

3F6 - Software Engineering and Design

Handout 16

Software Management
With Markup

Edward Rosten

Contents

1. Software Engineering
2. Software Life Cycle
3. Team Organisation
4. Product Development
5. Specification and High Level Design
6. Project Management
7. Quality Control
8. Module Testing
9. Profiling Tools
10. Code Reviews
11. Low Level Documentation
12. Final Testing and Release

Copies of these notes plus additional materials relating to this course can be found at:
<http://mi.eng.cam.ac.uk/~er258/teaching>.

Software Engineering

Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.

Software typically evolves in 4 main stages:

Specification defines the functionality of the software and the constraints on its operation. This will usually involve interaction with the *client* in order to understand the *requirements*.

Implementation produces the software to meet the specification. This is the *design and build* phase.

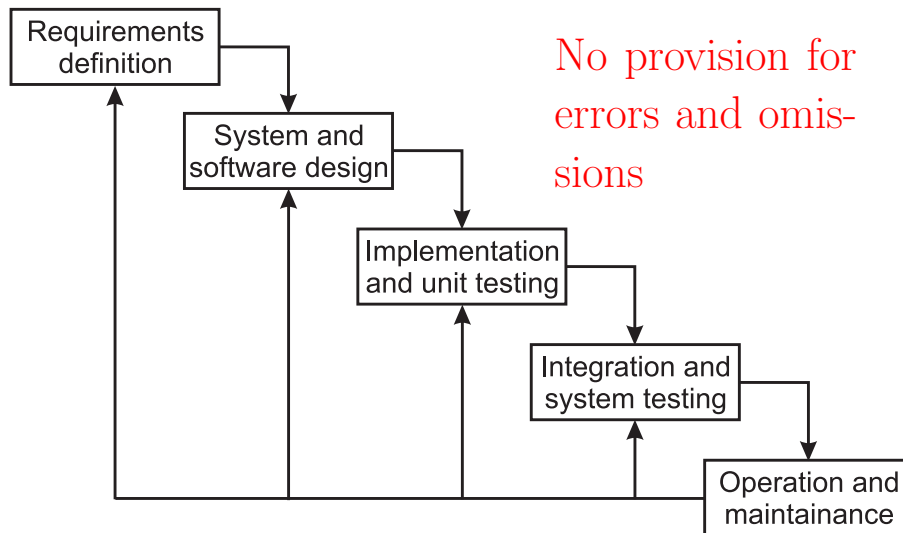
Validation ensures that the software does what it is supposed to do. This will be a continuation of the testing done in the development phase, but will usually be more comprehensive and more formal. The end of this process is marked by the client agreeing contract completion.

Evolution adapts the software to meet changing customer needs. This includes maintenance, enhancements and re-design.

These phases constitute the *software life cycle*.

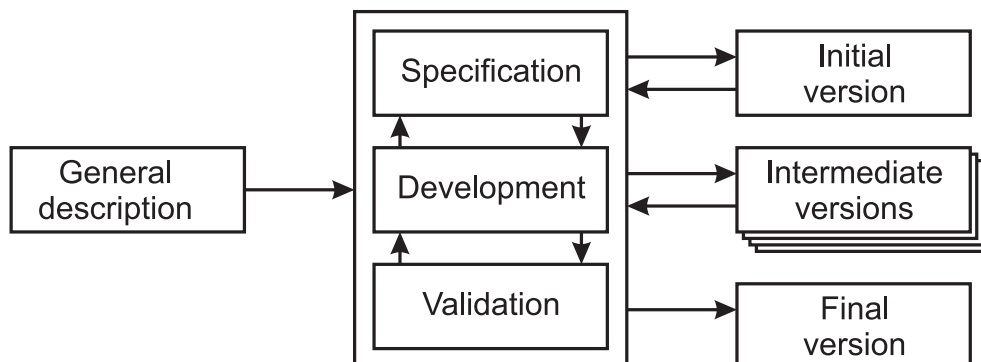
The Software Life Cycle

WaterFall Model



This view of the software life cycle is based on traditional engineering processes. It presents a good abstract view of what is involved but is somewhat idealistic.

Evolutionary Model



Real projects typically involve multiple iterations at each stage. Hence, the evolutionary model is a more pragmatic view of the software life cycle.

What can go wrong?

Project management **Risk high**

- poor cost estimation and scheduling
- Inadequate progress tracking
- Poor man-management

Specifications **Risk high**

- Failing to capture user requirements properly
- Specification was incomplete and/or imprecise

Design **Risk low**

- Poor choice of overall architecture
- Inconsistent module interfaces
- Code unreliable, slow, and crashes frequently.
- Confusing user interface

Testing **Risk medium**

- Inadequate

Documentation **Risk medium**

- Insufficient
- Badly written

End result is software which is delivered late, over-budget, runs too slowly, does not scale with increasing demand, unreliable and very difficult to use.

Costly: Denver International Airport

As part of a new airport for Denver, started in 1989, a fully automatic baggage handling system was specified. This was a large system, with 17 miles of track, 4,000 telecarts, 100 networked computers, 59 bar code readers, and 311 radio receivers. The contract was won by BAE automated systems, at a price of \$193 million, with a opening date of October 1993.

Delays producing the software caused the airport opening to be rescheduled again and again, and when it was finally tested it was a complete failure, with baggage lost, misrouted and damaged. In August 1994 it was decided to build a conventional baggage handling system at a cost of \$51 million.

The airport eventually opened 2 years late. The delays cost the airport \$3.2 billion in interest, operating and redesign costs.

Fatal: London Ambulance Service

To replace the old manual system of despatching ambulances from 999 calls, a new computerised system was commissioned. Consultants estimated that it would cost £1.5 million, and take 18 months. The winning consortium bid only £937,000, and estimated that it would take 6 months. It was 10 months late, and was rushed into service on 26 October 1992, with no proper testing or training. It was taken off-line the next day and they went back to pen and paper. What happened?

- Problems communicating with the vehicle radio systems
- Sometimes too many vehicles were dispatched, or none at all, with no way to check
- Messages scrolled off the top of the screen, and there were no scroll bars
- The whole user interface was poorly designed and confusing
- The system slowed down until it was unusable, and required regular reboots
- Due to a memory leak, the system crashed on 27 October 1992

Types of Software Project

There are many types of software project:

- 1 person making software for personal use.
- 2 or 3 people writing some research code.
- 5-9 people in a small start-up developing the first version of a new product.
- 10-50 people in a small/medium company (SME) maintaining and developing a relatively mature product.
- large software consultancy developing large bespoke software systems for clients e.g. a University financial management system.
- Microsoft, Oracle, Adobe, ...

Good practice varies across these as the emphasis and scale changes.

For the rest of this course, the focus will be on engineering development in the Start-Up → SME range of companies.

Team Organisation

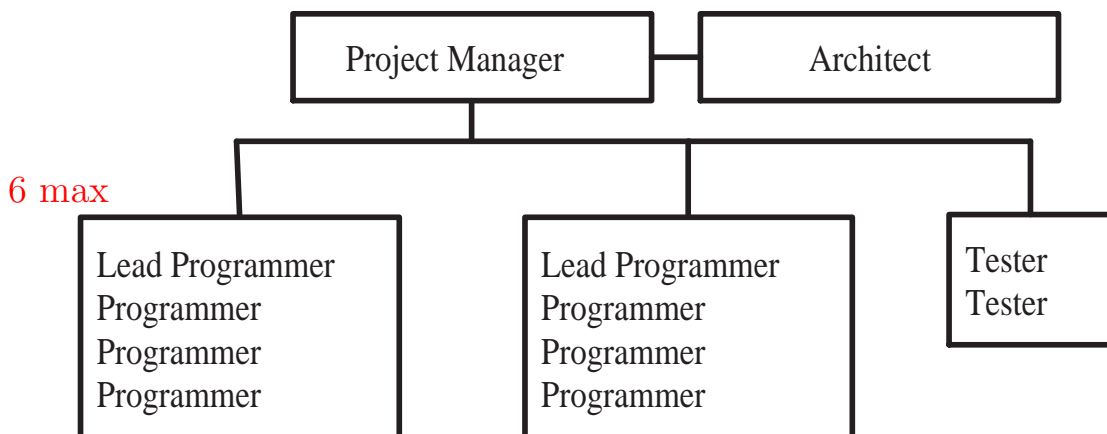
Architect: The principal designer, defines the overall architecture, module structure and all major interfaces, usually also an expert in the associated technology.

Project Manager: Responsible for scheduling the work, tracking progress and ensuring that all of the process steps are properly completed.

Lead Programmer: Leader of a programming team. Will typically spend 30% of his/her time managing the rest of the team.

Programmer: Implements specific modules and often implements module test procedures.

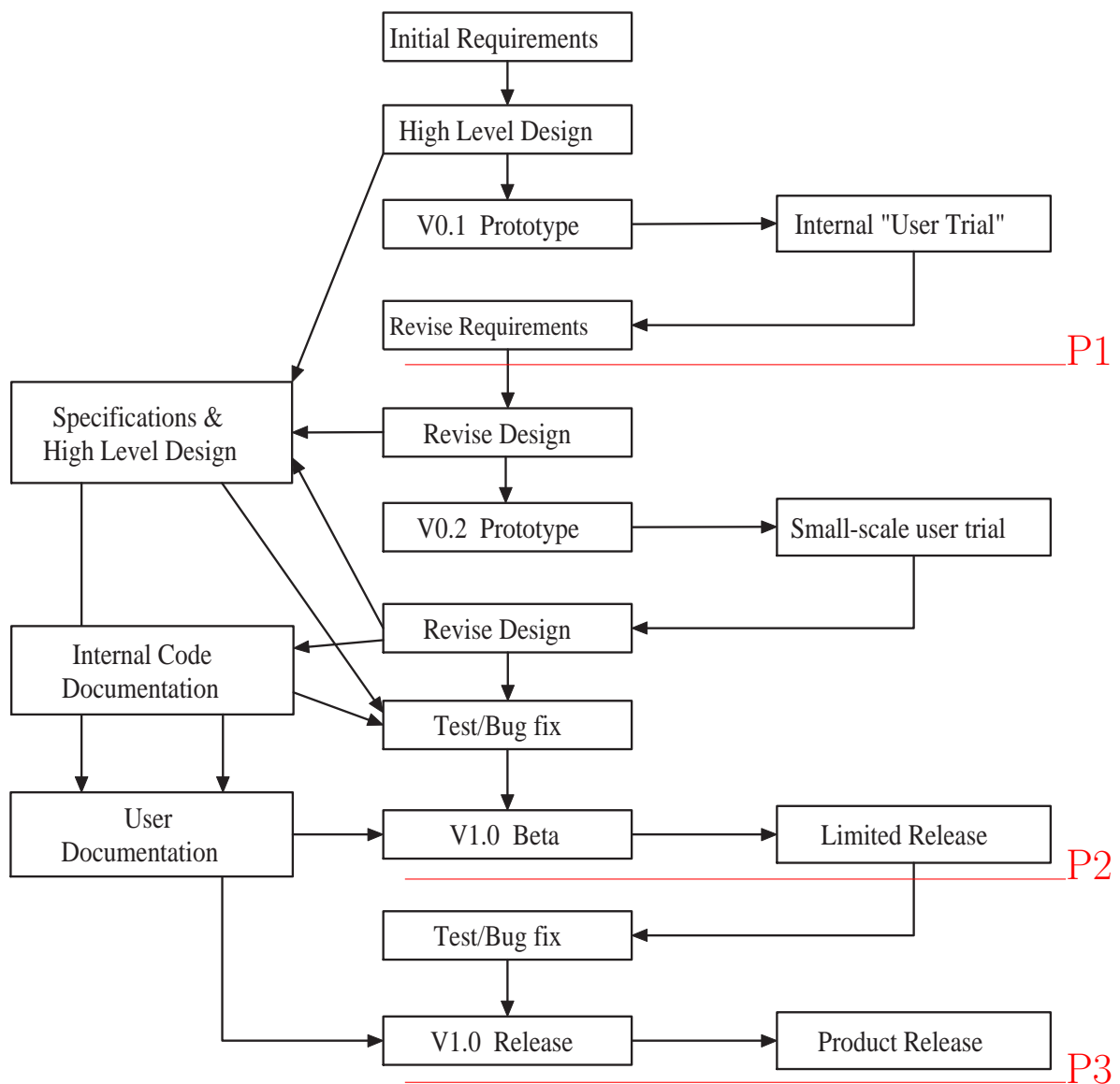
Tester: Designs test and validation procedures for the completed software. Tests are based on initial specification and will focus on the overall product, rather than the individual modules.



When manpower is limited, one individual may perform multiple roles but ideally they should be distinct.

Software Product Development

In the start-up/small company scenario, the focus will be on converting an idea or concept into a working product. The requirements will be loosely known and will evolve over time. The key is rapid prototype development and frequent user testing.



Specification and High Level Design

This is the job of the Architect.

The first step is to produce a high-level “white paper” setting out the vision, the major functionality, the user experience etc. If the software includes a user interface, show mock-ups and list use cases. If there are performance targets, state them. **Gaining Concensus**

This document should be readable by all sections of the company i.e. marketing, sales, technical staff. It should be agreed by senior management before going further.

Program specification and design:

1. divide software into modules (usually compilation units)
2. describe overall function of each module
3. define/document any special algorithms required
4. each module should define one or more class interfaces
5. document public functions/methods in each class

Assign modules to teams. Work with team leaders to assign module and/or class implementations to team members. Sit-in on initial meetings to verify understanding and resolve any issues.

Project Management

GOAL: ONTIME, ON BUDGET

The role of the project manager is: unpopular w. staff but CRUCIAL

1. Identifying required tasks and their dependencies
2. Assigning staff to tasks
3. Estimating task durations
4. Tracking progress through the life-time of the project
5. Advising senior management on progress
6. Rescheduling when necessary
7. Recording task information and completion rates.

(1), (2) and (3) will be in consultation with the Architect and Lead Programmers.

In practice, (3) is the hard part. Staff will consistently underestimate the required time, accumulated historical data generated by (7) provides a very valuable “sanity check”.

Tracking progress (4) should be via weekly meetings between the Project Manager and the Team leaders.

Milestones should be set at key points

- to signal progress to higher management
- to motivate staff

Project Scheduling Tools

Given a list of tasks, predicted durations and staff assignments, project management tools will provide:

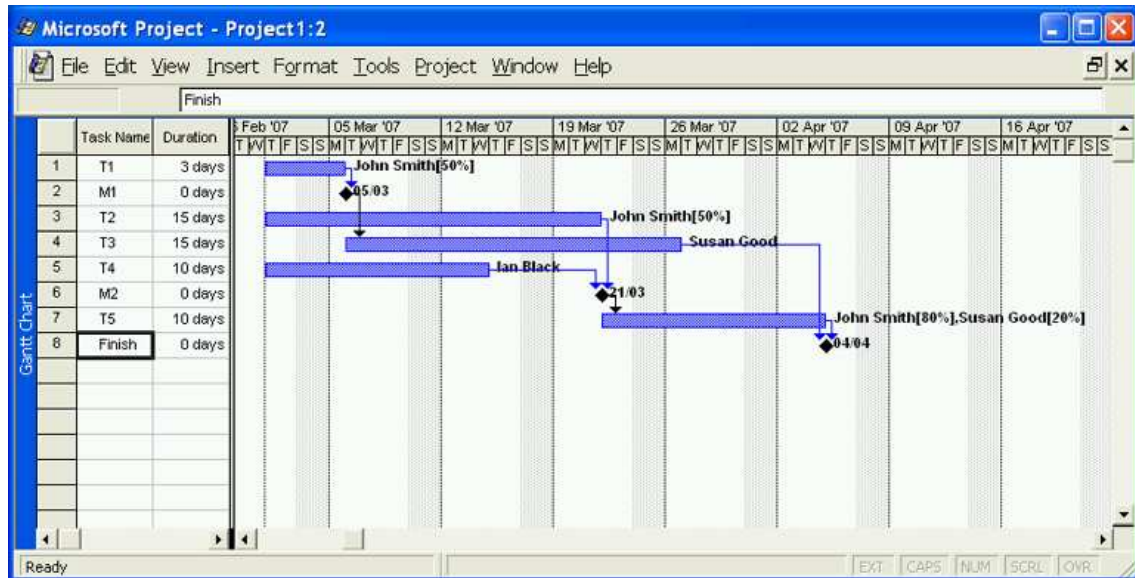
1. **Gantt charts** showing time bars, dependencies and progress.
2. **PERT diagrams** emphasising dependencies and critical paths.
3. Staff loading
4. Simplify rescheduling and performing “What if?” analysis.

Example Suppose that the following tasks have been identified:

Task	Duration (Days)	Who?	Dependencies
T1	3	John Smith[50%]	-
T2	15	John Smith[50%]	-
T3	15	Susan Good[100%]	T1 (M1)
T4	10	Ian Black[100%]	-
T5	10	JS[80%],SG[20%]	T2, T4 (M2)

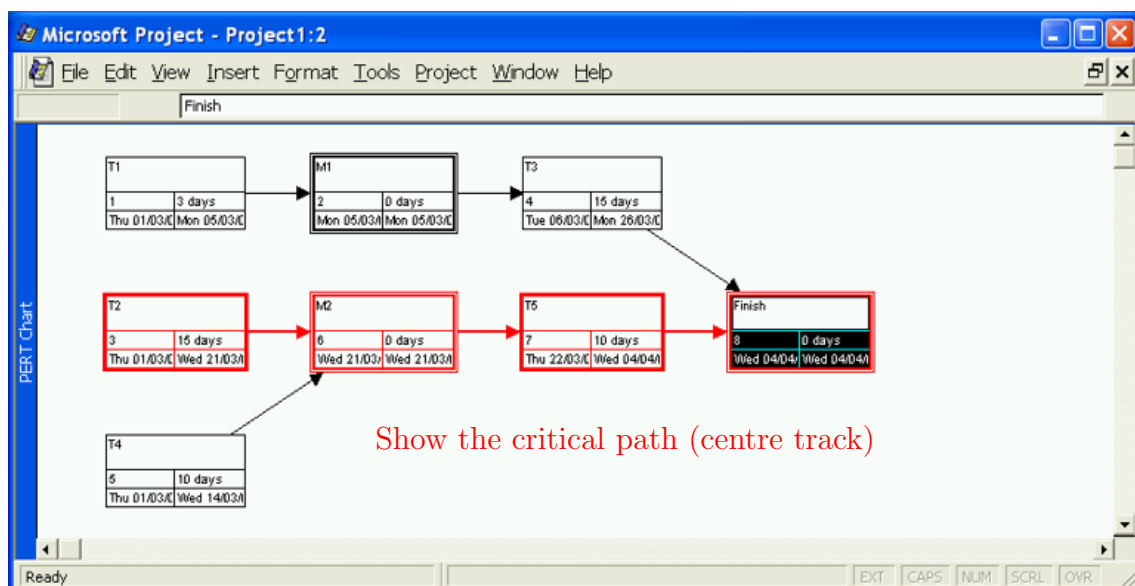
Example: Microsoft Project

A Gantt chart shows the tasks on a calendar:



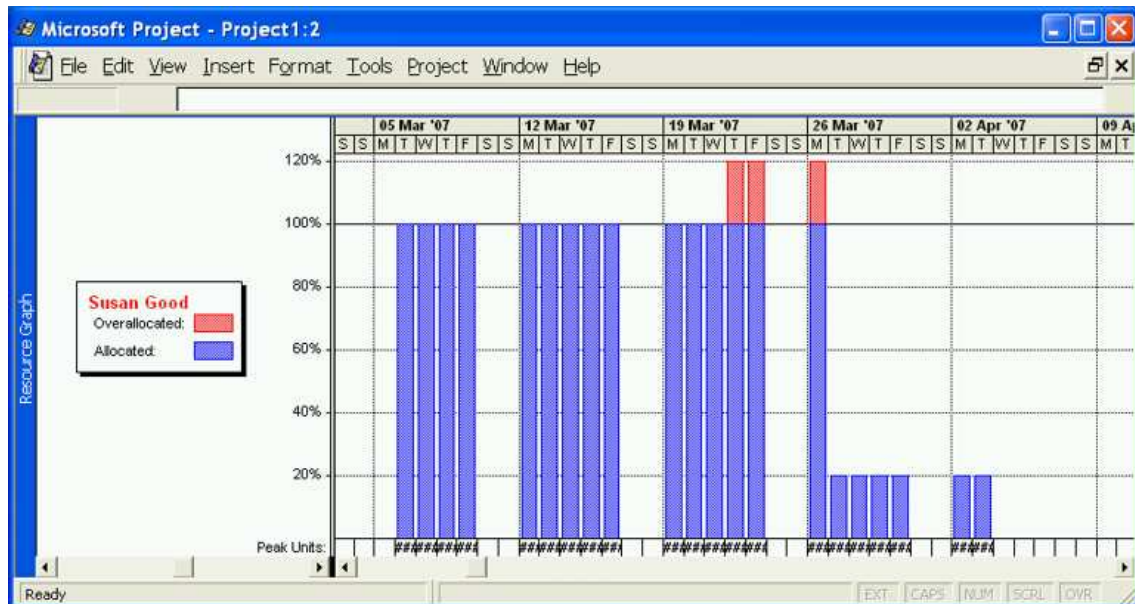
Must actively track progress (add today and color % task completion)

A PERT diagram is a form of *activity network*, which emphasises the relationships between tasks:



Show the critical path (centre track)

A Resource usage chart shows the loading on each staff member, e.g. the chart for Susan Good is



In this case, there is an overload in weeks 3/4 which needs to be resolved.

Delays and Rescheduling

Software projects are infamous for being late. There are many causes of over-runs, e.g:

- Programmers are optimists.
- Programmers are often reluctant to admit to delays.
- Sales apply pressure for an early delivery.
- Designs often need to be modified as a result of early testing.
- Debugging times are unpredictable.

It is often assumed that

$$\textit{work done} = \textit{time} \times \textit{people}$$

This is not true, hence, delays are rarely soluble by adding more manpower.

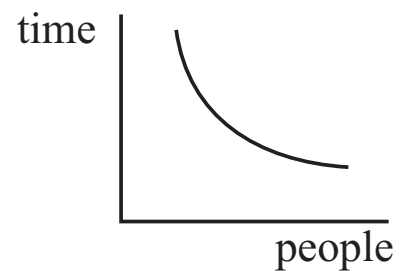
The only thing you can do with a late project is to

- cut features and/or
- reschedule

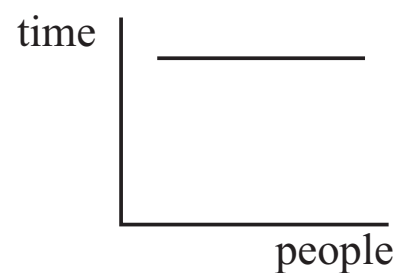
The Mythical Man-month

Brooks's Law: Adding manpower to a late software project makes it later.

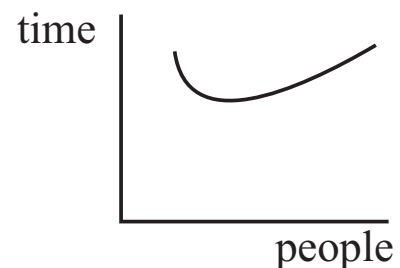
- An ideal project: infinitely partitionable, with no communication required.



- An unpartitionable project: cf “3 women cannot produce a baby in 3 months”.



- A project requiring communication: A software project has complex intercommunications, and the communication pathways increase as $\mathcal{O}(n^2)$. This extra communication can quickly overhaul any decrease due to partitioning.



Quality Control

Software development should not rely on testing/debugging to transform a poor quality product into a good one. Strict quality control must be applied throughout the development process via

1. a guide to software standards
2. source code control
3. module testing
4. the “nightly build”
5. weekly progress meetings
6. code reviews

Weekly progress meetings are essential at all levels. Typically, the Project Manager will meet with the Lead Programmers weekly, and the Lead Programmers will meet with the rest of their team at least once per week.

The principle is that nobody should be left alone for more than a week.

Progress meetings are not a waste of time - they are essential for keeping everyone on track.

Software Coding Standards

Every organisation should have a set of software coding standards defining

- essential banners e.g. license declarations
- naming conventions for constants, variables, types and functions, e.g. `LINESIZE`, `int myvar`, `class TreeNode`, `DoSomething()`.
- layout conventions including indentation, layout of control structures, placing of braces, etc
- deprecated programming styles, e.g.
 - `x=10, y=4;`
 - `if (p==0)` vs `if (p==NULL)`, etc.
- consistent use of alternative library functions, e.g.
 - `printf("x=%d\n",x)` vs `cout << "x=" << x << '\n'`
- use of comments
- etc

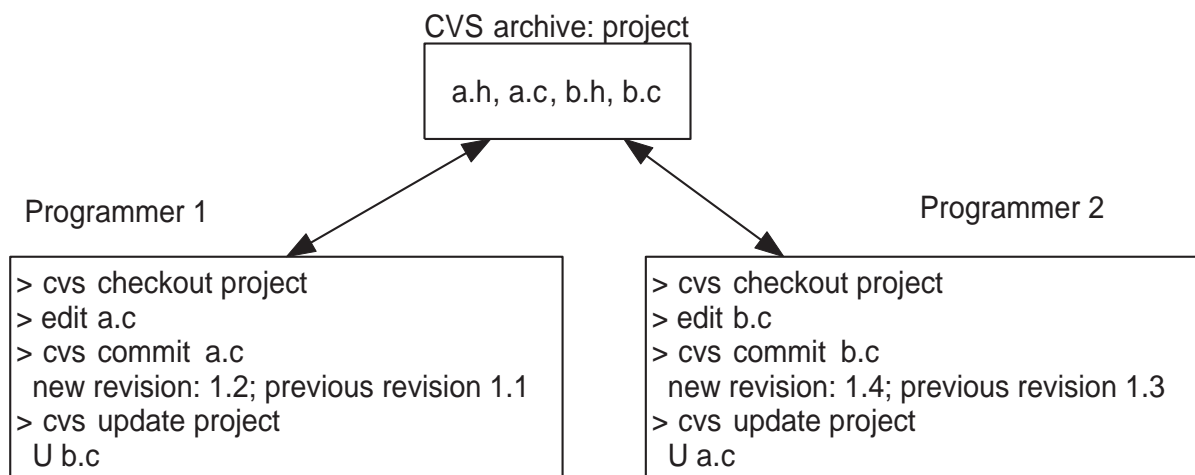
The aim is to ensure that all code looks as if it was written by the same person. This greatly simplifies understanding/maintenance of other people's code amongst the team.

Source Code Control

A source code control system provides

1. a central place to store all source code
2. a historical record of what has been done over time
3. a facility to record a set of sources as a “release”
4. an ability to reconstruct a project as it was at any time in the past
5. a facility to create separate code branches and merge them later

Example: Concurrent Versions System (CVS) CVS is popular, but rather old fashioned. In terms of features, it has been largely superseded. Subversion (SVN) provides a similar model but more modern features (commits are transactions). Newer systems such as git and Mercurial provide more advanced options.



Module Testing

Testing at the module level should be an integral part of the code development process. A good module testing strategy should include the following principles:

- All non-trivial class methods should have debugging code built-in.
- Debugging code should be included in product releases. Compile-time **DEBUG** directives should only be used to exclude debugging code if it has a significant impact on performance.
- Debugging should be enabled by switches set in a configuration file.
- Every module should have a test program to exercise it - this is often called a *test harness*.
- Where modules depend on other modules, the dependent modules should be completed and tested first
Software should be designed top-down and built bottom-up

The combination of test harness and built-in debugging code is used for three purposes:

1. debugging during code development
2. regression testing
3. post-release maintenance

Example: Testing a Stack Class

Consider a simple data structure implementing a stack of chars

```
class CharStack {
public:
    CharStack(int size);
    void push(char x);
    char pop();
private:
    int csize;    // size of stack
    int used;    // num chars on stack
    char *sp;    // stack pointer
    char *data;  // stack storage area
    bool trace;  // set true to debug
};
```

The `push` operation is *instrumented* as follows:

```
void CharStack::push(char x)
{
    *sp++ = x; ++used;
    if (trace)
        printf("CharStack: <...%c> %d of %d used\n",
              *(sp-1),used,csize);
}
```

`bool trace` more often `int trace` and used as a bit array

Now a test program can be written which can test the operation of the stack in detail by setting the **trace** variable and recording the output, e.g.

```
...
switch (testnum) {
    case 1:
        // test 1 - push n chars onto stack of size k
        CharStack *cstack = new CharStack(k);
        c = initialchar;
        for (int i=0; i<n; i++) {
            cstack->push(c); c++;
        }
        break;
    case 2:
        // test 2 - ....
        ...
}
```

The test program can pass the necessary test values via the command line, e.g.

```
> charstacktest 1 10 a 5
CharStack: <...a> 1 of 10 used
CharStack: <...b> 2 of 10 used
CharStack: <...c> 3 of 10 used
CharStack: <...d> 4 of 10 used
CharStack: <...e> 5 of 10 used
```

The programmer will use this facility during program development. It will also form an integral part of *Regression Testing*.

Regression Testing & the Nightly Build

When the **CharStack** class is judged to be complete and tested by the programmer, a *regression test* is constructed for it. Firstly, the required outputs are recorded:

```
> charstacktest 1 10 a 5 > charstacktest1.out
> charstacktest 2 a b c > charstacktest2.out
> etc
```

Then a test script is written, e.g.

```
charstacktest 1 10 a 5 > tempfile
diff tempfile charstacktest1.out >> logfile
charstacktest 2 a b c > tempfile
diff tempfile charstacktest2.out >> logfile
etc
```

Every night, all of the completed modules and their test harnesses are automatically:

- recompiled and
- all regression tests are run

This is called the *Nightly Build*. Every morning the logs are checked and any modules which failed their regression tests must be investigated and fixed.

The Nightly Build ensures that problems caused by changes and code redesigns are detected as soon as they occur.

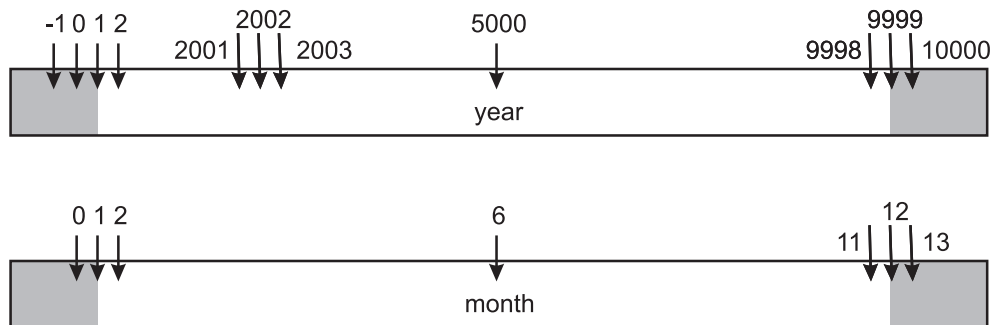
Choosing Good Test Data

Test data should be chosen by dividing test inputs into valid and invalid ranges (partitions). Test cases are then selected from the centres and boundaries of each partition. This is called *equivalence partitioning*.

Example: Date class The specification for a date class states that it should accept dates from 1 January 1 until 31 December 9999:

Date
<pre>+get_day(): int +get_month(): int +get_year(): int +set_date(int d, int m, int y)()</pre>

- Try values in the middle of the range and on the boundaries:



- Also try known problem values, e.g.

3-13 September 1752 29 February on non-leap years
 29 February 1900 29 February 2000

Use of Profiling Tools

A new module may pass all of its tests but may still not be fit for purpose. Three aspects of program behaviour will be of particular concern:

performance Computationally intensive methods might execute more slowly than expected. This can be due to inefficient loop structure, especially on inner loops; memory caching or generally inefficient architecture

test coverage Testing can only indicate the presence of bugs, it can never guarantee that there are none. One way to reduce the chances of missing bugs is to ensure that all possible paths through the code are properly tested.

memory leaks Memory leaks are notoriously difficult. They only show up when a program is run for a long time, and the source can be very difficult to track down. [cf the use of smart pointers in CORBA to try to minimise this problem.]

Fortunately, various tools (including the compiler itself) exist to help detect the above problems.

Example: Valgrind (linux) Purify(windows)

- a virtual machine using just-in-time (JIT) compilation techniques
- the original code is translated into a special form of machine code called IR
- IR is then executed on the virtual machine which has been instrumented to provide all of the necessary checks and statistics.

Valgrind can detect/monitor

- Use of uninitialized memory
- Reading/writing memory after it has been free'd
- Reading/writing off the end of malloc'd blocks
- Array bound violations
- Memory leaks
- Heap activity
- Race conditions in multi-threaded code

The problem with Valgrind and all similar profiling tools is that they run very slowly (4 to 5 times more slowly in the case of Valgrind). Hence, they are not so useful for real time programs.

Code Reviews

There are three types of code review:

Inspections These are quite formal and must follow a defined procedure. An inspection team consists of 5 or 6 people each of whom has a specific role: moderator, author, secretary, user-representative, standards-bearer, QA-inspector. **provides audit trail: ISO9000**

Walk-throughs These consist of 1 to 3 reviewers plus the author who manages the process. They are less formal than inspections and are usually the best compromise in a small company.

Readings A reading is like a walk-thru except that the emphasis is on the preparation phase and the actual meeting is just for reporting results. They are better than nothing but miss the benefit to the author of having to explain his code to the reviewers.

All three types of code review share common features

- the goal is to detect errors not to fix them
- the reviewers should be familiar with the overall design and the organisation's software standards
- higher management does not attend or see the outcome
- all participants must be circulated with all relevant documents before the meeting and must prepare for the meeting

Conduct of a Walk-Through

The reviewer team will typically consist of the lead programmer and/or the program manager plus one or two peers. The author is in control. The spirit should be similar to giving a technical presentation, not an examination.

Before the meeting each reviewer should be provided with

1. copies of all related design documents
2. a copy of the organisation's coding standards
3. copies of previous reviews on this or similar code
4. a list of any special use cases that the program manager wishes to focus on
5. the code to be reviewed

At the meeting, the author should give an overview of the code and its function, then go thru the code line-by-line. The reviewers should have read the code themselves before the meeting and they will ask for clarification and make comments. The author should mark these on the code listing.

After the meeting, the author should review all of the comments and modify the code accordingly.

Satisfactory code need only be reviewed once, but the reviewer team might decide to hold a second review if they have severe concerns.

Low Level Documentation

Good documentation of all product code is essential for future maintenance and development.

This documentation should sketch the class structures, the way that they interact and the primary data structures. Much of the detailed documentation can be generated automatically from the source code.

Example: Doxygen - analyses source code and uses special comments to automatically generate documentation in both browsable HTML and Latex/RTF for paper versions.

For example, consider the following source code fragment

```
/** \file a.h \author Steve Young \date 27-02-07 */

/// This is the base class. It is provided here simply to
/// illustrate how doxygen works.
class Base {
public:
    /// Construct a base with index \parm i
    Base(int i);

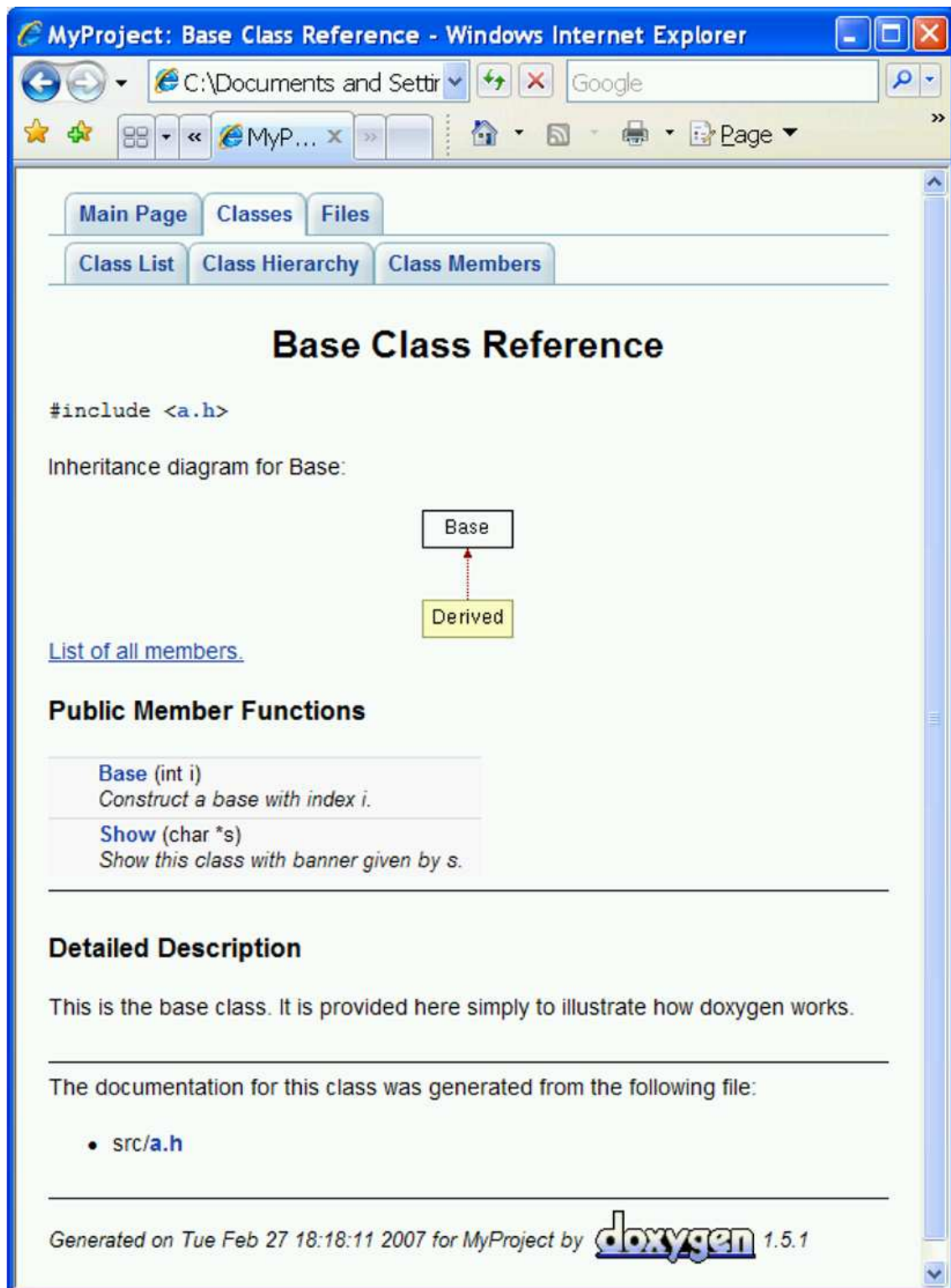
    /// Show this class with banner given by \parm s
    Show(char * s);

private:
    /// Private storage for base class
    short *private;
};

/// This is a derived class
class Derived:Base {
public:
    /// Show this class with banner given by \parm s
    Show(char * s);
};
```

Point out the Doxygen comments and tags

Automatically generated HTML documentation



The screenshot shows a Windows Internet Explorer browser window titled "MyProject: Base Class Reference". The address bar shows the path "C:\Documents and Settings\...". The browser has a single tab titled "MyP...". The page content includes navigation tabs for "Main Page", "Classes", and "Files", with sub-tabs for "Class List", "Class Hierarchy", and "Class Members". The main heading is "Base Class Reference". Below the heading, there is a code snippet: `#include <a.h>`. An inheritance diagram shows a box labeled "Base" with a red dashed arrow pointing to a box labeled "Derived". A link "[List of all members.](#)" is present. The "Public Member Functions" section lists two functions: "Base (int i) Construct a base with index i." and "Show (char *s) Show this class with banner given by s.". The "Detailed Description" section contains the text: "This is the base class. It is provided here simply to illustrate how doxygen works." and "The documentation for this class was generated from the following file:" followed by a bulleted list containing "src/a.h". At the bottom, a footer reads: "Generated on Tue Feb 27 18:18:11 2007 for MyProject by **doxygen** 1.5.1".

Integration Testing

Once all program components have been built and individually tested, they must be integrated to build the final system.

This is the stage of software development at which the Testers take control. During the code development process, they will have been designing a suite of system level tests. They may also have been building simulations of other system components (eg. hardware that the software integrates with), organising user interface testing panels, etc.

Testing at this stage will be addressing the following issues:

1. Does the integrated system work properly?
2. Does it support all of the use cases listed in the initial design requirements?
3. Does it support all of the functionality claimed in the documentation?
4. Does it scale to the required performance levels?
5. Is it possible to break the software?

Interface Testing

One of the most common types of error in complex systems are *interface errors*. Different teams of programmers may have misunderstandings about how to use a particular interface. These errors are not usually picked up in component testing since they occur due to the interaction between components.

Common errors include:

- Passing parameters of the wrong type, or in the wrong order.
- Ignoring expected pre-conditions, e.g. passing an unsorted array to a method expecting a sorted one.
- Calling functions in the wrong order, e.g. invoking methods on an object before it is created.

Functional tests might reveal interface errors, however, the most cost-effective means of finding interface errors is often a static technique such as program inspection.

Stress Testing

Once the system has been completely integrated it is possible to test the system for emergent properties such as performance and reliability.

Stress testing places the system under more and more load until the system fails. For example, a credit card transaction processing system may be designed to process 1000 transactions per second; stress testing tests the system up to, and beyond these specification limits.

The aim of stress testing is to test the failure behaviour of the system—to ensure that it suffers a *graceful failure*. There should be no data loss, and no sudden crashes.

Stress tests may also highlight problems that would be unlikely to come to light otherwise. For example, timing problems (e.g. race conditions) are more likely to occur under high load.

Bug Reporting and Triage

triage n. A process for sorting injured people into groups based on their need for or likely benefit from immediate medical treatment.

During integration testing, every bug discovered is recorded in a database. The Lead Tester will examine every bug and assign a priority to it. Typical, categories are:

critical	a showstopper - must be fixed before release
high	a high priority bug - fix if at all possible
medium	fix as many as possible
low	fix only if easy and obvious

The process of assigning categories to bugs is called *triage*.

As bug fixing progresses, the Project Manager will monitor the total bug count. When it falls to a level that he considers acceptable, he will authorise a release.

Summary

The key elements of good software management are

- Clear and unambiguous specification and high level design
- Accurate task scheduling and assiduous progress monitoring
- Due attention to quality control especially
 - Use of code reviews
 - Rigorous testing of individual modules
 - Extensive regression testing
- Keep staff fully engaged in progress through regular meetings and good communication