Software Engineering: A Definition

What is software engineering?

"…The establishment and use of sound engineering principles in order to obtain economically, software that is reliable, maintainable and works efficiently on real machines".

What have we been doing so far?

In simple terms most of us have simply been *programming*. That is, given a problem amenable to a software solution, most of us immediately delve straight into writing the solution (or the perceived solution) using the software tools and programming languages to hand, or ones that we are familiar with, with the following shortcomings.

- Little or no analysis of the problem is performed.
- Very little effort is expended on *designing* software structure or architecture.
- Little effort is made to validate or test our software in a rigorous manner.
- Maintenance is something that is ignored until later, hopefully indefinitely until you have left the company and it becomes somebody else's problem.

In essence we have been *hacking.*

"…But what's wrong with that, we got it to work didn't we?"

Ok, so you got it to work, *this time*. But what happens when the problem is *not* trivial or small and cannot be comprehended by one person. Perhaps it will involve a team of 20+ programmers. What happens when the deadlines are too short for one person to '*wing it*' and come up with a solution that is (just) acceptable to the customer (or lecturer marking it). That's when hacking starts to fail and *software engineering* is required.

So What Does Software Engineering Involve?

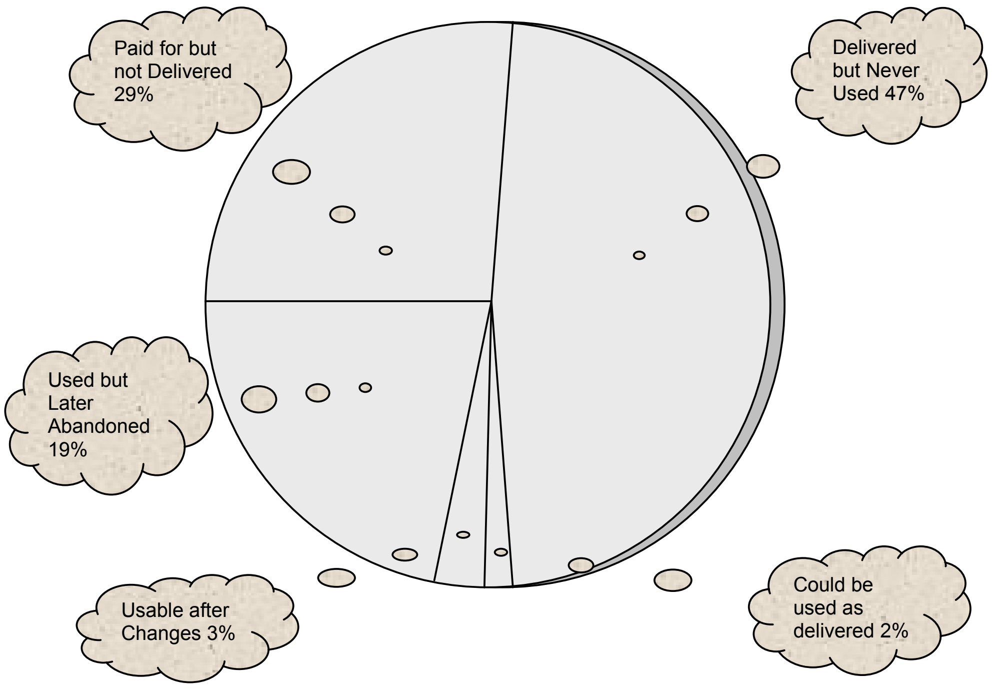
Apart from the above definition, software engineering is a set of analysis, design, programming, testing and maintenance techniques that have evolved over a number of decades to facilitate the creation, testing and maintenance of complex, high quality software.

It has been shown that such techniques, used correctly lead to the creation of software which is more likely to meet the needs and specification of its users and more likely to be produced on time and within budget. Such software is also likely to prove itself more reliable and maintainable than any "hacked" software.

I don't *really* need to learn any new techniques – Do I?

If you question the need for such techniques and think that you can "wing it" and survive, take a look at this graphical presentation shown below, which demonstrates the problems software production faced in the 70's when software *engineering* was rarely used.

This pie chart was produced by the American Department of Defence (DoD) and summarises the results of government contracts for software development projects delivered in the late 70's and early 80's.



Now imagine that you were the customer and the developer was say Ford Motor company and you were purchasing 100 ford Mondeos, how would you feel if only 2 of them worked as delivered? How could this have happened? Some common excuses are given below

- Incorrectly Analysed or specified Systems. Maybe the developer never consulted with the user in an attempt to really understand what they wanted. Of if they did, did they make an attempt to keep up with the possibly changing needs of the user?

- The project was mismanaged and could not be delivered on time.
- The product was not designed correctly to implement the specification.
- The product was out of date or too expensive by the time it was delivered.
- The product was not adequately tested prior to release.

Haven't we progressed since then?

In theory yes, but in practice, software is still not being developed to the same high standards as other engineering disciplines such as electrical, mechanical or civil. To give you some indication, here is a list of failed software developments to have grabbed the news headlines in the late 80's/90's

Classic Software Failures of the Late 80s- 90's

Mars orbitor 1999: This joint NASA/European Space Agency project crashed onto the surface of mars (instead of orbiting it) because NASA used imperial while ESA used metric.

Ariane 5: It's maiden flight on June 4, 1996 ended in the launcher being exploded because of a chain of software failures.

(See: http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html for details)

Taurus: A planned automated transaction settlement system for the London Stock Exchange. The project was cancelled in 1993 after having lasted more than five years. The project cost was around £75m; the estimated loss to customers was around £450m; and the damage to the reputation of the London Stock Exchange was huge. (See *Crash* by T.Collins)

Nimrod AWACS system: In the UK, the development of a Nimrod based Advanced early warning defence project was abandoned after countless 10's of millions of pounds of tax payers money had been spent on the project. It was estimated that due to mismanagement and slipping time-scales, the project would have been out-of-date by the time it had been introduced. Furthermore, the cost of making it work was greater than the cost of buying an American based system which already worked.

Denver: A baggage handling system where a complex system, involving around 300 computers, overran so badly as to prevent the airport opening on time, and then turned out to be extremely '*buggy*' and cost almost another 50% of its original budget of nearly $200m to make it work.

London Ambulance System: Where because of a succession of software engineering failures, especially defects in project management, a system was introduced that failed twice in the autumn of 1992. Although the monetary cost, at 'only' about £9m, was small by comparison with the other examples, it is believed that people died who would not have died if ambulances had reached them as promptly as they would have done without this software failure. (See "Crash" by Tony Collins)

Therac-25: Between 1985 and 1987 six people (at least) suffered serious radiation overdoses because of software-related malfunctions of the Therac-25 radiation therapy machine. Three of them are thought to have died of the overdoses. An important root cause was a lack of quality assurance, which led to an over-complex, inadequately tested, under-documented system being developed.

(See: http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html for full report)

Stanstead Airport (If I recall correctly) is still trying to develop a radar tracking system to monitor aircraft flying over UK and much of Europe. It is behind schedule and over budget.

Passport and N.I/Benefits Office. The backlog due to the issuing of new passports lead to lost holidays and compensation claims, plus over 1500 bugs were reported in the NI computer system

Year 2000 Bug: How much do you think this will end up costing?

For more modern examples, take a look at
http://www.baselinemag.com/print_article/0,3668,a=35839,00.asp

Aims and Objectives of this Course

The aim of this course is to teach the student how to *Analyse*, *design* and *engineer* software. It is **not** predominantly about programming or languages, although that is an integral part of the engineering process. Rather, software engineering is a set of techniques that will lead to the design of well-engineered, reliable and maintainable software systems that can evolve to meet the customer's present and future needs.

Software

Where does software fit into this design and engineering process? A textbook description of software might take the following form. Software is:

(1) Instructions (computer programs) that when executed provide the desired function and performance.
(2) Data structures that enable the programs to adequately manipulate information e.g. databases, spreadsheets etc.
(3) Documents that describe the operation and use of the programs.

A Comparison of Software *Behaviour* vs. that of other Engineered Products

To gain an understanding of software and ultimately an understanding of software engineering, it is important to examine the characteristics of software that make it different from other things that human beings build and engineered.

1. ***Software products do not have to abide by, and cannot be influenced by the laws of nature such as physics or chemistry.***

In effect this means that the usual *metrics* i.e. methods of measurement and assessment, that have been used to test the quality or functionality of say a mechanical or electrical product cannot be applied to a piece of software. In particular, you cannot, for instance use a Micrometer or an Oscilloscope to assess the quality, precision or accuracy of a piece of software.

Furthermore, it is possible to 'mash' together pieces of software to make the product work because there are no laws of physics to prevent it falling over. Try doing the same things with say a bridge and it'll end in tears and a lawsuit.

2. *Software Quality is hidden from its Users.*

Generally speaking, the customer only gets to see the "*outside*" or "*users eye view*" of the final software. The inner workings of software are not amenable to examination. The individual software *components* and the *boundaries/interfaces* between them are hidden. This is because the compiler has translated the original design into something completely different comprising 0's and 1's stored as pits on a CD ROM. Reverse engineering a software product to assess it's initial design quality is virtually impossible.

This is perhaps why hackers have existed for so long, as it's difficult to assess the quality of the product they produce. To them the motto is

> *"…As long as the product works, who cares if it's good or bad?"*

At the end of the day, the customer cannot tell whether they have been presented with a well-engineered product, or a "*lash-up*" put together by cowboys. Furthermore, once delivered, the quality of the product can only be assessed in terms of its *functionality*. Software contains no "*user-serviceable*" parts and nothing is interchangeable.

Contrast this with a motorcar for example where you can lift the lid and examine/replace the components such as engines and gearboxes, because the interfaces between the components are clearly identified (they are loosely coupled) and each has a well-defined and distinct role (they are highly cohesive).

3. *Software is often discreet in its behaviour*

In many ways software is inherently **unstable** as it often behaves like a *multi-way switch,* producing radically different outputs/results in response to even small changes in its data**.** This kind of behaviour makes testing particularly difficult.

For example the behaviour of a medical diagnosis program designed to ask for data about patients symptoms might come up with a completely different diagnosis if it knew the sex of the patient.

Mechanical devices by comparison are more *linear*, exhibiting predictable and well-defined behaviour throughout their range of acceptable inputs. There are few, if any discrete jumps in behaviour.

In summary, software's behaviour is not governed or restricted by any '*natural'* rules in the sense that a physical object is. In essence this means that the ability of software to produce either sensible or incredibly stupid results is limited only by the imagination and creativity (or stupidity) of the designer or programmer. There are no limits to what can be achieved.

4. Mechanical devices often recover automatically from overloads

For example, simply lifting off the throttle can cure over-revving a car. The engine will then return to a steady state. Once software is exposed to an overload, or data that is outside its specified range, it often falls into a heap requiring a reset to recover. This places a huge burden on the designer to include additional software to check the validity of all data and reject it before it gets processed.

One estimate is that 80-90% of all software written is there to validate and deal with user entered and intermediate (i.e. the results of calculations) data. Only 10-20% of code is written to actually implement the specification.

5. Software, until recently, could not be proven to work

More and more software is now being used in Safety-Critical applications, such as fly-by-wire aircraft or nuclear reactors. Unlike mechanical or electrical circuits for which there exists well-known equations for predicting and proving component and system behaviour (and indeed failure), no such metrics existed until recently to prove that software had been implemented in accordance with a specfication.

New formal mathematical techniques such as 'Z' and VDM, based on 1st order predicate calculus (complex set theory in simple terms) are only just emerging as a technique for specifying software requirements, but these techniques are complex. Furthermore, the uptake in their has been slow, even in fields such as safety critical systems where it could be argued that they would have most benefit. There are a number of reasons for this

1. Most safety critical software systems fail because the system has been incorrectly specified, miss-understood or incorrectly implemented. In order to use formal mathematics to specify the behaviour of the system, both the software developer and more importantly the customer both have to conversant in Z or VDM. This is often unrealistic and customers will not commit themselves to a legally binding specification they cannot understand.
2. The cost and time of specifying in Z or VDM is very large compared to less formal techniques.
3. Even if the system is specified in Z or VDM, there is no guarantee that the implementation of that system will be as rigorously performed. Indeed there is little chance that the implementation can even be formally checked against its specification.
4. If safety is the ultimate overriding concern, then there are many other factors to consider beside the software. For example, can you prove that the CPU in your computer behaves exactly in accordance with its specification at all times? What about the compiler and OS, do you have faith in them?

6. *Many Mechanical and Electrical systems often have well defined working boundaries and are constrained to work within them*

This makes it relatively easy for their designer to assess the suitability of their design for its intended purpose. For example, the mechanical designer of an engine for a car need only consider the interface between the engine and gearbox. That is, there is a well-defined, explicit specification about which the designer can explore different solutions.

In contrast a great deal of software is often designed with few and sometimes no boundaries placed upon it, restricted only by the ability of an individual to interpret/specify them, which makes designing such software to cope with all eventualities extremely difficult. Here are some classic examples: -

1. NORAD, the American computerised nuclear defence program, during it's 1st 24 hours of service, was taken to its highest state of alert (DEFCON 5), one step away from a launch, because software used to detect incoming threats had mistaken the earth's moon, emerging over the horizon as an all-out soviet nuclear attack.

2. When the UK developed it's first intelligent 'seek and destroy' intelligent torpedo, the programmers thought they were being clever when they built safeguards into the software that would automatically self-destruct the torpedo if it ever did a 180 degree turn and started heading back towards the submarine that launched it. Unfortunately, one of the 1st launches of the new torpedo resulted in it getting stuck in the subs tubes, so the captain turned around and headed back to port!!!

3. More recently (1999), an airbus A320 aeroplane overshot the runway coming in to land in monsoon conditions (i.e. extremely heavy rain). This particular type of aircraft uses computer controlled reverse engine thrust to assist in bringing the aircraft to a halt. However, when the plane touched down, there was so much water on the runway that the tyres immediately aqua-planned and failed to spin in the normal fashion. This led the computer to believe that the plane was still in the air and refused the pilots request to engage reverse thrust!!!

Although we can laugh at these failures, they highlight the very real problems that can and all too often do arise when software designers are forced to design systems which are too open-ended or lacking adequate specifications.

As evidence of this, try defining precisely and unambiguously the process of making a cup of coffee in the morning. Your specification should consider all possible problems and provide a solution to deal with each. Now compare this with a friend's solution.

A Comparison of Software *Production* vs. that of other Engineered Products

Software then has characteristics that are considerably different from those of hardware:

1. *Software is developed. It is not manufactured, in the classical sense.*

Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different.

- In both activities, high quality is achieved through good specification and design, but the manufacturing phase for hardware can introduce quality problems that are *non-existent* (or easily corrected) for software.

- Both activities are dependent on people, but the relationship between *'people applied'* and '*work accomplished*' is entirely different. Virtually 100% of the effort/manpower required to produce quality software is concentrated in the *design*. The reproduction of software can be made virtually 100% accurate.

  In comparison, hardware is much more difficult to reproduce requiring factories and personnel plus the purchase and delivery, assembly and storage of expensive raw and finished materials and quality assurance programs/personnel to maintain the quality of production.

- Software costs are concentrated in the design and engineering of the **first** product. This means that software projects cannot be managed as if they were manufacturing projects. The cost of producing duplicate software is trivial compared to the cost of producing the 1st product, further more software can be reproduced cheaply and effectively with little cost or manpower.

  In comparison, the cost of the producing hardware lies principally in the manufacturing and purchasing cost of the raw materials and the factory/personnel required making the product.

The above has important implications. For example, take Car production. The cost of producing the robots, presses, paint shops, production lines, quality assurance programs etc. that are put in place to manufacture just one model of a car are staggering in comparison to the cost of the design itself. This puts enormous pressure on the designer to get it right '*first time*' as making even minor changes to the design, or recalling a model for modification once production has started can be cripping

Contrast this to the approach of the software designer, who all too often adopts the attitude that changing or updating the product is cheap, or even free if they download the service pack from a web site. Some even think that they are doing the customer a 'favour' by updating the product from time to time!!