G53SQM Revision Notes

Craig Knott

Version 2.0

January 20, 2014

Contents

1 Introduction

Software engineering is much more chaotic than physical engineering for one simple reason. Half way through manufacturing, a physical product cannot be entirely changed, however in software development, it is very common for very large transformations of the software to occur. This module covers the best processes for software development, development in team working practices, proficiency and understanding of automated building and testing processes, and to help understand and adopt skills that will help with future software development work.

2 Certified Software Quality Engineers

Software quality engineers are required to have a few key characteristics, to ensure that the work that is completed by them and their team is of the highest quality possible. There are several main aspects to consider, and these are evaluated below:

- Software Quality **Management**
  Processes and activities to set strategic quality **goals** and **objectives**

- Software Quality **Engineering**
  Definition, planing and implementation of quality management goals. It also incorporates ways that we **prevent defects** and **build quality** into the software

- Software Quality **Assurance**
  Planned and systematic set of things needed to provide adequate **confidence that our processes work**

- Software Quality **Control**
  Planned and systematic set of actions and activities to monitor and measure the **conformance of our product to our goals**

- Software **Verification and Validation**
  Checking that products meet their **specified requirements** and **intended use**.

Some soft skills that a software quality manager should have: Leadership, Team building, Communication, and Conflict resolution.

3 Scrum

Scrum is a popular project management technique, which involves small teams or sets of teams and is better for less structured problems that have less defined requirements. It allows for work in changing requirements, allows us to product higher quality products, estimate better, and be more in control of the entire development process. The disadvantages are that is can be rather intensive, and there is a considerable cost to the customer to meet with the team very frequently. Scrum has a clearly defined life cycle: Sprint Planning Meeting, Sprint Work Schedule, Daily Scrum Meetings, Sprint Retrospective, and Slack time. There are three key members to a scrum: the product owner, some one who is a stake holder in the product and will determine what is most important to work on next; the scrum master, the person who ensures the scrum is progressing smoothly; and the development team, the people who will actually be working on the project. Scrum values Transparency, Inspection and Adaptation.

3.1 The Product Backlog & Stories

The product back log is a prioritised list of stories that will be used for the life time of the project. Stories, in the context of scrum, are a brief description of some task that the product owner wants the system to do. This is given a description, an ID, a name, a priority (or importance), a way to test or demo it, and an estimate in man-days or man-hours. These should be written in such a way that the product owner can understand what they are doing, and the development team can use them to produce the features. These initial estimates can be done with several techniques, two of which are Planning Poker, and Wideband Delphi. These methods both help all the members of the team understand the software more, and are easy and cheap to set up.

3.1.1 Planning Poker

Each member of the team is given a set of 13 or so cards, with numbers on each one (a non linear scale from 0.5 to 100, normally). One card has a coffee cup, and another has a question mark. Each story is pitched, and then the team place their estimates of man hours face down. Once all members have selected a time, they are all revealed, and an estimate is taken from an average of these. It is done face down so that no previous prediction can influence others. The question mark cup is used when you are unsure as to the time it should take, and the coffee cup indicates that we should take a break!

3.1.2 Wideband Delphi

Similar to planning poker, but much less restrictive. Instead of using cards, estimators simply write down their estimates. Each is then given chance to justify their choice - without telling the others what it was. This process is repeated until the estimates converge.

3.2 Sprint Planning Meeting

Before the sprint planning can begin, it is vital that the product backlog has been completed! This meeting is then the most important event in the entire scrum process, because it sets up the sprint to either succeed or fail. For this reason, it is important the product owner is present in this meeting, as their input is important, and they are the one who knows what they want the system to be. During this time, three aspects of the stories in the backlog are discussed: scope, estimates, and importance. At this point the product owner, and the development team discuss what features should be produced first, how long they should take, and what they entail. Notice that "quality" is *not* a variable here; this is because quality is not-negotiable. The product owner may attempt to convince the development team to cut corners on the internal quality, so that the product can be completed in time - however, you cannot have good external quality[1], without good internal quality[2]. This is why this meeting takes place; if the product owner wants a specific feature completed, we can potentially reduce the scope of this feature so that we can get it finished on time, or re-prioritise some other tasks - communication is key. Before the meeting is over, we estimate what tasks we will be able to complete during the sprint. We use two formulae for this, as we must calculate focus factor, and velocity.

```
Velocity = Last Sprint's Focus Factor * Man Hours
Focus Factor = Velocity / Man Hours
```

Velocity tells us how much we feel we can get done, and focus factor tells us how effectively the team are working.

[1] Quality as observed by the user
[2] Quality as observed by the developers (i.e. the code itself)

"Last Sprint's Focus Factor" is an example of the use of "Yesterday's Weather". Which is when we predict using the information from previous sprints. We use the formulae to decide how much work we can complete, and we pick as many of the high priority tasks as we can, that does not exceed this total time. To summarise, we discuss "CWEBS": Clarifications, Where/When the daily meeting takes place, Estimations, Backlog summaries, and Stories for this sprint.

3.2.1 How we prioritise

It is important, during the planning process, that we consider what should be completed first. This generally boils down to the four following variables: difficulty, prerequisites, what will the customer want to see, and who should be deciding these. The product owner and the development team generally work together on this, so that they can modify tasks and their time requirements as necessary, to ensure both parties are happy.

3.2.2 Importance

Importance and priority, whilst both expressing the same thing, are different from one another. When talking about priority, we say that the first item we should complete, is priority 1, and the last, priority n. However, when we then determine another task is even more important, the number scale begins to become a little less aesthetically pleasing. Importance on the other hard, works in the opposite way. The item to be complete first has importance n, and the product to complete last, has importance 1. This means that we can very easily add new elements to any where in the scale. Importance is generally the industry standard.

3.3 The Sprint Itself

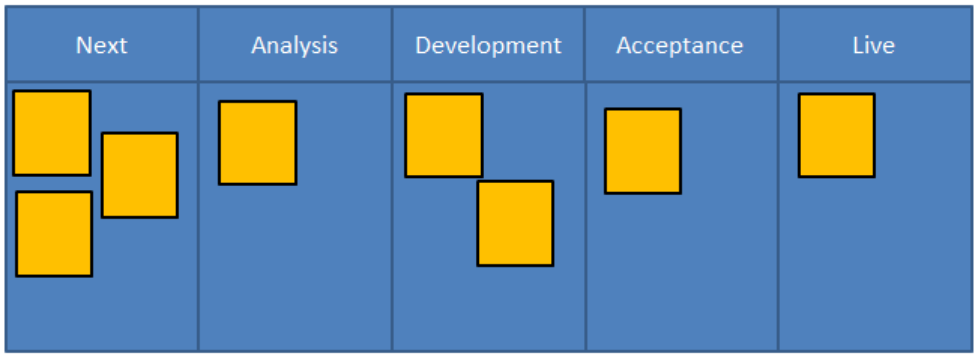
After all the planning has taken place, the actual development cycle can begin. The ideal length of this cycle is very much dependent on the teams, and the products, but a decent estimate is 3 weeks. Longer sprints allow for more momentum to build and for less administrative overhead, but shorter sprints can receive feedback much quicker, allowing them to learn and improve more frequently. During the sprint, the prioritised tasks assigned to this team from the sprint planning meeting are put on the sprint backlog. This is generally a kanban, with an associated burn down chart. These are then used to monitor the progress of the team, and visualise work flow. Each day of the sprint, a scrum meeting takes place. This meeting addresses four key points: What was completed yesterday, what should be completed today, what issues are you having, and updating of the Kanban and burn down chart. This meeting is useful so that all of the team members can be updated on the progress of the overall project, and they can acquire any help if they need it.

3.3.1 Burn Down Chart

A burn down chart is a graph that plots the estimated number of man hours remaining until the sprint's completion, against the dates in the sprint. At each scrum meeting, the burn down chart is update, to show the progress of the previous day, and to show how much work remains. It is then possible to calculate a trend line from the plotted points, known as the project velocity. This can then be used to determine if the project will be completed on time or not. A very steep project velocity means the project is progressively potentially too quickly, and some new tasks should be added, and a very shallow velocity implies there is too much work to complete, or the team have a very poor focus level.

3.3.2 Kanban

Kanban (literally "signboard" or "billboard") is a scheduling system for agile product. Kanban is a list of the stories that need to be completed in the split, which are split into categories, generally "Next", "Analysis", "Development", "Acceptance", and "Live". These sections of the board indicate how close to completion each of the tasks is. This helps to visualise the progress and work flow of the project, and any one from the team (or even the product owner), can easily determine the state of the current sprint.



3.4 Sprint Demos & Testing

The sprint demo is a very important part of the scrum process. In the demo, the team that worked on a particular story can demonstrate it functioning to the rest of the teams and the product owner. This has many benefits, including the team feeling good about themselves, for completing the work and receiving recognition for it, and also allows the rest of the teams and the product owner to see how far the project is coming along. It also helps to ensure that features are 100% finished and can be demonstrated accordingly, which stops teams from mostly finishing stories, and leaving them slightly unpolished, which causes complications later.

3.5 Sprint Retrospective

The sprint retrospective is a time after the sprint has completed, for the team to reflect on the sprint. The teams discuss what went well in the sprint, what did not go so well, and what improvements they can make for the next sprint. The team then actually vote in some manner on some of the improvements to specifically focus on in the next sprint, so that they are constantly improving. This is similar to the Japanese concept of Kaizen. This is also very similar to "Total Quality Management", combined with the "Defect Prevention Process", and "Six Sigma".

3.5.1 Kaizen

Kaizen, Japanese for "improvement" or "change for the best", refers to philosophy or practices that focus upon continuous improvement of processes in manufacturing, engineering, and business management. This is generally achieved by evaluation of past work, to determine what went well, what didn't, and what can be done to ensure the same mistakes are not made again. This conscious effort to focus on issues helps to improve the entire team, especially as they are able to work together to fix these issues.

3.6 Six Sigma

Six Sigma states that there should only be 3.4 defects in a piece of software, per 1,000,000 lines of code it has. This comes from standard deviation (hence the sigma), where 3.4/1,000,000 is the number of errors that a system has six standard deviations from the mean ($\pm$ 1.5).

3.6.1 Total Quality Management

TQM is the process of attempting to maximise quality, by focusing on customer satisfaction. There are four main steps to total quality management:

1. Customer Focus
2. Process Improvement
3. Human Side of Quality
4. Measurements and Analysis

3.6.2 The Defect Prevention Process

The DPP is a very simple set of rules that can be used to help improve the quality of a process or product, in its next iteration of development. It simply states that after completing work, we should evaluate how we did so, and identify any errors or mistakes that were made, or were an issue. We then design some strategy to combat this issue, at it's root, and then implement this strategy during our next work process. This helps us to constantly improve ourselves, and the quality of the work we product.

3.7 Slack Time

Slack time is a time in-between the end of a sprint, and the planning of the next sprint. This team is used to show that the members of a development team are appreciated, and to give them a short break from the intensities that is sprinting in a scrum. This time is usually allocated for the team to work on their own projects, or do some research into new technologies, so that they are still begin productive, but are not in a strict environment of work.

3.8 Potential Issues in Scrum

3.8.1 Disadvantages

As mentioned before, scrum is very intense. Teams are expected to work very hard during the sprint times, to ensure that all the stories they said they would be completed, would be. This is different to normal software development, where the team gives a ballpark figure of a year, and can work however they want for that time. In scrum, the product owner is constantly evaluating the system, seeing demonstrations, and can easily track the progress of the entire product. This means that there are far more, and far shorter deadlines than in most development methods. This also brings up the issue of cost for the product owner. As they must be with the development team for so much, the cost, in terms of both money, and time, is very large. This does end up producing them a much higher quality piece of software that is much more suited to their needs, but the cost of the software, and of the development process, can add up.

3.8.2 Scrum Variables

There are many, many variables and processes in place when working in a scrum, and unfortunately, there is no perfect set that every team can adopt. How many members should there be per team? How long should sprints be? What should the sprint goal of a sprint be? These questions all require much consideration, and it can often take multiple attempts at sprinting before an ideal set for a specific team can be identified. For this reason, the first few sprints of a scrum process can sometimes be quite unproductive, and the team may feel as though they are failing.

3.8.3 Location of teams

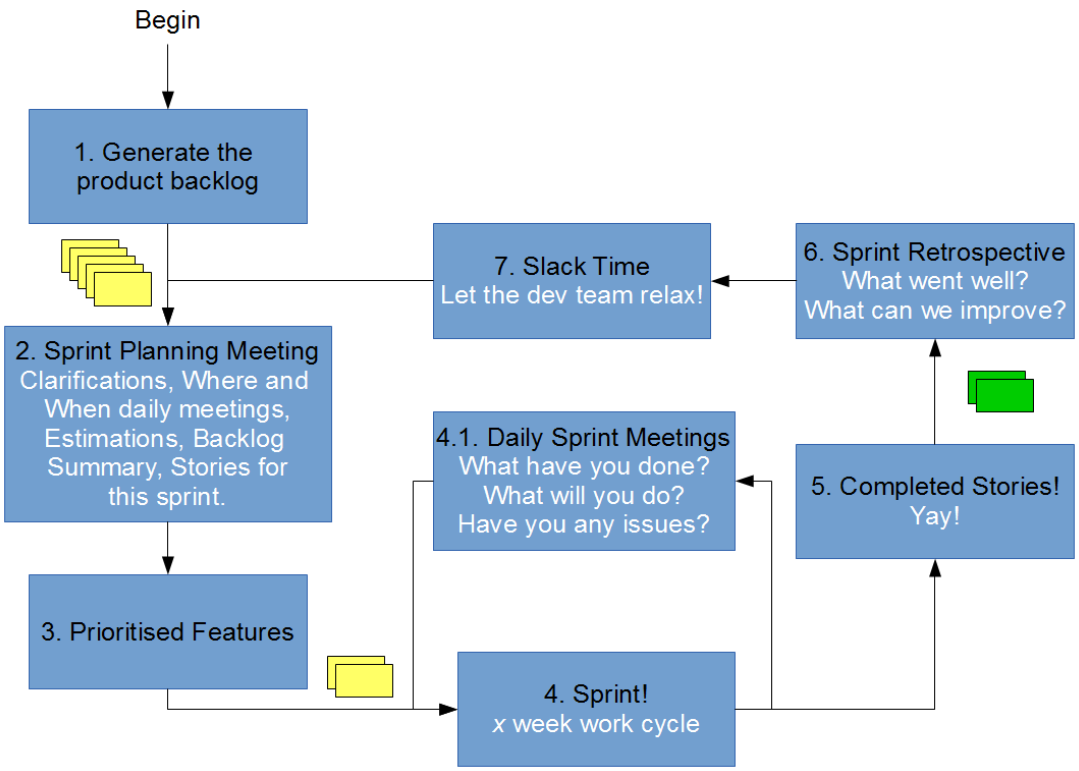
The majority of the benefits of scrum (especially when merged with XP), come from the co-location of the team. This can mean the geographically distributed teams may not work. However, as with most things in scrum, this depends entirely on how the scrum master deals with this issue and how they set up the teams accordingly. It is always important to keep the off-site members of the team involved, and not to give them a separate team, to promote communication in the scrum and to create a stronger team dynamic. It should also be noted that co-located teams should be able to see each other, and the board, hear each other, and be far enough away that they do not disturb other teams, or are disturbed by them. They should, however, be within walking distance of the product owner.

3.8.4 Dysfunctions of teams

There are five main dysfunctions that teams can have, and these should be carefully evaluated and resolved, to ensure that the team work together as well as possible. These dysfunctions are: (and can be remembered as: As Far As I Low (AFAIL) - a play on AFAIK (as far as I know)).

1. Absence of Trust
2. Fear of Conflict
3. Avoidance of Accountability
4. Inattentions to Global Success
5. Lack of Commitment

3.8.5 Scrum Life Cycle Diagram



4 Scrum and XP

Scrum is a project management framework[3], and XP (Extreme Programming) is a set of techniques employed to improve programming proficiency, and quality. The two can, therefore, very easily be combined and used to complement one another. Extreme Programming values Simplicity, Courage, Respect, Feedback and Communication. It is important, however, to consider the disadvantages of Extreme Programming, which are: a large cultural change to implement, cost to the customer, and it can potentially be inefficient.

4.1 Pair Programming

Pair programming is the process of having two developers working on the same feature at the same time. One of them programs, whilst the other offers help, advice, and looks for bugs and issues. This allows a great deal of knowledge sharing between the pair, helps reduce bugs, and improve the quality of the code. The issues with it are that the team members have to decide when to work together, and where, and that sometimes the dominant programmer will take over, completely negating the benefits of the process.

4.2 Test Driven Development

Test driven development is the process of writing a test of the feature you want, and then writing the code to pass this test. This inspires confidences, promotes a simple design, and naturally, reduces the number of bugs present in the system. The disadvantages are: that programmers like programming, not testing; that the initial set up is large; writing tests takes time; and it is very difficult to test for full completion. However, TDD does allow for adequate regression testing[4], due to the backlog of tests we produce.

4.3 Collective Code Ownership

The use of pair programming, and the frequent changing of partners to allow all members of the team to familiarise themselves with all aspects of the code base. This means that if a key member of the sprint is ill, the sprint is not in any danger, as all members of the team have knowledge of the entire software system.

4.4 Continuous Integration

Continuous integration is the use of a server that builds and tests the entire code base on a multitude of platforms, solving the "it works on my machine", issue, once and for all. This is generally used when you need to test on multiple different set ups, and there are multiple people working on separate features at the same time. The idea of continuous integration is to regularly merge your code with the mainstream, so that integration is much simpler. This also means that errors can be spotted much earlier, and it allows teams to see what affect their features have on the rest of the system. The disadvantages of continuous integration are that it takes a very long time to set up, and the tests have to be in depth and sophisticated, otherwise there is not worth having.

4.5 Coding Standard

Coding standards are a set of rules that the programmers in a team must abide by. This helps to create consistency within the code, and the other members of that team should then easily be able to identify what any segment of code is doing.

[3] Because it does not tell you exactly what to do

[4] Ensuring older tests do not fail due to adding new features

5 The Consumer Bill of Rights

The consumer bill of rights is a list of 10 rights that all consumers of software products have. There is no need to memorise all the of the points, however, it is important to know the general tenure of the rights. If you do wish to memorise the bill of rights, I find it easiest to remember it in the following way:

Centre for **D**isease **C**ontrol, **I**nstitution of **A**rt and **L**aw, **C**athode **R**ay **T**ube, **G**overnment. Which translates to:

1. C: Contracts
   The consumer should be allowed to see the contract before the sale.
2. D: Defects
   Any defects of the software should be made known.
3. C: Claims
   Any claims made about the software must be factual.
4. I: Information
   The consumer has the right to know, and dictate, the information of theirs that is used.
5. A: Access
   The company cannot prevent the consumer from accessing his/her personal information.
6. L: License
   The company cannot prematurely terminate the consumer's license.
7. C: Criticise
   The consumer is free to criticise, publish benchmarks, and have fair use of the software.
8. R: Reverse
   The consumer has freedom to reverse engineer the software.
9. T: Transferable
   Any mass market software should be transferable.
10. G: Govern
    All software within a product is governed by the laws that govern the product.

6 Git

6.1 About Git

In software development, Git is a distributed revision control and source code management (SCM) system with an emphasis on speed, which we use when we having more than one developer working on a project. The advantages of Git are that it is fast, and scalable, and that every Git working directory is a full-fledged repository with complete history and full version tracking capabilities, not dependent on network access or a central server. The disadvantages are that sometimes merging can be difficult, and that it is easy to accidentally delete work.

6.2 Commands

git clone $< url >$
Get a copy of project $< url >$, so you can look at or use the code

git branch
With no arguments, lists all branches
With a non-existent branch name, creates a new branch with the specified name
With $-v$, lists the last commit on each branch
With $-d$ and $branch - name$, deletes the specified branch

git checkout $< branch - name >$
With $-b$ and non-existent branch name, creates a new branch with the specified name and switches to it

git pull
Fetch from a remote repo and try to merge into the current branch
With $-f$, forces the merge

git add
Adds files and their contents to the staging area
"." or "*" as parameter to add all files
With $-A$ to add all files

git commit
Records a snapshot of the staging area
With $-a$ to add all files
With $-m$ to leave a commit message

git push
Push your new branches and data to a remote repository

git merge
Merge a specified branch into your current one

git fetch
Download new branches and data from a remote repository
With $--all$, gets all branches

git reset
Undo changes and commits
With $HEAD$ to undo the last commit and unstage files
With $--hard$ to undo the last commit and unstage files, and any changes in working directory

git reset
Show commit history of a branch

7 Metrics

`http://arisa.se/compendium/node88.html`
A metric is a quantitative measure of degree to which a system, component, or process possesses a given attribute (for instance, error density). This is a derived measure, and may often involve estimations where human processes are concerned. A measure, in this context, is defined as a quantitative indication of capacity, amount, size, extent, or dimension (CASED) of some attribute of a product or process (for instance, lines of code, numbers of errors, etc). You cannot get good external quality of software, if the internal quality is not also good.

The reason we measure software, is to determine the quality of the current product or process so that we can make informed comparisons. We can also predict the qualities of a product or process, and from there, improve on these qualities. We generally measure our metrics per person, per team, and per project, so that we can see origins and types of errors and defects in the product.

The metrics themselves can be evaluated on: products, which are the explicit results of software development activities, which have some deliverables, documentations and other artefacts produced; processes, which are activities related to the production of software; and even resources, which are inputs into the software development activities, such as hardware, knowledge, or people. The table below summarises some of the metrics that can be used to evaluate processes and products.

In general, there are two different types of measures: direct, and indirect. In direct measures, there is some quantifiable value to measure, like cost, speed, or lines of code. Indirect measures, however, need some sort of human estimate, such as functionality, quality, reliability and maintainability.

7.1 Size-Oriented Metrics

Size oriented metrics are concerned with the size of the software produced. These generally look at things such as Lines of Code (LOC), or Kilo-Lines of Code (KLOC), or Statement lines of code (SLOC, ignores whitespace). Typical measures for this include, errors/KLOC, defects/KLOC, cost/LOC, documentation pages/KLOC.

Line of code metrics are very easy to use, and easy to compute. However, they are also language and programmer dependant, and can be it counter productive if programmers are rewarded for having a high number of lines.

Lines of code can also be used to measure complexity, using *Halstead's Software Science*, which is a measure of entropy. To calculate the complexity, we use four variables:

n_1	number of distinct operators
n_2	number of distinct operands
N_1	total number of operators
N_2	total number of operands

For an example, see the following code fragment:

```
if (k < 3){
   if (k > 2){
      x = x * k;
   }
}
```

In this case, the distinct oper*ators* are: if, (,), {, }, >, <, =, *, and ;
The distinct oper*ands* are: k, 2, 3, and x. This gives our variables the values:

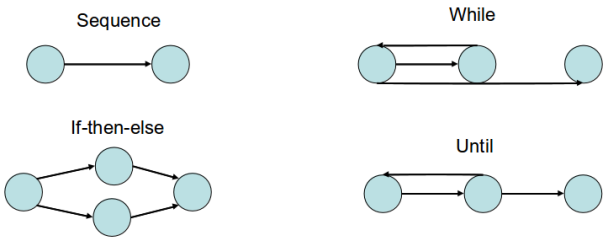
$$\begin{array}{llll} n_1 & 10 & N_1 & 15 \\ n_2 & 4 & N_2 & 7 \end{array}$$

Using these values, we can now work out various metrics that Halstead defined:

Program Length	$N = N_1 + N_2$
Program Vocabulary	$n = n_1 + n_2$
Estimated Length	$\hat{N} = n_1\ log_2\ n_1 + n_2\ log_2\ n_2$
Purity Ration	$PR = \frac{\hat{N}}{N}$
Volume	$V = N\ log_2\ n$
Difficulty	$D = \frac{n_1}{2} * \frac{N_2}{n_2}$
Program Effort	$E = D * V$

7.2 McCabe's Cyclomatic Complexity Measures

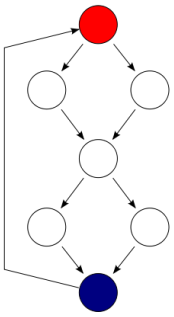
McCabe defined a set of metrics that are based on a control flow representation of the program being evaluated. A program graph is used to depict the control flow, in which nodes represent processing tasks (one or more code statements), and edges represent control flow between notes. See the diagram below for some examples:



Using these graphs, we can calculate the cyclomatic complexity of the program. This is a count of the number of linearly independent paths through the source code. For instance, source code with no control statements, would have a complexity of 1. However, introduces a single if statement into this code would increase the complexity to 2, as there are now two paths to traverse. This is represented by the formula:

M = E - N + 2P
where:
E is the number of edges
N is the number of nodes
P is the number of connected components (exit nodes)

For the following control sequence/source code fragment, the cyclomatic complexity would be 3 (as there are 9 edges, 7 nodes, and one connected component; 9-7+1 is 3).



```
if ( c1 ) {
  f1();
} else {
  f2();
}

if ( c2 ) {
  f3();
} else {
  f4();
}
```

7.3 McClure's Complexity Metric

McClure's Complexity Metric is much simpler than McCabe's, and simply adds the number of control variables (V), to the number of comparisons (C).

```
Complexity = C + V
```

7.4 Commit Metrics to Memory

If you wish to attempt to memorise some metrics to take into the exam (which is a good idea, because there may very well be a question on them), a compact way of remembering at least 13 is the following:

MMH SLEWFH LoC

(Where MMH sounds like the famous Starcraft player MMA, SLEWFH is a butchered spelling of sleuth, and LoC is just kind of there). These represent:

1. MMH
   McCabe, McClure and Halstead (the three complexity metrics defined earlier)

2. SLEWFH
   Size of the program, Load time of the program, Execution time of the program, What The Fucks/Hour[5], and Failures/Hour (the H is common to both)

3. LoC
   LoC is obviously Lines of Code, which breaks down into: Errors/LoC, Defects/LoC, Documentation/Loc, and Cost/LoC.

7.5 Measures of software quality

A common abbreviation used to evaluate a quality of a system is FURPS:

Functionality	Features of the system
Usability	Aesthetics, documentation
Reliability	Frequency of failure, security
Performance	Speed, throughput
Supportability	Maintainability

[5]How many times you exclaim, "WTF?" whilst looking at some code

However, this is certainly not the only set of rules to look for, other common encompassing terms include:

1. Correctness
   The degree to which a program operates according to the specification. This includes such metrics as Defects/KLOC, and Failures/Hours of operation

2. Maintainability
   Degree to which a program is open to change. With metrics such as average time to make a change, and the cost to correct.

3. Integrity
   The degree to which a program is resistant to an outside attack, observing the fault tolerance, security, and threats.

4. Usability
   How easy the software is to use, taking into account training time, skill level necessary to use, and increase in productivity from adopting the software.

7.6 Conclusion

Software metrics can give much useful information about the quality of code. The y allow comparison between projects, developers and team, over time. They suggest areas for improvement and give early warnings of problems (as in manufacturing process metrics). However, it is important that they are interpreted with some care as not all projects/tasks are equal. The management team need to know what quality problems exists, but developers should know what needs correcting.