Week 1: **01 INTRODUCTION TO SOFTWARE ENGINEERING**

WHAT IS A SOFTWARE?

A software system usually consists of a number of separate programs, configuration files, which are used to set up these programs, system documentation, which describes the structure of the system, and user documentation, which explains how to use the system and web sites for users to download recent product information.

There are two fundamental Types of Software Product:

(1) Generic products. These are stand-alone systems that are produced by a development organization and sold on the open market to any customer who is able to buy them. Examples of this type of product include software for PCs such as databases, word processors, drawing packages and project management tools.

(2) Customized products. These are systems which are commissioned by a particular customer. A software contractor develops the software especially for that customer. Examples of this type of software include control systems for electronic devices, systems written to support a particular business process and air traffic control systems.

An important difference between these types of software is that, in generic products, the organization that develops the software controls the software specification. For custom products, the specification is usually developed and controlled by the organization that is buying the software. The software developers must work to that specification.

WHAT IS SOFTWARE ENGINEERING?

Let us understand what Software Engineering stands for.

- **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 is all about developing products, using well-defined, scientific principles and methods.**

Software Engineering is an engineering branch associated with development of software product using well-defined scientific principles, methods and procedures. The outcome of software engineering is an efficient and reliable software product.

A formal definition of software engineering might sound something like, "An organized, analytical approach to the design, development, use, and maintenance of software."

More intuitively, software engineering is everything you need to do to produce successful software. It includes the steps that take a raw, possibly nebulous idea and turn it into a powerful and intuitive application that can be enhanced to meet changing customer needs for years to come.

One of the big differences between software engineering and most other kinds of engineering is that software isn't physical. It exists only in the virtual world of the computer. That means it's easy to make changes to any part of a program even after it is developed.

IEEE defines Software Engineering as:

(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

(2) The study of approaches as in the above statement.

Fritz Bauer, a German computer scientist, defines Software Engineering as:

"Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and work efficiently on real machines."

Software Engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification to maintaining the system after it has gone into use. In this definition, there are two key phrases:

(1) Engineering discipline. Engineers make things work. They apply theories, methods and tools where these are appropriate, but they use them selectively and always try to discover solutions to problems even when there are no applicable theories and methods.

(2) All aspects of software production. Software engineering is not just concerned with the technical processes of software development but also with activities such as software project management and with the development of tools, methods and theories to support software production.

HISTORY OF SOFTWARE ENGINEERING

The term 'software engineering' was suggested at conferences organized by NATO in 1968 and 1969 to discuss the 'software crisis'. The software crisis was the name given to the difficulties encountered in developing large, complex systems in the 1960s. It was proposed that the adoption of an engineering approach to software development would reduce the costs of software development and lead to more reliable software.

Key Dates in the History of Software Engineering are:

1968: NATO conference on software engineering. Publication of Dijkstra's note on the dangers of the goto statement in programs.

Early 1970s. Development of the notions of structured programming. Publication of Parnas's paper on information hiding. Development of Pascal programming language. Development of Smalltalk languages which introduced notions of object-oriented development.

Late 1970s. Early use of software design methods such as Yourdon and Constantine's structured design. Development of first programming environments.

Early 1980s. Development of the Ada programming language which included notions of structured programming and information hiding. Proposals for software engineering environments. CASE tools introduced to support design methods. Development of algorithmic approaches to software costing and estimation. Publication of the 1st edition of this book as the first student textbook on software engineering.

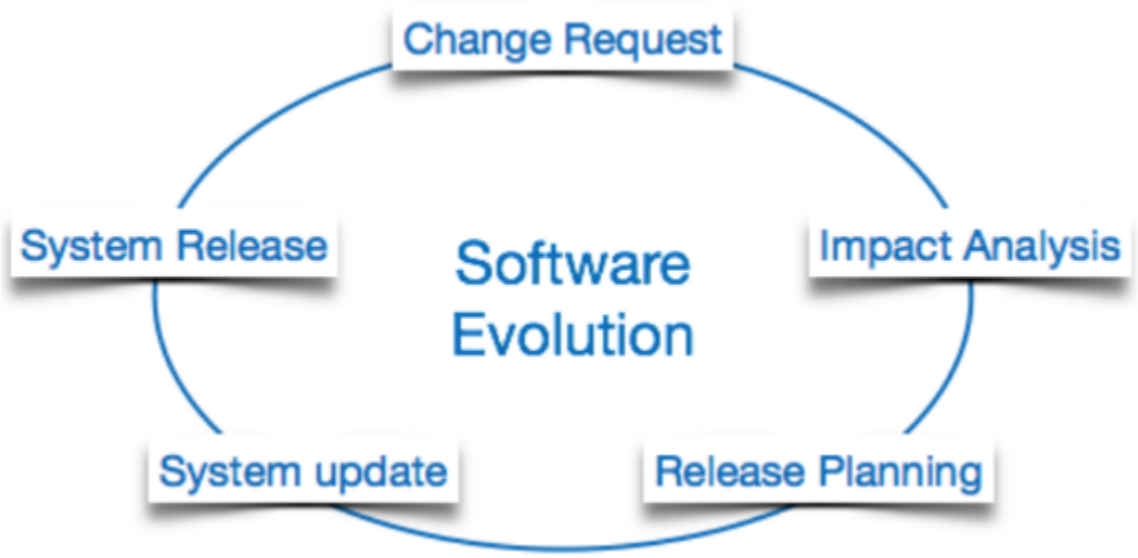
Late 1980s. Increased use of object-oriented programming through languages such as C++ and Objective-C. Introduction of object-oriented design methods. Extensive use of CASE tools.

Early 1990s. Object-oriented development becomes a mainstream development technique. Commercial tools to support requirements engineering become available.

Late 1990s. Java is developed and released in the mid-1990s. Increasing attention paid to notions of software architecture. Client-server distributed architectures are increasingly used. Notion of component-based software engineering is proposed. The UML is proposed, integrating several separately developed notations for representing object-oriented systems.

Early 2000s. Use of integrated development environments becomes more common. Use of stand-alone CASE tools declines. Use of the UML becomes widespread. Increasing use of scripting languages such as Python and PERL for software development. C# developed as a competitor to Java.

SOFTWARE EVOLUTION



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.

Evolution starts from the requirement gathering process. After which developers create a prototype of the intended software and show it to the users to get their feedback at the early stage of the software product development. The users suggest changes, on which several consecutive updates and maintenance keep on changing too. This process changes to the original software, till the desired software is accomplished.

Even after the user has the desired software in hand, the advancing technology and the changing requirements force the software product to change accordingly. Re-creating software from scratch and to go one-on-one with the requirement is not feasible. The only feasible and economical solution is to update the existing software so that it matches the latest requirements.

CHARACTERISTICS OF A GOOD SOFTWARE

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.

(1) Operational – tells us how well the software works in operations. It can be measured on:

a) Budget

b) Usability

c) Efficiency

d) Correctness

e) Functionality

f) Dependability

g) Security

h) Safety

(2) Transitional – This aspect is important when the software is moved from one platform to another:

a) Portability

b) Interoperability

c) Reusability

d) Adaptability

(3) Maintenance – This aspect briefs about how well the software has the capabilities to maintain itself in the ever-changing environment:

a) Modularity

b) Maintainability

c) Flexibility

d) Scalability

WHAT IS A SOFTWARE PROCESS?

A software process is the set of activities and associated results that produce a software product. There are four fundamental process activities that are common to all software processes. These are:

(1) Software specification where customers and engineers define the software to be produced and the constraints on its operation.

(2) Software development where the software is designed and programmed.

(3) Software validation where the software is checked to ensure that it is what the customer requires.

(4) Software evolution where the software is modified to adapt it to changing customer and market requirements.

BASIC TASK IN SOFTWARE ENGINEERING PROJECT

The following sections describe the steps you need to take to keep a software engineering project on track. These are more or less the same for any large project although there are some important differences.

(1) REQUIREMENTS GATHERING

The plan tells project members what they should be doing, when and how long they should be doing it, and most important what the project's goals are. They give the project direction. One of the first steps in a software project is figuring out the requirements. You need to find out what the customers want and what the customers need. After you determine the customers' wants and needs, you can turn them into requirements documents. Those documents tell the customers what they will be getting, and they tell the project members what they will be building.

(2) HIGH-LEVEL DESIGN

The high-level design includes such things as:

- decisions about what platform to use *(such as desktop, laptop, tablet, or phone)*
- what data design to use *(such as direct access, 2-tier, or 3-tier)*
- and interfaces with other systems *(such as external purchasing systems).*

(3) LOW-LEVEL DESIGN

The low-level design includes information about how that piece of the project should work. The design doesn't need to give every last nitpicky detail necessary to implement

the project's major pieces, but they should give enough guidance to the developers who will implement those pieces.

(4) DEVELOPMENT

The programmers continue refining the low-level designs until they know how to implement those designs in code. As the programmers write the code, they test it to find and remove as many bugs as they reasonably can.

(5) TESTING

Even if a particular piece of code is thoroughly tested and contains no (or few) bugs, there's no guarantee that it will work properly with the other parts of the system. One way to address both of these problems is to perform different kinds of tests. First developers test their own code. Then testers who didn't write the code test it. After a piece of code seems to work properly, it is integrated into the rest of the project, and the whole thing is tested to see if the new code broke anything.

(6) DEPLOYMENT

Deployment can be difficult, time-consuming, and expensive. Deployment might involve any or all of the following:

- **New computers for the back-end database**
- **A new network**
- **New computers for the users**
- **User training**
- **On-site support**
- **Parallel operations**
- **Massive bug fixing**

(7) MAINTENANCE

As soon as the users start pounding away on your software, they'll find bugs. If your application is successful, users will use it a lot, and they'll be even more likely to find bugs.

(8) WRAP-UP

There's one more important thing you should do: You need to perform a post-mortem. You need to evaluate the project and decide what went right and what went wrong. You need to figure out how to make the things that went well occur more often in the future.

Conversely, you need to determine how to prevent the things that went badly in the future.

PROFESSIONAL AND ETHICAL RESPONSIBILITY

Software engineers must accept that their job involves wider responsibilities than simply the application of technical skills. They must also behave in an ethical and morally responsible way if they are to be respected as professionals.

(1) Confidentiality. ***You should normally respect the confidentiality of your employers or clients irrespective of whether a formal confidentiality agreement has been signed.***

(2) Competence. ***You should not misrepresent your level of competence. You should not knowingly accept work that is outside your competence.***

(3) Intellectual property rights. ***You should be aware of local laws governing the use of intellectual property such as patents and copyright.***

(4) Computer misuse. ***You should not use your technical skills to misuse other people's computers.***

SOFTWARE PARADIGMS

Software paradigms refer to the methods and steps, which are taken while designing the software. These can be combined into various categories, though each of them is contained in one another:

- **SOFTWARE DEVELOPMENT PARADIGM**

This paradigm is known as software engineering paradigms; where all the engineering concepts pertaining to the development of software are applied. It includes various researches and requirement gathering which helps the software product to build. It consists of –

(1) Requirement gathering

(2) Software design

(3) Programming

- **SOFTWARE DESIGN PARADIGM**

This paradigm is a part of Software Development and includes –

(1) Design

(2) Maintenance

(3) Programming

- **PROGRAMMING PARADIGM**

This paradigm is related closely to programming aspect of software development. This includes –

(1) Coding

(2) Testing

(3) Integration