Software Engineering

Third Class

Lecture 1

Introduction to Software Engineering

1.1 Introduction

Virtually all countries now depend on complex computer-based systems. Most electrical products include a computer and controlling software. Industrial manufacturing and distribution is completely computerized, as is the financial system. Therefore, producing and maintaining software cost-effectively is essential for the functioning of national and international economies.

Software engineering is an engineering discipline whose focus is the cost effective development of high-quality software systems. Software is abstract and intangible. It is not constrained by materials, or governed by physical laws or by manufacturing processes. In some ways, this simplifies software engineering as there are no physical limitations on the potential of software. However, this lack of natural constraints means that software can easily become extremely complex and hence very difficult to understand.

The notion of *software engineering* was first proposed in 1968 at a conference held to discuss what was then called the 'software crisis'. This software crisis resulted directly from the introduction of new computer hardware based on integrated circuits. Their power made hitherto unrealizable computer applications a feasible proposition. The resulting software was orders of magnitude larger and more complex than previous software systems.

Early experience in building these systems showed that informal software development was not good enough. Major projects were sometimes years late. The software cost much more than predicted, was unreliable, was difficult to maintain and performed poorly. Software development was in crisis. Hardware costs were tumbling whilst software costs were rising rapidly. New techniques and methods were needed to control the complexity inherent in large software systems.

These techniques have become part of software engineering and are now widely used. However, as our ability to produce software has increased, so too has the complexity of the software systems that we need. New technologies resulting from the convergence of computers and communication systems and complex graphical user

interfaces place new demands on software engineers. As many companies still do not apply software engineering techniques effectively, too many projects still produce software that is unreliable, delivered late and over budget.
The tremendous progress since 1968 and that the development of software engineering has markedly improved software. Effective methods of software specification, design and implementation have been developed. New notations and tools reduce the effort required to produce large and com*plex* systems.
There is no single 'ideal' approach to software engineering. The wide diversity of different types of systems and organizations that use these systems means that we need a diversity of approaches to software development. However, fundamental notions of process and system organization underlie all of these techniques, and these are the essence of software engineering.

1.2 Software and Software Engineering

Software is (1) instructions (computer programs) that when executed provide desired function and performance, (2) data structures that enable the programs to adequately manipulate information, and (3) documents that describe the operation and use of the programs.
Software engineers are concerned with developing software products, i.e., software which can be sold to a customer. 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. *CustOmised (or bespoke) 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.

However, the line between these types of products is becoming increasingly unclear. More and more software companies are starting with a generic system and customizing it to the needs of a particular customer.

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. Engineers also recognize that they must work to organizational and financial constraints, so they look for solutions within these constraints.
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.

   In general, software engineers adopt a systematic and organized approach to their work, as this is often the most effective way to produce high-quality software. However, engineering is all about selecting the most appropriate method for a set of circumstances and a more creative, less formal approach to development may be effective in some circumstances. Less formal development is particularly appropriate for the development of web-based systems, which requires a blend of software and graphical design skills.

1.3 Software Characteristics

To gain an understanding of software (and ultimately an understanding of software engineering), it is important to examine the characteristics of software that make it different from other things that human beings build. When hardware is built, the human creative process (analysis, design, construction, testing) is ultimately translated into a physical form. If we build a new computer, our initial sketches, formal design drawings, and prototype evolve into a physical product (chips, circuit boards, power supplies, etc.).

Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware:

1-Software is developed or engineered; it is not manufactured in the classical sense. Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different. In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software. Both activities are dependent on people, but the relationship between people applied and work accomplished is entirely different. Both activities require the construction of a "product" but the approaches are different.

2- Software doesn't "wear out."

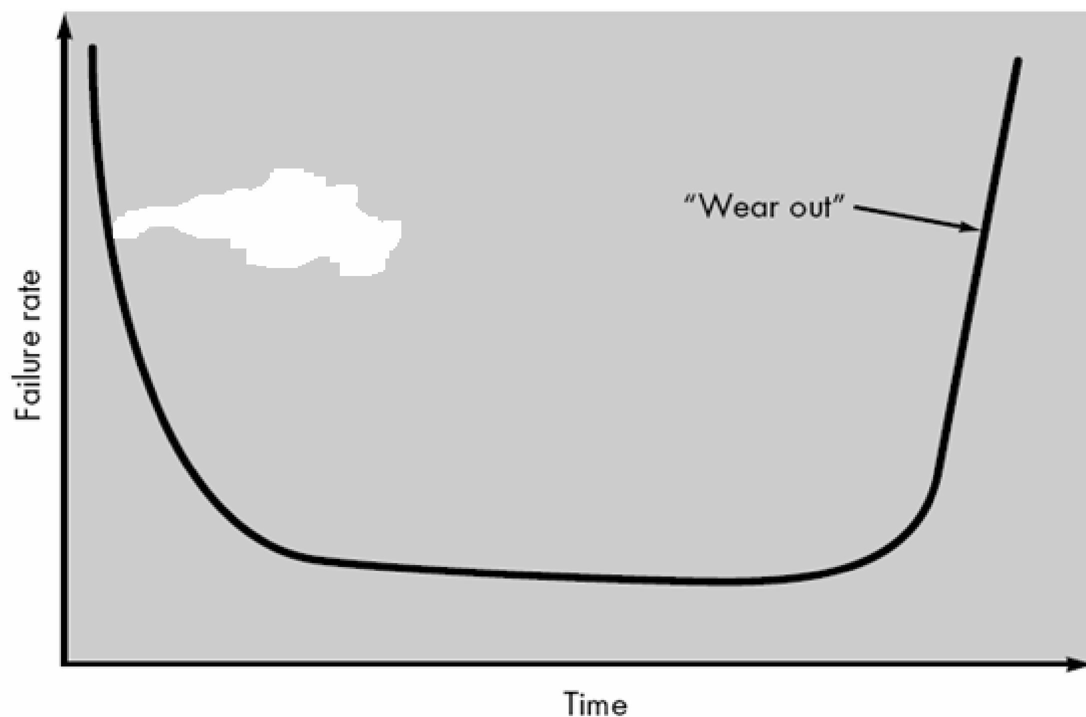

Figure 1: failure rate for hardware

Figure 1 depicts failure rate as a function of time for hardware. The relationship, often called the "bathtub curve," indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing defects); defects are corrected and the failure rate drops to a steady-state level (Ideally, quite low) for some period of time. As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies. Stated simply, the hardware begins to wear out.

Software is not susceptible to the environmental maladies that cause hardware to wear out. In theory, therefore, the failure rate curve for software should take the form of the “idealized curve” shown in Figure 2. Undiscovered defects will cause high failure rates early in the life of a program. However, these are corrected (ideally, without introducing other errors) and the curve flattens as shown.

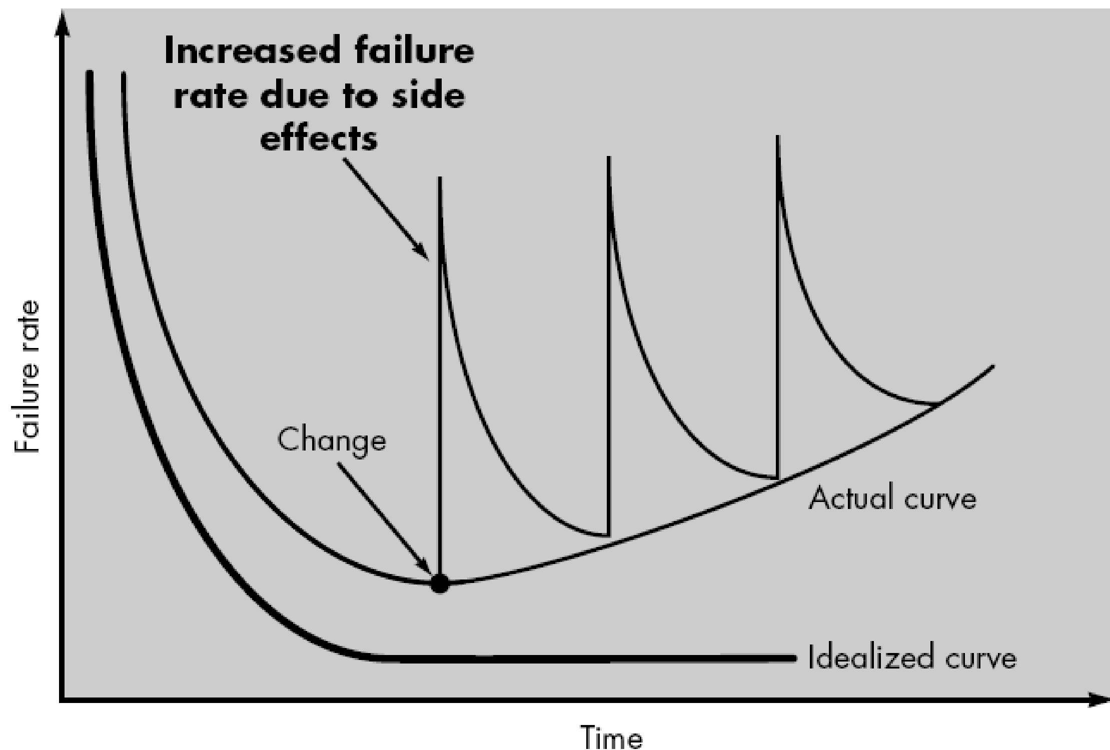

Figure2: Idealized and actual failure curves for software

The idealized curve is a gross oversimplification of actual failure models for software. However, the implication is clear—software doesn't wear out. But it does deteriorate. This seeming contradiction can best be explained by considering the “actual curve” shown in Figure.2. During its life, software will undergo change (maintenance). As changes are made, it is likely that some new defects will be introduced, causing the failure rate curve to spike as shown in Figure 2. Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again. Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

Another aspect of wear illustrates the difference between hardware and software. When a hardware component wears out, it is replaced by a spare part. There are no software spare parts. Every software failure indicates an error in design or in the process through which design was translated into machine executable code.

Therefore, software maintenance involves considerably more complexity than hardware maintenance.

3- Although the industry is moving toward component-based assembly, most software continues to be custom built.

Considering the manner in which the control hardware for a computer-based product is designed and built. The design engineer draws a simple schematic of the digital circuitry, does some fundamental analysis to assure that proper function will be achieved, and then goes to the shelf where catalogs of digital components exist. Each integrated circuit (called an *IC* or a *chip*) has a part number, a defined and validated function, a well-defined interface, and a standard set of integration guidelines. After each component is selected, it can be ordered off the shelf.

As an engineering discipline evolves, a collection of standard design components is created. Standard screws and off-the-shelf integrated circuits are only two of thousands of standard components that are used by mechanical and electrical engineers as they design new systems. The reusable components have been created so that the engineer can concentrate on the truly innovative elements of a design, that is, the parts of the design that represent something new. In the hardware world, component reuse is a natural part of the engineering process. In the software world, it is something that has only begun to be achieved on a broad scale.

A software component should be designed and implemented so that it can be reused in many different programs. In the 1960s, scientific subroutine libraries that were reusable in a broad array of engineering and scientific applications had been built. These subroutine libraries reused well-defined algorithms in an effective manner but had a limited domain of application. Today, the view of reuse has been extended to encompass not only algorithms but also data structure. Modern reusable components encapsulate both data and the processing applied to the data, enabling the software engineer to create new applications from reusable parts. For example, today's graphical user interfaces are built using reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms. The data structure and processing detail required to build the interface are contained with a library of reusable components for interface construction.

1.4 The attributes of good software

As well as the services that it provides, software products have a number of other associated attributes that reflect the quality of that software. These attributes are not directly concerned with what the software does. Rather, they reflect its behavior while it is executing and the structure and organization of the source program and associated documentation. Examples of these attributes (sometimes called nonfunctional attributes) are the software s response time to a user query and the understandability of the program code.

The specific set of attributes that you might expect from a software system obviously depends on its application. Therefore, a banking system must be secure; an interactive game must be responsive, and so on. These can be generalized into the set of attributes shown in Table 1, which are the essential characteristics of a well-designed software system.

Product characteristic	Description
Maintainability	Software should be written in such a way that it may evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable consequence of a changing business environment.
Dependability	Software dependability has a range of characteristics, including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, utilization, etc.
Usability	Software must be usable, without undue effort, by the type of user for whom it is designed. This means that it should have an appropriate user interface and adequate documentation.

Table 1:Essential attributes of good software