aims to maintain an environment in which it's possible for people to get on and do so!

Of course many would argue that the bottom line is always financial.

A manager is not the same as a leader...

Software project management

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

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

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?

Why do projects always slip?

Discussed in (optional but rather interesting 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.

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.

For example, by tracking metrics through the project, a manager can follow the progress of the project compared to what was planned.

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!

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) ?

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!!

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?

How to achieve reuse?

Your client is fairly mature in software development (CMM level 4, maybe). It is a service provider, and the provision of its service requires a high level of software support. That is, its core business is "IT intensive" although software development itself is not the organisation's core business.

The software developers are distributed throughout the UK in around 10 software development centres, each of which draws its customers from the rest of the organisation. The sites compete for customers.

The client has made several previous attempts at instituting a reuse programme, with limited success. Significant cross-site reuse has always been rare.

Analysing the problems

1. What can we reuse?
2. Why is it hard to achieve high levels of reuse?
3. What are the dangers of ill-thought-out reuse?
4. What are the characteristics of genuinely reusable components?
5. When can they 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?

©Perdita Stevens 1999

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)

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.

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?)

What are the characteristics of genuinely reusable components?

- architecturally compatible
- 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.

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?)

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.

What are the motivations for achieving high levels of reuse?

- saves money: don't need to pay to develop the reused component
- improved reliability, by cumulative debugging of reused components
- reduced market cycle time (quicker development)
- reduced effort (ditto)
- encourages modularity

Notice that though the most obvious saving is on development of the reused thing, the biggest saving seems to be due to higher quality, hence lower maintenance costs. (Martin Griss, HP)

©Perdita Stevens 1999

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?