# SOFTWARE ENGINNERING

AARTI SINGH
Singh12.aarti@gmail.com

ANANYA ANIKESH
ananyaanikesh@gmail.com

## ABSTRACT

The phrase 'software engineering' has many meanings. One central meaning is the reliable development of dependable computer-based systems, especially those for critical applications. This is not a solved problem. Failures in software development have played a large part in many fatalities and in huge economic losses. Software is intangible, complex, and capable of being transformed by a computer. Much effort has been devoted to overcoming the difficulties due to intangibility and complexity, but too little has been devoted to exploiting the third characteristic. A processoriented view of software may lead to substantial improvements in this and other respects.t his research paper tells about benefits of software engineering application its validation its problem design etc

***KEYWORDS:-*** *Softwarin engineering,research validation,benefits of software engineering,application,problem desgning,verification if software product*

## INTRODUCTION:-

Software developers have long aspired to a place among the ranks of respected engineers. But even when they have focused consciously on that aspiration [15; 3] they have made surprisingly little effort to understand the reality and practices of the established engineering branches. One notable difference between software engineering and physical engineering is that physical engineers pay more attention to their products and less to the processes and methods of their trade. Physical engineering has evolved into a collection of specialisations–electrical power engineering, aeronautical engineering, chemical engineering, civil engineering, automobile engineering, and several others. Within each specialisation the practitioners are chiefly engaged in normal design.

Software engineering, or, more generally, the development of software-intensive systems, has not yet evolved into adequately differentiated specialisations, and has therefore not yet established normal design practices. There are, of course, exceptional specialised areas such as the design of compilers, file systems, and operating system. The aspiration to 'software engineering' expresses a widely held belief that software development practices and theoretical foundations should be modelled on those of the established engineering branches

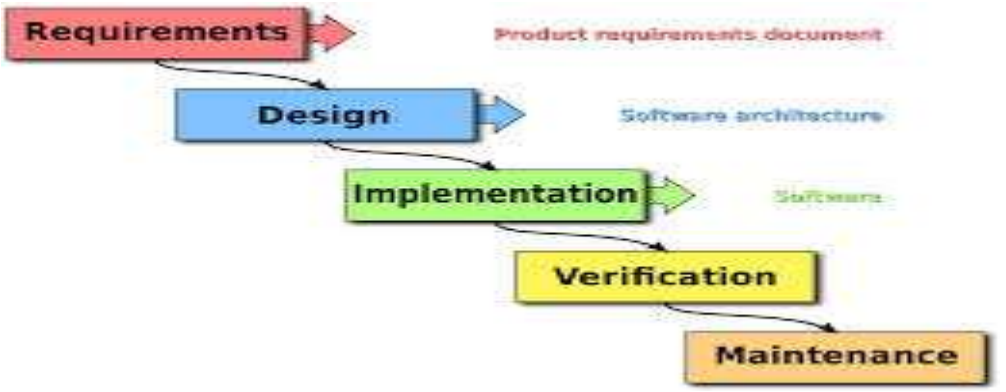
## SOFTWARE HISTORY

Two streams may be distinguished in the evolving modern practice of software development since it began in the 1940s. One may be called the formal stream. Programs are regarded as mathematical objects: their properties and behaviour can be analysed formally and predictions of the results of execution can be formally proved or disproved. The other stream may be called the intuitive stream. Programs are regarded as structures inviting human comprehension: the results of their execution can be predicted—not always reliably—by an intuitive process of mental enactment combined with

some informal reasoning. Both streams have a long history. A talk by Alan Turing in 1949 [Turing49] used assertions over program variables to construct a formal proof of correctness of a small program to compute the factorial function. Techniques of program structuring, devised and justified by intuition, came to prominence in the 1960s with the control structures of Algol 60 [Naur60], Conway's invention of coroutines [Conway63], and the class concept of Simula67 [Dahl72]. Dijkstra's advocacy of restricted control flow patterns in the famous GO TO letter [Dijkstra68] rested on their virtue of minimising the conceptual gap between the static program text and its dynamic execution: the program would be more comprehensible. In further developments in structured programming the two streams came together. A structured program text was not only easier to understand: the nested structure of localised contexts allowed a structured proof of correctness based on formal reasoning. At this stage the academic and research communities made an implicit choice with far-reaching consequences. Some of the intellectual leaders of those communities were encouraged by the success and promise of formal mathematical techniques to focus their attention and efforts on that stream. They relaxed, and eventually forsook, their interest in the intuitive aspects of program design and structure. For those researchers themselves the choice was fruitful: study of the more formal aspects of computing stimulated a rich flow of results in that particular branch of logic and mathematics. For the field of software development as a whole this effective separation of the formal and intuitive streams was a major loss. The formal stream flowed on, diverging further and further from the concerns and practices of realistic software development projects. The intuitive stream, too, flowed on, but in increasing isolation. Systems became richer and more complex, and the computer's role in them became increasingly one of intimate interaction with the human and physical world. Software engineering came to be less concerned with purely symbolic computation and more concerned with the material world and with the economic and operational purposes of the system of which software was now only a part. Development projects responded increasingly to economic and managerial imperatives and trends rather than to intellectual or scientific disciplines.

## SOFTWARE ENGINEERING

Structured programming was ideally suited to what we may call pure programming. The archetypical expository examples of pure programming are calculating the greatest common divisor of two integers, sorting an array of integers, solving the travelling salesman problem, or computing the convex hull of a set of points in 3-space. These problems proved surprisingly fertile in stimulating insights into program design technique, but they were all limited in a crucial way: they required only computation of symbolic output results from symbolic input data. The developer investigates the problem world, identifies a symbolic computational problem that can usefully be solved by computer, and constructs a program to solve it. The user captures the input data for each desired program execution and presents it as input to the machine. The resulting output is then taken by the same or another user and applied in some way to guide action in the problem world.




## 1. SOFTWARE ARE NOT SOFTWARE ENGINEERING PRODUCT

A common usage speaks of the physical and human world as the 'environment' of a computer-based system. This usage is seriously misleading. The word 'environment' suggests that ideally the surrounding physical and human world affects the proper functioning of the software either benignly or not at all. If the environment provides the right temperature and humidity, and no earthquakes occur, the software can get on with its business independently, without interference from the world

## 2. SOFTWARE TECHNOLOGY MATURATION

Redwine and Riddle reviewed a number of software technologies to see how they develop and propagate. They found that it typically takes 15-20 years for a technology to evolve from concept formulation to the point where it's ready for popularization. They identify six typical phases:

• Basic research. Investigate basic ideas and concepts, put initial structure on the problem, frame critical research questions.

• Concept formulation. Circulate ideas informally, develop a research community, converge on a compatible set of ideas, publish solutions to specific subproblems.

• Development and extension. Make preliminary use of the technology, clarify underlying ideas, generalize the approach.

• Internal enhancement and exploration. Extend approach to another domain, use technology for real problems, stabilize technology, develop training materials, show value in results.

• External enhancement and exploration. Similar to internal, but involving a broader community of people who weren't developers, show substantial evidence of value and applicability.

• Popularization. Develop production-quality, supported versions of the technology, commercialize and market technology, expand user community. Redwine and Riddle presented timelines for several software technologies as they progressed through these phases up until the mid-1980s. He presented a similar analysis for the maturation of software architecture in the 1990s

## 3.RESEARCH VALIDATION IN SOFTWARE ENGINEERING

Type of validation Examples

Analysis I have analyzed my result and find it satisfactory through ...ormal analysis) … rigorous derivation and proof (empirical model) … data on controlled use(controlled … carefully designed statistical experiment) experiment

Experience My result has been used on real examples by someone other than me, and the evidence of its correctness / usefulness / effectiveness is …alitative model) … narrative(empirical model, … data, usually statistical, on practice (notation, tool) … comparison of this with similar results in technique) actual use
Example Here's an example of how it works on (toy example) … a toy example, perhaps motivated by reality (slice of life) …a system that I have been developing

Evaluation Given the stated criteria, my result... (descriptive model) … adequately describes the phenomena of interest … (qualitative model) … accounts for the phenomena of interest… (empirical model) … is able to predict … because …, or … gives results that fit real data … Includes feasibility studies, pilot projects

Persuasion I thought hard about this, and I believe... (technique) … if you do it the following way, … (system) … a system constructed like this would … (model) … this model seems reasonable Note that if the original question was about feasibility, a working system, even without analysis, can be persuasive

## APPLICATION OF SOFTWARE ENGINEERING

Software engineering (SE) is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software. It is the application of engineering to software because it integrates significant mathematics, computer science and practices whose origins are in engineering. It is also defined as a systematic approach to the analysis, design, assessment, implementation, testing, maintenance and reengineering of software, that is, the application of engineering to software

## BENEFITS OF SOFTWARE ENGINEERING

### Software is Complex

- Malleable
- Intangible
- Abstract
- Solves complex problems
- Interacts with other software and hardware
- Not continuous

## CONCLUSION:-

Software engineering will benefit from a better understanding of the research strategies that have been most successful. The model presented here reflects the character of the discipline: it identifies the types of questions software engineers find interesting, the types of results we produce in answering those questions, and the types of evidence that we use to evaluate the results.

## REFRENCES

1. Impact Project. "Determining the impact of software engineering research upon practice. Panel summary, Proc. 23rd International Conference on Software Engineering (ICSE 2001), 2001

2. William Newman. A preliminary analysis of the products of HCI research, using pro forma abstracts. Proc 1994 ACM SIGCHI Human Factors in Computer Systems Conference (CHI '94), pp.278-284.

3. Samuel Redwine, et al. DoD Related Software Technology Requirements, Practices, and Prospects for the Future. IDA Paper P-1788, June 1984.

4. S. Redwine & W. Riddle. Software technology maturation. Proceedings of the Eighth International Conference on Software Engineering, May 1985, pp. 189-200.

5. Mary Shaw. Prospects for an engineering discipline of software. IEEE Software, November 1990, pp. 15-24.

[Jackson00] Michael Jackson; Problem Frames: Analysing and Structuring Software Development Problems; Addison-Wesley, 2000.

[Klein03] Gary Klein; Intuition at Work; Doubleday, 2003. [Naur60] J W Backus, F L Bauer, J Green, C Katz, J McCarthy, A J Perlis, H Rutishauser, K Samelson, B Vauquois, J H Wegstein, A van Wijngaarden, M Woodger, ed Peter Naur;

Report on the Algorithmic Language ALGOL 60; Communications of the ACM Volume 3 Number 5, pages 299-314, May, 1960.