The Resolution of the Software Crisis Through 'Software Engineering'

Wing Yung Chan
Trinity College

In this essay, I will argue that the problems that software engineering was created to solve are potentially intractable to an engineering perspective, and that despite the parallels that the creation of software has with other engineering fields, its future development may emerge out of different silos of thought. Furthermore, although the application of engineering principles in computing have led to notable insights, the roots that these principles address may in fact be transient and part of the current environment, soon to dissolve away as computer science matures.

Originally, the creation of software was known as programming. Programmers were mainly mathematicians by trade, often specialising in computation or logic. The term 'software engineering' was birthed in 1968 when the NATO Science Committee sponsored what they called the 'Working Conference on Software Engineering'. During the conference, Fritz Bauer proposed the following definition for Software Engineering, *'[Software engineering is] the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.*'[1] The juxtaposition of the word 'Engineering' with 'Software' was intentionally provocative at that time, and since then, both the name and the subject matter that it encapsulates have been controversial and contested.

[1] *"Software Engineering, Report on a conference sponsored by the NATO Science Committee",* Peter Naur and Brian Randell, 1969

Engineering has the connotation of trying to manage complexity via practical means. Science, which is notably used in the phrase 'Computer Science', has a more theoretical bias. For many people, calling the subject 'Software Engineering' was an admission of failure, in that their theoretical focus was proving to be infeasible in the 'real world'.

Even at the conference, the association of the programming process as fundamentally an engineering exercise was contested, '*During general discussion concerning theory and practice at the 1969 NATO conference, I.P. Sharp came at the issue from an entirely different angle, arguing that one ought to think in terms of "software architecture" (design), which would be the meeting ground for theory (computer science) and practice (software engineering). "Architecture is different from engineering," he maintained and then added, "I don't believe for instance that the majority of what [Edsger] Dijkstra does is theory—I believe that in time we will probably refer to the 'Dijkstra School of Architecture'.*"'[2] This perspective has found a home with a particular area of software, namely databases, where 'Database Architect' is the de facto job title.

Another view of programming is as an art form. This was made famous by Donald Knuth, Professor Emeritus at Stanford University and author of the seminal multi-volume work, 'The Art of Computer Programming', in which he described programming as a creative endeavour mixing mathematical rigour with ingenious solutions, '*The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music.*'[3]

At the dawn of computing, it was initially thought that the construction of computers would be most problematic at the electrical or mechanical level. Engineers on a computer project would be mostly concerned with the hardware aspect of design and implementation. Software, in contrast with hardware, was virtual and intangible, and as such seemed protected from faults or defects that might arise. As an example, the term 'bug', which had been used in engineering circles centuries earlier, became part of the computer science vernacular in 1947, when a moth stuck inside a relay tube stopped the computer[4] from working correctly[5] . At such an early time, it was difficult to see how such a bug could ever be found in software.

However, problems with software emerged as early as 1949, when the EDSAC (Electronic Delay Storage Automatic Calculator) computer began operation. This was one of the earliest

[2] "*Finding a History for Software Engineering",* M. S. Mahoney, *IEEE Annals of the History of Computing, Vol 26. Issue 1*. 2004
[3] "*The Art of Computer Programming Vol. 1"*, D. E. Knuth, *Addison-Wesley*. 1998
[4] The computer was the Harvard Mark II, invented by Howard Aiken and Grace Hopper.
[5] Log Book With Computer Bug, *National Museum of American History*, Smithsonian Institution. 1947

practical stored-program electronic computers. Yet the second program ever written for it was incorrect. The program written to enumerate the prime numbers was wrong due to a coding error. Sir Maurice Wilkes, founder of the Computer Laboratory in Cambridge and creator of the afore-mentioned EDSAC, is known for saying, "*…I realized that a large part of my life from then on was going to be spent in finding mistakes in my own programs.*"[6]

As projects grew in complexity, the number of bugs or defects also increased, and this led to public and embarrassing failures. The relevance of the 'engineering' debate was especially poignant at the time with the release of IBM's flagship operating system, the OS/360, which accompanied the System/360 mainframe computer in 1964. IBM at that point was the unquestionable market leaders for the mainframe computers, and so all eyes were on them. The operating system was a complex software system which has become famous for being both a success and a failure. The technology allowed companies to unlock new avenues in business[7], however, the project was late and over-budget. The project manager, Fred Brooks, wrote about his personal experiences in the seminal book, "The Mythical Man-Month: Essays on Software Engineering"[8], where the insights he gained have led people to realise that software might be less dependent on science and more on engineering.

The conference in 1968 was in fact a culmination of discussions that begun the year before around the increasing concern of how to correctly handle '*the difficulties of meeting schedules and specifications on large software projects*'.[9] In this light, it seems perfectly logical to see how an engineering perspective might prove advantageous, since engineering has developed principles and methodologies to deliver important and complex systems. The phrase 'software crisis' was coined during this pivotal meeting by F. L. Bauer. The report quotes David and Fraser: "*There is a widening gap between ambitions and achievements in software engineering. This gap appears in several dimensions: between promises to users and performance achieved by software, between what seems to be ultimately possible and what is achievable now and between estimates of software costs and expenditures. The gap is arising at a time when the consequences of software failure in all its aspects are becoming increasingly serious.*"

[6] "*Expert C Programming*", Peter Van der Linden, *Prentice Hall Professional*. 1994
[7] "I work in an environment of some fourteen large installations using OS/360. These are complex systems, being used for many very sophisticated applications. People are doing what they need to do, at a much lower cost than ever before, and they seem to be reasonably satisfied." Hastings during the NATO Conference. 1968.
[8] "*The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2nd Edition)*", Frederick P. Brooks, *Addison-Wesley Professional*. 1995.
[9] "*Software Engineering, Report on a conference sponsored by the NATO Science Committee*", Peter Naur and Brian Randell, 1969 (Page 6)

With the increase in the computing potential due to improvements in hardware, there was a shared expectation that this would be exploited through the creation of software systems. Gordon Moore observed that the number of transistors that can be placed on a chip was doubling every 24 months.[10] This rate of exponential increase has set the agenda for chip manufacturers since 1965 who have fulfilled the prediction that Moore made. Even more amazing is that hardware has improved in almost every way. The size and cost of transistors has also decreased exponentially. But despite the potential that hardware offers us, the software is unable to capitalise on it. Even worse, the numbers of bugs scales with the size of the project, so as larger projects are undertaken, it seems we are doomed to witness failures on even larger scales. This result was actually predicted at the conference in 1968 by Opler, '*I am concerned about the current growth of systems, and what I expect is probably an exponential growth of errors.*'[11]

After the conference, companies began applying engineering principles to software projects but by 1990, the subject was still not a congruent whole, with M. Shaw writing, '*Software engineering is not yet a true engineering discipline, but it has the potential to become one.*'[12] In this period of uncertainty, there have been reports advocating the importance of software engineering as well as reports criticising its utility. Yasushi Sato, assistant professor at the National Graduate Institute for Policies Studies in Japan, wrote a report on NTT's Role in the development of Software Engineering in Japan. In this report, it is suggested that an NTT project, DIPS, may have been '*a major factor for the survival of Japanese computer manufacturers.*'[13] A study conducted by Brenda Whittaker, Senior Consultant for KPMG Consulting reports that in 1995, '*31 per cent of software projects [in the USA] will be cancelled before completion, and more than half the projects will cost an average of 189 per cent of their original estimates.*'[14] By 1997, R. Baber made an evaluation of the state of software engineering[15], in which he predicated that, '*software development as practiced today is a pre-engineering phase of what can and should – and hopefully will – become a true engineering discipline sometimes in the not too distant future. Software development is not yet an engineering discipline, at least not in the sense commonly accepted by engineers in today's traditional fields, for example, civil, mechanical, and electrical engineering.*'

[10] "*Excerpts from A Conversation with Gordon Moore: Moore's Law*", Intel. 2005. Note: the original Moore's law was made in 1965, although he clarified it in a 2005 video interview.
[11] "*Software Engineering, Report on a conference sponsored by the NATO Science Committee*", Peter Naur and Brian Randell, 1969 (Opler, Page 10)
[12] "*Prospects for an Engineering Discipline of Software"*, M. Shaw, *IEEE Software, vol. 7, no. 6.* 1990 (Page 15)
[13] "*An Inconspicuous Giant: NTT's Role in the Development of Software Engineering in Japan*", Yasushi Sato, *IEEE Annals of the History of Computing, vol 32 Issue 4*. 2010 (Page 3)
[14] "*What went wrong? Unsuccessful information technology projects"*, Information Management & Computer Security, Vol 7 Issue 1. Brenda Whittaker, *MCB UP Ltd*. 1999
[15] "*Comparison of Electrical "Engineering" of Heaviside's Times and Software "Engineering" of Our Times*", R. Baber, *IEEE Annals of the History of Computing Vol. 19, Issue 4.* 1997

The handling of complexity seems at first glance to be exactly how engineering approaches can directly help. Projects such as nuclear reactors, bridges and racing cars are incredibly complex, and yet engineering has managed to deliver them. Gilette said during the conference, '*We are in many ways in an analogous position to the aircraft industry, which also has problems producing systems on schedule and to specification. We perhaps have more examples of bad large systems than good, but we are a young industry and are learning how to do better.*'[16] However, one can argue that software must be treated somewhat uniquely. Complexity scales with not only the number of components but also the interactions between them. A simple demonstration of this is incidentally one of Fred Brooks' conclusions from his essay, which has become a mantra in software engineering, 'Adding people to a late project makes it later'. This is because despite the added man-power, the increase of interactions between people can be exponential. The number of interactions is manageable for small teams but becomes unwieldy for a team larger than eight. Interestingly, this insight has led to the emergence of large corporations made up of small teams, a practice championed by search engine giant, Google[17]. In a similar way, because components in software are not fixed like in a bridge, they are free to interact with everything else and a software project with a similar number of components to any other engineering project will almost always have a greater number of interactions.

The success or failure of a large software project, as with most engineering projects can be categorised by certain performance metrics: cost versus budget, time taken, number of defects, completion to meet specification. In particular, the prevalence of defects in software has led to a culture of regarding a project as successful if the number of critical bugs is below a certain threshold of tolerance. This is a far cry from the origins of programming as a mathematical exercise of rigorous deduction and perhaps this culture can by definition never create perfect software.[18] Practically, this has three implications. Firstly, software is not perfect and is liable to break, sometimes spectacularly. A lot of research has gone into the chronicling of software disasters[19], for example the Ariane 5 Missile disaster, which blew up after 37 seconds of flight, destroying its payload of US $370 million worth of satellites because of a

[16] "*Software Engineering, Report on a conference sponsored by the NATO Science Committee*", Peter Naur and Brian Randell, 1969 (Page 10)
[17] "*Google Thinks Small*", Q. Hardy. http://www.forbes.com/global/2005/1114/054A.html
[18] Java, currently (2011) the second most popular programming language in the world, comes with a disclaimer, "JAVA TECHNOLOGY IS NOT FAULT TOLERANT AND IS NOT DESIGNED, MANUFACTURED, OR INTENDED FOR USE OR RESALE AS ONLINE CONTROL EQUIPMENT IN HAZARDOUS ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, SUCH AS IN THE OPERATION OF NUCLEAR FACILITIES, AIRCRAFT NAVIGATION OR COMMUNICATION SYSTEMS, AIR TRAFFIC CONTROL, DIRECT LIFE SUPPORT MACHINES, OR WEAPONS SYSTEMS, IN WHICH THE FAILURE OF JAVA TECHNOLOGY COULD LEAD DIRECTLY TO DEATH, PERSONAL INJURY, OR SEVERE PHYSICAL OR ENVIRONMENTAL DAMAGE."
[19] One example is Software 'Runaways: Monumental Software Disasters', R. Glass, *Prentice Hall*. 1997

small software bug.[20] It caused the death of 28 American soldiers in Dharan, Saudi Arabia when the patriot missile system failed to intercept a Scud missile.[21] This was traced to mistake in code which only manifested itself when the system was left on for more than a hundred hours.

I will now present two major contributions of engineering in the methodology of software creation, which are designed to mitigate the afore-mentioned failures. Firstly, several novel approaches to the project lifecycle have been proposed. The project lifecycle denotes the stages of creating large software systems. Secondly, a large set of principles have emerged to detect and remove bugs as efficiently and completely as possible.

Creating software requires the translation from a list of goals, normally written in natural language, into software, which is code written in a programming language. This translation is done as a sequence of stages, successively increasing the detail of the specification until you reach the ultimate level of detail, which is the code itself. The waterfall model, which originates in the manufacturing and construction industries, describes a sequential design process, in which progress is visualised as flowing downwards through the phases of Requirements, Design, Implementation, Verification (Test) and Maintenance. This was described in the software context by Winston W. Royce in 1970[22], although it was described even earlier under the name of the 'stagewise model' in June 1956[23]. However, this model has proved to be too inflexible, leading to projects running out of budget and/or time because problems are detected at the later stages, by which time, it is too late to start again.

Another model is Iterative and Incremental Development (IID), which originates from a 1950s project in designing the X-15 hypersonic jet.[24] Team members from this project went on to be part of NASA's Project Mercury, which applied IID to software in the early 1960s. Iterative and Incremental Development entails the splitting up of the requirements into groups, and then designing, coding and testing each group individually in a series of 'increments'. This had several useful characteristics. Firstly, by breaking the problem into smaller sections, you reduced the complexity of the project into effectively several mini-projects. Secondly, this allowed the release of a working system (with capabilities increasing as time progressed) to the client for evaluation and feedback much earlier than before. In fact, according to Winston Royce's son, Walker, Winston had himself seen the limitations of the

[20] The computer was storing integers as 16 bits, which means that any number larger than that overflowed and this caused an exception in the program.
[21] "*Patriot Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia*", *IMTEC-92-26*. 1992
[22] "*Managing the Development of Large Software Systems*", W. Royce, *Proceedings of IEEE WESCON*. 1970
[23] "Symposium on advanced programming methods for digital computers", H.D. Benington, *Office of Naval Research*.1956
[24] "*The X 15 Lessons Learned*", W.H. Dana, *NASA Dryden Research Facility*. 1993

waterfall model, '*He was always a proponent of iterative, incremental, evolutionary development. His paper described the waterfall as the simplest description, but that it would not work for all but the most straightforward projects. The rest of his paper describes [iterative practices] within the context of the 60s/70s government contracting models (a serious set of constraints).*'[25]

The idea of creating working systems early on in the project as prototypes became the cornerstone behind the Spiral Model of development. This elegant model was proposed by Barry Boehm in 1986[26], in which the software is built up in stature as a series of prototypes moving around the project lifecycle whilst becoming more and more feature rich (hence the spiral), until it finally becomes the finished product.

In order to speed up the delivery of projects, smaller software houses proposed 'agile' methods. These methods aim to reduce the amount of time spent on design and documentation, in order to focus on coding. They build on IID to provide usable products as early as possible to let the client decide what features should be added, rather than imposing that all the requirements be decided at the very beginning. A plethora of agile methodologies became available at the turn of the 21st century, with the best known being extreme programming (Beck 1999), Scrum (Schwaber and Beedle, 2001), Crystal (Cockburn, 2001), Adaptive Software Development (Highsmith, 2000), DSDM (Stapleton, 1997) and Feature Driven Development (Palmer and Felsing, 2002).[27] This form of design was different to traditional engineering approaches and this break away from tradition created a divide between traditionalists and supporters of agile methods. As a result of this, coupled with the difficulty in realising some of the principles underlying agile methods, there is currently limited take up of this methodology.

Once code has been written, the system has to be tested. With small projects, defect count was small, so testing was a small part of the process. However, Glenford Myers notes in his book, 'The Art of Software Testing', that '*in 1979, it was a well-known rule of thumb that in a typical programming project approximately 50 percent of the elapsed time and more than 50 percent of the total cost were expended the program or system being developed. Today, a quarter of the century later, the same is still true*'.[28] The problem with testing is elegantly expressed by Edsger Dijkstra, '*Program testing can be used to show the presence of bugs, but never to show their*

[25] "*Iterative and Incremental Development: A Brief History*", C. Larman, Cover Article, *IEEE Computer June 2003.*
[26] "*A Spiral Model of Software Development and Enhancement"*, B. Boehm, *ACM SIGSOFT Software Engineering Notes*. 1986 (Pages 14-24).
[27] "*Software Engineering 8th Edition",* I. Sommerville, *Addison-Wesley*. 2007 (Page 396)
[28] "*The Art of Software Testing",* G. Myers, *John Wiley & Sons Inc*. 1979

absence'.[29] From an engineering perspective, testing involves showing that the system does what it is required to do, and also doesn't do what it shouldn't do, all in the shortest amount of time possible.

Other approaches have been discovered to speed up the finding of bugs. An interesting discovery was the realisation that waiting to the test phase to discover the bug was not only exorbitantly expensive, but also inefficient. P. Pierce wrote in a 1996 article for the Northcon conference, '*The earlier a defect is found in the development process, the lower the cost to correct it. If an error in logic is discovered during the review of the Software Requirements Document, the cost is much lower than if it is not found until validation testing or after the product is shipped*'.[30] A practical solution is to enforce 'Software Inspections' which involve a manual walkthrough of the code to find defects. Although this is not a practice that was widely adopted until more recently, studies done in the 1980s indicate that inspections '*are more effective for defect discovery than program testing*'.[31] In 1986, M. E. Fagan[32] reported that more than 60% of errors can be detected with informal program inspections. This was followed up by Mills et al. in 1987[33], who suggested that with a formal approach this gives estimates of a 90% error detection rate.

A more direct approach to solving the problem is to invert the waterfall model and make testing the focus of the project. This has been called Test-Driven Development since 2003[34], and is considered an offshoot of Extreme Programming that has become important in its own right. In this paradigm, the developer writes a test suite of failing automated test cases before writing any actual code for the system. Then, as the program is developed, its progress can be tracked against this large suite of test cases. Initially, there is a large overhead in creating the test cases, but with the advent of practical automated testing, it can provide a clear metric for tracking progress. The automation of testing is hoped to reduce both testing time and human resources required to do manual testing.

Automated testing requires a translation of requirements into test cases which can decide whether a system works. Often in the software process, there are many languages to contend with: the business language of the requirements; the formal language of the specification; the code for implementation (which may itself be made up of multiple languages); the testing and verification language. No matter what process or methodology you apply, this translation

[29] "*Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27-31 October 1969".* J. Buxton and B. Randell, *Software Engineering Techniques*. 1970
[30] "*Software verification and validation*", P. Peirce, *Northcon/96*. 1996 (Pages 268)
[31] "*Software Engineering 8th Edition",* I. Sommerville, *Addison-Wesley*. 2007 (Page 522)
[32] "*Advances in software inspections"*, M. E. Fagan, *IEEE Trans. on Software Engineering*. 1986 (Pages 744-51)
[33] "*Cleanroom software engineering*", H. D. Mills et al, *IEEE Software*. 1987 (Pages 19-25)
[34] "*Test-Driven Development by Example",* K. Beck, *Addison-Wesley*. 2003

between languages and phases is the source of considerable complexity, but is there another way?

An alternate branch of thought has been developed to deal with this problem indirectly. It stems from the idea that Mathematicians don't need to test their theorems. Once a conjecture has been proved, it becomes a theorem, uncontested and invariant through the ages. What if you could write a program like a mathematical theorem and prove its correctness? Furthermore, if the proof of its correctness was actually manifested in the design phase, this would save both time and cost that would have been spent on explicit testing. Instead of finding bugs, you would spend time creating systems that were not just a refinement of the initial requirements, but provably satisfy them.

Rather than being a pipedream, languages which can be verified for correctness originated not long after the NATO conference itself, for example ML (1970), Ada (1980), Haskell (1990). These languages are naturally mathematical in formulation, and are usually referred to as functional languages, because of the similarity with mathematical functions. Functional languages are necessarily 'high level', which means that they are abstracted away from the underlying hardware. Functional programming allows the programmer to focus on the problem domain at hand, rather than worrying on how the processor will implement it.

Furthermore, two emergent properties of high level functional programming are particularly advantageous in the pursuit of bug-free programs. Firstly, at the lowest level, code eventually becomes machine code, a sequence of instructions that the computer can process. It may take several instructions to work out (a + b) + c. As you abstract away from the hardware, each line of code has more 'power', which means it translates to multiple lines of machine code. So a high level functional language is incredibly succinct, which is good when designing complex systems. Secondly, functional programs are strongly-typed, which means that they often force the definition of types such as a Dog type or a Cat type. This enables type checking, which would stop you from creating a Dog object which was actually a Cat[35]. These errors are often called 'incidental', and are common at lower levels but are caught by the compilers[36] of higher level languages.

Although poetry and humour benefit much from the ambiguity of languages, beauty in mathematics and science is most commonly seen as simplicity and clarity. The two most

[35] Catching errors at compile-time means that if the program has type-errors, it will not compile. This stops software from being run till it is free of at least these types of errors.
[36] A compiler is a program that transforms source code from one language to another, usually to a lower-level language.

popular languages[37], JAVA and C have a lot of ambiguity which makes programming quicker but also leads to bugs. Both of these languages have what is known as an ambiguous grammar, which means that there are statements that are valid which could have more than one meaning. It is up to the compiler software to decide what it really means. Understandably, this lack of preciseness means that it is very hard to prove code is correct.

Languages do exist that are free of ambiguous grammar. One is called SPARK[38], an offshoot of the language ADA, which sets out to be free of ambiguity, meaning that any compiler of the language will give the same result each time. SPARK is grounded on mathematical logic and formalism, and while traditional coders may not welcome the intensity of mathematical rigour, it is precisely this painstaking rigour that may be the solution to the current prevalence of bug-riddled software.

Despite the benefits, functional programming has had a slow adoption rate in businesses because of the high threshold of skill required to write code, as well as concerns about its performance. However, in a demonstration of its potential benefits, Magnus O. Myreen's[39] PhD thesis, 'Formal verification of machine-code programs', presents a practical method of translating properties of programs into properties of functions, '*the problem of proving properties of programs is reduced, via fully-automatic deduction, to a problem of proving properties of recursive functions.*'[40] With this power, the number of transformations has been reduced and for the most part automated, meaning that the software development process can be significantly shortened.

In 2001, Ulf Wiger, published an article entitled, '*Four-fold increase in productivity and quality: Industrial strength programming in telecom-class products*'.[41] Using the language Erlang, which is a functional language with an emphasis on concurrency (the ability to simultaneously run programs at the same time), Ericsson, which is a Swedish telecommunications company, was able to reportedly achieve a momentous gain in both productivity and quality, including a 99.999% availability rate for its products.

A major problem that software projects face is the estimation of how long a project would take to complete. This is crucial for budgeting as costs for these projects are not dominated by equipment costs but man-hours. A good programmer can write a hundred lines of code a

[37] *TIOBE Programming Community Index for October 2011- http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html*
[38] SPARK was designed by Bernard Carré and Trevor Jennings. It is also described as High Integrity Ada, "*High Integrity Ada: The SPARK Approach",* J. Barnes, *Addison-Wesley*. 1997
[39] Incidentally, Myreen was my lecturer in 2010 for a course on the functional language, ML.
[40] "*Formal verification of machine-code programs"*, Magnus O. Myreen, *Trinity College, Cambridge*. 2008
[41] "*Four-fold increase in productivity and quality: Industrial strength programming in telecom-class products"*, U. Wiger, *Third Workshop on Formal Design of Safety Critical Embedded Systems*. 2001

day, and in small projects, this often happened. However, in large projects, the figure is rather sobering. Estimates from IBM taken in one year show that programmers working on an operating system wrote 1,500 LOC (lines of code) per man-year[42]. If you assume a 280-day working year, this equates to just five lines of code a day! AT&T, the telecoms giant published figures of just 600 LOC per man-year.[43] Furthermore, as part of Brooks' research for his book on software engineering, he found that writing in a higher level language did not reduce the number of lines of code written per man-year. Since in a higher level language each line is 'worth' multiple lines of lower language, this was a significant result, and prompted Brooks to write, '*Much of the complexity in a software construct is, however, not due to conformity to the external world but rather to the implementation itself—its data structures, its algorithms, its connectivity. Growing software in higher-level chunks, built by someone else or reused from one's own past, avoids facing whole layers of complexity.*'[44]

There are in fact many paradigms for high-level language. Fifth-generation languages (5GL) were heavily advocated by Japan between 1982 and 1993. 5GLs are languages that allow the computer to essentially solve the problem without the programmer. Prolog (1972), OPS5 (1970s), Mercury (1995) are examples of 5GL programs. The lesser constraint on human involvement could drive the cost of projects down while providing automatic verification techniques as part of its code. It was abandoned in the 1990s because solving problems automatically turned out to be much harder that first thought. However, if an algorithm is found to make this transformation efficiently, this would have widespread consequences.[45]

One of the main reasons that is cited for large scale project failure is the changing of requirements during the project. This is done for many reasons, perhaps there has been a legal change forcing the need for new features, or there has been a change in strategy from the business requiring a change to be reflected in the software. In a survey from Canadian businesses in 1997, 51% of projects identified that a '*Change in scope of technology, functionality or business case'* was a risk that was not addressed in the project planning phase.[46] The fatality that such a change in requirements often entails can be explained by the fact that the design stage (formal specification) is structured rigidly to solve specifically the list of requirements. Roger Pressman writes in his book, 'Software Engineering: A Practitioner's Approach', that '*as time passes, the cost impact grows rapidly—resources have been committed, a design framework as been established, and change can cause upheaval that*

[42] "*The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2nd Edition)"*, Frederick P. Brooks, *Addison-Wesley Professional*. 1995. (Page 237, 8.6)
[43] As above. (See Page 237, 8.7)
[44] As above. (See Page 212)
[45] "*Fifth-Generation Computers*", R. Grigonis. 1980s.
[46] "*What went wrong? Unsuccessful information technology projects*", Information Management & Computer Security, Vol 7 Issue 1. Brenda Whittaker, *MCB UP Ltd*. 1999

requires additional resources and major design modification.'[47] Many have remarked that this problem is not the fault of software or its creators. It is in fact a product of an unrealistic expectation of management on the tractability of software coupled with the intrinsic complexity of a large system. The tractability of software and its apparent flexibility because it is a virtual platform is in many ways a myth because the complexity is still there, regardless of the fact that it is not physical. S. Gill noted in the conference, '*It is of the utmost importance that all those responsible for large projects involving computers should take care to avoid making demands on software that go far beyond the present state of technology unless the very considerable risks involved can be tolerated.*'[48] Even now, software engineering is still unable to remove the impact that a change of requirements has in the software development process.

In conclusion, although the effectiveness of engineering techniques in software has enabled the creation of large, successful projects, it may in fact not be the right way to approach the problem. I believe Mahoney eloquently describes the state of affairs in his article entitled 'Finding a History for Software Engineering', '*Software engineering began as a search for an engineering discipline on which to model the design and production of software. That the search continues after 35 years suggests that software may be fundamentally different from any of the artifacts or processes that have been the object of traditional branches of engineering: It is not like machines, nor masonry structures, nor chemical processes, nor electric circuits, nor semiconductors.'* Computer science has also struggled to find its own identity amongst the more established disciplines of Mathematics, Natural Science and Engineering. It is hoped that in time, software creation will eventually fall in line with one of the more established disciplines so that it develops with a greater integrity, but with research into advanced computation techniques and the promise of functional programming, perhaps it will emerge as an altogether separate beast in its own right.

[5,546 Words]

Wing Yung Cha

[47] "*Software Engineering: A Practitioner's Approach*", R. Pressman, *McGraw-Hill Higher Education. 7th ed*. 2009
[48] "*Software Engineering, Report on a conference sponsored by the NATO Science Committee*", Peter Naur and Brian Randell, 1969 (Gill, Page 10)