Chapter 1

Introduction

1.1 History of Software Engineering

It is very unfortunate that we often have very little interest in the history of any subject and many of us even thinks that the concepts and ideas are propagated and advertised as being new, but were existed decades ago perhaps under a different terminology. As said by Glass (1) that, “The most frequent mistake is the assumption that progress in those early days was slow and plodding and that not much was happening in the field.”. It is worthwhile to consider the past and to investigate how terms and concepts originated and evolved day by day as we see, know and understand today. Computer has come a long way since it was first introduced somewhere in the late 1940s. Since then it has evolved steadily throughout the ages and applications of various types of software have reached heights that were not thought to be possible. From its beginnings, writing software has evolved into a profession concerned with how best to maximize the quality of software and of how to create it. Software engineering is a rather relative term when we consider the word “engineering”. The first appearance of the two words came about in the 1950s. The basic problem software engineers had was that one could not see a physical development in the software— all done virtually or on paper. So it was difficult to develop software without a proper model. While one had, to learn that the origins of software cannot be clearly distinguished from the hardware, it will be even harder to try to predict the future of software and hardware. Will further levels of abstraction evolve? In any case, we speculate that software is still in its infancy and the origins of software will gradually come to include the present times. It is fascinating to be a part of history, especially one that has such an impact on the world.
1.1 History of Software Engineering

1.1.1 The Origins of Software

1.1.1.1 The Early Electronic Computers

In 1837, Charles Babbage designed and planned to build the Analytical Engine, a programmable calculator that was supposed to be driven by programs on punch cards, enabling the machine to perform all operations possible to modern computers. The idea of programming the machine with the cards originally came from Ada Lovelace, the protégé of Babbage, who is often regarded as the world's first programmer. However, just like the Difference Engine, the Analytical Engine failed and was never built during Babbage's lifetime. Further, in 1837, ENIAC- the first large digital computer that could be reprogrammed was built and there was still no concept or need for portable software. In fact, there was no place where the instructions for running the machine were stored. Instead, every time the ENIAC was to compute a new problem, it had to be set up a new -- that meant re-cabeling all the hardware units manually. In the subsequent years, the manual setup was changed.

In 1944, the Mark I was developed at Harvard and in 1945 the Whirlwind at MIT was built in which punched cards were being used to determine the order of operations. It was the Harvard Mark I that spun the modern creation of software in the United States. Grace Hopper was assigned the responsibility by the Navy to create programs for the Mark I and it was in her daily routines that she saw the need for easier reuse of code and later the compiler. Her ideas were not perfect but seemed to create a series of events which would quickly lead us to the modern idea of compiles and assemblers.

1.1.1.2 The Early Days of Software

In 1804, the idea of software was first observed with the loom of Jacquard. Paper cards with punched holes controlled the weaving of the pattern on the loom, enabling more complex patterns and faster production times. His machine created the first need for software and with it the first negative reaction to programmable machines. The high-tech loom changed the weaving profession, effectively lowering the required skill set and limiting the number of people needed to operate. This loom demonstrated three aspects of software:

1. logical structure
2. representation
3. interaction with the physical device
1.1 History of Software Engineering

– a concept that would loosen throughout the evolution of software. Jacquard's most significant invention was perhaps that of the punch cards – they would remain in use for well over a hundred years. Some significant uses would be Hollerith's tabulating machine for the U.S. Census, the UNIVAC and the machines built by IBM.

In 1936, Alan Turing invented the theoretical design of a computer – the Turing Machine, proving that the Halting problem is unsolvable \(2\). The Turing Machine was designed to function as a systematical working human, using only three operations:

1. read from a tape
2. write to a tape
3. move the read/write head.

With only these operations, the Turing Machine can compute anything that a functioning computer is able to calculate, and the Church Turing thesis \(3\) extends its capabilities by conjecturing that any function calculable in the common sense can be computed by a Turing Machine. The theoretical implications of a Turing Machine are vast, but the significant feature in this instance is the software that arose from the conception of a Turing Machine. A Turing Machine proposed that a mechanical machine would simply execute a set of instructions, which could then be easily copied or moved to a new machine. Additionally, it proposed a so called Universal Turing Machine. “A man provided with paper, pencil, and rubber, and subject to strict discipline, is in effect a universal Turing Machine \(4\)”. An Universal Turing Machine that could read and simulate the behavior of other machines, given as input on the tape.

In those days, the software – mainly consisting of instructions by punch cards – was always specific to one kind of machine. The limitations of punch cards and tape reels forced extra processing time just to sort the files in an accessible manner, occupying 25% of all processing time. In order to be effective, code had to be precise and compact. Only many years later would the software evolve to become largely independent from the hardware. During these first years of software development, the study and perfecting of algorithms was quickly becoming its own subject area. Some of the best algorithms in place today were created and implemented prior to the 1970s. This, along with the constant improvements in hardware and the realization that computers can be used for more than just mathematics, created a growing need for programmers, directly creating a need for formal education in the area. Colleges began implementing computer science departments in the 1960s. The first programs seemed to focus on languages and practical
applications while contemporary programs focus more on theories and practices. These thought of Universal Turing Machine and related experiments had many direct influences on the years to follow. Even though it took some time for them to be realized, can be seen as the earliest conceived notions on realization of compilers and interpreters and were finally implemented in 1952. Similarly, it can also be seen as the earliest conceived notions of emulators and simulators, which were only recently implemented efficiently in 1997.

In 1954, while the first languages were being created and used, Laning and Zierler developed the first assembler. In 1957, IBM released FORTAN and COBOL was popularized by the United States Government in 1960. It was during this time of innovation that the first examples of open source occurred. SHARE was a group of IBM users that joined forces to, if nothing else but share frustrating experiences. They managed to create many libraries of code, reducing the amount of redundant work between members. As the group grew, SHARE and IBM seemed to form a symbiotic relationship, in that SHARE created more profits for IBM and IBM in return placed great weight on SHARE's opinions and references. It was a great example of a developer community.

The term “software engineer” arose in 1968, when people speculated that lack of engineering approach was causing the software crisis. The NATO Science Committee sponsored two major software conferences, one in 1968 and the other in 1969. These conferences gave the initial boost required for software engineering and many mark these events as the official birth period of software engineering. Perhaps the most important occurrence in the advent of software was the unbundling of software from hardware by IBM in 1969, under pressure by the U.S. Government, and the rise of software companies. The industry slowly lost its focus on hardware, creating an expectation of reliable software. Likewise, the costs of software development started to exceed the costs for hardware.

Between 1969 and 1972, the programming language C was developed by Dennis Ritchie at AT & T's Bell Labs with other just like its direct ancestor B. Strangely enough, the initial motivation was to enable the programmers to play the video game Space Travel on a PDP11. As it happened, and almost at the same time the operating system Unics was being developed for a PDP11 machine at Bell Labs written in Assembler. The researchers soon found that the finished high level language C would enable them to make Unics. In 1973, it become necessary that the labeled UNIX to be portable to almost any other machine. So most parts of it were then rewritten for that purpose. The success of both C and UNIX was closely tied together. The fact that AT & T, being a regulated monopoly was not able to sell UNIX for profit. Along with its high portability, it led
1.1 History of Software Engineering

The impressive networking, file handling and user management capabilities of UNIX influenced the early days of the Internet and many operating systems which are widely used today, such as Linux, *BSD or MacOS. Similarly, C was soon standardized, adapted to the PC and used by system programmers everywhere.

1.1.2 The Chronological Evolution of Software Engineering

1.1.2.1 During 1940s: The Pioneering Era

The most important development was that new computers were coming out almost every year or two, rendering existing ones obsolete. During 1940s, computer hardware was application-specific and so scientific and business tasks needed different machines. Software people had to rewrite all their programs to run on these new machines. Programmers did not have computers on their desks and had to go to the “machine room”. Jobs were run by signing up for machine time or by operational staff. Jobs were run by putting punched cards for input into the machine’s card reader and waiting for results to come back on the printer. Due to the need to frequently translate old software to meet the needs of new machines, high-order languages like FORTRAN, COBOL, and ALGOL were developed. Hardware vendors gave away systems software for free as hardware could not be sold without software. A few companies sold the service of building custom software but no software companies were selling packaged software. The field was so new that the idea of management by schedule was non-existent. Making predictions of a project’s completion date was almost impossible.

1.1.2.2 1945 to 1950s: Debated What Engineering Might Mean for Software!

The term software engineering first appeared in the late 1950s and early 1960s. Programmers have always known about civil, electrical, and computer engineering and debated what engineering might mean for software. During 1950s people believed understanding was, “Engineer software like you engineer hardware”\(^5\). First, software was much easier to modify than was hardware, and it did not require expensive production lines to make product copies. One changed the program once and then reloaded the same bit pattern onto another computer, rather than having to individually change the configuration of each copy of the hardware. This ease of modification led many people and organizations to adopt a “code and fix” approach to software development, as compared to the
1.1 History of Software Engineering

exhaustive “Critical Design Reviews” that hardware engineers performed before committing to production lines and bending metal (measure twice, cut once). Many software applications became more people-intensive than hardware-intensive; even SAGE became more dominated by psychologists addressing human-computer interaction issues than by radar engineers. Another software difference was that software did not wear out. Thus, software reliability could only imperfectly be estimated by hardware reliability models, and software maintenance was a much different activity than hardware maintenance. By the 1960s, however, people were finding out that software phenomenology differed from hardware phenomenology in significant ways.

1.1.2.3 1960s: The Software Crisis and Origins of Software Engineering

During 1960s, the code-and-fix was the primary software development approach followed by the industry. But during the 1960s, 70s and 80s brought about the so called software crisis. Software was invisible, it did not weigh anything, but it cost a lot. This time frame became a very bumpy road for software developers and engineers. During this period many of the problems in software development were highlighted. Initially the software crisis was defined in terms of productivity, but later it turned out to be defined in terms of quality. Many software projects ran over budget or over schedule. Software engineering has come a long way since the 1960s and the first attempts to make our field into an engineering discipline. The philosophy during 1960s was “think outside the box”. Repetitive engineering would never have created the Arpanet or Engelsb's mouse-and-windows GUI. Have some fun prototyping; it is generally low-risk and frequently high reward. Respect software's differences. You cannot speed up its development indefinitely. Since it is invisible, you need to find good ways to make it visible and meaningful to different stakeholders. Avoid cowboy programming. The last-minute all-nighter frequently does not work, and the patches get ugly fast.

This situation led the NATO Science Committee (5) to convene two landmark Software Engineering conferences in 1968 (Garmisch, Germany) and 1969, attended by many of the leading researcher and practitioners in the field. In these conferences the difficulties and pitfalls of designing complex systems were frankly discussed. These particular conferences gave the field its initial boost. Many believe these conferences marked the official start of the profession of software engineering.

Software generally had many more states, modes, and paths to test, making its specifications much more difficult. Winston Royce, in his classic 1970 paper, said, In order to procure a $5 million hardware device, I would expect a 30 page specification would
provide adequate detail to control the procurement. In order to procure $5 million worth of software, a 1500 page specification is about right in order to achieve comparable control\(^5\). It was hard to tell whether it was on schedule or not, and if you added more people to bring it back on schedule, it just got later, as Fred Brooks explained in the “Mythical Man-Month”\(^5\). A few projects caused loss of life\(^5\). Eventually the hard work of many software companies paid off and reviled the path towards a brighter future of software engineering.

1.1.2.4 1970s Synthesis and Antithesis : Formality and Waterfall Processes

The main reaction to the 1960s code-and-fix approach involved processes in which coding was more carefully organized and was preceded by design, and design was preceded by requirements engineering. The philosophy during 1970s was to eliminate errors early. Even better, prevent them in the future via root cause analysis. Determine the system's purpose. Without a clear shared vision, you are likely to get chaos and disappointment. Goal-question metric is another version of this. Avoid Top-down development and reductionism. COTS, reuse, IKIWISI, rapid changes and emergent requirements make this increasingly unrealistic for most applications.

More careful organization of code was exemplified by Dijkstra's\(^5\) famous letter to ACM Communications, “Go To Statement Considered Harmful”. The Bohm-Jacopini result showing that sequential programs could always be constructed without goto's led to the Structured Programming movement. This movement had two primary branches. One was a “formal methods” branch that focused on program correctness, either by mathematical proof\(^5\) or by construction via a “programming calculus”. The other branch was a less formal mix of technical and management methods, “top-down structured programming with chief programmer teams”, pioneered by Mills and highlighted by the successful New York Times application led by Baker.

The success of structured programming led to many other “structured ” approaches applied to software design. Principles of modularity were strengthened by Constantine's concepts of coupling (the degree of interdependency between two modules- to be minimized) and cohesion (the degree of functional strength of a module - to be maximized), by Parnas's increasingly strong techniques of information hiding, and by abstract data types. A number of tools and methods\(^5\) employing structured concepts were developed, such as structured design; Jackson's structured design and programming, emphasizing data considerations and Structured Program Design Language.

Requirements-driven processes were well established in the 1956 SAGE process model,
1.1 History of Software Engineering

but a stronger synthesis of the 1950s paradigm and the 1960s crafting paradigm was provided by Royce’s version of the “waterfall” model (5). During 1970s Software engineering was mostly qualitative.

1.1.2.5 1985 to 1989s : No Silver Bullet

The cost of owning and maintaining software in the 1980s was twice as expensive as developing the software. During the 1990s, the cost of ownership and maintenance increased by 30% over that of 1980s. In 1995, statistics showed that half of surveyed development projects were operational, but were not considered successful. The average software project overshoots its schedule by half. Three-quarters of all large software products delivered to the customer are failures that are either not used at all, or do not meet the customer’s requirements.

For decades, solving the software crisis was paramount to researchers and companies producing software tools. Almost every new technology and practice from the 1970s to the 1990s was trumpeted as a silver bullet to solve the software crisis. Tools, discipline, formal methods, process and professionalism were touted as silver bullets. Debate about silver bullets raged over the following decade. Advocates for Ada, components and processes continued arguing for years that their favorite technology would be a silver bullet. Skeptics disagreed. The search for a single key to success never worked. In 1986, Fred Brooks (6) published his No Silver Bullet article, arguing that no individual technology or practice would ever make a 10-fold improvement in productivity within 10 years. Eventually, almost everyone accepted that no silver bullet would ever be found. Yet, claims about silver bullets pop up now and again even today. Some interpret no silver bullet to mean that software engineering failed. Others interpret no silver bullet as proof that software engineering has finally matured and recognized that projects succeed due to hard work. However, Brooks goes on to say, “We will surely make substantial progress over the next 40 years; an order of magnitude over 40 years is hardly magical ...”. All known technologies and practices have only made incremental improvements to productivity and quality. Yet, there are no silver bullets for any other profession, either. However, it could also be said that there are, in fact, a range of silver bullets today, including lightweight methodologies spreadsheet calculators, customized browsers, in-site search engines, database report generators, integrated design-test coding-editors with memory/differences/undo and specialty organizations that generate niche software at a fraction of the cost of totally customized website development. Nevertheless, the field of software engineering appears too complex and diverse for a single “silver bullet” to
improve most issues, and each issue accounts for only a small portion of all software problems.

1.1.2.6 1990 to 1999s : Prominence of the Internet

With the start of the 1900s came as never before seen phenomenon called the “Internet” and World Wide Web (WWW) brought out opportunities like never before. The rise of the Internet led to very rapid growth in the demand for international information display/e-mail systems on the WWW. Programmers were required to handle illustrations, maps, photographs, other images in addition to simple animation at a rate never before seen with few well known methods to optimize image display or storage. The growth of browser usage running on the HTML language changed the way in which information display and retrieval was organized. The widespread network connections led to the growth and prevention of international computer viruses on Windows computers. The vast proliferation of spam e-mail became a major design issue in e-mail systems along with flooding communication channels and requiring semi automated pre-screening. Keyword search systems evolved into web based search engines, and many software systems had to be redesigned for international searching depending on Search Engine Optimization (SEO) techniques. Human natural language translation systems were needed to attempt to translate the information flow in multiple foreign languages with many software systems being designed for multi-language usage based on design concepts from human translators. Typical computer user bases went from hundreds or thousands of users to many-millions of international users.

1.1.2.7 2000 to Present : Lightweight Methodologies

With the expanding demand for software in many smaller organizations, the need for inexpensive software solutions led to the growth of simpler and faster methodologies that developed running software from requirements to deployment in much quicker and easier way. The use of rapid prototyping evolved to entire lightweight methodologies such as Extreme Programming (XP), which attempted to simplify many areas of software engineering, including requirements gathering and reliability testing for the growing vast number of small software systems. Very large software systems still used heavily documented methodologies with many volumes in the documentation set. However, smaller systems had a simpler and faster alternative approach to managing the development and maintenance of software calculations and algorithms, information storage or retrieval, and display.
1.1 History of Software Engineering

The 21st century brought out some of the best programmers and programs of all time. Software became user friendly and easy to use. Programmers were looking for easier and better ways to write down codes. Life for both the software engineer and the end user became much, much easier.

1.1.3 Current Trends in Software Engineering

Software engineering is relatively a young discipline and is still developing. The directions in which software engineering is developing recently includes:

- **Component Based Software Development (CBSD)**: Component Based Software Development aims to construct complex software systems by means of integrating reusable software components. This approach promises to alleviate the software crisis at great extents following the principle of abstraction.

- **Agile Philosophy**: Agile software development guides software development projects that evolve rapidly with changing expectations and competitive markets. Proponents of this method believe that heavy and document driven processes (like TickIT, CMM and ISO 9000) are fading in importance. Some people believe that companies and agencies export many of the jobs that can be guided by heavyweight processes. Agile philosophy are incorporated in the extreme programming, scrum and lean software development.

- **Aspects Orientation**: Aspects help software engineers to deal with quality attributes by providing tools to add or remove boilerplate code from many areas in the source code. Aspects describe how all objects or functions should behave in particular circumstances. For example, aspects can add debugging, logging or locking control into all objects of particular types. Researchers are currently working to understand how to use aspects to design general purpose code. Related concepts include generative programming and templates.

- **Experimental Software Engineering**: Experimental software engineering is a branch of software engineering interested in devising experiments on software, in collecting data from the experiments and in devising laws and theories from this data. Proponents of this method advocate that the nature of software is such that we can advance the knowledge on software through experiments only.
• **Model driven Approach** : Model driven design develops textual and graphical models as primary design artifacts. Development tools are available that use model transformation and code generation to generate well organized code fragments that serve as a basis for producing complete applications.

• **Software Product Lines** : Software product lines are a systematic way to produce families of software systems instead of creating a succession of completely individual products. This method emphasizes extensive, systematic and formal code reuse to try to industrialize the software development process.

The Future of Software Engineering (FOSE) held at ICSE 2000 documented the state of the art of SE and listed many problems to be solved over the next decade. The FOSE tracks at the ICSE 2000 and the ICSE 2007 conferences also help identify the state of the art in software engineering.

Software has formed an everlasting place in society, because humanity could never return to life without it. Some people even hypothesize that we are already living in the age of artificial intelligence because humans are now capable of doing things not normally humanly possible.