© 2014 IJIRT | Volume 1 Issue 11 | ISSN: 2349-6002

SOFTWARE CRISIS

Kartik Rai, Lokesh Madan, Kislay Anand
Student (B.tech VI^{tIh} sem) Department of Computer science
Dronacharya College Of Engineering,Gurgaon-122506

Abstract- **Software Crisis is restricting the growth of computing field due to huge gap between hardware developments and making use of it through non availability of software systems and competent software development staff. So many problems have enforced the emergence of cognitive type discipline of Software Engineering and its persistence for not meeting the gap imposed. Nature of software is different than systems visible while in operation. Special emphasis is given to identify the problems being faced which need to be addressed for providing solutions to overcome the software crisis. The basic premise of this paper is that unless the problems at the software industry level are solved, no number of technical and project management tools can be of much help in overcoming the software crisis. The author examines the existence of the software crisis, its causes and its serious impact on every walk of life. The nature of software development is discussed, considering it as a craft and as an engineering discipline. After evaluating various reasons and causes, the issue is opened for researchers to get it addressed. Issues like education, professionalization, programmer's productivity, and human factors are discussed. Action on these recommendations requires crossing organizational boundaries, and viewing the problem from a macro perspective. In the age pervasive computing the direction is almost set but no "Silver Bullet" solution is available has been realized by the community. The focus of mechanical systems development is diminishing due to realization of software community that reuse is the solution but with developers' competence.**

Index Terms- Software Crisis, growth of computing, Software Engineering, cognition type discipline, Issues and problems in development of software, programmer, software engineer, software developer, software person, software practitioner and software professional.

I. INTRODUCTION

_ Software is often too complex to be entirely understood by a single individual. We can try to manage complexity by dividing the system into subsystems, but,as systems grow, the interaction between subsystems increases non-linearly.

_ It is notoriously difficult to establish an adequate and stable set of requirementsfor a software system. Often there are hidden assumptions, there is no analyticprocedure for determining when the users have told the developers everything they need to know, and developers and users do not have a common understanding of terms used.

_ The interaction between the different parts of a system makes change difficult.Software is essentially thought stuff (that is, the result of a thought process)and much of what is important about software is not manifest in the programsthemselves (such as the reasons for making design decisions).

_ A requirements specification for a system contains, perhaps implicitly, an application domain model (for example, describing the rules of air traffic). Developmentof application domain theories is very difficult.

All these aspects are directly related to written communication. Managing complexity depends on an ability to document the interfaces (parameters and functionality) of the modules involved. Requirements are an important reference for the whole process and should, therefore, be unambiguously and accessibly described for everyone.

To keep track of changes it is important to document what exactly has been changed.Software can be made more visible by describing non-material artifacts, such as the overall design of a program. Domain models should, just like the requirements, be well documented. Thus, software engineering can benefit from good techniques to describe systems (programs, subsystems, etc.)

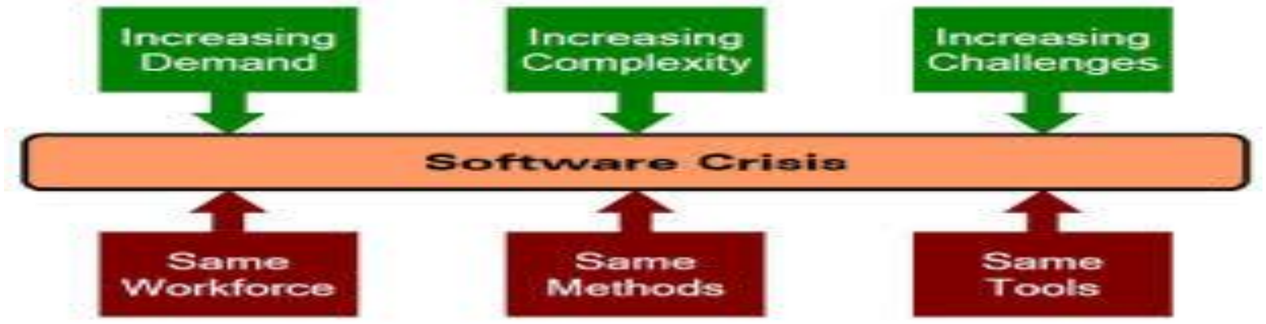
Since last 20-25 years, there has been a rapid increase in the development of programs using a computer. Also, the difficulty level of software has increased to a greater extent. In other words, a drastic change has occurred in the development of computer programs. In order to make the programs more and more predictable, different types of patterns have been created. But the software industry is still many years away from becoming a mature engineering discipline. Even in today's society, software is viewed with suspicion by many individuals, such as senior managers and customers, as something similar to black magic. The result is that software is one of the most difficult artifacts of the modern world to develop and build. Developers work on techniques that cannot be measured or reproduced. All this, lead to a new concept called 'software crisis'. It has become the longest continuing crisis in the engineering world, and it continues unabated.

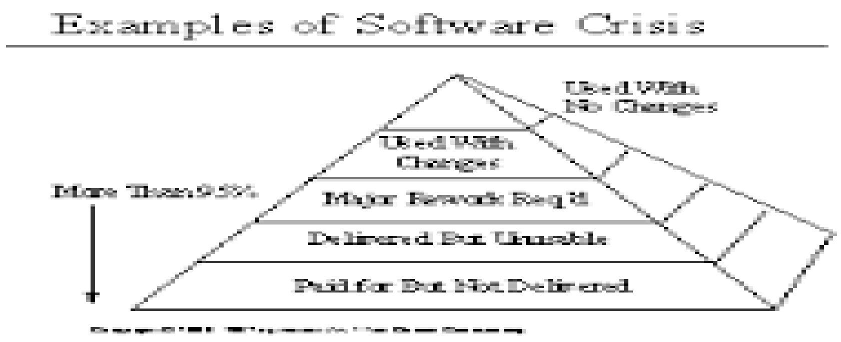
II. SOFTWARE CRISIS

The difficulty of writing the code for a computer program which is correct and understandable is referred to as software crisis. The term software crisis revolves around three concepts: complexity, change and the expectations. This term was given by F. L. Bauer at the first NATO Software Engineering Conference in 1968 at Garmisch, Germany. Current System design approach is exceedingly empirical. It is unable to cope with increasing systems complexity. A number of problems in software development were identified in 1960s, 1970s, and 1980s. The problems that software projects encountered were: the projects ran over-budget, caused damage to property even to life. Despite our rapid progress, the software industry is considered by many to be in a crisis. Some 40 years ago, the term "Software Crisis" emerged to describe the software industry's inability to provide customers with high quality products on schedule.

In general it refers to poorly written, hard to read, error-prone software that often lacks good documentation.

Software crisis is also referred to the inability to hire enough qualified programmers. It has become the longest continuing "crisis" in the engineering world and it continues unabated. The most visible symptoms of the software crisis are late delivery, over budget; Product does not meet specified requirements, inadequate documentation. One of the most serious complaints against software failure is the inability to estimate with acceptable accuracy the cost, resources, and schedule necessary for a software project. Conventional assessment methods have always produced positive results which contribute to the too well-known cost infested and schedule slippage. As the world becomes more and more dependent on computers and as the complexity of software systems continues to rise, the crisis can only get worse. It is particularly severe in vital segments of the economy such as the health and financial services, and the transportation, manufacturing, communication, power generation, and defense industries.





Factors Contributing To The Software Crisis

The factors contributing to the software crisis are numerous. If we examine them in detail then we find what they fall into four different categories which when combined highlight the gravity of the crisis. These main factors are named as follows:-

- Schedule and budget overshooting
- Productivity issues
- Quality issues
- Software maintenance problem

Examination of factors

To examine in detail the issue of the software crisis, it is necessary to determine the factors which contribute towards each set of problems.

 Schedule and budget over shooting

The development of software applications is a thriving business. The money invested in software technology in USA alone is billions of dollars annually; in 1986 this was estimated at 30 billion dollars (see [COACARD 1986]). The estimated average annual growth rate since 1980 has been about 12%. Today the cost of software is typically about 80% of the cost of a complete computer system in contrast to the situation in the late 1950s when the cost of hardware was the major ingredient of the total cost of the system. Business is now approaching the situation where the hardware represents the inexpensive ’core’ surrounded by the precious software layers.

A highly skilled designer will be well paid and his involvement in a project introduces high labour costs. The software development process is labour intensive and typically necessitates a high level of skilled-man time units. Any increase in the time factor produces more cost for the product. The user is complementary to the designer. The user uses the system and enjoys the usefulness of the system which is designed. There is no sense in designing and implementing a system which will not be put into practice. Sometimes the user supplies unrealistic and over ambitious requirements to the designer and a good amount of designer time can be involved in indentifying the invalidity of the user’s requirements. The time required to carry out the feasibility check of user requirements is also typically included in the schedule. Even a slight deviation from the schedule can lead to cost increases.

The computer environment can be defined as the collection of hardware and software resources which are utilized in order to achieve a particular effect, normally defined by a problem description, but including also the tools used to realize a solution to the problem. These resources include the tools for the designer who relies on them in any process of system Development. The collection of tools for each project costs money which contributes to the development expenditure and increases the cost of the product. Moreover, any variation in the environment in which a tool operates or any malfunctioning of the tool causes by delays which ultimately means more cost. Poor estimates of resources and inadequate financial planning also contribute to higher costs; poor quality and cheap resources are rarely reliable and typically cost money. A designer who is being unproductive adds additional expenditure so contributing to greater cost.

A contract in the form of legally binding agreements on both sides is a compromise between the designer and the user or potential user. This typically includes the time scale to be followed in the development process, (generally called the schedule), the budget forecast to meet the financial needs in accord with the schedule drawn up and the requirements of the user for the software product i.e. the specification of the system. A contract with incomplete, unclear, vague and ambiguous requirements which are subjected to several possible interpretations reduces the possibility of a successful outcome and can lead to faulty or misdirected effort. Time, effort and resources absorbed after completion of the schedule leads to an ultimate increase in cost. Ad hoc development - performed without caring for documentation, chronological recording, and verifying the system against the contract - generates confusion and ambiguities leading to some maintenance problems.

Typically delays in producing the software systems are either because there is an overshoot in the budget or delay in delivery of the system. On such occasions, the user refuses to accept the product due to its poor quality. Moreover, potential advantages occur in hitting the markets before competitors do.

 Productivity issues

The main challenge to software developers today is to increase the output per time period of each worker. To understand the issue, let us refer to the industrial revolution in the 18th century. With the achievements related to such a revolution, it became possible for one worker to produce the same amount of work with a machine as hundreds had previously done with their hands. To create an information revolution of a similar kind it is likewise required that one worker must be employed to even greater effect in producing code. This becomes possible if computer software factories are established to produce quality software in bulk. Unfortunately, software production factories are not in use on a large scale; thus a lot of effort form scientists and software engineers is expended. It is worth noting that productivity can be measured, e.g. as the ratio of the number of lines of source code produced per unit of time; care needs to be exercised to account for blank lines of commentary, several statements packed into a single line, etc. Even the software process itself can significantly affect the metric.

The main strands of the software productivity issue are people, the process, the product and the machines. Each strand has many threads. The number of factors conspiring to reduce productivity is large and numerous causes are interrelated. Small systems can be produced by an individual designer facing minimal problems. The problems arising from individuals are due to their limitations in capabilities, years of experience, language experience, and experience with similar problems; the production of large systems by a team of designers offers a further set of problems. Individual factors, team factors and the factors which are due to the nature and degree of communication among the members of a team influence the extent of the productivity. It is an old saying that a chain is only as strong as its weakest link; this applies to teams of people as well as to metal chains .Wrong selection of languages, unsuitable development methodologies, lack of attention to verification, validation and the system environment - all these limit productivity at a fundamental level. On the other hand the productivity level is greatly enhanced if a good language which suits the problem requirements is chosen, a well defined and thoroughly worked out methodology is adopted for design, the design process includes ‘walk throughs’ and milestones, implementation is carried out structurally, verified automatically and an effective and supportive environment is used.

The problems associated with the product itself are: the kind of software, the quality requirements and the human interface. A software product of poor quality is not attractive to the user. Moreover, to produce high quality software needs time, commitment and other resources (e.g. tools) which affect the level of productivity. If the product is not accepted by the user then the time of a skilled designer is wasted. Low quality, large software products which do not meet the actual development tie requirements and are developed without proper control and modularity typically cause future maintenance problems. Organizations which use computer based systems invest enormous efforts to avoid the pitfalls of such systems and follow the wisdom of a (well known) proverb: “It is better to die instead of remaining alive and enduring continuously bleeding wounds”. This produces serious setbacks to the economic and financial resources of organizations and causes a decrease in the morale level of team.

Factors (from the computer system (machines)) conspiring against productivity include characteristics of the machine, its constraints and its behaviour. Software production which is carried out without keeping these factors in view is unlikely to suit the purpose for which it was designed. Unsuitable systems which do not permit the proper control of the hardware cannot generally be modified easily to produce a successful system and take up the development resources in an unproductive fashion. Recently, they have been many new hardware machines on the market with different characteristics and constraints; their individual behaviour hinders the bulk production of software, leading to concerns about portability and techniques for achieving this. An important aspect of this is the availability or other wise of tools and the appropriateness of techniques for achieving productivity. Thus source control systems, data bases, separate compilation facilities are all aspects of these concerns.

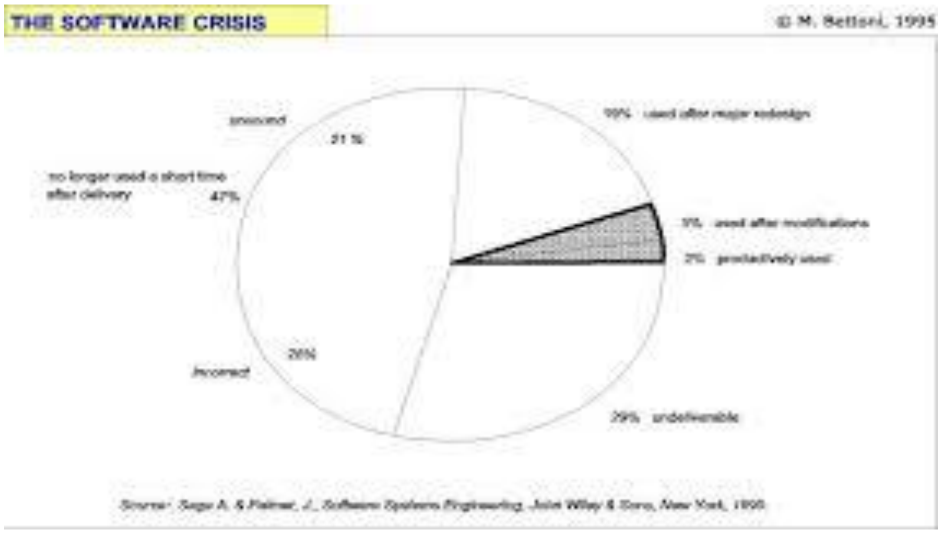
- **Quality issues**

There is little value in computer systems if they are not reliable, secure and efficient. The emphasis must be on the quality of software product; indeed the presence of quality is a prime objective of software engineering. A simple definition of the quality is contained in [Ratcliff 1987], where an approach is adopted which is based upon the observation of defining is quality of software products as their total ‘usefulness’ to the users. And total usefulness is further sub-divided into current usefulness, which deals with the usage of the software in the existing context, and potential usefulness which contributes to the usage of the same system in any other context resulting from slight variation of the environments, changing requirements, varying applications, etc

Software Crisis in terms of statistics in 1990's

* 31 % of projects canceled
* 52.7% cost an average of 189% over budget
* 84% are late or over budget (91% for large companies.)
* The average system is delivered without 58% of proposed functionalities
* $81 billion in 1995 for cancelled projects
* $51 billion in 1995 for over-budget projects

Only 16.2% of software projects are completed on-time and on-budget. In larger companies, a meager 9% of technology projects come in on-time and on-budget. In addition, about one third of all projects will be canceled before they ever get completed. Further results indicate 53% of projects will cost an average of 189% of their original estimates. In financial terms this analysis revealed that over $100 billion in cancellations and $60 billion in budget over runs occur in the Software Sector annually.



III. CAUSES

Software engineering today is in severe crisis. The situation is particularly grim because this crisis is not widely acknowledged by the software development industry. The causes of software crisis were linked to the overall complexity of the software process and the relative immaturity of software engineering as a profession. The main reason for the crisis is the lack of a sound software construction methodology with which to manage the high complexity of modern applications. The notion of a software crisis emerged at the end of the 1960s. An early use of the term is in Edsger Dijkstra's ACM Turing Award Lecture, "The Humble Programmer" (EWD340), given in 1972 and published in the Communications of the ACM. Dijkstra says,

"The major cause of the software crisis is] that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem." - Edsger Dijkstra

* The cost of owning and maintaining software in the 1980's was twice as expensive as developing the software.
* During the 1990's, the cost of ownership and maintenance increased by 30% over the 1980's.

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.
To explain the present software crisis in simple words, consider the following. The expenses that organizations all around the world are incurring on software purchases compared to those on hardware purchases have been showing a worrying trend over the years.
Not only are the software products turning out to be more expensive than hardware, but they also present a host of other problems to the customers: software products are difficult to alter, debug, and enhance; use resources no optimally; often fail to meet the user requirements; are far from being reliable; frequently crash; and are often delivered late. Among these, the trend of increasing software costs is probably the most important symptom of the present software crisis.

Software crisis: The present scenario
The Software Crisis began 4 decades ago and continues today. In the 60s, we began to speak of a "software crisis". A thirty year long "crisis" was beginning. A world-wide research effort also began.
Today, the situation is quite different. We have a Science of Programming. We know a great deal about how to design and document software, but the "Software Crisis" continues unabated!
The software crisis continues because the communication between Computer Scientists and those who write software, including the Engineers, has been very poor. Current software standards, are weak, superficial, and are not based on software science. Process oriented "standards" are empty because there are no product/document standards.

IV. SOLUTION

Over the last twenty years many different paradigms have been created in attempt to make software development more predictable and controllable. While there is no single solution to the crisis, much has been learned that can directly benefit today's software projects. One of the possible solutions to the software crisis is the study of software engineering. It is believed that the only satisfactory solution to the present software crisis can possibly come from a spread of software engineering practices among the engineers, coupled with further advancements in the software engineering discipline itself. Software engineering is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use. As a solution to this software crisis, we must apply a disciplinary artistry; using tools that help us manage this complexity. The skilled systems engineer, can through the use of these techniques and by the application of systems engineering methods and project management skills, reduce the demands placed on software engineers, hence reducing the software engineering effort and also reducing the total development cost. But still, there is no single approach which will prevent all the consequences of software crisis in all cases. While there is no single solution to the crisis, much has been learned that can directly benefit today's software projects. It is our human inability to deal with complexity that lies at the root of the software crisis. It has been noted frequently that we are experiencing a software crisis, characterized by our inability to produce correct, reliable software within budget and on time. No doubt, many of our failures are caused by the inherent complexity of the software development process, for which there often is no analytical description.
Through the use of computer-aided symbolic specification techniques and simulation, and with an understanding of the software development process, the skilled systems engineer can contribute to the resolution of the software crisis. The skilled systems engineer, can through the use of these techniques and by the application of systems engineering methods and project management skills, reduce the demands placed on software engineers, hence reducing the software engineering effort and also reducing the total development cost.
In software engineering, the possible solution to software metrics is the use of proper software metrics and the proper utilization of these metrics. For the implementation of this solution to the problem of software crisis some pre-requisites are there.

They are:
1. Knowledge of basic statistics and experimental design.
2. Basic understanding of commonly used software life cycle models, at least to the level covered in an introductory senior or graduate-level software engineering course

3. Experience working as a team member on a software development project.

In addition, for maximum utility in analytic studies and statistical analyses, metrics should have data values that belong to appropriate measurement scales. Software engineering is still a very young discipline. There are encouraging signs that we are beginning to understand some of the basic parameters that are most influential in the processes of software production.

For the projects which are delivered late must adopt the following methodology:

Project Planning & Scheduling

Project planning means creating work breakdown, and then allocate responsibilities to the developers over time. Project planning consists of construction of various tasks, timelines and essential pathways including Gantt charts and PERT charts and different written plans for various situations. It is quite usual in software development process to work backward from the project end date which results in complete software project failure. It is impossible that a project can be completed efficiently from the planning stage to the implementation stage. Allocation of roles and responsibilities has to be clearly defined. Proper scheduling is also required before the start of the project. It includes the time scheduling, teams scheduling.

For the projects running out of budget, cost estimation methodology must be applied:

Cost Estimation

Cost estimation is mainly involved the cost of effort to produce the software project. But it's not limited to the effort only. It also includes the hardware and software cost, training the employees and customer, travelling to the customer, networking and communication costs. Cost estimation should be done as a part of the software process model. Cost estimation needs to be done well before the start of the project **development.**

Failure of the budgeting for the cost of the project results in complete disaster. Development tools, cost and hardware cost also need to be estimated first.

In order to cope up with the increasing system complexity, risk management should be applied:

V. RISK MANAGEMENT

Risk management is an important factor towards software project failure if it's not managed timely and effectively. As nothing can be predicted that what will happen in future so we have to take the necessary steps in the present to take any uncertain situation in the future. Risk management means dealing with a concern before it becomes a crisis. Project managers have to identify the areas where the risk can be and how it can affect the development of the project. Risk can be of technical nature or non technical. After the risk is identified there is a need to make the categories of that risk. Risk analysis is the process of examining the project results and deliverables after the risk analysis and applying the technique to lower the risk. After the risk is analyzed, the next step is to priorities the risk. At first focus on the most sever risk first; and les sever later. Managing the risk to achieve the desired results and deliverables is done through controlling the risk at its best.

VI. CONCLUSION

Thus, we have discussed software crisis, its causes, the present status and the possible solution to this crisis. Software engineering appears to be one of the few options available to tackle software crisis. Software engineering is the application of a systematic, disciplined, quantifiable approach to development, operation, and maintenance of software; that is, the application of engineering to software.

It is believed that the only satisfactory solution to the present software crisis can possibly come from a spread of software engineering practices among the engineers, coupled with further advancements to the software engineering discipline itself. The solution being advocated is to place a special emphasis on fault tolerance software engineering which would provide a set of methods, techniques, models and tools that would exactly fit application domains, fault assumptions and system requirements and support disciplined and rigorous fault tolerance throughout all phases of the life cycle. Also, the software must not be considered equivalent to a widget, i.e. a gadget.

REFERENCES

Books referred:
Software engineering: concepts and techniques - Peter Naur
Software engineering- Richard H. Thayer
Software engineering-Rajiv mal
Software engineering-kk aggarwal
Websites and links:
en.wikipedia.org/wiki/Software_crisis
www.apl.jhu.edu/Classes/Notes/.../SoftwareEngineeringOverview.PDF
http://www.unt.edu/benchmarks/archives/1999/july99/