Software Engineering
Part 1

In this part, we look at

Developing software system is generally a quite complex and time consuming process. Moreover, the nature and complexity of software requirements have drastically changed in the last few decades and users all over the world have become much more demanding in terms of cost, schedule and quality. These three parameters, all being desirable, have an apparent contradiction at times which can only be resolved by optimum design of software using well established software engineering methodologies.

Software Engineering Methodologies constitute the framework that guides us in optimally developing the software systems. These frameworks define the different phases of software development such as planning, requirements analysis, design, testing and maintenance. The choice of which methodology to use in a specific development process is closely related to the size, complexity, reliability and maintainability of the software, and to the environment it is supposed to function in.

Software Engineering concerned with designing, testing, coding of the programs, implementation, maintenance, reliability, and performance with respect to "efficiency, accuracy, storage space required, and etc", management of software, cost and time factor of software development

As software engineers, we use our knowledge of computers and computing to help solve problems. Often the problem with which we are dealing is related to a computer or an existing computer system, but sometimes the difficulties underlying the problem have nothing to do with computers.

Therefore, it is essential that we first understand the nature of the problem. In particular, we must be very careful not to impose computing machinery or techniques on every problem that comes our way.

- We must solve the problem first,
- Then, if need be, we can use technology as a tool to implement our solution.

1.1 What is Software Engineering?

Solving Problems

Most problems are large and sometimes tricky to handle, especially if they represent something new that has never been solved before. So we must begin investigating it by **analyzing** it, that is, by breaking the problem into pieces that we can understand and try to deal with. Thus we can thus describe a large problem as a collection of small problems and their interrelationships. Figure 1.1 illustrates how analysis works.

Once we have analyzed the problem, we must construct our solution from components that address the problem's various aspects. Figure 1.2 illustrates this reverse process: **Synthesis** is the putting together of a large structure from small building blocks.

Thus, any problem-solving technique must have two parts:

- analyzing the problem to determine its nature, and then
- synthesizing a solution based on our analysis.

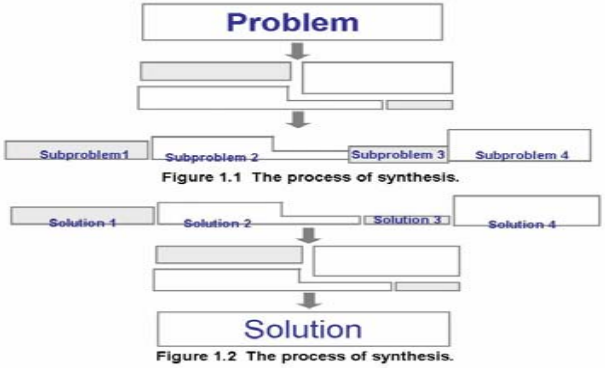

Figure 1.1 The process of synthesis.
Figure 1.2 The process of synthesis.

To help us solve a problem, we employ a variety of methods, tools, procedures, and paradigms. A **method** or **technique** is a formal procedure for producing some result. A **tool** is an instrument or automated system for accomplishing something in a better way. This "better way" can mean that the tool makes us more accurate, more efficient, or more productive or that it enhances the quality of the resulting product. A **procedure** is a combination of tools and techniques that, in concert, produce a particular product. Finally, a **paradigm** it represents a particular approach or philosophy for building software. Just as we can distinguish between two paradigms like object-oriented development from procedural ones. One is not better than other; each has its advantage and disadvantages, and there may be situations when one is more appropriate than another.

Software engineers use tools, techniques, procedures, and paradigms to enhance the quality of their software products. Their aim is to use efficient and productive approaches to generate effective solutions to problems.

Where Does the software Engineer Fit In?

We can view computing by concentrating on the computers and programming languages themselves, or we can view them as tools to be used in designing and implementing a solution to a problem. Software engineering takes the latter view, as shown in Figure 1.3.

Instead of investigating hardware design or proving theorems about how algorithms work, a software engineer focuses on the computer as a problem-solving tool. We will see later in this chapter that a software engineer works with the functions of a computer as part of a general solution, rather than with the structure or theory of the computer itself.

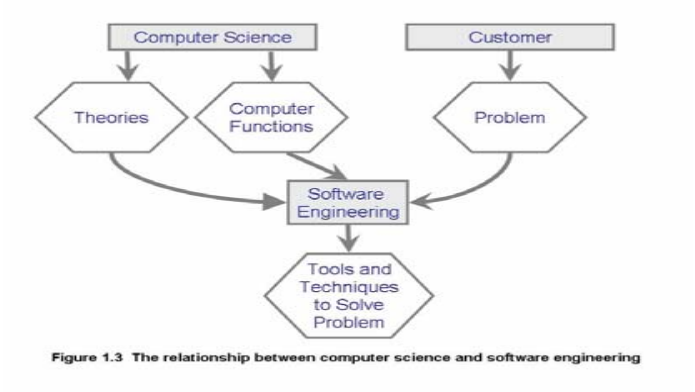

Figure 1.3 The relationship between computer science and software engineering

How Successful Have Been?

Writing software is an art and as well as a science, and it is important for you as a student of computer science to understand why. Computer scientists and software engineering researchers study computer mechanisms and theories about how to make them more productive or efficient. However, they also design computer systems and write programs to perform tasks on those systems, a practice that involves a great deal of art, ingenuity, and skills. There may be many ways to perform a particular task on a particular system, but some are better than others. One way may be more efficient, more precise, easier to modify, easier to use, or easier to understand. Any hacker can write code to make something work, but it takes the skills and understanding of a professional software engineer to produce code that is robust, easy to understand and maintain, and does its job in the most efficient and effective way possible. Consequently, software engineering is about design and developing high-quality software.

Before, we examine what is needed to produce quality software systems. Let us look back to see how successful we have been. Are users happy with their existing software systems? Yes or No. Software has enabled us to perform tasks more quickly and effectively than ever before. Consider life before word processing, spreadsheets, electronic mail, or sophisticated telephony, for example. And software has supported life-sustaining or life-saving advances in medicine, agriculture, transportation, and most other industries.

However, software is not without its problems. Often systems function, but exactly as expected. Often, we talk about "bugs" in software, meaning many things that depend on the context. A **bug** can be a mistake in interpreting a requirement, a syntax error in apiece of code, or the (as-yet-unknown) cause of a system crash.

A **fault** occurs when a human makes a mistake, called an **error**, in performing some software activity. A single error can generate many faults, and a fault can reside in any development or maintenance product. A **failure** is a departure from system's required behaviors. It can be discovered before or later or after system delivery, during testing, or during operation and maintenance.

Thus, a fault is an inside view of the system, as seen by the eyes of the developers, whereas a failure is an outside view: a problem that the user sees. Not every fault corresponds to a failure; for example if the faulty code is never executed or a particular state is never entered, then the fault will never cause the code to fail. Figure 1.4 shows the genesis of a failure

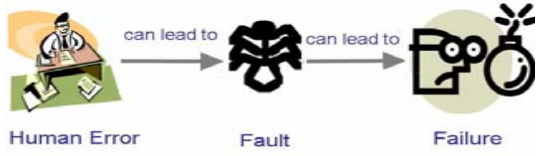

Figure 1.4 How human error causes a failure.

Program vs. Software, and Software Engineering

A **program** is an algorithm expressed using a precise notation which can be executed by a computer. In other words, a program is a sequence of instructions that tell the computer what to do. **Software** is not merely a collection of computer programs. Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of **hardware.**

Software is:

- **Instructions** (computer programs)—that when executed provide desired function and performance,
- **Data Structures**—that enable programs to be adequately manipulate information, and
- **Documents**—that describe the operation and use of the programs.

IEEE defines **software** as the collection of computer programs, procedures, rules, and associated documentation and data.

Engineering is the analysis, design, construction, verification, and management of technical (or social) entities. [By a single entity we mean computer software]

IEEE defines **Software Engineering** as the systematic approach or a discipline to the development, operation, maintenance, and retirement of software.

The work associated with software engineering can be categorized into three generic phases (i.e., definition, development, and support), regardless of application are, project size, or complexity each phase.

The **definition phase** focuses on **what**: That is, during definition phase, the software engineer attempts to identify:

- What information is to be processed?
- What function and performance are desired?
- What system behavior can be expected?
- What interfaces are to be established?
- What design constraints exist?
- What validation criteria are required to define a successful system?

The **development phase** focuses on **how**: That is, during development phase, the software engineer attempts to define:

- How data are to be structured?
- How function is to be implemented?
- How interfaces are to be characterized?
- How the design will be translated into a programming language (or nonprocedural language)?
- How testing will be performed?

The **support phase** focuses on change associated with error correction, adaptations required as the software's environment evolves, and changes due to enhancements brought about by changing customer requirements. The support phase reapplies the steps of the definition and development phases but does so in the context of existing software.

Four types of **change** encountered during the support phase:

- **Correction**—even with a best quality assurance activities. It is likely that the customer will uncover defects in the software. Corrective Maintenance changes the software to correct defects.
- **Adaptation**—over time, the original environment (e.g., CPU, OS, business rules, external product characteristics) for which the software was developed is likely change. Adaptive maintenance results in modification to the software to accommodate changes to its external environment.
- **Enhancement**—as software is used, the user will recognize additional functions that will provide benefit. Perfective Maintenance extends the software beyond its original functional requirements.
- **Prevention**—computer software deteriorates due to change, and because of this, Preventive maintenance, often called. Software engineering must be conducted to enable the software to serve the needs of its end user. In essences, preventive maintenance makes changes to computer programs so that they can be more easily corrected, adapted, and enhanced.

The phases and related steps described in our generic view of software engineering are complemented by a number of **Umbrella Activities**. Typically activities in this category include:

- Document Preparation and Production
- Reusability Management
- Measurement
- Risk Management
- Software Project Tracking and Control
- Formal Technical Reviews
- Software Quality Assurance
- Software Configuration Management

These umbrella activities are applied throughout the software process.

1.2 Software Engineering Problem

The problem of Scale

A fundamental problem of software engineering is the problem of scale; development of a very large system requires a very different set of methods compared to developing a small system. In other words, the methods that are used for developing small systems generally **do not scale up** to large systems. An example will illustrate this point. Consider the problem of counting people in a room versus taking a census of a country. Both are essentially counting problems. But the methods used for counting people in a room (probably juts go row-wise or column-wise) will just not work when taking a census. Different set of methods will have to be used for counting a census, and the census problem will require considerably more management, organization, and validation, in addition to counting.

Thought there is no universally acceptable definition of what is a "small" project and what is a "large" project, one can use the definition used in the cost construction model (COCOMO) cost model to get an idea of scale. According to this model, a project is small if its size in thousands of delivered lines of code (KDLOC) is 2 KDLOC, intermediate if the size is 8 KLOOC, medium if the size is 32 KDLOC, and large if the size is 128 KDLOC (or larger).

Cost, Schedule, and Quality

Like all engineering discipline, software engineering is driven almost by three major factors: cost, schedule, and quality.

The **cost** of developing a system is the cost of the resources used for the system, which in the case of software, are the manpower, hardware, software, and other support resources. Generally, the manpower component is predominant, as software development is largely labor-intensive and the cost of computing systems is now quite low. Hence, the cost is considered to be the total number of person-months spent in the project.

Schedule is an important factor in many projects. Business trends are dictating that the time to market of a product should be reduced; that is, the cycle time from concept to delivery should be small. Any business with such a requirements will also require that the cycle time for building a software needed by the business be small.

One of the major factors driving any production discipline is quality. IEEE defines **Quality** as the degree to which a system, component, or process meets specified requirements, and customer or user needs or expectations.

We can view quality of a software as having three dimensions:

- **product operation**—deals with quality factors such as correctness, reliability, efficiency and usability.
- **product transition**—deals with quality factors like portability, reusability, and interoperability.
- **product revision**—deals with quality factors such as maintainability, flexibility, and testability.

Attribute	Definition
Correctness	the extent to which a program satisfies its specification.
Reliability	the property that defines how well the software meets its requirements.
Efficiency	a factor in all issues relating to the execution of software; it includes such consideration as response time, memory requirements and throughput.
Usability	the effort required to learn and operate the software properly, is an important property that emphasizes the human aspect of the system.
Portability	the effort required to transfer the software from one hardware configuration to another.
Reusability	the extent to which parts of the software can be reused in other related applications.
Interoperability	the effort required to couple the system with other systems.
Maintainability	the effort required to locate and fix errors in operating programs.
Flexibility	the effort required to modify an operational program (perhaps to enhance its functionality).
Testability	the effort required to test to ensure that the systems or a module performs its intended function.

The Problem of Consistency

Though high quality, low cost (or high productivity), and small cycle time are the primary objectives of any project, for an organization there is another goal: consistency. An organization involved in software development does not just want low cost and high quality for a project, but it wants these consistently. In other words, a software development organization would like to produce consistent quality with consistent productivity. Consistency of performance is an important factor for any organization, it allows an organization to predict the outcome of a project with reasonable accuracy, an to improve its processes to produce higher-quality products and to improve its productivity.

1.3 The Software Engineering Approach

Once we understand the system's nature, we ready to begin its construction.

Building a System

Software projects progress in a way similar to the house-building process. Software development involves users, customers, and developers. If we asked to develop a software system for a customer, the first step is meeting with the customer to determine the requirements. These requirements describe the system, before knowing the boundary, the entities, and the activities, it is impossible to describe the software and how it will interact with its environment.

Once requirements are defined, we create a **system design** to meet the specified requirements. The system design shows the customer what the system will look like from the customer's perspective.

The system design phase of a software project describes only appearance and functionality. The design is then reviewed by the customer. When approved, the overall system design is used to generate the designs of the individuals programs involved. The basis for our discussion is a well-defined description of the software project as a system; the system design includes a complete description of the functions and interactions involved.

When the **programs** have been written, they are tested as individual pieces of code before they can be linked together. This first phase of testing is called **module** or **unit testing**. Once we are convinced that the pieces work as desired, we put them together and make sure that they work properly when joined with others. This second testing phase is often referred to as **integration testing**, as we build our system by adding one piece to the next until the entire system is operational. The final **testing phase**, is called **system testing**, involves a test of a whole system to make sure that the functions and interactions specified initially have been implemented properly. In this phase, the system is compared with the specified requirements; the developer, customer and users check that the system serves its intended purpose. At last, the final product is **delivered**.

Thus, development of software includes the following activities:

- Requirements Analysis and Definition,
- System Design,
- Program Design,
- Writing the Programs (Program Implementation),
- Unit Testing,
- Integration Testing,
- System Testing,
- System Delivery, and
- Maintenance.

In an ideal situation, the activities are performed one at a time; when you reach the end of the list, you have a competed software project. However, in reality, many of the steps are repeated. We define a **software development process** as any description of software development that contains some of the nine activities listed before, organized so that together they produce tested code. In part 2, we will explore several of the different development processes that are used in building software. Subsequent Parts will examine each of the sub-processes and their activities, from requirements analysis through maintenance.

1.4 Who Does Software Engineering?

A key component of software development is communication between customer and developer; if that fails, so too will the system. We must understand what the customer wants and needs before we can build a system to help solve the customer's problem. To do this let us turn out attention to the people involved in software development. The number of people working on software development depends on the project's size and degree of difficulty.

Usually, the participants in a project fall into one of three categories:

- **Customer**—is the company, organization, or person who is paying for the software system to be developed.
- **Developer**— is the company, organization, or person who is building the software system for the customer. This category includes any managers needed to coordinate and guide the programmers and testers.
- **User**—is the person or people who will actually use the system: the ones who sit at the terminal or submit the data or read the output.

Figure 1.6 shows the basic relationships among the three types of participants.

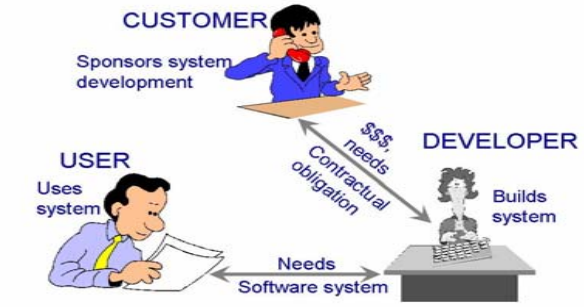

Figure 1.6 Participants in software development.

1.5 Members of the Development Team

Earlier in this Part, we saw that customers, users and developers play major roles in the definition and creation of the new product. The developers are software engineers. But each engineer may specialize in a particular aspect of development. The first step in any development process is finding out what the customer wants and documenting the requirements. **Analysis** is the process of breaking things into their components parts so that we can understand them better. Thus the development team includes one or more **requirement analysts** to work with the customer, braking down what the customer wants into discrete requirements.

Once the requirements are known and documented, analysts work with **designers** to generate a system-level description of what the system is to do. In turn, the designers work with **programmers** to describe the system in such a way that programmers can write lines of code that implement what the requirements specify. After the code is generated, it must be tested. Often the first testing is done by the programmers themselves; sometimes, additional **testers** are also used to help catch faults that the programmers overlook. When units of code are integrated into functioning groups, a team of testers works with the implementation team to verify that as the system is built up by combining pieces, it works properly and according to specification. When the development team is comfortable with the functionality and quality of the system, attention turns to the **customer**. The test team and the customer work together to verify that the complete system is what the customer wants;

they do by comparing how the system works with the initial set of requirements. Then, **trainers** show users to use the system.

For many software systems, acceptance by the customer does not mean the end of the developer's job. If faults are discovered after the system has been accepted, a **maintenance team** fixes them. Moreover, the customer's requirements may change as time passes, and corresponding changes to the system must be made. Thus, maintenance can involve **analysts** who determine what requirements are added/changed, **designers** to determine where in the system design the change should made, **programmers** to implement the changes, **testers** to make sure that the changed system still runs properly, and **trainers** to explain to users how the change effects the use of the system.
Figure 1.5 illustrates how the roles of the development team correspond to the steps of development.

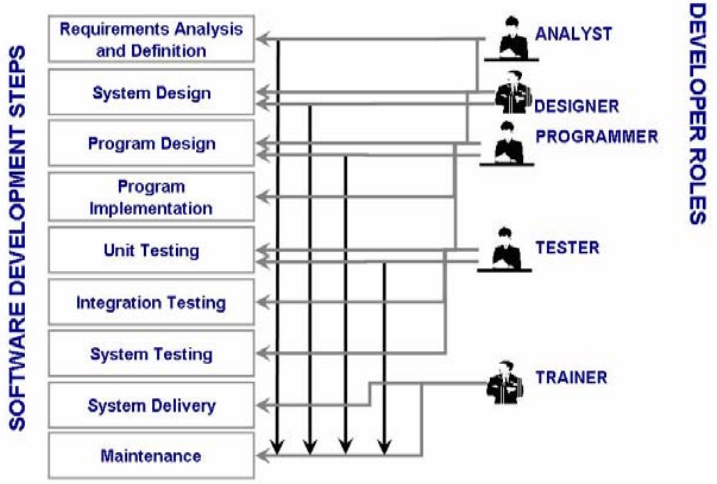

Figure1.5 The roles of the development team

Librarians prepare and store documents tat are used during the life of the system, including requirements specifications, design descriptions, program documentation, training manuals, test data, schedules, and more. Working with the librarians are the members of a **configuration management teams**. Configuration management involves maintaining a correspondence among the requirements, the design, the implementation, and tests. This cross-reference tells developers what program to alter if a change in requirements is needed, or what parts of a program will be affected if an alteration of some kind is proposed. Configuration management staff also coordinates the different versions of a system that may be built and supported.

1.6 How Has Software Engineering Changed?

We must acknowledge that most system do not stand by themselves. They interface with other systems, either to receive or to provide information. Developing such systems is complex simply because they require a great deal of coordination with the systems with which they communicate. This complexity is especially true of systems that are being developed concurrently.

The Nature of the Change

Today's software-based systems are far different and more complex. Typically, they run on multiple systems, sometimes configured in a client-server architecture with distributed functionality. Software performs not only the primary functions that the user needs, but also network control, security, user-interface presentation and processing, and data or object management. The traditional approach to the development, which assumes a linear progression of development activities, where one begins only when its predecessor is complete (see Part 2), is no longer flexible or suitable for today's system

In his Stevens lecture, Wasserman (1996) summarized these changes by identifying seven key factors that have altered software engineering practice, illustrated in Figure 1.6:

- Critically of time-to-market for commercial product
- Shift in the economics of computing: lower hardware costs and greater development/maintenance costs
- Availability of powerful desktop computing
- Extensive local- and wide-area networking
- Availability and adoption of object-oriented technology
- Graphical user interfaces using windows. Icons, menus, and pointers
- Unpredictability of the waterfall model of software development.

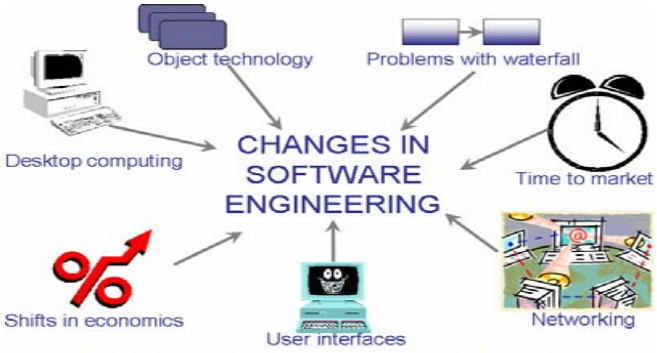

Figure1.6 The key factors that have changed software development

Wasserman's Discipline of Software Engineering

Wasserman (1996) points out that any one of the eight technological changes would have a significant effect on the software development process.

- **Abstraction**. An abstraction is a description of the problem at some level of generalization that allows us to concentrate on the key aspects of the problem without getting mired in the details.
- **Analysis and Design Methods and Notations**. Analysis and design methods offer us more than a communicate medium. They allow us to build models and check them for completeness and consistency.
- **User Interface Prototyping**. Prototyping means building a small version of a system, usually with limited functionality, that can be used to
    - help the user or customer identify the key requirements of a system and

    - demonstrate the feasibility of a design or approach.

  Prototyping is often used to design a good **user interface**: the part of the system with which the user interacts,
- **Software Architecture**. A system's architecture describes the system in of a set of architectural units, and a map of how the units relate to one another. The more independent the units, the more modular the architecture and the more easily we can design and development the process separately. Wasserman (1996) points out that there are at least five ways that we can partition the system into units:
    - **modular decomposition**: based on assigning functions to modules
    - **data-oriented decomposition**: based on external data structures
    - **event-oriented decomposition**: based on events that the system must handle
    - **outside-in design**: based on user inputs to the system
    - **object-oriented design**: based on identifying classes of objects and their interrelationships.
- **Software Process**. Since the late 1980s, many software engineers have paid careful attention to the process of development software, as we as to the product that result. The organization and discipline in the activities have been acknowledged to contribute to the quality of the software and to the speed with which it is developed. Wasserman (1996) suggests that different types of software need different processes. In particular, he suggests that enterprisewide applications need a great deal of control whereas individual and departmental applications can take advantage of rapid application development, as illustrate in Figure 1.7.

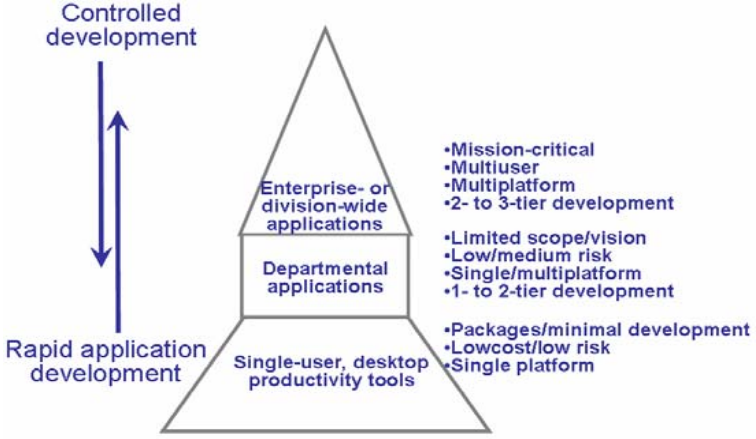

Figure 1.7 Differences in development (Wasserman 1996).

- **Reuse**. In software development and maintenance, we often take advantage of the commonalities across applications by reusing items from previous development. For example, we can reuse the same operating system or database management system from one development to the next, rather than building a new one each time. However, establishing long-term, effective reuse program can be difficult, because there are several barriers:
    - it is sometimes faster to build a small component that to search for one in a repository of reusable components

- it may extra time to make a component general enough to be reusable easily by other developers in the future.
- it is difficult to document the degree assurance and testing that have been done, so that a potential reuser can feel comfortable about the quality of the component.
- it is not clear who is responsible if a reused component fails or needs to be updated.
- it can be costly and time-consuming to understand and reuse a component written by someone else.
- there is often a conflict between generality and specificity.

- **Measurements**. Improvement is a driving force in software engineering research: improving our processes, resources, and methods so that we produce and maintain best products. But sometimes we express improvement goal generally, with no quantitative description of where we are and we would like to go. For this reason, software measurement has become a key aspect of good software engineering practice. By quantifying where we can go and what we can, we describe our actions and their outcomes in a common mathematical language that allows us to evaluate our progress. In addition, a quantitative approach permits us to compare progress across disparate projects.
- **Tools and Integrated Environments**. For many years, vendor touted CASE (computer-aided software engineering) tools, where standardized, integrated development environment would enhance software development. However, we have seen how different developers use different processes, methods, and resources, so a unifying approach is easier said than done. Wassermann (1990) has identified five issues that must be addressed in any tool integration:
  - **platform integration**: the ability of tools to interoperate on a heterogeneous network
  - **presentation integration**: commonality of user interface
  - **process integration**: linkage between the tools and the development process
  - **data integration**: the way tools share data
  - **control integration**: the ability for one tool to notify and initiate action in another

1.7 Project Management

As stated earlier, a phased development process is central to the software engineering approach. However, a development process does not specify how to allocate resources to the different activities in the process. Nor does it specify things like schedule for the activities, how to divide work within a phase, how to ensure that each phase is being done properly, what risks for the project are and how to mitigate them, etc. Without properly handling these issues, it is unlikely that the cost and quality objectives can be met. These types of issues generally fall in the domain of **project management**. This implies that even for properly using a phased development process, project management must be treated as an integral part of the software development.

The management activities in a project typically involve around a **plan**. A software plan forms the baseline that is used heavily for project monitoring and control during project execution. This makes planning the most important management activity in a project. All project management activities require information, upon which the management decisions are based. Otherwise, even the essential questions—is the schedule is being met, what is the extent of cost overrun, is quality objectives being met, etc.—cannot be answered.