ST. XAVIER'S COLLEGE

MAITIGHAR, KATHMANDU

Software Engineering

Theory Assignment #1.1

Submitted by:

Aashish Raj Shrestha

013BSCCSIT002

Submitted to:

Er. Sanjay K Yadav Lecturer, Department of Computer Science St. Xavier's College	

Date of Submission: 24th August, 2016

Table of Contents

Introduction to Software Engineering

Software Engineering is the sub discipline of Computer Science that attempts to apply engineering principles to the creation, operation, modification and maintenance of the software components of various systems. As with much of Computer Science, the subject of Software Engineering is at a very early stage in its development. It is much more of an art than a science; and at present has little in common with classical engineering [1]. Software engineering is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use [2].

Someday, Software Engineering may well be concerned with the application of well-understood scientific methods to the construction, operation, modification and maintenance of software. Today, however, Software Engineering is concerned with finding ways in which to produce working software for predictable costs in predictable time. When the problems involved are very simple or when only one person is involved, implementing software to meet their own needs, there isn't much to be said, and we are a long way from having any scientific principles for the production of software. Therefore, the major focus of software engineering today is on well-tested heuristics for the production of software to solve complex problems when many people are involved in the process, as users, as analysts, as programmers, as managers, etc. Therefore most of the issues in Software Engineering are concerned with interactions among people, rather than with the production of software [1].

Software engineers use their computer science skills to create products of practical use and economic value. Software engineers are ethically responsible for the correctness, suitability, and safety of their projects. When possible, software engineers apply scientific and mathematical knowledge to their work. A software development process is a process by which user needs are translated into a software product. Software development processes are comprised of specific software development practices [2].

Software engineering is especially challenging because software is a tractable medium, requirements often change, and competitive pressures cause schedule pressure [2].

Socio Technical System

Socio-technical systems design (STSD) methods are an approach to design that considers human, social and organizational factors [3], as well as technical factors in the design of organizational systems. They have a long history and are intended to ensure that the technical and organizational aspects of a system are considered together. The outcome of applying these methods is a better understanding of how human, social and organizational factors affect the ways that work is done and technical systems are used. This understanding can contribute to the design of organizational structures, business processes and technical systems. Even though many managers realize that socio-technical issues are important, socio-technical design methods are rarely used. We suspect that the reasons for their lack of use are,

Primarily, difficulties in using the methods and the disconnection between these methods and both technical engineering issues, and issues of individual interaction with technical systems [4].

Critical System

IEEE defines critical software as *the software whose failure could have an impact on safety, or could cause large financial or social loss [6].*

'Failure' can result in significant economic losses, physical damage, threats to human life or the system's environment or the existence of organization which operates the system i.e. system failure can have severe human or economic consequence [5]. 'Failure' in this context does NOT mean failure to conform to a specification but means any potentially threatening system behavior [8].

If the system failure results in significant economic losses, physical damages or threats to human life than the system is called ***critical system****.* As system failure is uncertain, we might face it at any time and at any moment. One of the cause of system failure can be because of the use of critical software.

Examples of Critical Systems Are [8]:

- Communication systems such as telephone switching systems, aircraft radio systems, etc.
- Embedded control systems for process plants, medical devices, etc.
- Command and control systems such as air-traffic control systems, disaster management systems, etc.
- Financial systems such as foreign exchange transaction systems, account management systems, etc.

There are three types of critical system [7]:

1. ***Safety-critical systems:*** A system whose failure may result in injury, loss of life or serious environmental damage. An example of a safety-critical system is a control system for a chemical manufacturing plant.
2. ***Mission-critical systems:*** A system whose failure may result in the failure of some goal-directed activity. An example of a mission-critical system is a navigational system for a spacecraft.
3. ***Business-critical systems:*** A system whose failure may result in very high costs for the business using that system. An example of a business-critical system is the customer accounting system in a bank.

The high costs of failure of *critical systems* means that trusted methods and techniques must be used for development. Consequently, critical systems are usually developed using well-tried techniques rather than newer techniques that have not been subject to extensive practical experience. Rather than embrace new techniques and methods, critical systems developers are naturally conservative. They prefer to use older techniques whose strengths and weaknesses are understood, rather than new techniques which may appear to be better but whose long-term problems are unknown.

Expensive software engineering techniques that are not cost-effective for non-critical systems may sometimes be used for critical systems development. For example, formal mathematical methods of software development have been successfully used for safety and security critical systems. One reason why these formal methods are used is that it helps reduce the amount of testing required. For critical systems, the costs of verification and validation are usually very high—more than 50% of the total system development costs.

Software Process

The systematic approach that is used in software engineering is sometimes called a software process. A software process is a sequence of activities that leads to the production of a software product. There are four fundamental activities that are common to all software processes. These activities are [9]:

1. Software specification, where customers and engineers define the software that is 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 reflect changing customer and market requirements.

Different types of systems need different development processes. For example, real-time software in an aircraft has to be completely specified before development begins. In e-commerce systems, the specification and the program are usually developed together. Consequently, these generic activities may be organized in different ways and described at different levels of detail depending on the type of software being developed.

Project Management

Software project management is an essential part of software engineering. Projects need to be managed because professional software engineering is always subject to organizational budget and schedule constraints. The project manager's job is to ensure that the software project meets and overcomes these constraints as well as delivering high-quality software. Good management cannot guarantee project success. However, bad management usually results in project failure: the software may be delivered late, cost more than originally estimated, or fail to meet the expectations of customers [10].

The success criteria for project management obviously vary from project to project but, for most projects, important goals are:

1. Deliver the software to the customer at the agreed time.
2. Keep overall costs within budget.
3. Deliver software that meets the customer's expectations.
4. Maintain a happy and well-functioning development team.

Software project management is crucial to the success of a project. The basic task is to plan the detailed implementation of the development process to ensure that the cost and quality objectives are met. It specifies what is needed to meet the cost, quality, and schedule objectives. For this purpose we need monitoring and control. Monitoring obtains information from the development process and exerts the required control over it. Figure 1.1 shows where monitoring and control is carried out in project management [11]:

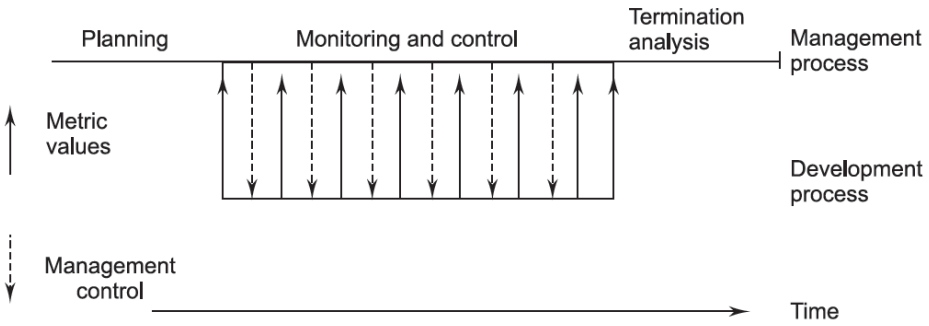

Figure 1.1: Phases of Project Management

Monitoring and control systems are an important class of a real-time system. They check sensors and provide information about the system's environment and take actions depending on the sensor reading. Monitoring systems take action when some exceptional sensor value is detected. Control systems continuously control hardware actuators depending on the value of associated sensors.

Consider the following example: A burglar alarm system is to be implemented for a building. This uses several different types of sensors. These include movement detectors in individual rooms, window sensors on ground floor windows, which detect if a window has been broken, and door sensors, which detect a door opening on corridor doors. There are 50 window sensors, 30 door sensors, and 200 movement detectors in the system.

When a sensor detects the presence of an intruder, the system automatically calls the local police and, using a voice synthesizer, reports the location of the alarm. It switches on lights in the rooms around the active sensor and sets off an audible alarm. The alarm system is normally powered by the main power source but is equipped with a battery backup. Power loss is detected using a separate power circuit monitor that monitors the main voltage. It interrupts the alarm system when a voltage drop is detected.

Software Requirement

The requirements describe the "what" of a system, not the "how."[13] The requirements for a system are the descriptions of what the system should do—the services that it provides and the constraints on its operation. These requirements reflect the needs of customers for a system that serves a certain purpose such as controlling a device, placing an order, or finding information [12].

Then the **analysis of requirement and specification** produces a large document and contains a description of what the system will do without describing how it will be done. The resultant document is known as the software requirement specification (SRS) document.

A *software requirements specification* (SRS) is a document that is created when a detailed description of all aspects of the software to be built must be specified before the project is to commence. It is important to note that a formal SRS is not always written. In fact, there are many instances in which effort expended on an SRS might be better spent in other software engineering activities. However, when software is to be developed by a third party, when a lack of specification would create severe business issues, or when a system is extremely complex or business critical, an SRS may be justified [14].

An SRS document must contain the following [13]:

- Detailed statement of problem.
- Possible alternative solution to problem.
- Functional requirements of the software system.
- Constraints on the software system.

The SRS document must be precise, consistent, and complete. There is no scope for any ambiguity or contradiction in the SRS document. An SRS document may be organized as a problem statement, introduction to the problem, functional requirements of the system, non-functional requirements of the system, behavioral descriptions, and validation criteria.

The term 'requirement' is not used consistently in the software industry. In some cases, a requirement is simply a high-level, abstract statement of a service that a system should provide or a constraint on a system. At the other extreme, it is a detailed, formal definition of a system function. Davis (1993) explains why these differences exist [12]:

If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not predefined. The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization's needs. Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the requirements document for the system.

Some of the problems that arise during the requirements engineering process are a result of failing to make a clear separation between these different levels of description. I distinguish between them by using the term 'user requirements' to mean the high-level abstract requirements and 'system requirements' to mean the detailed description of what the system should do. User requirements and system requirements may be defined as follows:

1. User requirements are statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate.

2. System requirements are more detailed descriptions of the software system's functions, services, and operational constraints. The system requirements document (sometimes called a functional specification) should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers.

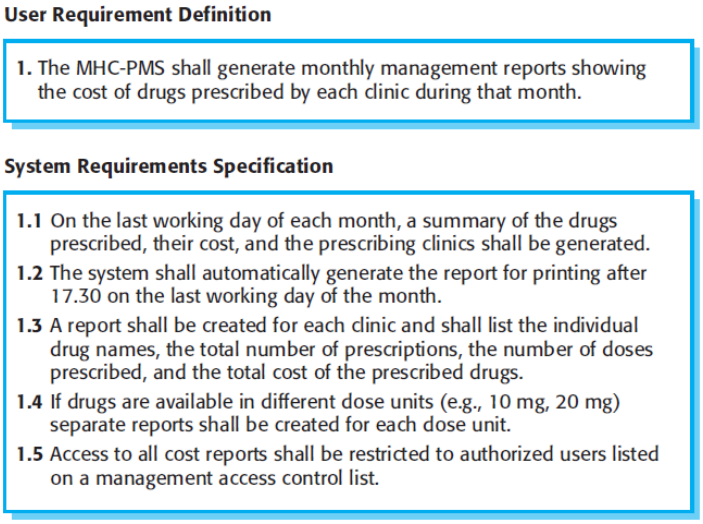

Figure 1.2: Users and System Requirement

Different levels of requirements are useful because they communicate information about the system to different types of reader. Figure 1.2 illustrates the distinction between user and system requirements. This example from a mental health care patient management system (MHC-PMS) shows how a user requirement may be expanded into several system requirements. You can see from Figure 1.2 that the user requirement is quite general. The system requirements provide more specific information about the services and functions of the system that is to be implemented [12].

Requirements Engineering Process

The software requirements reflect the needs of customers for a system that serves a certain purpose such as controlling a device, placing an order, or finding information. The process of finding out, analyzing, documenting and checking these services and constraints is called requirements engineering (RE) [12].

A requirement is a feature of the system or a description of something the system is capable of doing in order to fulfill the system's purpose [16].

According to Sommerville & Sawyer (1997), selection of RE process depends on many things: organization, systems engineering and software development process, type of the software developed, etc. All processes do not fit to all organizations. Usually good requirements engineering process includes the following activities [15]:

1. Requirements elicitation, where the system requirements are discovered through consulting the stakeholders, from system documentation, domain knowledge and market studies.
2. Requirements analysis and negotiation, where the requirements are analyzed in detail, and they are accepted by the stakeholders.

3. Requirements validation, where the consistency and completeness of the requirements is checked. These activities also need support to accommodate changes in the requirements. In embedded computer systems, due to physical constraints (e.g., timing, heat production), system requirements engineering is very important, even more important than software requirements engineering.

For Example, we take requirements engineering consists of the following processes [16]:

- Requirements gathering (elicitation).
- Requirements analysis and modeling.
- Requirements documentation.
- Requirements review.
- Requirements management.

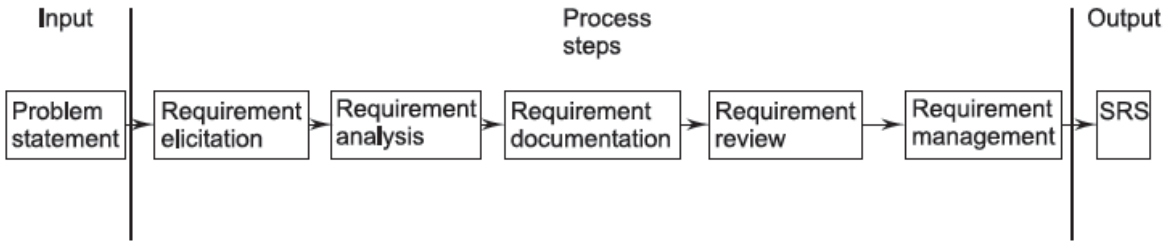

Figure 1.3: Process Steps of Requirements Engineering

Critical System Specification

The diversity of possible users means that the requirements document has to be a compromise between communicating the requirements to customers, defining the requirements in precise detail for developers and testers, and including information about possible system evolution. Information on anticipated changes can help system designers avoid restrictive design decisions and help system maintenance engineers who have to adapt the system to new requirements [17].

The level of detail that you should include in a requirements document depends on the type of system that is being developed and the development process used. Critical systems need to have detailed requirements because safety and security have to be analyzed in detail. When the system is to be developed by a separate company (e.g., through outsourcing), the system specifications need to be detailed and precise. If an in-house, iterative development process is used, the requirements document can be much less detailed and any ambiguities can be resolved during development of the system.

Architectural Design

Large systems are always decomposed into subsystems that provide some related set of services. The initial design process of identifying these subsystems and establishing a framework for subsystem control and communication is called architectural design [19].

Architectural design is concerned with understanding how a system should be organized and designing the overall structure of that system [20]. The architectural design defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented. The architectural design representation—the framework of a computer-based system—is derived from the requirements model [18].

In the model of the software development process, architectural design is the first stage in the software design process. It is the critical link between design and requirements engineering, as it identifies the main structural components in a system and the relationships between them. The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components.

Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.

Architectural design methods have a look into various architectural styles for designing a system. These are [19]:

- Data-centric architecture
- Data-flow architecture
- Object-oriented architecture
- Layered architecture

Data-centric architecture involves the use of a central database operation of inserting and updating it in the form of a table. Data-flow architecture is central around the pipe and filter mechanism. This architecture is applied when input data takes the form of output after passing through various phases of transformations. These transformations can be via manipulations or various computations done on the data. In object-oriented architecture the software design moves around the clauses and objects of the system. The class encapsulates the data and methods.

Layered architecture defines a number of layers and each layer performs tasks. The outer-most layer handles the functionality of the user interface and the innermost layer mainly handles interaction with the hardware.

In practice, there is a significant overlap between the processes of requirements engineering and architectural design. Ideally, a system specification should not include any design information. This is unrealistic except for very small systems. Architectural decomposition is usually necessary to structure and organize the specification. Therefore, as part of the requirements engineering process, you might propose an abstract system architecture where you associate groups of system functions or features with large-scale components or sub-systems. You can then use this decomposition to discuss the requirements and features of the system with stakeholders [20].

Application Architecture

Application systems are intended to meet a business or organizational need. All businesses have much in common—they need to hire people, issue invoices, and keep accounts, and so on. Businesses operating in the same sector use common sector specific applications. Therefore, as well as general business functions, all phone companies need systems to connect calls, manage their network, issue bills to customers, etc. Consequently, the application systems used by these businesses also have much in common [21].

These commonalities have led to the development of software architectures that describe the structure and organization of particular types of software systems. Application architectures encapsulate the principal characteristics of a class of systems. For example, in real-time systems, there might be generic architectural models of different system types, such as data collection systems or monitoring systems. Although instances of these systems differ in detail, the common architectural structure can be reused when developing new systems of the same type.

The application architecture may be re-implemented when developing new systems but, for many business systems, application reuse is possible without reimplementation. We see this in the growth of Enterprise Resource Planning (ERP) systems from companies such as SAP and Oracle, and vertical software packages (COTS) for specialized applications in different areas of business. In these systems, a generic system is configured and adapted to create a specific business application.

For example, a system for supply chain management can be adapted for different types of suppliers, goods, and contractual arrangements.

As a software designer, you can use models of application architectures in a number of ways:

1. ***As a starting point for the architectural design process.*** If you are unfamiliar with the type of application that you are developing, you can base your initial design on a generic application architecture. Of course, this will have to be specialized for the specific system being developed, but it is a good starting point for design.

2. ***As a design checklist.*** If you have developed an architectural design for an application system, you can compare this with the generic application architecture. You can check that your design is consistent with the generic architecture.

3. ***As a way of organizing the work of the development team.*** The application architectures identify stable structural features of the system architectures and in many cases, it is possible to develop these in parallel. You can assign work to group members to implement different components within the architecture.

4. ***As a means of assessing components for reuse.*** If you have components you might be able to reuse, you can compare these with the generic structures to see whether there are comparable components in the application architecture.

5. ***As a vocabulary for talking about types of applications.*** If you are discussing a specific application or trying to compare applications of the same types, then you can use the concepts identified in the generic architecture to talk about the applications.

Object Oriented Design

To understand the object-oriented point of view, consider an example of a real-world object—the thing you are sitting in right now—a chair. **Chair** is a subclass of a much larger class that we can call **PieceOfFurniture.** Individual chairs are members (usually called instances) of the class **Chair**. A set of generic attributes can be associated with every object in the class **PieceOfFurniture**. For example, all furniture has a cost, dimensions, weight, location, and color, among many possible attributes. These apply whether we are talking about a table or a chair, a sofa or an armoire. Because **Chair** is a member of **PieceOfFurniture**, **Chair** inherits all attributes defined for the class [22].

Requirements modeling (also called analysis modeling) focuses primarily on classes that are extracted directly from the statement of the problem. These *entity classes* typically represent things that are to be stored in a database and persist throughout the duration of the application (unless they are specifically deleted).

Design refines and extends the set of entity classes. Boundary and controller classes are developed and/or refined during design. *Boundary classes* create the interface (e.g., interactive screen and printed reports) that the user sees and interacts with as the software is used. Boundary classes are designed with the responsibility of managing the way entity objects are represented to users.

Controller classes are designed to manage (1) the creation or update of entity objects, (2) the instantiation of boundary objects as they obtain information from entity objects, (3) complex communication between sets of objects, and (4) validation of data communicated between objects or between the user and the application.

In the object-oriented design approach, the basic abstractions are not the real-world functions, but are the data abstraction where the real-world entities are represented, such as picture, machine, radar system, customer, student, employee, etc [23].

In this design, the functions are grouped together on the basis of the data they operate on, such as in class person, function displays are made member functions to operate on its data members such as the person name, age, etc.

In this approach, the state information is not represented in a centralized shared memory but is implemented/distributed among the objects of the system.

An object-oriented system is made up of interacting objects that maintain their own local state and provide operations on that state. The representation of the state is private and cannot be accessed directly from outside the object. Object-oriented design processes involve designing object classes and the relationships between these classes. These classes define the objects in the system and their

interactions. When the design is realized as an executing program, the objects are created dynamically from these class definitions [24].

Object-oriented systems are easier to change than systems developed using functional approaches. Objects include both data and operations to manipulate that data. They may therefore be understood and modified as stand-alone entities. Changing the implementation of an object or adding services should not affect other system objects. Because objects are associated with things, there is often a clear mapping between real-world entities (such as hardware components) and their controlling objects in the system. This improves the understandability, and hence the maintainability, of the design.

To develop a system design from concept to detailed, object-oriented design, there are several things that you need to do [24]:

1. Understand and define the context and the external interactions with the system.
2. Design the system architecture.
3. Identify the principal objects in the system.
4. Develop design models.
5. Specify interfaces.

Like all creative activities, design is not a clear-cut, sequential process. You develop a design by getting ideas, proposing solutions, and refining these solutions as information becomes available. You inevitably have to backtrack and retry when problems arise. Sometimes you explore options in detail to see if they work; at other times you ignore details until late in the process. Consequently, we have deliberately not illustrated this process as a simple diagram because that would imply design can be thought of as a neat sequence of activities. In fact, all of the above activities are interleaved and so influence each other.

We illustrate these process activities by designing part of the software for the wilderness weather station. Wilderness weather stations are deployed in remote areas. Each weather station records local weather information and periodically transfers this to a weather information system, using a satellite link.

Distributed System Architecture

Virtually all large computer-based systems are now distributed systems. A distributed system is one involving several computers, in contrast with centralized systems where all of the system components execute on a single computer. Tanenbaum and Van Steen (2007) define a distributed system to be [29]:

". . .a collection of independent computers that appears to the user as a single coherent system."

Obviously, the engineering of distributed systems has a great deal in common with the engineering of any other software. However, there are specific issues that have to be taken into account when designing this type of system. These arise because the system components may be running on independently managed computers and they communicate across a network.

Coulouris et al. (2005) identify the following advantages of using a distributed approach to systems development:

1. *Resource sharing* A distributed system allows the sharing of hardware and software resources—such as disks, printers, files, and compilers—that are associated with computers on a network.

2. *Openness* Distributed systems are normally open systems, which means that they are designed around standard protocols that allow equipment and software from different vendors to be combined.

3. *Concurrency* In a distributed system, several processes may operate at the same time on separate computers on the network. These processes may (but need not) communicate with each other during their normal operation.

4. *Scalability* In principle at least, distributed systems are scalable in that the capabilities of the system can be increased by adding new resources to cope with new demands on the system. In practice, the network linking the individual computers in the system may limit the system scalability.

5. *Fault tolerance.* The availability of several computers and the potential for replicating information means that distributed systems can be tolerant of some hardware and software failures (see Chapter 13). In most distributed systems, a degraded service can be provided when failures occur; complete loss of service only occurs when there is a network failure.

For large-scale organizational systems, these advantages mean that distributed systems have largely replaced mainframe legacy systems that were developed in the 1990s. However, there are many personal computer application systems (e.g., photo editing systems) that are not distributed and which run on a single computer system. Most embedded systems are also single processor systems.

Distributed systems are inherently more complex than centralized systems. This makes them more difficult to design, implement, and test. It is harder to understand the emergent properties of distributed systems because of the complexity of the interactions between system components and the system infrastructure. For example, rather than the performance of the system being dependent on the execution speed of one processor, it depends on the network bandwidth, the network load, and the speed of all of the computers that are part of the system. Moving resources from one part of the system to another can significantly affect the system's performance.

Furthermore, as all users of the WWW know, distributed systems are unpredictable in their response. The response time depends on the overall load on the system, its architecture and the network load. As all of these may change over a short time, the time taken to respond to a user request may vary dramatically from one request to another.

The most important development that has affected distributed software systems in the past few years is the service-oriented approach. Much of this chapter focuses on general issues of distributed systems.

Example: Client–server architectures are usually thought of as distributed systems architectures but the logical model of independent services running on separate servers can be implemented on a single

computer. Again, an important benefit is separation and independence. Services and servers can be changed without affecting other parts of the system [29].

An Example: Clients may have to know the names of the available servers and the services that they provide. However, servers do not need to know the identity of clients or how many clients are accessing their services. Clients access the services provided by a server through remote procedure calls using a request-reply protocol such as the http protocol used in the WWW. Essentially, a client makes a request to a server and waits until it receives a reply.

Figure 1.4 is an example of a system that is based on the client–server model. This is a multi-user, web-based system for providing a film and photograph library. In this system, several servers manage and display the different types of media. Video frames need to be transmitted quickly and in synchrony but at relatively low resolution. They may be compressed in a store, so the video server can handle video compression and decompression in different formats. Still pictures, however, must be maintained at a high resolution, so it is appropriate to maintain them on a separate server [25].

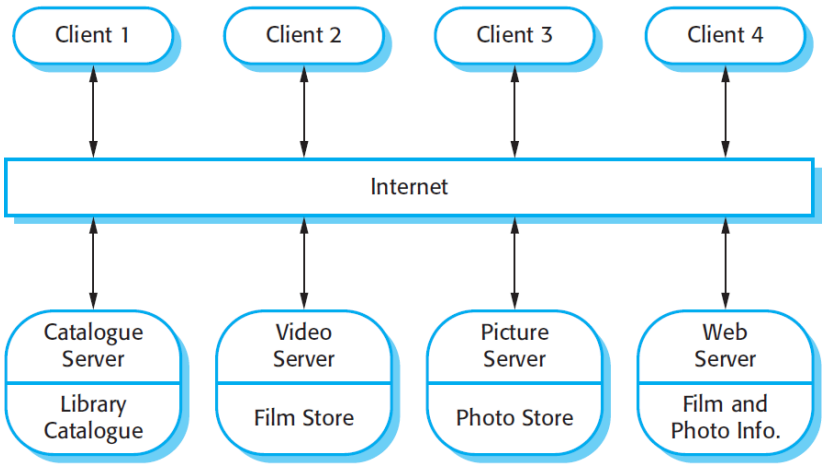

Figure 1.4: A client server architecture for a film library

The catalog must be able to deal with a variety of queries and provide links into the web information system that includes data about the film and video clips, and an e-commerce system that supports the sale of photographs, film, and video clips. The client program is simply an integrated user interface, constructed using a web browser, to access these services.

The most important advantage of the client–server model is that it is a distributed architecture. Effective use can be made of networked systems with many distributed processors. It is easy to add a new server and integrate it with the rest of the system or to upgrade servers transparently without affecting other parts of the system [25].

Designers of distributed systems have to organize their system designs to find a balance between performance, dependability, security, and manageability of the system. There is no universal model of system organization that is appropriate for all circumstances so various distributed architectural styles

have emerged. When designing a distributed application, you should choose an architectural style that supports the critical non-functional requirements of your system.

In this section, we discuss five architectural styles [30]:

1. **Master-slave architecture**, which is used in real-time systems in which guaranteed interaction response times are required.

2. **Two-tier client–server architecture**, which is used for simple client–server systems, and in situations where it is important to centralize the system for security reasons. In such cases, communication between the client and server is normally encrypted.

3. **Multitier client–server architecture**, which is used when there is a high volume of transactions to be processed by the server.

4. **Distributed component architecture**, which is used when resources from different systems and databases need to be combined, or as an implementation model for multi-tier client–server systems.

5. **Peer-to-peer architecture**, which is used when clients exchange locally stored information and the role of the server is to introduce clients to each other. It may also be used when a large number of independent computations may have to be made.

Real Time Software Design

Real-time software: Real-time software deals with a changing environment and is multi-user in which many users can perform or work with a software at the same time. First, it collects the input and converts it from analog to a digital, control component that responds to the external environment and performs the action. The software is used to monitor, control, and analyze real-world events as they occur. Examples include Rocket launching, games, etc. [31]

Embedded System and Real-Time Software Design: The design process for embedded systems is a systems engineering process in which the software designers have to consider in detail the design and performance of the system hardware. Part of the system design process may involve deciding which system capabilities are to be implemented in software and which in hardware. For many real-time systems embedded in consumer products, such as the systems in cell phones, the costs, and power consumption of the hardware are critical. Specific processors designed to support embedded systems may be used and, for some systems, special-purpose hardware may have to be designed and built [32].

This means that a top-down software design process, in which the design starts with an abstract model that is decomposed and developed in a series of stages, is impractical for most real-time systems. Low-level decisions on hardware, support software, and system timing must be considered early in the process. These limit the flexibility of system designers and may mean that additional software functionality, such as battery and power management, has to be included in the system.

Given that embedded systems are reactive systems that react to events in their environment, the most general approach to embedded, real-time software design is based on a stimulus-response model. A stimulus is an event occurring in the software system's environment that causes the system to react in some way; a response is a signal or message that is sent by the software to its environment.

You can define the behavior of a real-time system by listing the stimuli received by the system, the associated responses, and the time at which the response must be produced. For example, Figure 1.5 shows possible stimuli and system responses for a burglar alarm system.

Stimulus	Response
Single sensor positive	Initiate alarm; turn on lights around site of positive sensor.
Two or more sensors positive	Initiate alarm; turn on lights around sites of positive sensors; call police with location of suspected break-in.
Voltage drop of between 10% and 20%	Switch to battery backup; run power supply test.
Voltage drop of more than 20%	Switch to battery backup; initiate alarm; call police; run power supply test.
Power supply failure	Call service technician.
Sensor failure	Call service technician.
Console panic button positive	Initiate alarm; turn on lights around console; call police.
Clear alarms	Switch off all active alarms; switch off all lights that have been switched on.

Figure 1.5: Stimuli and responses for a burglar alarm system

Stimuli come from sensors in the system's environment and responses are sent to actuators, as shown in Figure 1.6. A general design guideline for real-time systems is to have separate processes for each type of sensor and actuator (Figure 1.7). These actuators control equipment, such as a pump, which then makes changes to the system's environment. The actuators themselves may also generate stimuli. The stimuli from actuators often indicate that some problem has occurred, which must be handled by the system.

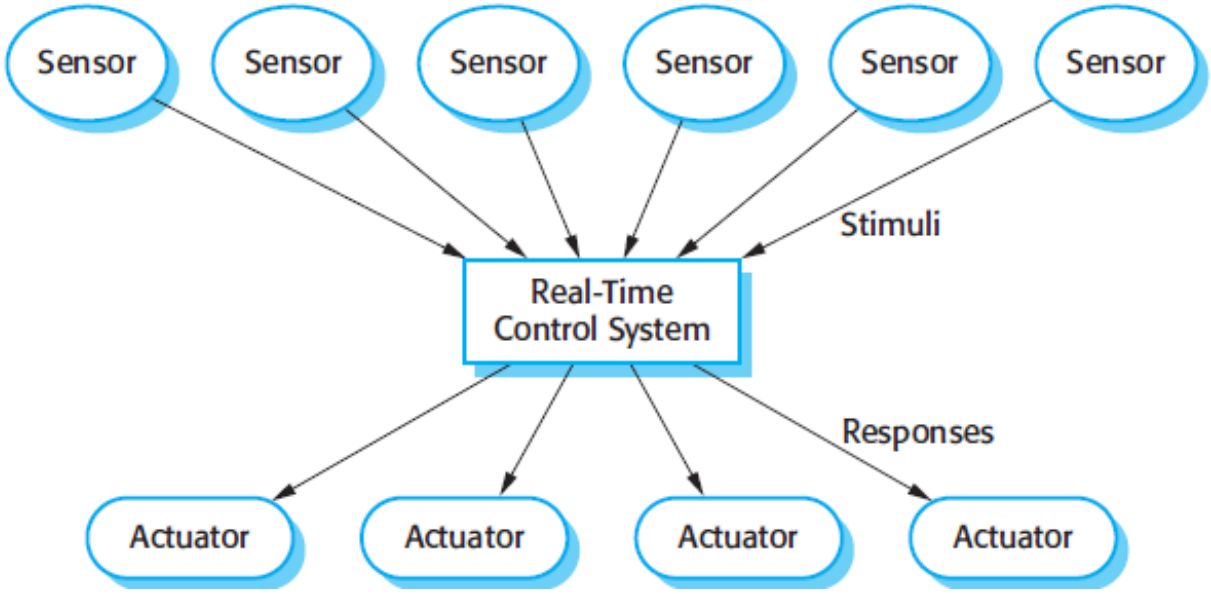

Figure 1.6: A general model of an embedded real-time system

For each type of sensor, there may be a sensor management process that handles data collection from the sensors. Data processing processes compute the required responses for the stimuli received by the system. Actuator control processes are associated with each actuator and manage the operation of that actuator. This model allows data to be collected quickly from the sensor (before it is overwritten by the next input) and allows processing and the associated actuator response to be carried out later.

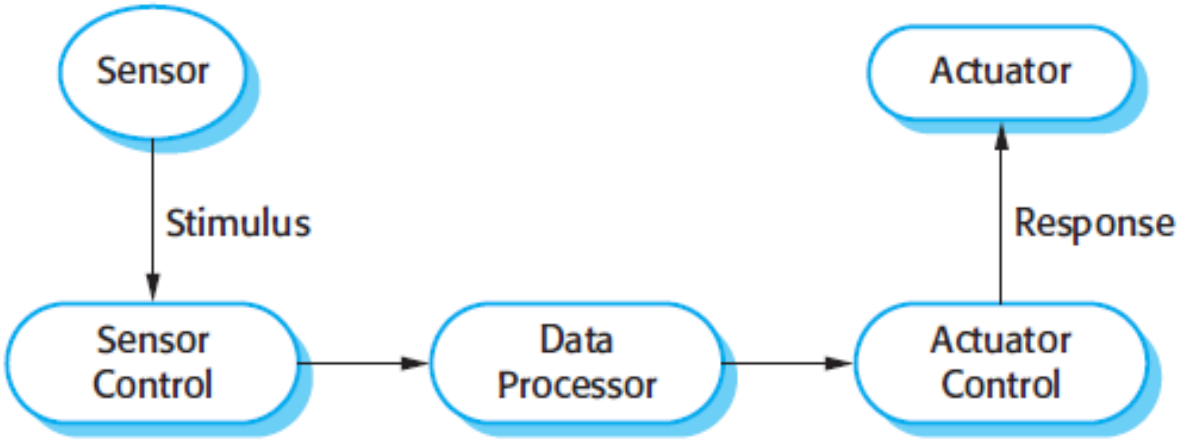

Figure 1.7: Sensor and actuator processes

A real-time system has to respond to stimuli that occur at different times. You therefore have to organize the system architecture so that, as soon as a stimulus is received, control is transferred to the correct handler. This is impractical in sequential programs. Consequently, real-time software systems are normally designed as a set of concurrent, cooperating processes. To support the management of these processes, the execution platform on which the real-time system executes may include a real-time

operating system. The functions provided by this operating system are accessed through the run-time support system for the real-time programming language that is used.

Stimuli fall into two classes [32]:

1. *Periodic stimuli.* These occur at predictable time intervals. For example, the system may examine a sensor every 50 milliseconds and take action (respond) depending on that sensor value (the stimulus).

2. *Aperiodic stimuli.* These occur irregularly and unpredictably and are usually signaled using the computer's interrupt mechanism. An example of such a stimulus would be an interrupt indicating that an I/O transfer was complete and that data was available in a buffer.

User Interface Design

The process of designing the way in which system users can access system functionality and the way that information produced by the system is displayed [37].

User interface design creates an effective communication medium between a human and a computer. Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype [36].

A software engineer designs the user interface by applying an iterative process that draws on predefined design principles.

If software is difficult to use, if it forces you into mistakes, or if it frustrates your efforts to accomplish your goals, you won't like it, regardless of the computational power it exhibits, the content it delivers, or the functionality it offers. The interface has to be right because it molds a user's perception of the software.

User interface design begins with the identification of user, task, and environmental requirements. Once user tasks have been identified, user scenarios are created and analyzed to define a set of interface objects and actions. These form the basis for the creation of screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items. Tools are used to prototype and ultimately implement the design model, and the result is evaluated for quality.

User scenarios are created and screen layouts are generated. An interface prototype is developed and modified in an iterative fashion.

An interface prototype is "test driven" by the users, and feedback from the test drive is used for the next iterative modification of the prototype [36].

Although there is significant literature on the design of human/computer interfaces, relatively little information has been published on metrics that would provide insight into the quality and usability of the interface [33].

Sears [34] suggests that *layout appropriateness* (LA) is a worthwhile design metric for human/computer interfaces. A typical GUI uses layout entities—graphic icons, text, menus, windows, and the like—to assist the user in completing tasks. To accomplish a given task using a GUI, the user must move from one layout entity to the next. The absolute and relative position of each layout entity, the frequency with which it is used, and the "cost" of the transition from one layout entity to the next all contribute to the appropriateness of the interface.

A study of Web page metrics [35] indicates that simple characteristics of the elements of the layout can also have a significant impact on the perceived quality of the GUI design. The number of words, links, graphics, colors, and fonts (among other characteristics) contained within a Web page affect the perceived complexity and quality of that page.

It is important to note that the selection of a GUI design can be guided with metrics such as LA, but the final arbiter should be user input based on GUI prototypes. Nielsen and Levy [Nie94] report that "one has a reasonably large chance of success if one chooses between interface [designs] based solely on users' opinions. Users' average task performance and their subjective satisfaction with a GUI are highly correlated." [33]

Software Reuse

Reuse: Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code [41].

Software Reuse: Software reuse is a technique that leads to the overall productivity of an organization that is involved in developing many products, but developing reusable modules is harder than developing modules for one's own use, thus reducing the productivity of the group that is developing reusable modules as part of their product development [40].

From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language. The only significant reuse or software was the reuse of functions and objects in programming language libraries [41].

However, costs and schedule pressure meant that this approach became increasingly unviable, especially for commercial and Internet-based systems. Consequently, an approach to development based around the reuse of existing software emerged and is now generally used for business systems, scientific software, and, increasingly, in embedded systems engineering.

Software reuse is possible at a number of different levels [41]:

1. *The abstraction level:* At this level, you don't reuse software directly but rather use knowledge of successful abstractions in the design of your software. Design patterns and architectural patterns are ways of representing abstract knowledge for reuse.

2. *The object level:* At this level, you directly reuse objects from a library rather than writing the code yourself. To implement this type of reuse, you have to find appropriate libraries and discover if the

objects and methods offer the functionality that you need. For example, if you need to process mail messages in a Java program, you may use objects and methods from a JavaMail library.

3. *The component level:* Components are collections of objects and object classes that operate together to provide related functions and services. You often have to adapt and extend the component by adding some code of your own. An example of component-level reuse is where you build your user interface using a framework. This is a set of general object classes that implement event handling, display management, etc. You add connections to the data to be displayed and write code to define specific display details such as screen layout and colors.

4. *The system level:* At this level, you reuse entire application systems. This usually involves some kind of configuration of these systems. This may be done by adding and modifying code (if you are reusing a software product line) or by using the system's own configuration interface. Most commercial systems are now built in this way where generic COTS (commercial off-the-shelf) systems are adapted and reused. Sometimes this approach may involve reusing several different systems and integrating these to create a new system.

By reusing existing software, you can develop new systems more quickly, with fewer development risks and also lower costs. As the reused software has been tested in other applications, it should be more reliable than new software. However, there are costs associated with reuse [41]:

1. The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs. You may have to test the software to make sure that it will work in your environment, especially if this is different from its development environment.

2. Where applicable, the costs of buying the reusable software. For large off-theshelf systems, these costs can be very high.

3. Problem tracking, where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

Software configuration management tools support each of the above activities. These tools may be designed to work together in a comprehensive change management system, such as ClearCase (Bellagio and Milligan, 2005). In integrated configuration management systems, version management, system integration, and problem-tracking tools are designed together. They share a user interface style and are integrated through a common code repository.

Alternatively, separate tools, installed in an integrated development environment, may be used. Version management may be supported using a version management system such as Subversion (Pilato et al., 2008), which can support multi-site, multiteam development. System integration support may be built into the language or rely on a separate toolset such as the GNU build system. This includes what is perhaps the best-known integration tool, Unix make. Bug tracking or issue tracking systems, such as Bugzilla, are used to report bugs and other issues and to keep track of whether or not these have been fixed.

Component Based Software Engineering

Component Based Development: Commercial off-the-shelf (COTS) software components, developed by vendors who offer them as products, provide targeted functionality with well-defined interfaces that enable the component to be integrated into the software that is to be built. The *component-based development model* incorporates many of the characteristics of the spiral model. It is evolutionary in nature [39], demanding an iterative approach to the creation of software. However, the component-based development model constructs applications from prepackaged software components [38].

Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented classes or packages16 of classes. Regardless of the technology that is used to create the components, the component-based development model incorporates the following steps (implemented using an evolutionary approach):

1. Available component-based products are researched and evaluated for the application domain in question.
2. Component integration issues are considered.
3. A software architecture is designed to accommodate the components.
4. Components are integrated into the architecture.
5. Comprehensive testing is conducted to ensure proper functionality.

The component-based development model leads to software reuse, and reusability provides software engineers with a number of measurable benefits. Your software engineering team can achieve a reduction in development cycle time as well as a reduction in project cost if component reuse becomes part of your culture [38].

In the software engineering context, reuse is an idea both old and new. Programmers have reused ideas, abstractions, and processes since the earliest days of computing, but the early approach to reuse was ad hoc. Today, complex, high-quality computer based systems must be built in very short time periods and demand a more organized approach to reuse.

Component-based software engineering (CBSE) is a process that emphasizes the design and construction of computer-based systems using reusable software "components." Clements [45] describes CBSE in the following way:

[CBSE] embodies the "buy, don't build" philosophy espoused by Fred Brooks and others. In the same way that early subroutines liberated the programmer from thinking about details, [CBSE] shifts the emphasis from programming software to composing software systems. Implementation has given way to integration as the focus. [42]

Also, Component-based software engineering identifies, constructs, catalogs, and disseminates a set of software components in a particular application domain. These components are then qualified, adapted, and integrated for use in a new system. Reusable components should be designed within an

environment that establishes standard data structures, interface protocols, and program architectures for each application domain. [43]

Component-based software engineering (CBSE) emphasizes reusability—that is, the creation and reuse of software building blocks. Such building blocks, often called *components,* must be cataloged for easy reference, standardized for easy application, and validated for easy integration. Bennatan [46] suggests four software resource categories that should be considered as planning proceeds:

Off-the-shelf components. Existing software that can be acquired from a third party or from a past project. COTS (commercial off-the-shelf) components are purchased from a third party, are ready for use on the current project, and have been fully validated.

Full-experience components. Existing specifications, designs, code, or test data developed for past projects that are similar to the software to be built for the current project. Members of the current software team have had full experience in the application area represented by these components. Therefore, modifications required for full-experience components will be relatively low risk.

Partial-experience components. Existing specifications, designs, code, or test data developed for past projects that are related to the software to be built for the current project but will require substantial modification. Members of the current software team have only limited experience in the application area represented by these components. Therefore, modifications required for partial-experience components have a fair degree of risk.

New components. Software components must be built by the software team specifically for the needs of the current project.

Ironically, reusable software components are often neglected during planning, only to become a paramount concern later in the software process. It is better to specify software resource requirements early. In this way technical evaluation of the alternatives can be conducted and timely acquisition can occur [44].

Software Evolution

Regardless of its application domain, its size, or its complexity, computer software will evolve over time. Change drives this process [47]. For computer software, change occurs when errors are corrected, when the software is adapted to a new environment, when the customer requests new features or functions, and when the application is reengineered to provide benefit in a modern context. Over the past 30 years, Manny Lehman [48] and his colleagues have performed detailed analyses of industry-grade software and systems in an effort to develop a *unified theory for software evolution.* The details of this work are beyond the scope of this book, but the underlying laws that have been derived are worthy of note [49]:

The Law of Continuing Change (1974): Software that has been implemented in a real-world computing context and will therefore evolve over time (called *E-type systems*) must be continually adapted else they become progressively less satisfactory.

The Law of Increasing Complexity (1974): As an E-type system evolves its complexity increases unless work is done to maintain or reduce it.

The Law of Self-Regulation (1974): The E-type system evolution process is self-regulating with distribution of product and process measures close to normal.

The Law of Conservation of Organizational Stability (1980): The average effective global activity rate in an evolving E-type system is invariant over product lifetime.

The Law of Conservation of Familiarity (1980): As an E-type system evolves all associated with it, developers, sales personnel, users, for example, must maintain mastery of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.

The Law of Continuing Growth (1980): The functional content of E-type systems must be continually increased to maintain user satisfaction over their lifetime.

The Law of Declining Quality (1996): The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.

The Feedback System Law (1996): E-type evolution processes constitute multilevel, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

The laws that Lehman and his colleagues have defined are an inherent part of a software engineer's reality [47].

During the process of software evolution, many objects are produced; for example, files, electronic documents, paper documents, source code, executable code, and bitmap graphics. A version control tool is the first stage towards being able to manage multiple versions. Once it is in place, a detailed record of every version of the software must be kept. This report includes [50]:

- The name of each source-code component, including the variations and revisions.
- The versions of the various compliers and linkers used.
- The name of the software staff who constructed the component.
- The date and the time at which it was constructed.

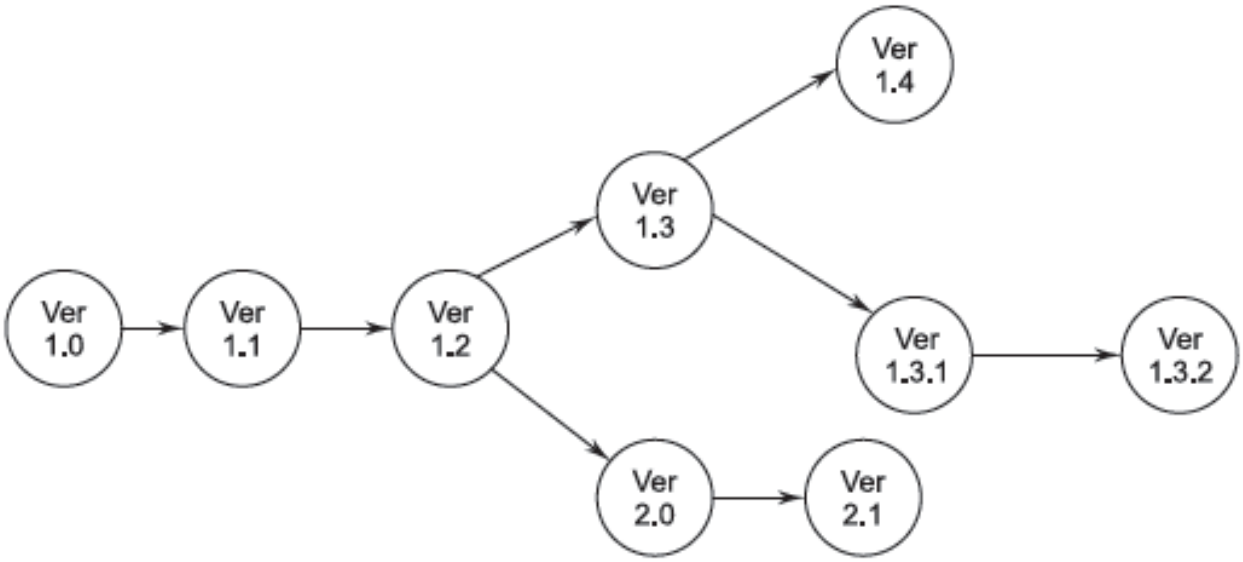

Figure 1.8: An Evolutionary Graph for a Different Versions of an Item

The above evolutionary graph (Figure 1.8) depicts the evolution of a configuration item during the development life-cycle. The initial version of the item is given version number Ver 1.0. Subsequent changes to the item which could be mostly fixing bugs or adding minor functionality is given as Ver 1.1 and Ver 1.2. After that, a major modification to Ver 1.2 is given a number Ver 2.0, and at the same time, a parallel version of the same item without the major modification is maintained and given a version number 1.3.

Commercial tools are available for version control, which perform one or more of the following tasks [50]:

- Source-code control
- Revision control
- Concurrent-version control

There are many commercial tools, such as Rational Clear Case, Microsoft Visual Source Safe, and a number of other tools to help version control.

Verification and Validation

Software testing is one element of a broader topic that is often referred to as verification and validation (V&V). *Verification* refers to the set of tasks that ensure that software correctly implements a specific function [51]. *Validation* refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm [52] states this another way:

Verification: “Are we building the product right?”

Validation: “Are we building the right product?”

The definition of V&V encompasses many software quality assurance activities.

Verification and validation (V and V) is the name given to the checking and analysis process that ensures that software conforms to its specifications and meets the needs of the customers who are paying for that software. Variation and validation is a whole life-cycle process. It starts with requirements reviews and continues through design reviews and code inspection to product testing. There should be V and V activities at each stage of the software development process. These activities ensure that the results of process activities are as specified [54].

Verification and validation includes a wide array of SQA activities: technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, usability testing, qualification testing, acceptance testing, and installation testing. Although testing plays an extremely important role in V&V, many other activities are also necessary.

Testing does provide the last bastion from which quality can be assessed and, more pragmatically, errors can be uncovered. But testing should not be viewed as a safety net. As they say, "You can't test in quality. If it's not there before you begin testing, it won't be there when you're finished testing." Quality is incorporated into software throughout the process of software engineering. Proper application of methods and tools, effective technical reviews, and solid management and measurement all lead to quality that is confirmed during testing.

Miller [53] relates software testing to quality assurance by stating that "the underlying motivation of program testing is to affirm software quality with methods that can be economically and effectively applied to both large-scale and small-scale systems." [51]

Within the V and V process, ***two techniques*** of system checking and analysis may be used. [54]

1. **Software inspections.** Software inspection analyzes and checks system representations, such as the requirements document, design diagrams, and the program source code. They may be applied at all stages of the process. Inspections may be supplemented by some automatic analysis of the source text of a system or associated documents. Software inspections and automated analyzes are static V and V techniques as they do not require the system to be executed.

2. **Software testing.** Software testing involves executing an implementation of the software with test data and examining the outputs of the software and its operational behavior to check that it is performing as required. Testing is a dynamic technique of verification and validation because it works with an executable representation of the system.

Verification [54]

Verification is the process of determining whether the output of one phase of software development confirms to that of its previous phase.

OR

Verification involves checking of functional and non-functional requirements to ensure that the software confirms to its specifications.

Validation [54]

Validation is the process of determining whether a fully developed system confirms to its requirement specifications.

OR

Validation is an analysis process that is done after checking conformance of the system to its specifications.

Thus, the goal of the verification and validation process is to establish confidence in the customer that the software system is 'fit for the customer.' It doesn't mean that the software system is free from errors.

Software Cost Estimation

Cost estimation is the next step for projects. The cost of a project is derived not only from the estimates of effort and size but from other parameters, such as hardware, travel expenses, telecommunication costs, training costs, etc. Figure 1.9 depicts the cost-estimation process [55].

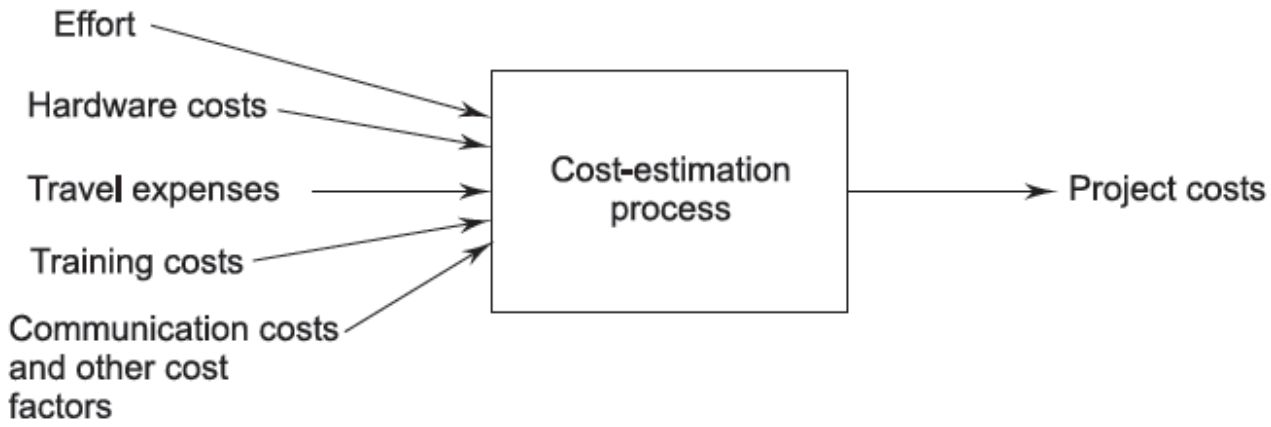

Figure 1.9: Cost-Estimation Process

Figure 1.10 depicts the project-estimation process.

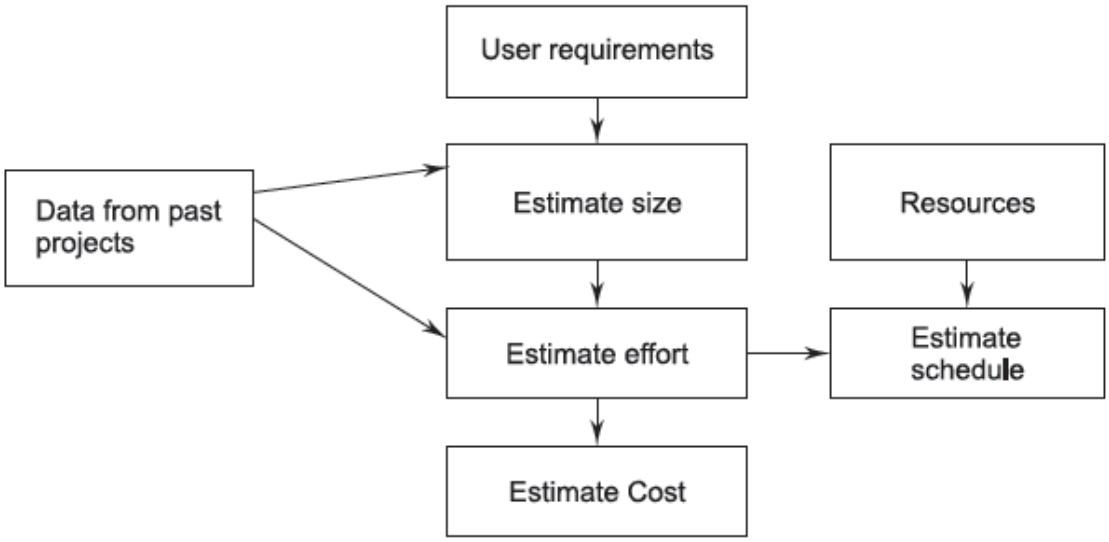

Figure 1.10: Project-estimation Process

Once the estimation is complete, we may be interested to know how accurate the estimates are. The answer to this is "we do not know until the project is complete." There is always some uncertainty associated with all estimation techniques. The accuracy of project estimation will depend on the following [55]:

- Accuracy of historical data used to project the estimation.
- Accuracy of input data to various estimates.
- Maturity of an organization's software-development process.

The following are some of the reasons cost estimation can be difficult:

- Software-cost estimation requires a significant amount of effort. Sufficient time is usually not allocated for planning.
- Software-cost estimation is often done hurriedly, without an appreciation for the actual effort required and is far from realistic.
- Lack of experience for developing estimates, especially for large projects.
- An estimator uses the extrapolation technique to estimate, ignoring the non-linear aspects of the software-development process.

Reasons for Poor/Inaccurate Estimation

The following are some of the reasons for poor and inaccurate estimation:

- Requirements are imprecise. Also, requirements change frequently.
- The project is new and is different from past projects handled.
- Non-availability of enough information about past projects.
- Estimates are forced to be based on available resources.
- Cost and time tradeoffs.

If we elongate the project, we can reduce overall costs. Usually, customers and management do not like long project durations. There is always the shortest possible duration for a project, but it comes at a cost.

The following are some of the problems associated with estimates:

- Estimating size is often skipped and a schedule is estimated, which is of more relevance to management.
- Estimating size is perhaps the most difficult step, which has a bearing on all other estimates.
- Let us not forget that even good estimates are only projections and subject to various risks.
- Organizations often give less importance to collection and analysis of historical data of past development projects. Historical data is the best input to estimate a new project.
- Project managers often underestimate the schedule because management and customers often hesitate to accept a prudent realistic schedule.

Project-estimation Guidelines

Some guidelines for project estimation are as follows [55]:

- Preserve and document data pertaining to past projects.
- Allow sufficient time for project estimation especially for bigger projects.
- Prepare realistic developer-based estimates. Associate people who will work on the project to reach a realistic and more accurate estimate.
- Use software-estimation tools.
- Re-estimate the project during the life-cycle of the development process.
- Analyze past mistakes in the estimation of projects.

Quality Control

The terms ‘quality assurance’ and ‘quality control’ are widely used in manufacturing industry. Quality assurance (QA) is the definition of processes and standards that should lead to high-quality products and the introduction of quality processes into the manufacturing process. Quality control is the application of these quality processes to weed out products that are not of the required level of quality [56].

In the software industry, different companies and industry sectors interpret quality assurance and quality control in different ways. Sometimes, quality assurance simply means the definition of procedures, processes, and standards that are aimed at ensuring that software quality is achieved. In other cases, quality assurance also includes all configuration management, verification, and validation activities that are applied after a product has been handed over by a development team [56]. Here we use the term ‘quality assurance’ to include verification and validation and the processes of checking that quality procedures have been properly applied.

Quality control encompasses a set of software engineering actions that help to ensure that each work product meets its quality goals. Models are reviewed to ensure that they are complete and consistent.

Code may be inspected in order to uncover and correct errors before testing commences. A series of testing steps is applied to uncover errors in processing logic, data manipulation, and interface communication. A combination of measurement and feedback allows a software team to tune the process when any of these work products fail to meet quality goals [57].

The goal of software quality control, and in a broader sense, quality management in general, is to remove quality problems in the software. These problems are referred to by various names—*bugs, faults, errors,* or *defects* to name a few [58].

As software engineers, we want to find and correct as many errors as possible before the customer and/or end user encounter them. We want to avoid defects—because defects (justifiably) make software people look bad.

Software Testing

Software testing involves executing an implementation of the software with test data and examining the outputs of the software and its operational behavior to check that it is performing as required. Testing is a dynamic technique of verification and validation because it works with an executable representation of the system [54].

Testing is a set of activities that can be planned in advance and conducted systematically. For this reason a template for software testing—a set of steps into which we can place specific test-case design techniques and testing methods—should be defined for the software process [59].

A number of software-testing strategies have been proposed in the literature. All provide the software developer with a template for testing and all have the following generic characteristics:

- To perform effective testing, a software team should conduct effective formal technical reviews. By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works “outward” toward the integration of the entire computer-based system.
- Different testing techniques are appropriate at different points in time.
- Testing is conducted by the developers of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

References

[1] "Some notes for Software Engineering -- Introduction", bernstein-plus-sons.com, 2016. [Online]. Available: http://www.bernstein-plus-sons.com/.dowling/POCSS08/SE_Introduction.html [Accessed: 20-August-2016].

[2] "An Introduction to Software Engineering", agile.csc.ncsu.edu, 2016. [Online]. Available: http://agile.csc.ncsu.edu/SEMaterials/Introduction.pdf [Accessed: 20-August-2016].

[3] "Socio-technical system", iwc.oxfordjournals.org, 2016. [Online]. Available: http://iwc.oxfordjournals.org/content/23/1/4.full#fn-1 [Accessed: 20-August-2016].

[4] "Socio-technical systems: From design methods to systems engineering", iwc.oxfordjournals.org, 2016. [Online]. Available: http://iwc.oxfordjournals.org/content/23/1/4.full [Accessed: 20-August-2016].

[5] "Critical Systems", ifs.host.cs.st-andrews.ac.uk, 2016. [Online]. Available: http://www.cs.ccsu.edu/~stan/classes/cs530/slides/se-03.pdf [Accessed: 20-August-2016].

[6] "IEEE Standard Glossary of Software Engineering Terminology", idi.ntnu.no, 2016. [Online]. Available: http://www.idi.ntnu.no/grupper/su/publ/ese/ieee-se-glossary-610.12-1990.pdf [Accessed: 20-August-2016].

[7] "Critical Systems", ifs.host.cs.st-andrews.ac.uk, 2016. [Online]. Available: https://ifs.host.cs.st-andrews.ac.uk/Books/SE9/Web/Dependability/CritSys.html [Accessed: 20-August-2016].

[8] "Critical Systems Engineering", courses.cs.washington.edu, 2016. [Online]. Available: https://courses.cs.washington.edu/courses/cse466/05sp/pdfs/lectures/L12-Critical_Systems.pdf [Accessed: 20-August-2016].

[9] Ian Sommerville, "Software engineering," *in Software Engineering*, 9th Ed. Boston, Massachusetts 02116, 2011, pp. 9

[10] Ian Sommerville, "Project Management," *in Software Engineering*, 9th Ed. Boston, Massachusetts 02116, 2011, pp. 594

[11] B. B. Agrawal, "Monitoring and Control for Design," *in Software Engineering & Testing*, Ed. Massachusetts, 2010, pp. 146

[12] Ian Sommerville, "Requirements Engineering," *in Software Engineering*, 9th Ed. Boston, Massachusetts 02116, 2011, pp. 83

[13] B. B. Agrawal, "Waterfall Model," *in Software Engineering & Testing*, Ed. Massachusetts, 2010, pp. 38

[14] Roger Pressman, "Software Requirements Specification Template," *in Software Engineering: A Practitioner's Approach*, 7th Ed. New York, NY 10020, 2005, pp. 123

[15] I. Sommerville, P. Sawyer. *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, 1997.

[16] B. B. Agrawal, "Requirement Engineering," *in Software Engineering & Testing*, Ed. Massachusetts, 2010, pp. 53-55

[17] Ian Sommerville, "The software requirement document," *in Software Engineering*, 9th Ed. Boston, Massachusetts 02116, 2011, pp. 91-93

[18] Roger Pressman, "Design within the context of software engineering," *in Software Engineering: A Practitioner's Approach*, 7th Ed. New York, NY 10020, 2005, pp. 217

[19] B. B. Agrawal, "Architectural Design," *in Software Engineering & Testing*, Ed. Massachusetts, 2010, pp. 123-124

[20] Ian Sommerville, "Architectural Design," *in Software Engineering*, 9th Ed. Boston,Massachusetts 02116, 2011, pp. 148

[21] Ian Sommerville, "Application architectures," *in Software Engineering*, 9th Ed. Boston,Massachusetts 02116, 2011, pp. 164-165

[22] Roger Pressman, "OBJECT-ORIENTED ANALYSIS AND DESIGN CONCEPTS," *in Software Engineering: A Practitioner's Approach*, 7th Ed. New York, NY 10020, 2005, pp. 866

[23] B. B. Agrawal, "FUNCTIONAL-ORIENTED VERSUS THE OBJECT-ORIENTED APPROACH," *in Software Engineering & Testing*, Ed. Massachusetts, 2010, pp. 143-144

[24] Ian Sommerville, "Object-oriented design using the UML," *in Software Engineering*, 9th Ed. Boston,Massachusetts 02116, 2011, pp. 178-179

[25] Ian Sommerville, "Client–server architecture," *in Software Engineering*, 9th Ed. Boston,Massachusetts 02116, 2011, pp. 161-163

[26] Ian Sommerville, "Architectural design decisions," *in Software Engineering*, 9th Ed. Boston,Massachusetts 02116, 2011, pp. 151-153

[27] Ian Sommerville, "Performance Testing," *in Software Engineering*, 9th Ed. Boston,Massachusetts 02116, 2011, pp. 227

[28] Ian Sommerville, "Include timeouts when calling external components," *in Software Engineering*, 9th Ed. Boston,Massachusetts 02116, 2011, pp. 362

[29] Ian Sommerville, "Distributed software engineering," *in Software Engineering*, 9th Ed. Boston,Massachusetts 02116, 2011, pp. 479-481

[30] Ian Sommerville, "Architectural patterns for distributed systems," *in Software Engineering*, 9th Ed. Boston,Massachusetts 02116, 2011, pp. 490

[31] B. B. Agrawal, "SOFTWARE APPLICATIONS," *in Software Engineering & Testing*, Ed. Massachusetts, 2010, pp. 15

[32] Ian Sommerville, "Embedded systems design," *in Software Engineering*, 9th Ed. Boston,Massachusetts 02116, 2011, pp. 540-543

[33] Roger Pressman, "User Interface Design Metrics," *in Software Engineering: A Practitioner's Approach*, 7th Ed. New York, NY 10020, 2005, pp. 635

[34] Sears, A., "Layout Appropriateness: A Metric for Evaluating User Interface Widget Layout, *IEEE Trans. Software Engineering*, vol. SE-19, no. 7, July 1993, pp. 707–719.

[35] Ivory, M., R. Sinha, and M. Hearst, "Empirically Validated Web Page Design Metrics," webtango.berkeley.edu, [Online]. Available: http://webtango.berkeley.edu/papers/chi2001/. [Accessed: 21- August- 2016].

[36] Roger Pressman, "User Interface Design," *in Software Engineering: A Practitioner's Approach*, 7th Ed. New York, NY 10020, 2005, pp. 312

[37] Ian Sommerville, "user interface design," *in Software Engineering*, 9th Ed. Boston,Massachusetts 02116, 2011, pp. 747

[38] Roger Pressman, "Component-Based Development," *in Software Engineering: A Practitioner's Approach*, 7th Ed. New York, NY 10020, 2005, pp. 50-51

[39] Nierstrasz, O., S. Gibbs, and D. Tsichritzis, "Component-Oriented Software Development," *CACM*, vol. 35, no. 9, September 1992, pp. 160–165.

[40] B. B. Agrawal, "Representative Qualities," *in Software Engineering & Testing*, Ed. Massachusetts, 2010, pp. 95

[41] Ian Sommerville, "Implementation issues," *in Software Engineering*, 9th Ed. Boston,Massachusetts 02116, 2011, pp. 193-196

[42] Roger Pressman, "Component-Based Development," *in Software Engineering: A Practitioner's Approach*, 7th Ed. New York, NY 10020, 2005, pp. 303

[43] Roger Pressman, "Component Level Design," *in Software Engineering: A Practitioner's Approach*, 7th Ed. New York, NY 10020, 2005, pp. 310

[44] Roger Pressman, "Reusable Software Resources," *in Software Engineering: A Practitioner's Approach*, 7th Ed. New York, NY 10020, 2005, pp. 696

[45] Clements, P., "From Subroutines to Subsystems: Component Based Software Development," *American Programmer*, vol. 8, no. 11, November 1995.

[46] Bennatan, E. M., *Software Project Management: A Practitioner's Approach*, 3d ed., McGraw-Hill, 2000.

[47] Roger Pressman, "Maintenance and Reengineering," *in Software Engineering: A Practitioner's Approach*, 7th Ed. New York, NY 10020, 2005, pp. 761-763

[48] Lehman, M., and L. Belady, *Program Evolution: Processes of Software Change*, Academic Press, 1997.

[49] Lehman, M., et al., "Metrics and Laws of Software Evolution—The Nineties View," *Proceedings of the 4th International Software Metrics Symposium (METRICS '97)*, IEEE, 1997, [Online]. Downloadable: www.ece.utexas.edu/~perry/work/papers/feast1.pdf. [Accessed: 21- August- 2016].

[50] B. B. Agrawal, "SOFTWARE-VERSION CONTROL," *in Software Engineering & Testing*, Ed. Massachusetts, 2010, pp. 199-200

[51] Roger Pressman, "Verification and Validation," *in Software Engineering: A Practitioner's Approach*, 7th Ed. New York, NY 10020, 2005, pp. 450-451

[52] Boehm, B., *Software Engineering Economics*, Prentice Hall, 1981.

[53] Miller, E., "The Philosophy of Testing," in *Program Testing Techniques*, IEEE Computer Society Press, 1977, pp. 1–3.

[54] B. B. Agrawal, "VERIFICATION AND VALIDATION," *in Software Engineering & Testing*, Ed. Massachusetts, 2010, pp. 85-86

[55] B. B. Agrawal, "Estimating Cost," *in Software Engineering & Testing*, Ed. Massachusetts, 2010, pp. 209-211

[56] Ian Sommerville, "Quality Management," *in Software Engineering*, 9th Ed. Boston,Massachusetts 02116, 2011, pp. 652

[57] Roger Pressman, "Quality Control," *in Software Engineering: A Practitioner's Approach*, 7th Ed. New York, NY 10020, 2005, pp. 412

[58] Roger Pressman, "Bugs, Errors, and Defects," *in Software Engineering: A Practitioner's Approach*, 7th Ed. New York, NY 10020, 2005, pp. 417

[59] B. B. Agrawal, "Software Testing," *in Software Engineering & Testing*, Ed. Massachusetts, 2010, pp. 161