Module 1

Introduction to Software Engineering

Lesson 1

Basic Issues in Software Engineering

Specific Instructional Objectives

At the end of this lesson the student will be able to:

- Identify the scope and necessity of software engineering.
- Identify the causes of and solutions for software crisis.
- Differentiate a piece of program from a software product.

Scope and necessity of software engineering

Software engineering is an engineering approach for software development. We can alternatively view it as a systematic collection of past experience. The experience is arranged in the form of methodologies and guidelines. A small program can be written without using software engineering principles. But if one wants to develop a large software product, then software engineering principles are indispensable to achieve a good quality software cost effectively. These definitions can be elaborated with the help of a building construction analogy.

Suppose you have a friend who asked you to build a small wall as shown in fig. 1.1. You would be able to do that using your common sense. You will get building materials like bricks; cement etc. and you will then build the wall.

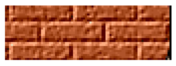

Fig. 1.1: A Small Wall

But what would happen if the same friend asked you to build a large multistoried building as shown in fig. 1.2?

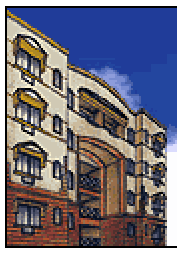

Fig. 1.2: A Multistoried Building

You don't have a very good idea about building such a huge complex. It would be very difficult to extend your idea about a small wall construction into constructing a large building. Even if you tried to build a large building, it would collapse because you would not have the requisite knowledge about the strength of materials, testing, planning, architectural design, etc. Building a small wall and building a large building are entirely different ball games. You can use your intuition and still be successful in building a small wall, but building a large building requires knowledge of civil, architectural and other engineering principles.

Without using software engineering principles it would be difficult to develop large programs. In industry it is usually needed to develop large programs to accommodate multiple functions. A problem with developing such large commercial programs is that the complexity and difficulty levels of the programs increase exponentially with their sizes as shown in fig. 1.3. For example, a program of size 1,000 lines of code has some complexity. But a program with 10,000 LOC is not just 10 times more difficult to develop, but may as well turn out to be 100 times more difficult unless software engineering principles are used. In such situations software engineering techniques come to rescue. Software engineering helps to reduce the programming complexity. Software engineering principles use two important techniques to reduce problem complexity: abstraction and decomposition.

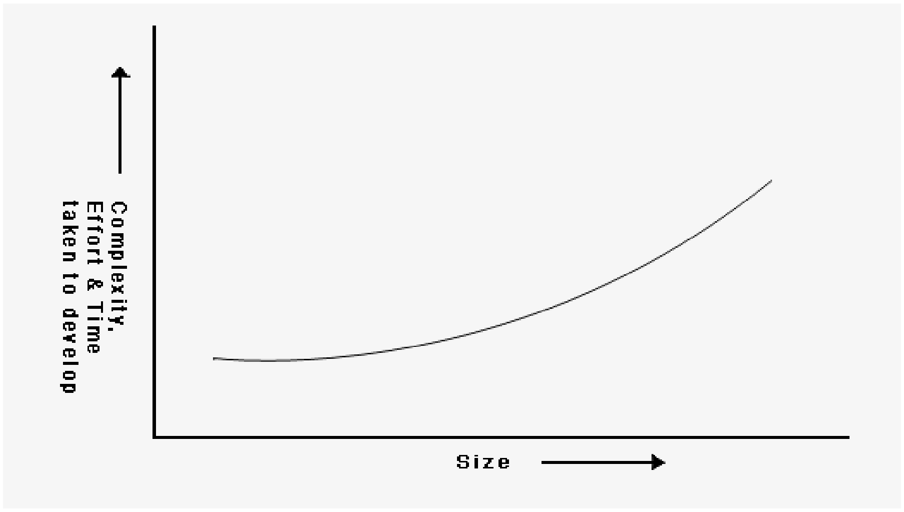

Fig. 1.3: Increase in development time and effort with problem size

The principle of abstraction (in fig.1.4) implies that a problem can be simplified by omitting irrelevant details. In other words, the main purpose of abstraction is to consider only those aspects of the problem that are relevant for certain purpose and suppress other aspects that are not relevant for the given purpose. Once the simpler problem is solved, then the omitted details can be taken into consideration to solve the next lower level abstraction, and so on. Abstraction is a powerful way of reducing the complexity of the problem.

The other approach to tackle problem complexity is decomposition. In this technique, a complex problem is divided into several smaller problems and then the smaller problems are solved one by one. However, in this technique any random decomposition of a problem into smaller parts will not help. The problem has to be decomposed such that each component of the decomposed problem can be solved independently and then the solution of the different components can be combined to get the full solution. A good decomposition of a problem as shown in fig.1.5 should minimize interactions among various components. If the different subcomponents are interrelated, then the different components cannot be solved separately and the desired reduction in complexity will not be realized.

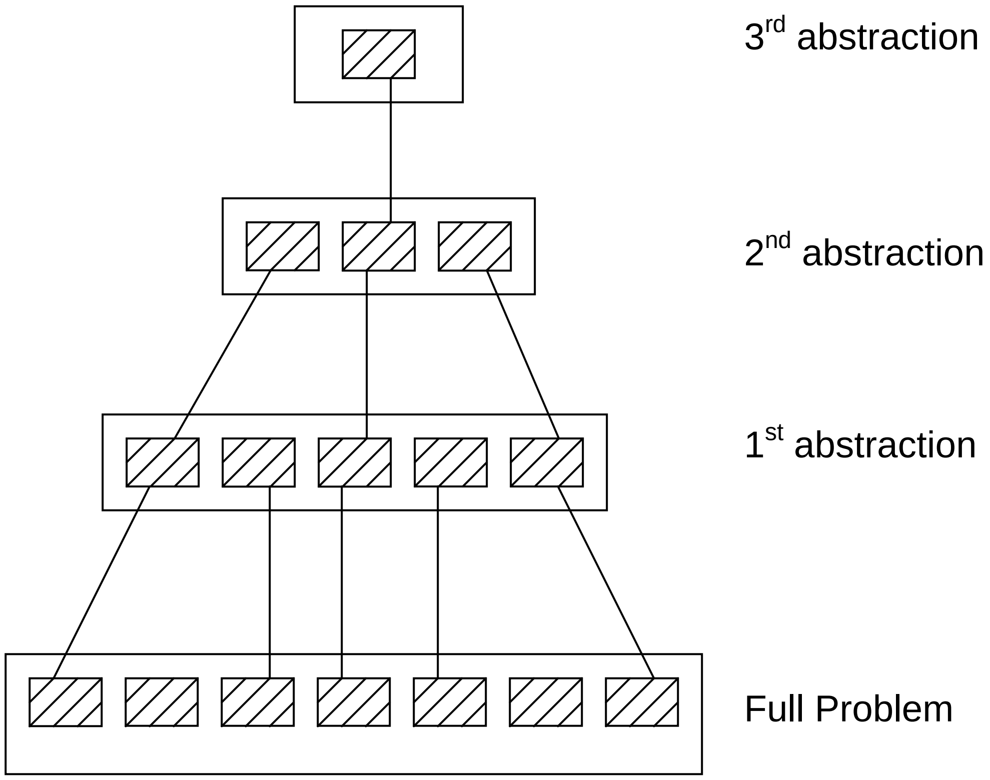

Fig. 1.4: A hierarchy of abstraction

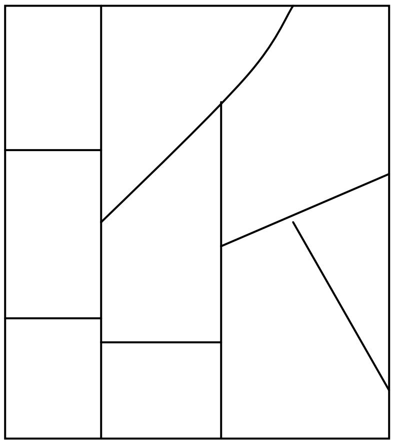

Fig. 1.5: Decomposition of a large problem into a set of smaller problems.

Causes of and solutions for software crisis.

Software engineering appears to be among the few options available to tackle the present software crisis.

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 (as shown in fig. 1.6)

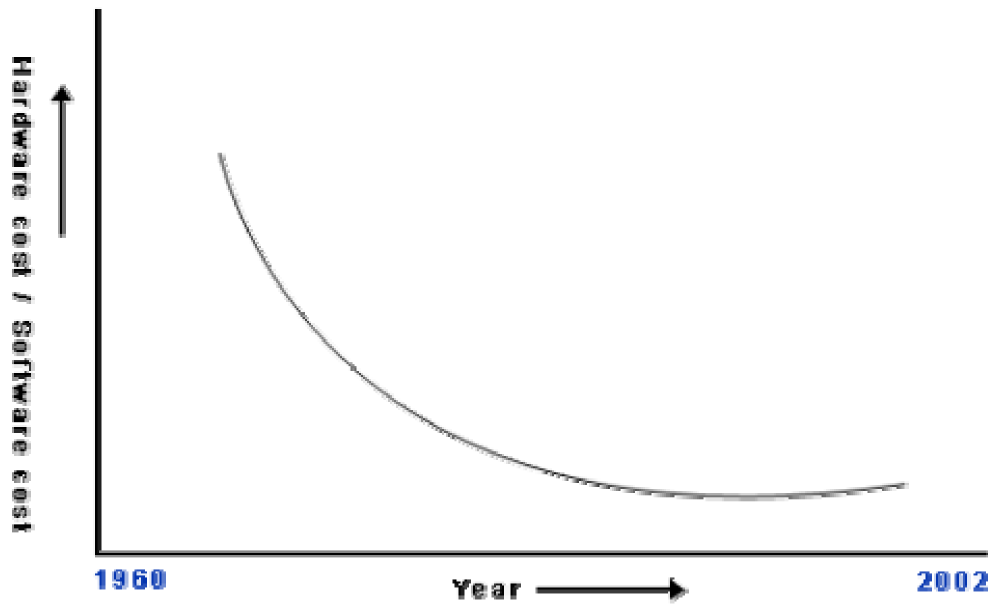

Fig. 1.6: Change in the relative cost of hardware and software over time

Organizations are spending larger and larger portions of their budget on software. 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 non-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. Remember that the cost we are talking of here is not on account of increased features, but due to ineffective development of the product characterized by inefficient resource usage, and time and cost over-runs.

There are many factors that have contributed to the making of the present software crisis. Factors are larger problem sizes, lack of adequate training in software engineering, increasing skill shortage, and low productivity improvements.

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.

Program vs. software product

Programs are developed by individuals for their personal use. They are therefore, small in size and have limited functionality but software products are extremely large. In case of a program, the programmer himself is the sole user but on the other hand, in case of a software product, most users are not involved with the development. In case of a program, a single developer is involved but in case of a software product, a large number of developers are involved. For a program, the user interface may not be very important, because the programmer is the sole user. On the other hand, for a software product, user interface must be carefully designed and implemented because developers of that product and users of that product are totally different. In case of a program, very little documentation is expected, but a software product must be well documented. A program can be developed according to the programmer's individual style of development, but a software product must be developed using the accepted software engineering principles.