COMP3009 – Software Quality Assurance and Project Management

Revision Notes

1) Introduction

This module looks at the following things:

- Software development processes
- Capability Maturity Modelling
- ISO 9000
- SAIV, RAD and RUP (development processes)
- Metrics of software development
- Residual Bug content (detecting bugs in code)
- Re-use metrics (how the reuse of code is measured)
- Risk management and 'what if' planning

The **definition** for Software Quality Assurance is as follows (from the IEEE):

- A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established requirements
- A set of activities designed to evaluate the process by which the products are developed or manufactured (this contrasts to Quality Control which I think looks at the quality of the products, not the process)
- A systematic, planned set of actions necessary to provide adequate confidence that the software development process or the maintenance process of a software system product conforms to established functional requirements as well as with the managerial requirements of keeping the schedule and operating within the budgetary confines.

2) Capability Maturity Modelling (CMM)

Introduction

When the importance of software was being recognised in the 1970's, software development processes were created to ensure high quality software was being built. A process is the set of procedures, methods and strategies. You may have the tools, infrastructure, people and engineers in the organisation. The process is how you integrate all of these factors. A process is a higher view of IT and of the development lifecycle.

This is a development model that was originally developed as a tool to objectively assess the ability of government contractors' processes to conduct a software project (such as how often meetings take place, training, configuration management,

testing). Despite it originating from software development, it is used as a general model to aid in other areas such as project management, software maintenance, human capital management and risk management.

It is a model that allows organisations to **improve their software process** by defining the stages the organisation goes through as they improve the process. It is used as a guide for selecting process improvement strategies by first determining the current process capabilities of the organisation and then identify the issues that are most critical to the software quality and the process improvement.

A CMM has several requirements to fulfil;

- Needs to set goals for senior management
- Identify priorities and process improvement
- Identify process capability of organisations
- Predicting future process performance of projects
- Ability to compare to other organisations in the industry

The CMM was first published in 1988, and as a book titled ‘Managing the Software Process’ one year later, so it has been around for about 20 years or so.

CMM has since been superseded by CMMI (Capability Maturity Model Integration), which was created to sort the problem of corporations using multiple CMMs at the same time.

If is one of the ‘traditional’ or ‘heavy weight’ strategies, meaning there is a lot of effort put into the planning and quality assurance, which is a similar stance to the V shape model.

A maturity model is a set of structured levels that describe how well the behaviours, practices and processes of an organisation can reliably and sustainably produce required outcomes. There are several things that it can provide, such as a place to start with software development, a common language and shared vision or a way to improve your organisation.

CMM could also be used as a benchmark for comparison and as an aid to understanding things (such as comparing the processes of different companies that have something in common, such as they are all software houses).

Capability Maturity Model Integration (CMMI) is improved from CMM by having many maturity models integrated into each other and not having it so specific to software.

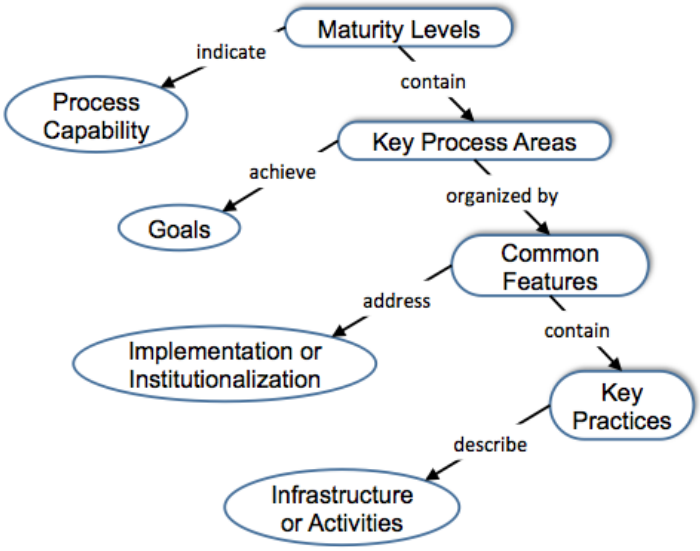
Structure

There are 5 aspects to the CMM;

- **Maturity Levels** – These 5 levels describe the process maturity, where the highest level (5) is the ideal state, where processes are systematically managed by process optimisation and continuous process involvement
- **Key Process Areas** – This is a collection activities that are performed together to achieve a set of goals that are considered important
- **Goals** – These describe the states that must exist for a key process area to be implemented in an effective and lasting way. How the goals have been

accomplished is an indication of the organisation's capability at a given maturity level.

- **Common Features** – These are practices that implement and institutionalise a key process area. There are 5 types of common features:
  - Commitment to Perform
  - Ability to Perform
  - Activities Performed
  - Measurement and Analysis
  - Verifying Implementation
- **Key Practices** – These are elements of the infrastructure and practice that contribute most effectively to the implementation of the KPAs.



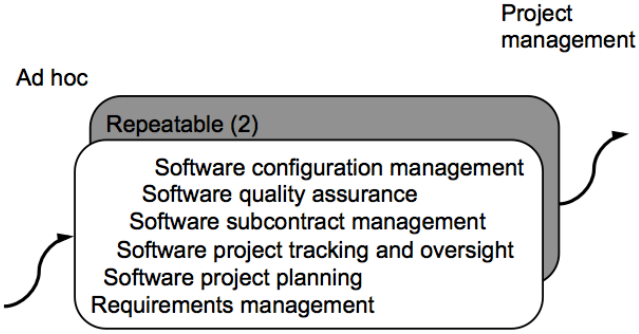
So maturity levels indicate the process capability of the organisation. These levels contain Key Process Areas that define the goals an organisation needs to achieve to go up to the next level. Each KPA is organised by a Common Features that address the implementation of the KPA. Each common feature contains key practces that describe the elements of the infrastructure and practice that will contribute most effectively to the implementation of the KPA.

Levels

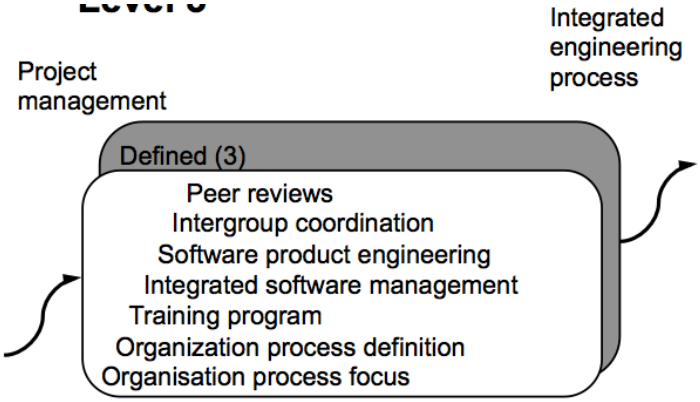
The levels are based on the belief that the higher a level a corporation is at, the more predictable and effective is the corporation's software process is. There are 5 levels, and in each level there are Key Process Areas which characterise that level, and each KPA has 5 definitions identified (Goals, Commitment, Ability, Measurement and Verification).

The levels are defined as follows:

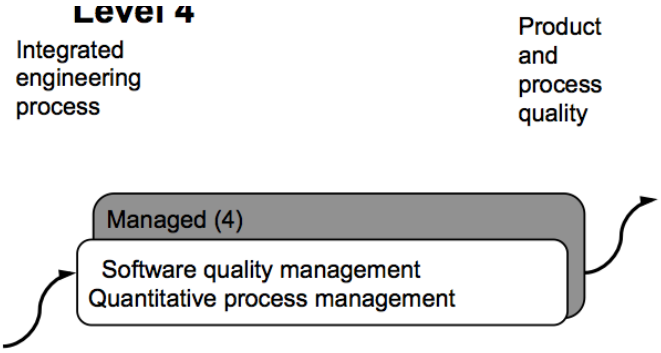
1) Initial – This is where the process is undocumented and in a state of dynamic change. It is uncontrolled and results in a chaotic or unstable environment for the processes. Requirements are gathered, something happens and an output is produced.



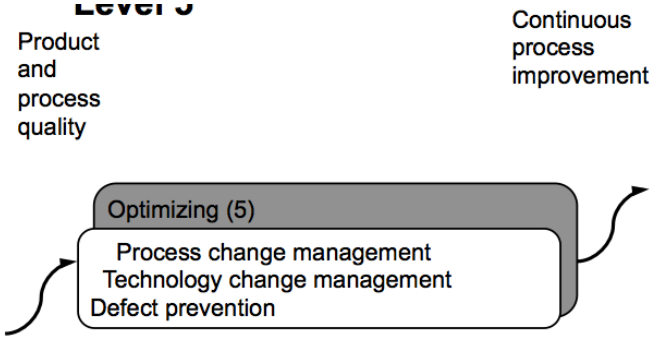
2) Repeatable - Where the process can be repeated with possibly consistent results. Discipline is unlikely to be rigorous. Progress is visible at defined points/milestones.



3)Defined – This is where the process is documented as a standard process and can be improved over time.



4)Managed – The process is monitored using process metrics and can use these metrics to improve the process over time.



5)Optimising – A process at this stage has its focus on incrementally improving the process. Another focus is determining the common causes of process variation.

The typical environment of a company that is at level 1 in CMM can have managers say things like 'I'd rather have it wrong than have it late' and 'The schedule is the main thing as promotions and pay raises are based on meeting it.'

Peer reviews can help raise organisation up the maturity levels.

Responsibilities

Responsibilities for implementing Key Process Areas can either belong to:

- **The Project** – This is mainly responsible for addressing many key process areas
- **The Organisation** – This is responsible for addressing other KPAs

Project responsibilities include acting on the requirements management, software project planning, tacking and oversight, subcontractor management, configuration management, integrated software management, peer reviews, defect prevention.

Organisation Responsibilities include acting on software quality assurance, organisation process focus, training, technology and process change management.

Example

An example of Goals of a KPA are shown below:

1. Software estimates are documented for use in planning and tracking the software project.

2. Software project activities and commitments are planned and documented.

3. Affected groups and individuals agree to their commitments related to the software project.

Following on, an example of Key Practices are shown below:

Software Project Planning

Activity 9

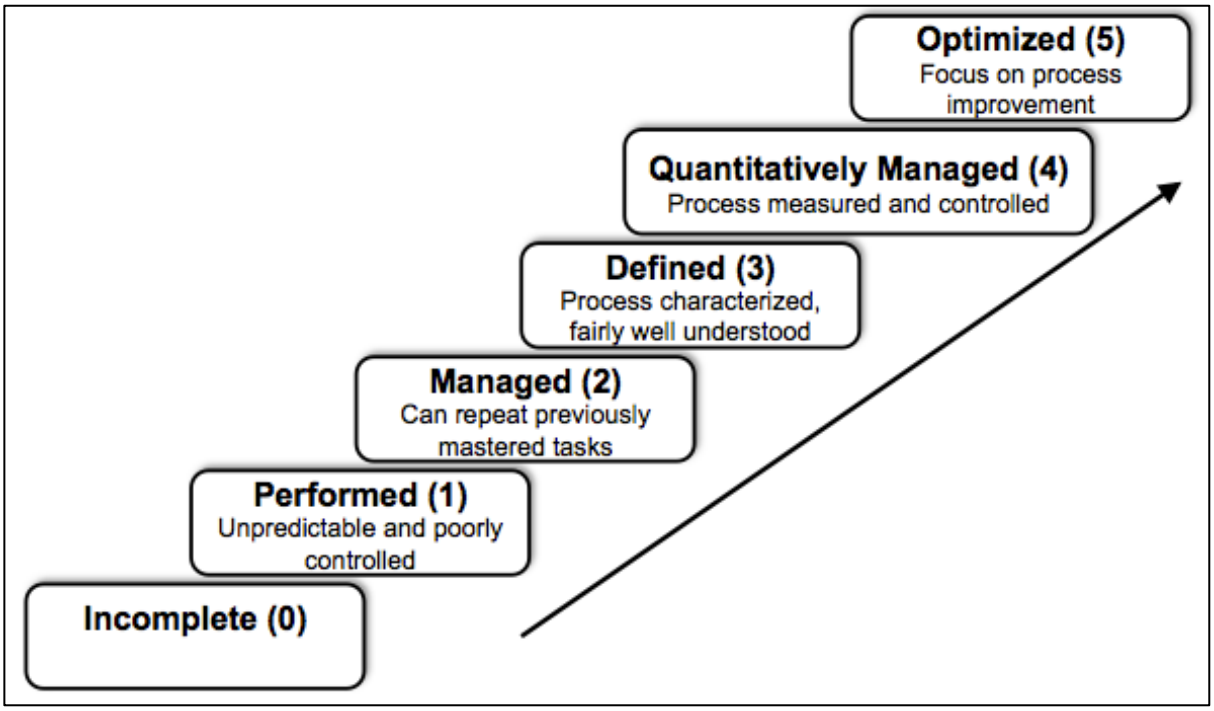
Estimates for the size of the software work products (or changes to the size of software work products) are derived according to a documented procedure:

This procedure typically specifies that ....

So the goals are just that; goals or aims to improve the software development process. An activity is a method that will most effectively meet the goal.

CMMI

CMMI has six levels if the maturity framework (instead of the five in CMM).



This is very similar to CMM except there is a level 0 which marks no process control whatsoever.

3) ISO9001

ISO9001 is part of the ISO9000 family of standards that relate to quality management systems and are used to ensure organisations meet the needs of the customers and other stakeholders.

ISO9001 deals with the requirements an organisation has to meet to have the standard met (and assumedly receive ISO9000 certification).

In the ISO9001 standard, there is a **quality policy** which is a formal statement from management and is closely linked to the business and marketing plan and to the customer needs. It is intended that this policy is understood by employees at all levels of the organisation, with each employee needing measurable objectives to work towards.

Decisions about the quality of the system need to be made based on recorded data and the system is regularly audited and evaluated for conformance and effectiveness.

Records need to kept regarding the source of raw materials and where products are assembled so if there is any problems, then they can be tracked to the source of the issue.

Businesses need to plan for each stage of the development of a new product.

Regular audit and meetings are needed to review performance.

Potential problems in the running of the business (ie, a supplier ceases trading) needs to be planned for with plans on how to deal with the situation.

Summary

ISO9001 basically covers the management of development, training, testing, standards and metrics. Furthermore, the acceptance, delivery and document control of a process. This contrasts to CMM which covers the entire process, going down to things such as prototyping and code re-use procedures.

4) Total Quality Management

There are a couple of potential problems with thinking with processes. If there is too much of a focus on the **people** of the company when implementing a process, then there will be resistance from people unhappy with **change**. If there is too much of a focus on **tools** that do not fit in with the process then this will lead to ineffective automation. Too much of a focus on **procedures** that do not match the process will lead to unusable procedures.

There is an intimate tie between the quality of the product (software system) and the quality of the process that was followed to make the product. This link is identified with the **Total Quality Management (TQM)** movement used in manufacturing and service industries.

TQM is the application of quantitative methods and human resources to improve the material and services supplied to an organisation, improve all of the processes within an organisation and to improve the degree to which the needs of the customer are met, not and in the future.

Common Points

There are several points that are common in the quality movement:

- Quality Improvement is the responsibility of the management
- Quality Improvement looks at fixing the process and not the people
- Quality Improvement must be measured
- Rewards and incentives are necessary to establish and maintain an improvement effort
- Quality improvement is a continuous process

TQM is applied to software at the system and organisational levels. This contrasts to CMM where it only applies to software.

It also takes the view that the quality of products and processes is the responsibility of everyone who is involved with the creation or consumption of the products and services offered by an organisation. This includes the workforce, the suppliers and distributors.

5) Costs of Software Projects

When you are discussing costs with a client, there are contrasting opinions. The client will want to have a fixed price (for their convenience), but the developer does not want to give exact values due to the unpredictability of software development and how difficult it is to estimate.

So what things make up the cost of developing software? One key cost is the time and materials of the project where the developer is concerned with the negotiation between the time and materials, whereas the client is concerned with the cost. In some cases (mainly the Government), the client is willing to pay as much as it costs to get the product finished and done.

Cost Estimation

Developers need a way to estimate the cost of a software project. There are several different methods for estimating outlined below.

Parkinson's Law states that work will fill the time available for the project. This means that if you have a set amount of time to do the project, then people will work to fill that time

The **analogy** estimation will look at previous projects and their costs and draw an analogy to the current project at hand. A downside is that different projects are unique, so the estimate given could be wrong. Furthermore, the specification of the project could change at a later stage. It is a simple technique that can be done by a salesperson.

Price To Win is a technique used when you are bidding for a contract. If you know what other competing organisations are bidding, then you can come in with a bid that undercuts them. However, a good client will keep people's bids a secret. The danger of using this method it is possible to become a loss leader, where you bid a value so low that you will not make a profit on it. This is similar to commerce where you sell a product for less than its cost to produce.

Top Down is where we have a high level design in mind, and know the components that are involved in developing the system, and then drill down from this high abstraction. You can then ask experts for the costs of each little bit of the project.

Bottom Up is used if we have time to design down to the software module level, then we can give estimates for the cost of each module. This will give more accurate estimates, but will take a while to get that estimate, with the possibility of another competing company making a bid before you can.

Parametric models can be used to estimate costs, but also project duration and staffing levels. Cost estimation is related to the staffing patterns, the amount of development parallel to the team size and the delivery date. Several models exist, such as PUTNAM, SLIM, Boehm, COCOMO, GECOMO.

Accuracy

The accuracy of each method can vary. For example, the analogy, Parkinson's Law and price to win can be inaccurate by up to 400%

Top down has an inaccuracy of roughly 200%-300%. Bottom up provides a better estimate with it having a 10% error. Parametric model has a 20% error and Outline System Design has a 10% error.

The estimate given at the beginning of the project will be less accurate than an estimate given later on in the project. So as the project continues, a more accurate estimation for project costs is given.

6) Parametric Models

Parametric models can be used to estimate project costs, staffing levels and project duration. There are a number of models that exist and are outlined below.

COCOMO

Constructing Cost Model (COCOMO) is an algorithmic software cost estimation model. It uses a regression formula with parameters that are derived from historical project data and current project characteristics.

There are two versions of COCOMO; COCOMO 81 is the original that followed the Waterfall model, and COCOMO II is the successor introduced in 2000.

COCOMO 81

COCOMO consists of a hierarchy of three increasingly details forms (see below)

Basic COCOMO

This is good for providing quick estimates of software costs, but its accuracy is limited due to a lack of factors to account for difference in project attributes. It does estimates this as a function program size which is expressed in thousands of lines of code.

COCOMO classes software projects into one of the following 3 types:

- **Organic projects** – These are ones where there are small teams with a good experience working with 'less than rigid' requirements. This is probably relating to Agile methods
- **Semi-detached Projects** – These are medium teams with mixed experience working with a mix of rigid and less than rigid requirements
- **Embedded projects** – These are projects with tight constraints (such as hardware, software and operational constraints).

To calculate the cost, the following equations are used:

Effort Applied (E) = $a_b(KLOC)^{b_b}$ **[man-months]**
Development Time (D) = $c_b(\text{Effort Applied})^{d_b}$ **[months]**
People required (P) = Effort Applied / Development Time **[count]**

The constants in the formula is dependant on the type of project that is in question, with the values shown below:

Software project	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semi-detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

This method is good for quick estimates of software costs, but it does not account for the differences in hardware constraints, personnel quality and experience and other factors

Intermediate COCOMO

This computes an estimate based on the program size (as in basic COCOMO) and a set of cost drivers that include hardware, personnel and project attributes. There are 4 of these cost drivers:

- **Product Attributes** – This encompasses overall product qualities, such as the reliability, the size of the application database and the complexity of the product
- **Hardware Attributes** – These take into consideration given constraints for the project, such as memory constraints, run-time performance constraints and the volatility of the virtual machine environment
- **Personnel Attributes** – This looks at the people in the organisation and their abilities, such as analyst capabilities, software engineering capability, applications experience and programming language experience.
- **Project Attributes** – These are factors relating to the project itself, such as the use of software tools, the application of software engineering methods and the required development schedule.

In total, there are 15 attributes and each receive a rating from 1-6 ranging from ‘very low’ to ‘extra high’.

Detailed COCOMO

This is the big one; it incorporates all of the characteristics from the intermediate version with the addition of an assessment of the cost driver’s impact on each step of the software engineering process.

7) Project Factors

There are many stages in software development, and each will have their own metrics to determine if it is a success.

Correctness is used to determine the extent to which a program satisfies its specifications and fulfils the user's mission objectives. This can be measured in by a percentage, but the easiest way is to do '1-(faults/lines of code)'. This means the number of faults in the software are relative to the number of lines of code.

Efficiency is the amount of computing resources and code that is required by a program to support a function. This is measured by 1-(actually utilisation/allocated resources)

Flexibility is the effort required to modify an operational program. This metric is calculated by 1-(0.05 x average labour days to change). It is possible that this metric could be a minus number if the average number of labour days is large. This means that flexibility is tied to the amount of work someone has to do to change the program.

Integrity is the extent to which software or data by unauthorised persons can be controlled. This is described by 1-(faults/lines).

Interoperability describes the amount of effort to couple one system with another, and is an effort metric. There are two key factors in determining this metric; the size of the system and the time taken to couple. Therefore, the metric is measured y 1-(effort to couple/effort to develop).

Maintainability is the effort to locate and fix a defect in an operational system. This is measured by 1-(0.1 x average labour days to fix a bug), therefore it is dependent on the amount of time to fix.

Portability is the effort to transport a program from one hardware configuration and/or software system to another environment. This is based on time and therefore is measured by 1-(effort to transport / effort to develop)

Reliability is the extent to which a program can be expected to perform its intended function with required precision. This is measured based on the faults in the code, therefore it is measured by 1-(faults/lines of code).

Reusability is another one, but it is covered in a later lecture/revision section I think

Verifiability is the amount of effort required to test a program to ensure that it performs its intended functions. This is relative to the amount of effort used to develop the system and therefore it is measured by 1-(effort to verify/effort to develop).

All of this can be easily remembered as CEFIIMPRRV.

8) Metrics

There are many metrics that can be used to measure things, but the metrics that are useful will be dependent on the application.

Reliability defines the **probability** that the software will not fail in a given period of time. Alternatively, it is the probability that a given function will perform on demand. The view on reliability will be dependent on what your project goals are. For example,

a time-critical system will need to perform within a set of time parameters and therefore could be seen as failing if it take too long. Alternatively, a general piece of software would fail if it crashes or does not complete a task that it is supposed to.

Rate of continuous failures is the opposite of **reliability** where it measures the **rate** at which failures occur in the system. The two metrics are linked, where the greater the rate of continuous failures, the lower the reliability.

MMTR (Mean Time To Repair) measures how easy it is to fix a system if it goes down; if it is easy to repair, then you will have less effort to be put into fixing the system. The metric is the average amount of time it takes to bring a software system back up.

MTTF (Mean Time To Failure) is the average amount of time it takes for a system to fail. This metric can also be found on hard disk drives.

MTBF (Mean Time Between Failures) details the average amount of time between each instance of the software failing. This is calculated by looking at the previous performance of the system.

Availability measure the **probability** that the software will be functioning at a given time in the future.

Application of Metrics

There are many applications the metrics described above can be applied to.

Reliability is used in the domain where failure is very undesirable, such as avionics where any failure can result in something catastrophic like loss of life.

Rate Of Continuous Failures is where frequent failures are undesirable and is therefore used in producing operating systems.

MTBF is used where software stabilisation is critical, such as control systems and software packages.

Availability is important when downtime is really undesirable, such as systems in telecommunications.

When developing software, a metric called **time to target** can be used to measure how long it will take until the software has reached a suitable level of reliability or some other metric.

Reliability Growth Models

Usually in development, the inter-failure times for the software tend to get larger as development goes on. It is possible to plot the probabilities of MTTF on a curve with the actual value of the MTTF being in the middle of the curve. This is an example of a **reliability growth model**.

The accuracy of these models can be very variable, but the accuracy of these can be assessed and measured, meaning you can be a better judge of what models are best. However, the best way to measure the accuracy is to measure the current reliability of the software, which can only be found out in the future! Therefore, a key assumption here is the conditions of use in the future will be the same as those of the past, therefore allowing accuracy to be measured. However, most software test regimes do not emulate operational usage, hence this is often an overseen aspect of testing.

Any predictions made will be probabilistic as it is not possible to know what will be input into the system next, therefore we use a mathematical model, statistical inference and a prediction procedure and analyse past predictions to assess accuracy.

Jelinski & Moranda

The best known reliability growth model is Jelinski & Moranda. It is based on the homogenous Poisson point process because it uses randomness and uniformity of the failure rate.

The failure rate is proportional to the number of residual errors (denoted 'N'), O is the proportionality constant and we estimate both N and O by the maximum likelihood principle. Maximum likelihood basically takes measurements from a sample of the population and ensure that these values are the most likely in the probability distribution produced. These input measurements are the past performance of the system.

So we first get a set of data (x1, x2, x3...) that are the time intervals between errors. Then we will fund the density for Xi.

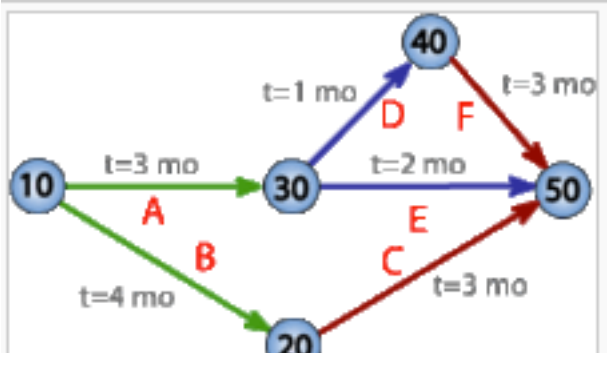
An example of this is with a space craft mission where by the software going into the space craft was observed to have 41 system anomalies in 8 months or so. The average time to the next error is 19.4 days. The length of the space mission is a certain number of days. All of this information is combined to produce a probability of the mission being successful is 66.2%.

Reliability Growth Models Conclusion

While it is possible to predict reliability, there is no universally best model as reliability is dependent on the application being built and the way that it has be produced (either through the application design or via the process under which the software was developed).

9) Critical Path Method

Critical Path Method is a tool/algorithm that deals with the scheduling of a set of project activities. It is used to identify which activities in a project determine



the length of the overall project from the relations between the activities and their durations.

To produce a Critical Path, you first need to obtain a list of activities that are required to complete the project and their estimated durations. In addition, the dependencies each activity has on other activities need to be noted (i.e. where a activity is needed to be completed before another activity may commence.

A diagram can be built to display the activities, dependencies and the durations. Each activity will have the following metrics attached to them:

- **ES** – Earliest Start – This is the earliest time that the activity may start based on the finish times of earlier dependent activities
- **EF** – Earliest Finish – This is the earliest time the activity can finish
- **LS** – Latest Start – This is the latest time the activity can start
- **LF** – Latest Finish – This is the latest time the activity can finish without delaying the project
- **Duration** – The duration of activity

Each activity will be denoted by a letter. You start with the initial activity and then from it add activities that depend on it to build the critical path network. From this, you will be able to identify which activities are critical for the project to finish on time (i.e. their earliest and latest start/finish times are equal) and which ones are not. Non-critical activities can have a 'float' associated with them which describes the amount of additional time that can be added to its duration without affecting the finish time of the project.

This can then be expanded to include resource allocations (such as having only one JCB to dig two trenches) or logical constraints.

While critical path methods are very useful, there are a few areas in software engineering when critical paths do not apply. Such examples include where a project does not have a fixed start or end point, where requirements are shifting/changing, new events will appear out of the blue, agile methods and iteration-style development does not fit in, unforeseen bugs taking project priority.

PERT

Program Evaluation and Review Technique (PERT) is a model for project management designed to analyse an represent the tasks involved in completing a project. This is usually used in conjunction with the Critical Path Method.

It is an event-oriented technique as opposed to a 'start and completion' oriented technique and is therefore used in projects where time is the overall factor (as opposed to cost).

A **PERT Chart** is very similar to a Critical Path diagram, but events (that link the activities) are numbers in 10s (to allow for other events to be added at a later date). For each activity, an optimistic, pessimistic and most likely estimations for its duration are given. An expected time is also given, which is an average of the 3

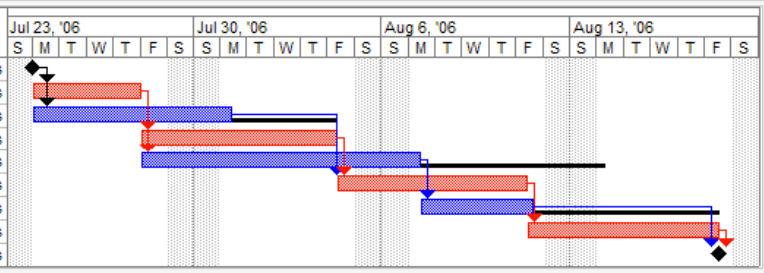
previously mentioned estimates (with the most likely time being given a weighting of 4 and the optimistic and pessimistic estimations given a weighting of 1).

An example of a PERT Chart table is shown below:

Activity	Predecessor	Time estimates			Expected time
		Opt. (*O*)	Normal (*M*)	Pess. (*P*)	
A	—	2	4	6	4.00
B	—	3	5	9	5.33
C	*A*	4	5	7	5.17
D	*A*	4	6	10	6.33
E	*B, C*	4	5	7	5.17
F	*D*	3	4	8	4.50
G	*E*	3	5	8	5.17

Once this step is complete, one can draw a Gantt chart or a network diagram.

ID	Task Name	Predecessors	Duration
1	Start		0 days
2	a	1	4 days
3	b	1	5.33 days
4	c	2	5.17 days
5	d	2	6.33 days
6	e	3,4	5.17 days
7	f	5	4.5 days
8	g	6	5.17 days
9	Finish	7,8	0 days



A Gantt chart created using Microsoft Project (MSP). Note (1) the critical path is in red, (2) the slack is the black lines connected to non-critical activities, (3) since Saturday and Sunday are not work days and are thus excluded from the schedule, some bars on the Gantt chart are longer if they cut through a weekend.

10) Number of Bugs

As mentioned before, reliability growth models can be used to estimate the number of bugs still in the software. There are a number of models that can be used:

- Jelinski & Moranda
- Bayesian Jelinski & Moranda
- Littlewood – Verral.

There is not best model, but the Jelinski & Moranda should be easy to understand.

The problem with measuring software reliability is by knowing how many bugs there are to begin with in the software; this is not possible to know!

Testedness is a metric that measures the amount of effort put into testing. If only 10% of the system has been tested, then the testing process will not be very effective in finding bugs. However, if you know the test coverage, then you will not necessarily know the testedness of a system, however looking back on previous project can aid making this metric more accurate.

Usually the accuracy of various models are very reliable. However, one consideration in testing is that when the project is handed over to the client, the conditions in which the software was tested in the developer's hands will be different to the conditions inside the client's workplace. This means that there are a higher

number of bugs discovered during handover. Cutover is the process of transitioning from one system to another.

Problems with Software Handover

There are several potential issues client environment where the software is being deployed:

- The client may have a different hardware and/or software configuration
- There may be viruses on the client computer
- Calendar related events (such as the Millennium Bug)
- The client may have a different security policy to the developers, meaning the developed system may be exposed to threats that it has not seen before or need to comply with different anti-virus or firewalls.

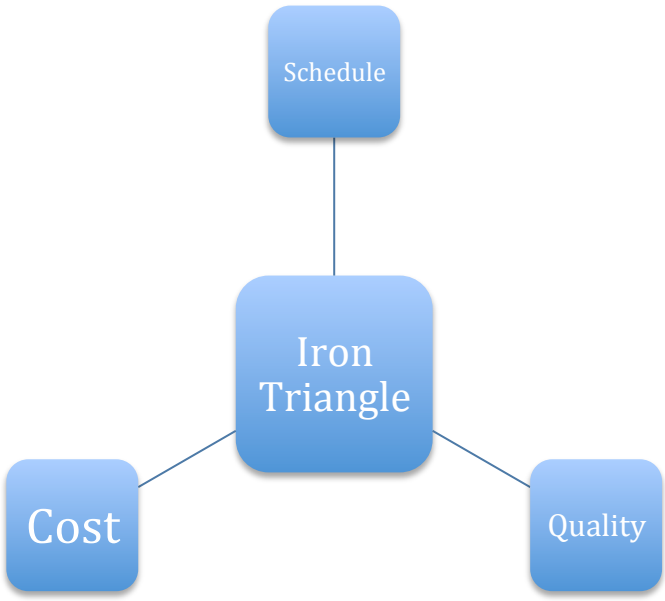
Reality

Collecting data sets for the aforementioned models when the system is difficult because the environment changes. Everyone needs to get used to the fact that the software is 100% bug free. There are some valid reasons for delivering software that has bugs in it. An example would be to meet the Christmas sales period.

SAIV

SAIV (Schedule as an Independent Variable) is a development paradigm whereby development of a system is downsized and minor features are dropped to allow for the system to meet its schedule. This treats the schedule as an independent variable.

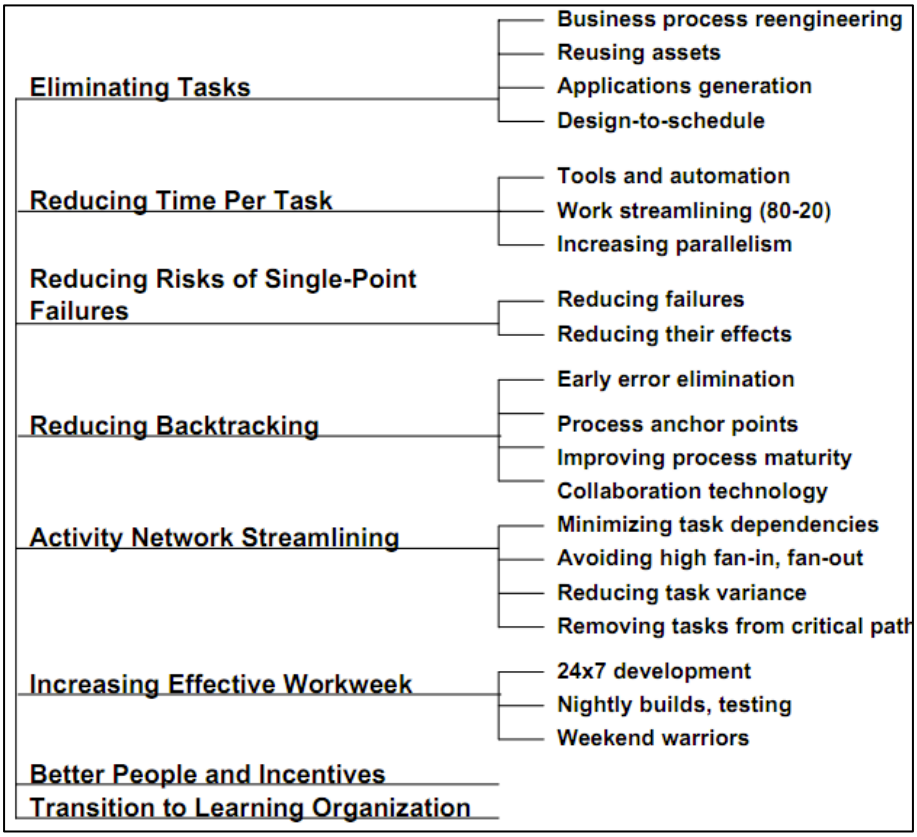
There are 3 key conflicting qualities to any project: **Quality, Cost** and **Schedule**. These can be arranged in a triangle; you can go towards one of these three aspects and maximise it. If you want to meet a delivery date, then it is going to cost more and there will be more bugs.



Common sense in development will suggest that developing the core elements of the project should be done first and then any additional requirements be prioritised and added after the core elements have been implemented. As the delivery date approaches, it is wise to drop features to meet the schedule. When you are prioritising features, it is often best to see what features will cost more time than

others and build a cumulative chart adding together the costs of the features left to implement. Usually, **20% of your features will take 80% of your time**.

Another crucial point is that if you save time on an activity, you will only save time on the whole project if it is a critical activity. In Rapid Application Development (RAD), the Opportunity Tree is a tool that will allow you to help identify and cut the critical path. The opportunity tree will give an overview of aspects that will reduce the amount of time on taken on the project and give actual activities that will fulfil these



aspects.

11) Software Trends

Back in the 60s and 70s, software companies were very large, but these days they are turning into **smaller** ones. Furthermore, with thanks to the Internet, real teams that need to have face to face communication are now virtual teams where they may never meet each other to get a project completed.

Big companies still exist and they have a hierarchy towards telecommuters (who can be part-time employees or freelancers). The way that you interact with people has also changed, whereby information used to be relayed by memo and now is available on company Intranets.

Software used to be a specialised art, but now it is easily accessible with kids in college dabbling with code. Software engineering now needs to be responsive to

train people to changes in the market as opposed to having a large formal process which was less suitable to changes.

There used to be large maintenance tasks to keep systems up to date and add new features. But today, software is more disposable and much easier to rewrite thanks to things like Object Orientation and open frameworks and libraries.

As hardware resources were limited in the 60s and 70s, writing efficient code was of paramount importance. Today is different; you can get away with making inefficient code thanks to the speed and plentiful resources being provided by today's hardware. As a result, there is less of a need for coders and more of a need for architects. This means the need for communication between humans is more necessary than ever.

Economic Trends

In software development's infancy, specialised knowledge was expensive to obtain. Today, as there is a lot more software around that can be reused and therefore the cost to develop a million lines of code has gone down.

Beforehand, the whole ecosystem was technology driven (where new hardware was released and new software needed to be written for it) but it is now driven by the market. This is thanks to how there are many languages, different size processors and how code is now designed to be backwards compatible with previous hardware. People today are now looking at how they can make money from creating applications.

Because of the move towards development teams being distributed around the world, the old cost models are a little obsolete. **Rapid Application Development** should be the development style of choice in this global market with the use of **SAIV** being used too. This new approach is popular after penalty clauses for the late delivering of software projects is becoming common.

Today's State

When developing software today, you need to follow a process that suits your needs as a company or an individual; this is critical to ensure quality software being produced. When writing code, it is best to look at integrating products together and seeing at what is available and share.

Architectures that are being developed should be more open, flexible and less concerned optimisation due to the advances in hardware.

Reasons For Failure

There are many factors that can cause software projects to fail. Common ones are a shortage of resources, people, time and/or tools. Operating under a mature process is not enough; a process tailored to the company's is what's needed. In addition, we need to have constant communication between the clients and the developer to ensure the project is going as what the client expects and to improve the overall quality of the software. The development team should also work well together. With all of these factors present, then the software project should be a success.

12) Artificial Bug Insemination

Finding out how many bugs exist in software is a nigh on impossible task. An analogy that was explained in the lecture was to find out how many squirrels exist in the common. Suggestions included shooting the squirrels on sight (but where would you send the hunters?), place nuts as bait and count the number that come and eat it (but how many will be counted twice?), measure the amount of food that is eaten and calculate it from that the number of squirrels. We could put cameras around the common, but then where would we place the cameras?

Profitable Vandalism

One way to find out how effective your testing regime is and how many bugs exist is to deliberately place bugs into the code without letting the testers know.

If one were to place 6 bugs deliberately into the system, and the next week testers found 8 bugs (2 of which were the ones we put in). From this, we know that 6 bugs were novel and can then get an idea of the test coverage and work out how many bugs are in the system. If we know these two factors, we can then work out the target time to get the system delivered to the client. So with 8 bugs found and 6 were genuine bugs, then we can estimate that 24 bugs existed at the beginning of the week where 18 were genuine at 6 were artificial. If the test coverage continues like this, then it would take 2 weeks to get to zero residual bugs.

This technique is rarely used, but it has been used in some projects. If this ever happens, the bugs implanted in the code will need to be documented well and taken out when the project ships.

Drawbacks

There is no guarantee that this method will be accurate. In addition there are many problems with this method. It is a waste of the tester's time by deliberately placing bugs in the code for them to find. In addition, if a tester solves a deliberately inserted bug, then this might have a knock on effect and create more bugs. Also, there is a risk that you antagonise the testers and decrease moral. Also, it is essential that you make detailed notes of the bugs inserted other wise you will loose them and may lurk around in the system.

13) Reuse and Metrics

In a project, there may be many things that you reuse. This does not have to be code, but manuals, documentation and project plans.

However, reusing code can save a lot of time and headaches in the implementation stage. There are several things that promote and lend to code reuse. The two most familiar will be the concept of object orientation (and how properly structured code can have all or parts of it reused). Another one is **component based software engineering (CBSE)** where systems are produced in components such that they are separate areas of concern.

There are a lot of commercial examples of reuse; C++ STL (Standard Template Library), Java Beans, COBRA, APIs, Microsoft MFC Library.

Module reliability

Say there is a project that has produced code that is going into a reuse library. Later on, another project reuses this code, but a few bugs are found in the code. This is fixed and progress continues.

So over time, the reused code will be thoroughly tested and there will be fewer and fewer bugs in that code. There is a model called **Gaffney's Model** that tries to put a metric on the number of times a piece of code has been reused and how reliable it is. It works by first obtaining figures for the amount of codebase that is reused from previous projects and the number of bugs existing in both the new and the reused parts. From this we can calculate the benefit of the amount of reuse that has happened in the system. We do not need to know the maths for this for the exam.

Encouraging

So you are the manager of a company and you wish to encourage the amount of code reuse that occurs in your development teams. What methods can you use to make this happen?

You could give the programmer a **bonus** every time they reuse some code from the company's reuse library. The downside to this is that someone could use the most mundane and simple bits of code that are simple to write in order to maximise the bonus. The alternative is to give people additional holiday instead of monetary bonuses.

In order for code reuse to be a convenience and not a chore, the code modules will need to be easy to find, indexed and well documented. Furthermore the modules will need to be 'clean' (I'm assuming that means it is free from debugging printfs and rubbish comments). This means there is an overhead to set up reuse modules and therefore another approach is to give a bonus to the people creating the reusable code.

14) Risk Management

Big projects need risk management; many things can go wrong in software development that can affect the finance, schedule or the quality of the product. There specialist software engineers that deal with risk management; Pressman, Charette, Elaine Hall.

This material is not examinable, but there is nothing stopping you from drawing from it in the exam as examples.

If certain events happen to a software project, then the developers could loose money, deliver bad quality deliverables, ruin the prestige of the company and alienate clients. **You cannot eliminate risks** from software projects; all you can do is know what to do in the event of risks arising. Therefore three steps are required to be taken in risk management:

- Identify all the risks
- Assess the risks in terms of probability of occurring and the impact on the company/project
- What to do when the risks arise.

You do not react to risks, you pre-empt them.

Identify Risks

The first step is to identify the risks. This cannot be done by one person alone (the manager), but needs to be done by all employees as they have a different perspective on the project.

There are different types of risks around that have different types of impacts on the project. Two metrics are assigned to each risk; the **probability** of the risk occurring and the **impact** the risk will have on the project. When risks are thought off, the will be added to a database for the managers to evaluate.

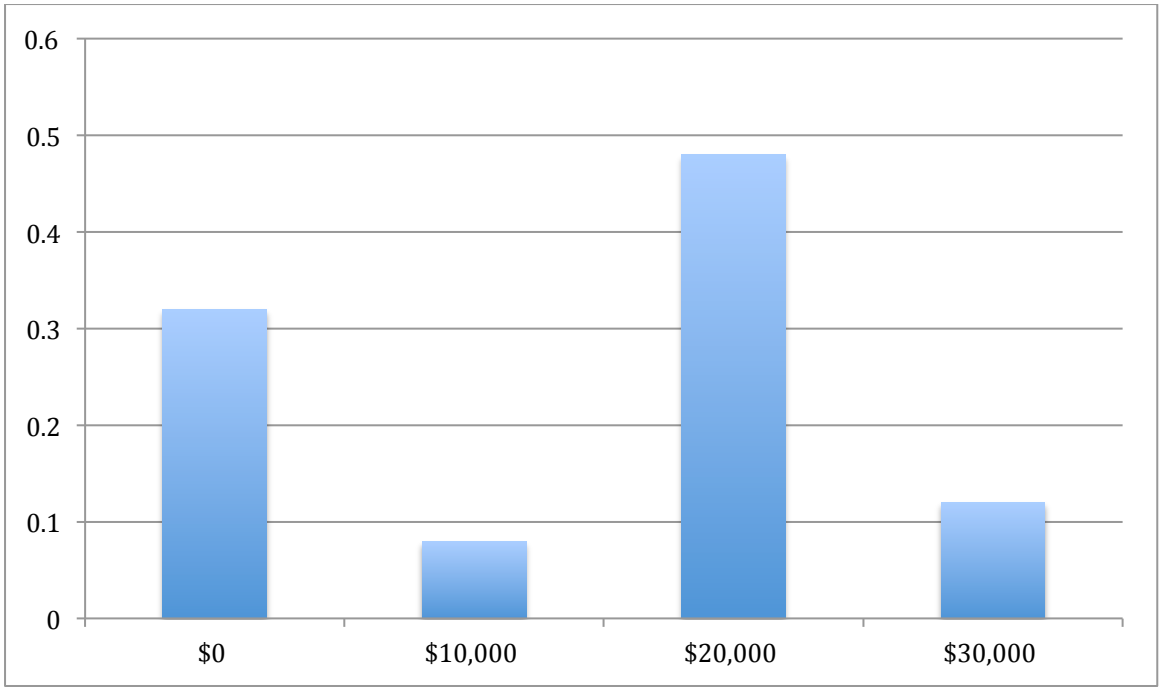
Software risks involve uncertainty and loss, therefore things need to be quantified so we can handle them correctly. For this, we can use the binomial distribution and Monte Carlo Simulation.

Monte Carlo Simulation

The Monte Carlo Simulation is a method of producing discrete random variables as input for simulations.

Before the simulation commences, we define a range of possible events that can happen and assign them a probability. Based on the probabilities of these events, we find the possible values that the random variable we will use in the simulation can take. Some of these values will map to the event in such a way that it matches the probability for that event. A random number generator (such as a dice) is then used to produce these random values.

A Risk Management example for this would be if we had 2 risks; Risk A has a 0.2 probability of occurring and will cost £20000 if it happens, Risk B has a 0.6 chance of occurring and will cost £10000 if it occurs. This means we can have a 5-sided fair die to represent the values of a discrete random variable. Risk A will have the value of ‘1’ assigned to it and Risk B will have ‘1’, ‘2’ and ‘3’. Each event will have a roll of the die in each ‘round’. If we roll a ‘1’, then Risk A will occur and cost £10000. The next roll will be for Risk B and if it rolls a ‘4’ Risk B will not happen. This round will cost the project £10000. This process is repeated several times to build a histogram of the values (below):



Interesting Points

From the CMM lecture, when you check the quality of a product, then you need to check the quality of the entire production line. If the materials you are using are of high quality and are coming from high quality suppliers, then you will probably have the foundations for a high quality product. This is similar to that of the Ford Transit production line located near Southampton. This technique could be applied to software too. The key difference here is that software is not physical and is trivial to replace, where as Transit vans are not.