

## Evaluation & Optimization of Software Engineering

Ashiqur Rahman<sup>1</sup>, Asaduzzaman Noman<sup>2</sup>, Atik Ahmed Sourav<sup>3</sup>,  
Shakh Md. Alimuzjaman Alim<sup>4</sup>,

<sup>1</sup>(M.Sc in Information Technology (IT), Jahangirnagar University, Bangladesh)

<sup>2</sup>(CSE, Royal University of Dhaka, Bangladesh)

<sup>3</sup>(CSE, Royal University of Dhaka, Bangladesh)

<sup>4</sup>(EEE, Royal University of Dhaka, Bangladesh)

**ABSTRACT:** The term is made of two words, software and engineering. Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called software product. Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods. The outcome of software engineering is an efficient and reliable software product. IEEE defines software engineering as: The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software.

**Keywords** –Programming, IEEE, FORTRAN, COBOL, Client.

### I. INTRODUCTION

Software engineering is the branch of computer science that creates practical, cost-effective solutions to computing and information processing problems, preferentially by applying scientific knowledge, developing software systems in the service of mankind. This paper covers the fundamentals of software engineering, including understanding system requirements, finding appropriate engineering compromises, effective methods of design, coding, and testing, team software development, and the application of engineering tools. The process of developing a software product using software engineering principles and methods is referred to as software evolution. This includes the initial development of software and its maintenance and updates, till desired software product is developed, which satisfies the expected requirements. The process of developing a software product using software engineering principles and methods is referred to as software evolution. This includes the initial development of software and its maintenance and updates, till desired software product is developed, which satisfies the expected requirements

Software paradigms refer to the methods and steps, which are taken while designing the software.

- Requirement gathering
- Software design
- Programming

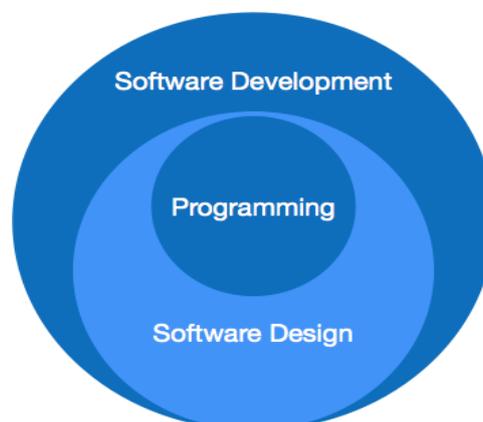


Figure 1: Software paradigm

## II. RELATED WORK

From its beginnings in the 1960s, writing software has evolved into a profession concerned with how best to maximize the quality of software and of how to create it. Quality can refer to how maintainable software is, to its stability, speed, usability, testability, readability, size, cost, security, and number of flaws or "bugs", as well as to less measurable qualities like elegance, conciseness, and customer satisfaction, among many other attributes. How best to create high quality software is a separate and controversial problem covering software design principles, so-called "best practices" for writing code, as well as broader management issues such as optimal team size, process, how best to deliver software on time and as quickly as possible, work-place "culture", hiring practices, and so forth. The most important development was that new computers were coming out almost every year or two, rendering existing ones obsolete. 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. 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. Computer hardware was application-specific. Scientific and business tasks needed different machines. 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 notion of reuse flourished. As software was free, user organizations commonly gave it away. Groups like IBM's scientific user group SHARE offered catalogues of reusable components. Academia did not yet teach the principles of computer science. Modular programming and data abstraction were already being used in programming. For decades, solving the software crisis was paramount to researchers and companies producing software tools. 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 the 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.

## III. WHY SOFTWARE ENGINEERING?

**Large software** - It is easier to build a wall than to a house or building, likewise, as the size of software become large engineering has to step to give it a scientific process.

**Scalability**- If the software process were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing one.

**Cost**- As hardware industry has shown its skills and huge manufacturing has lower down the price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.

**Dynamic Nature**- If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.

**Quality Management**- Better process of software development provides better and quality software product. A software product can be judged by what it offers and how well it can be used. This software must satisfy on the following grounds:

- Operational
- Transitional
- Maintenance

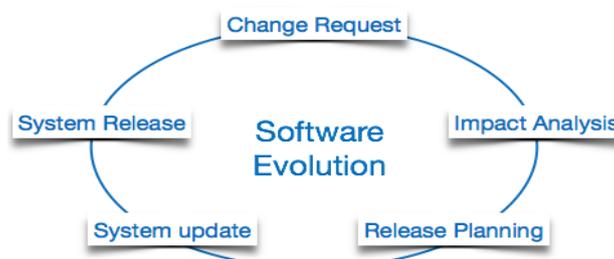


Figure 2: Software Evolution

- Process models define structured set of activities, tasks, milestones and work products required to develop high quality software.
- All software process models accommodate generic framework activities but each applies a different emphasis to these activities.

The deployment of the system includes changes and enhancements before the decommissioning or sunset of the system. Maintaining the system is an important aspect of SDLC. As key personnel change positions in the organization, new changes will be implemented, which will require system updates.

#### IV. OPERATIONAL, TRANSITIONAL & MAINTENANCE

Operational process tells us how well software works in operations. It can be measured on:

- Budget
- Usability
- Efficiency
- Correctness
- Functionality
- Dependability
- Security
- Safety

Transitional aspect is important when the software is moved from one platform to another:

- Portability
- Interoperability
- Reusability
- Adaptability

Maintenance aspect briefs about how well a software has the capabilities to maintain itself in the ever-changing environment:

- Modularity
- Maintainability
- Flexibility
- Scalability

#### V. SOFTWARE MYTHS

The development of software requires dedication and understanding on the developers' part. Many software problems arise due to myths that are formed during the initial stages of software development. Unlike ancient folklore that often provides valuable lessons, software myths propagate false beliefs and confusion in the minds of management, users and developers. Managers, who own software development responsibility, are often under strain and pressure to maintain a software budget, time constraints, improved quality, and many other considerations. Common management myths are listed below:

- Software standards provide software engineers with all the guidance they need - Learn to use them
- People with modern computers have all the software development tools they need - Need good CASE tools
- Adding people is a good way to catch up when a project is behind schedule - Late addition makes it later
- Giving software projects to outside parties to develop solves software project management problems - Need to learn how to manage and control software projects
- A general statement of objectives from the customer is all that is needed to begin a software project - spend time to have very good understanding of customer requirements
- Project requirements change continually and change is easy to accommodate in the software design
  - Understand the requirements first then write codes. It costs more to change later
- Once a program is written, the software engineer's work is finished
  - It is only the beginning
- There is no way to assess the quality of a piece of software until it is actually running on some machine. The only deliverable from a successful software project is the working program.
  - Practice formal/peer review
- Software engineering is all about the creation of large and unnecessary documentation not shorter development times or reduced costs
  - Better quality leads to reduced work



Figure 3: Flow chart of processing.

**VI. Acknowledgements Capability Maturity Model Integration**

Capability Maturity Model Integration (CMMI) is a process improvement training and appraisal program and service administered and marketed by Carnegie Mellon University (CMU) and required by many DoD and U.S. Government contracts, especially in software development. CMU claims CMMI can be used to guide process improvement across a project, division, or an entire organization. CMMI defines the following maturity levels for processes: Initial, Managed and Defined. Currently supported is CMMI Version 1.3. CMMI is registered in the U.S. Patent and Trademark Office by CMU. CMMI was developed by a group of experts from industry, government, and the Software Engineering Institute (SEI) at CMU. CMMI models provide guidance for developing or improving processes that meet the business goals of an organization. A CMMI model may also be used as a framework for appraising the process maturity of the organization. By January of 2013, the entire CMMI product suite was transferred from the SEI to the CMMI Institute, a newly created organization at Carnegie Mellon. CMMI originated in software engineering but has been highly generalized over the years to embrace other areas of interest, such as the development of hardware products, the delivery of all kinds of services, and the acquisition of products and services. The word "software" does not appear in definitions of CMMI. This generalization of improvement concepts makes CMMI extremely abstract. CMMI currently addresses three areas of interest:

1. Product and service development — CMMI for Development (CMMI-DEV),
2. Service establishment, management, — CMMI for Services (CMMI-SVC), and
3. Product and service acquisition — CMMI for Acquisition (CMMI-ACQ).

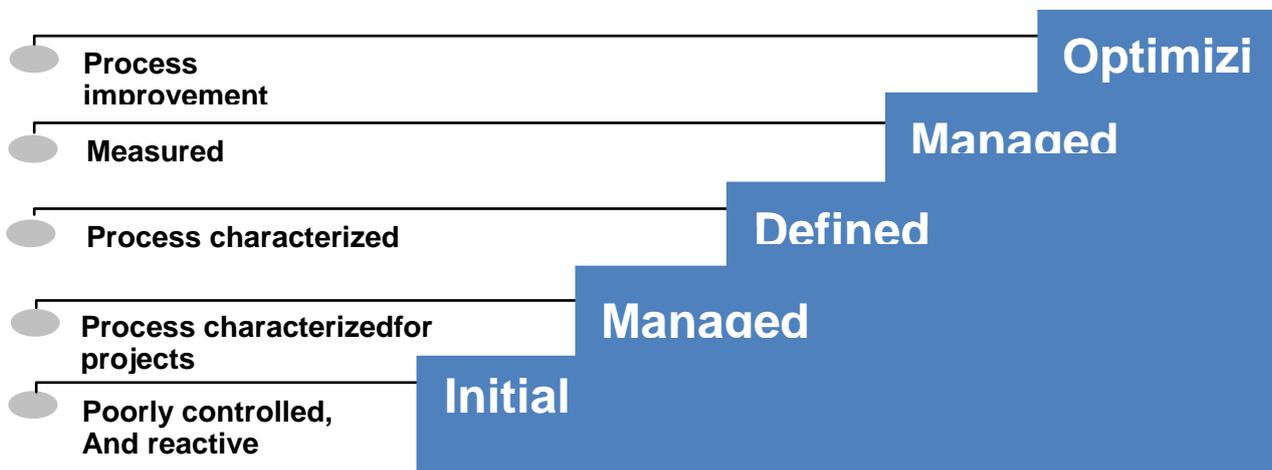


Figure 4: Maturity Levels

A process area (PA) is a cluster of related practices in an area that, when performed collectively, satisfy a set of goals considered important for making significant improvement in that area. Practices are actions to be performed to achieve the goals of a process area. All CMMI process areas are common to both continuous and staged representations. A process area is NOT a process description.

Level	Focus	Process Areas
5 Optimizing	<i>Continuous Process Improvement</i>	Organizational Innovation and Deployment Causal Analysis and Resolution
4 Quantitatively Managed	<i>Quantitative Management</i>	Organizational Process Performance Quantitative Project Management
3 Defined	<i>Process Standardization</i>	Requirements Development Technical Solution Product Integration Verification Validation Organizational Process Focus Organizational Process Definition Organizational Training Integrated Project Management for IPPD Risk Management Integrated Teaming Integrated Supplier Management Decision Analysis and Resolution Organizational Environment for Integration
2 Managed	<i>Basic Project Management</i>	Requirements Management Project Planning Project Monitoring and Control Supplier Agreement Management Measurement and Analysis Process and Product Quality Assurance Configuration Management
1 Initial		



Figure 5: Process Area



Figure 6: The Capability Levels

**V. CONCLUSION**

Software engineering is about teams and it is about quality. The problems to solve are so complex or large, that a single developer cannot solve them anymore. Software engineering is also about communication. Teams do not consist only of developers, but also of testers, architects, system engineers, customer, project managers, etc. Software projects can be so large that we have to do careful planning. Implementation is no longer just writing code, but it is also following guidelines, writing documentation and also writing unit tests. But unit tests alone are not enough. The different pieces have to fit together. And we have to be able to spot problematic areas using metrics. Once we are finished coding, that does not mean that we are finished with the project: for large projects maintaining software can keep many people busy for a long time. Since there are so many factors influencing the success or failure of a project, we also need to learn a little about project management and its pitfalls, but especially what makes projects successful. And last but not least, a good software engineer, like any engineer, needs tools, and you need to know about them.

**REFERENCES**

- [1] Victor R. Basili. The experimental paradigm in software engineering. In Experimental Software Engineering Issues: Critical Assessment and Future Directives. Proc of Dagstuhl-Workshop, H. Dieter Rombach, Victor R. Basili, and Richard Selby (eds), published as Lecture Notes in Computer Science #706, Springer-Verlag-1993.
- [2] Geoffrey Bowker and Susan Leigh Star: Sorting Things Out: Classification and Its Consequences. MIT Press, 1999.
- [3] Frederick P. Brooks, Jr. Grasping Reality through Illusion - Interactive Graphics Serving Science. Proc 1988 ACM SIGCHI Human Factors in Computer Systems Conf (CHI '88) pp. 1-11.
- [4] Rebecca Burnett. Technical Communication. Thomson Heinle 2001.
- [5] Thomas F. Gieryn. Cultural Boundaries of Science: Credibility on the line. Univ of Chicago Press, 1999.
- [6] ICSE 2002 Program Committee. Types of ICSE papers. <http://icse-conferences.org/2002/info/paperTypes.html>
- [7] Impact Project. Determining the impact of software engineering research upon practice. Panel summary, Proc. 23rd International Conference on Software Engineering (ICSE 2001), 2001.
- [8] Ellen Isaacs and John Tang. Why don't more non-North-American papers get accepted to CHI? <http://acm.org/sigchi/bulletin/1996.1/isaacs.html>