CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

Software products have significantly changed the facets of the work and daily life. Software products can be seen in the working place, offices, cars, the Internet etc. Moreover, almost all electronic equipment contains some kind of software. Software helps in supporting and manufacturing industry, education, entertainment, health and financial services, economic analysis, research, management activities, and many other domains. Software development has already become one of the largest industries in the world. Software engineering as a discipline emerged in the late 1960s and became increasingly important during the past 40 years. The prime objective of software engineering is to bring sound engineering disciplines to the development of software products to improve their quality. However, unlike other engineering disciplines, software engineering is still a young discipline, yet has its extraordinary characteristics. It’s inherently complicated, multi-disciplinary and multi-dimensional nature made it a unique discipline which requires specific solutions by combining knowledge from different disciplines. Creation of reliable, maintainable, etc. software systems that satisfy customer requirements is a hard task, which is steadily becoming more and more difficult. The size and complexity of software systems are growing constantly. In many cases, small teams of developers cannot deliver new product versions on a schedule time [GSK2010] because of the ever increasing number of failures. As the number of developers grows, the management of the development team as well as the systematic and efficient communication and document sharing among its members becomes essential. Software development organizations now frequently use third party libraries, components and services in their projects. They also experience pressure from their customers to deliver customized and easily maintainable systems quickly and cheaply. Software Engineering studies the methods and techniques that help develop reliable, efficient, maintainable, evolvable, etc.
software systems on time and on budget alleviating the above-mentioned difficulties [JJJ1998][JMV2005]. The definition given as per IEEE standard that: “Software engineering is the application of a systematic, disciplined, quality approach to the development, operation and maintenance of software; that is, the application of engineering to software” [IEEE1990].

A definition of software engineering is given by [DAN1992] which described its different facets as: “Software engineering is that form of engineering applies a systematic, disciplined, quantifiable approach, the principles of computer science design, engineering, management, mathematics, psychology, sociology and other disciplines as necessary and sometimes just plain invention, to create, develop, operate and maintain cost effective, reliably correct, high-quality solutions to software problems” .Software engineering has been defined differently depending on the perspectives, such as computer science or engineering.

The definition given by H. Pham [PET1968] is “Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines”. Most of the software systems have a high inherent complexity; so the engineering principles have to be used in their development to make it better; Software is nothing but the collection of instructions or the computer programs when executed by desired functions to obtain desired output or to meet the required performance. The two main concern of software engineers are: software quality and reliability. Software reliability means trustworthiness or dependability i.e. the ability of a program to perform a required function under given condition for a given period of time. In general, Software Engineers follow a systematic and organized methodology for their work, as this proved the most effective way to produce high-quality software. However, engineering is all about selecting the most correct method for a given set of circumstances and a more innovative informal approach to development may be effective in some circumstances.

Component Based Software approach is modularizing systems as software components as a new programming paradigm. Component Based Software Development (CBSD) [LUI2002] is gaining popularity as the component
technology for the construction of high-quality, evolvable, large software systems, developed in time and affordable way. CBSD support the development and plug and play, reusable software for reducing developing costs and efforts for improving the flexibility and reliability of the any application due to the use or reuse of software components. As per the researcher that adaptability is more important than reuse. In the days of competition, new version and maintenance of software is crucial. So it is required to design a complex system by assembling highly cohesive loosely coupled components. The cost of redesigning of such adoptable components or replacing by a better component is minimized. One approach to create component-based software using object-oriented programming is based on interface based programming. However interface based program support distributed systems. Computer program is inherently distributed. Interface-based programming in the Component based approach is extended to distributed systems with component object models in a distributed pattern.

Component-based software engineering (CBSE) is a branch of software engineering that emphasizes the separation of concepts in form of independent functions applying software system life cycle model. It is a complete reuse-based approach to define requirement, designing, implementing and composing loosely coupled independent components into systems. This gives benefits to both the short-term and the long-term for the software developer itself and for organizations. The components are considered as starting point for service-orientation. In Service-Oriented Architectures (SOA), a component is converted into a service and subsequently and adds some more characteristics to make it usable. The components are an initiator and terminating of events and play a major and individual role for Event Driven Architectures (EDA). The concept of re-use is not a started with this CBSE only. But this concept in software engineering context is using since many years. But the introduction of object oriented programming system gave a new idea and this approach changed the technology in the field of software engineering. Today complex and high quality software systems are designed efficiently using component based approach in a short span of time. The importance of Component Based
development lies in its efficiency. It takes only a few person efforts to assemble the software system because the components are designed to be integrated with ease.

Software reuse is technique in which already designed software’s code in software system is used. There are many benefits of software reuse. It saves the potential, cost, time etc. There result [KHO2009] is increase in the product quality, delivery on time, under budgeted development and also decrease in the maintenance cost. From the above discussed benefits, software reuse plays an important role in software development. When a reuse of component is need to select the proper component. For reusing there is a need to retrieve the components form the repository which is a special database containing a large number of components that may be reused in development process. Before retrieving a component, a search that component is required. There are a lot of component having similar name, similar functionality etc. that make the searching process difficult and ambiguous. Each component is stored with a name generally a string. So to search a component keyword based searching is applied. So the existing string matching algorithms can be applied on the component searching. Some technique are earlier developed which can be used to search a component which are based on the string matching algorithms are described below. They have their own benefits and drawbacks but do not fulfill all the requirements. Advancement in Component Based Approach is Component-based Software approach is for the rapid assembly of flexible software components. CBSE combines constituents of software architecture, modular design, software verification, configuration and its deployment. Software project is designed to be compatible and collaboration with the software architecture community. In CBSE with the Quality of Software Architectures (QoSA) conference and the International Symposium on Architecting Critical Systems (ISARCS), is calculated as part of the federated Comp Architecture event. The theoretical concept of component specification, composition, analysis and verification are ready to face research challenges. While the use of models and methods for component software development is increasing. New trends in global services like distributed systems architectures, and large scale software systems with their high-
performance.

1.2 VARIOUS APPROACHES FOR SOFTWARE ENGINEERING

There are three major approaches in this world such as structured, object-oriented, and component-based for developing any software system.

1.2.1 Structured Approach

This approach contains basic steps of a software development process such as analysis, design, implementation, testing, and maintenance. This thesis focuses on only analysis and design steps and their models. Each of the elements of the analysis model [ROG1997] provides information that is required to create a design model. The flow of information during software design is illustrated in

![Analysis and Design Models for Structured Approach](image)

**Figure 1.1 Analysis and Design Models for Structured Approach [ROG1997]**

Figure 1.1. This approach supports a variety of elements such as a data dictionary, data flow diagrams, state transition diagrams, entity-relationship diagrams; process specifications, control specifications and data object descriptions for analysis model. The design phase produces a data design, an architectural design, an interface design, and a procedural design with the help of various methods and techniques such as transaction mapping and transform mapping for architectural design and structured programming, graphical design notation, tabular design notation, and program design.
language for procedural design.

Data flow diagrams model the transformations of data as it flows through a system and are the focus of SA/SD (Structured Analysis/Structured Design). A data flow diagram consists of processes, data flows, actors, and data stores. Starting from the top-level data flow diagram, SA/SD recursively divides complex processes into sub diagrams, until many small processes are left that are easy to implement. When the resulting processes are simple enough, the decomposition stops, and a process specification is written for each lowest-level process. Process specifications may be expressed with decision tables, pseudo code, or other techniques.

The data dictionary contains details missing from data flow diagrams. The data dictionary defines data flows and data stores and meaning of various names. State transition diagrams model time dependent behavior. Most state transition diagrams describe control processes or timing of function execution and data access triggered by events.

Entity relationship (ER) diagrams are used for showing the relationships between data base entities that indicates process specifications. Each ER data element corresponds to one data flow diagram data base as individual component. In early days, for the design phase, the most favorite technique was the structured programming, to produce procedural design. It was developed in languages such as Pascal, Ada and C. The traditional definition of structured programming refers to any software development technique that includes structured design and results in the development of a structured program. Structured programming [VIJ2014] allows programs to be broken down into functions or procedures, which is written with detailed knowledge of the inner workings of other blocks, thus using a top-down design approach or step wise refinement [HUI1999]. Large scale systems, built using this approach, are often deployed on only mainframes and minis. They feature as mainframe-based or other non-relational database systems. Therefore, both feeling the hard competition, and to improve software development was the reason for moving into object-oriented approach in industry [HER1999].
1.2.2 Object-Oriented Approach

The steps of software development mentioned above are common for all software engineering approaches. Therefore, analysis and design phases are inevitable for object-oriented approach, as well. In this approach, design is divided into four different steps as illustrated in Table 1.1. [VIJ2014] [HIL2002].

**Table 1.1 Analysis and design phases for object-oriented approach**

<table>
<thead>
<tr>
<th>Phase</th>
<th>Techniques</th>
<th>Key Deliverables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis</td>
<td>Collaboration Diagrams Class and Object Models</td>
<td>Analysis Models</td>
</tr>
<tr>
<td></td>
<td>Analysis Modeling</td>
<td></td>
</tr>
<tr>
<td>System Design</td>
<td>Deployment Modeling Component Modeling Package</td>
<td>Overview design and implementation architecture</td>
</tr>
<tr>
<td></td>
<td>Modeling</td>
<td></td>
</tr>
<tr>
<td>Class Design</td>
<td>Class and Object Modeling Interaction Modeling</td>
<td>Design Models</td>
</tr>
<tr>
<td></td>
<td>State Modeling Design Patterns</td>
<td></td>
</tr>
<tr>
<td>Interface Design</td>
<td>Class and Object Modeling Interaction Modeling</td>
<td>Design Models with interface specification</td>
</tr>
<tr>
<td></td>
<td>State Modeling Design Patterns</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Prototyping Modeling</td>
<td></td>
</tr>
<tr>
<td>Data Management</td>
<td>Class and Object Modeling Interaction Modeling</td>
<td>Design Models with database specification</td>
</tr>
<tr>
<td>Design</td>
<td>State Modeling Design Patterns</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Modeling Package Modeling</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Design Patterns</td>
<td></td>
</tr>
</tbody>
</table>

Object-oriented approach implements a way for implementing real-world problems to abstractions. It is an obvious human nature that wants to transform the development of a large and complex system into the development of a set of less complicated sub-systems. Object-oriented approach offers conceptual structures that support this sub-division and try to provide a mechanism to support the reuse of program code, design, and analysis models. This approach uses classes and objects as the main constructs from analysis to implementation. It normally involves using an object oriented language such as C++ or Java that has a feature of encapsulation, inheritance and polymorphism and the ability for objects to invoke each other. The invoking by another object is an important constraint
when it is applied to distributed systems. [HER1999]. UML (Unified Modeling Language) contains a number of concepts that are used to describe systems and the ways in which the systems can be broken down and modeled. The UML Specification defines the terms class and object as follows [HIL2002]. Class is a ‘Definition of objects that have the same attributes, operations, methods and relationships.’ Moreover, the purpose of a class is to declare a collection of methods, operations and attributes that describe the structure and behavior of objects. An object is an instance from a class. It is structured and behaves according to its defined class. Interface is defined as a group of externally visible operations. The interface contains no internal structure; it has no attributes, no associations, and only abstract operations.

Vijay and Devender Kumar [VIJ2014] explained Object-orientation (1) Encapsulation, which is also known as information hiding, provides the internal implementation of the object without requiring any change to the application that uses it. (2) Inheritance is the characteristics of the derived class to inherit some of its properties or methods from a parent. (3) Polymorphism is producing different results but it has same name of method. In this approach de facto standard notation [HIL2002], UML, reveals analysis and design phases of software development. Use cases specify the functionality that the software system will offer from the user’s perspective. They are used to document the scope of the system and the developer’s understanding of what it is that the users require. Classes might interact to deliver the functionality of the use case and the set of classes is known as collaboration. Collaborations can also be represented in various ways that reveal their internal details. The collaboration diagram is probably the most useful one. In addition to collaboration diagram, class diagram also represents these collaborations in detail. The class diagram is fundamental to object-oriented analysis. Through successive evolution, it provides flexible systems architecture and a low-level basis for the allocation of data and method behavior to individual classes object instances.

A sequence diagram is a sequence diagram to show an interaction between tasks drawn in a time sequence. Sequence diagrams can be drawn at different levels of development life cycle detail and to meet different purposes
at several stages. In object-orientation some algorithmic approaches are used, that is, Pseudo-code, and Activity diagrams. The UML specification defines the state as a condition during the life of an object or an interaction during which it satisfies some condition, performs some action, or waits for some events. State charts describe that apparently.

Class Responsibility Collaboration (CRC) cards methodology provide an effective method for exploring the possible ways of allocating responsibilities to classes and instruction to fulfill the responsibilities. CRC cards can be used at several different stages of a project different purpose. The object-oriented approach allows development-time with reuse of same class[HER1999]. When it was compared to previous approaches then it was found that it enhances developer’s ability to design and coding of software. However, this level of reuse has clearly fallen short of addressing the needs of large-scale development. The object-oriented approach has facilitated development of large-scale projects, but it has been mainly limited to the use of one technology on one platform. It has not really developed technologies and models for interoperability, but rather has been mostly focused on the development of one single system. In the 80s, neither interoperability nor portability was a major issue.

![Diagram showing developer's and user's points of view, object-oriented vs monolithic application, and traditional system](image)

**Figure 1.2 Object-Oriented Applications [HER1999]**

The need for open systems was already there, but the technology to resolve
the issues was not. This made it difficult to address portability and interoperability [HER1999]. Besides the insufficiencies of two features of object-oriented, reusability and interoperability, their payloads are described above. The reason for these shortcomings can be explained as the object-oriented approach changed the way, applications were built, but it did not change the nature of the applications themselves. This is illustrated in Figure 1.2. When software is developed using object-oriented approach then end user would get a monolithic application. In brief, it can be stated that the object-oriented approach is applied for the help of the functional developer, not for the end user [HER1999].

1.2.3 Component-Based Approach

This approach is expected to make the development and maintenance of software systems easy. The Gartner Group, for example, estimated that by 2003 that 70% of new applications will be prepared as a combination of pre-assembled and newly created components integrated to form complex business systems [VIJ2014]. The resulting increase in reuse should dramatically improve time-to-market [JJJ1998], software lifecycle costs, and quality [COL2012]. Basic concepts, promises, modeling languages that help for analysis and design and this approach and its seed, CBD, are taken from previous approaches and intermediary approaches such as distributed objects and distributed systems.

Figure 1.3 CBD maturity phases [HER1999]

Figure 1.3 depicts the transformation that occurs after object-oriented
approach. The distributed object approach extends the object-oriented approach with the ability to invoke remote objects using an object request. The distributed system approach means a development approach for building systems that are distributed, are often multi-tier [HER1999].

Most of the company trying to develop their software by component-based development approach while developing application for distributed system development. While this can deliver significant benefits, such as allowing the technical bridging of heterogeneous systems. It does not decrease the cost of development. It is slightly addressed by using object-oriented techniques but not enough to make a big difference. Here, now distributed component approach is at the full swing in industry and getting more benefit out of it. These industries have accepted new technology such as Enterprise Java Beans to reduce cost and development time for distributed systems. It solves the issue of interoperability of distributed systems and the integration of such component that is developed at the remote area [HER1999]. Despite the industrial revolution by the new technology i.e. component-based technology introduced abstraction and lower-level mechanisms that incorporated into a comprehensive software engineering process [DOG2003].

1.3 HISTORICAL DEVELOPMENT OF SOFTWARE COMPONENT BASED SYSTEM

The idea that software should be modularized and be available in component form became prominent in 1969 [MCL1969]. The conferences were organized to manage widely spread software crisis in software industry. As subsequent development of pipes and filters into the UNIX operating system was the first implementation of this idea. [RAI2011] defined the modern concept of a software component and called them Software ICs and set out to a new start of an infrastructure and market for concepts of developing independent these components by inventing the C ++ programming language and summarized this view in concept of Object-Oriented Programming[BOS2000]. [HOF2000. Component Based Software Engineering (CBSE) has already become widely adopted software development paradigm. Component-based approach helps to deal with the growing complexity of software, enables reuse of already implemented software parts, facilitates the independent development, etc.
This approach is practice oriented and only thus only a limited amount of work in theoretical available in the literature [SZY1998], [CRN2002].

Land R. [LAN2002] explained that the research areas of CBSE [and of intersection of software architectures. The software architecture is the organized structure of software that consists of components whereas the CBSE deals in components and its behavior and the technology required for their integration. One of the worldwide-known publications on component-based software [Szy1998]. This book provides an extended rationale for introducing the component-based approach into the software development field. It demonstrates the differences between obsolete object-oriented and component-oriented development paradigms and explains why the former does not completely meet the requirements of the modern software market.

Moreover, it provides necessary information for making easy step from OOP to CBSE. All essential definitions of the basic entities of the component-based approach e.g. component interface etc., are discussed in [SZY1998]. The author stresses the multi-functional nature of a component: a unit of composition, unit of independent deployment, unit of encapsulation etc. A detailed taxonomy on the existing components models (COM, CORBA, and JavaBeans) was presented explaining their basic principles, rules, advantages, and disadvantages. Additionally, the book catalogs a number of valuable practical guidelines for the development of components, component frameworks, and component architectures. Finally, the future of components was speculated: technical tensions, market evolution, and software developer evolution. Another extensive publication on component-based theory. The Leavens G. T., Sitaraman M., [LEA2000] present the underlying theory for multiple component-oriented solutions. Further, they focus mostly on two main issues: a) description syntax for components and their interconnection and b) semantics for component behavior and interaction. This book contains only high-level abstractions of component-based systems and does not concentrate on particular component technology.

A good collection of the foundations in CBSE is presented by Bachman F. et al. [BAC2000]. The report highlights the main concepts, such as component, interface, component model, component framework, etc. Besides the
foundations, the authors also sketch a vision for necessary future research in the field, namely, derivation of the properties of component compositions from the properties of components. There are two other interesting collections of articles in component-based software engineering field [CRN2002]. The first collection Heineman G. T. and W. T. Councill [HEI2001] presents various aspects of component-based software engineering. It starts with definitions of components and different views of component technologies. A component infrastructure is responsible for the operational context of components and for supplementary units that ensure the proper functioning of the components (with a certain quality level). Component architecture usually integrates a number of components, responsible for certain tasks, using pre-defined rules and constraints. This also contains a number of overviews on component technologies like COM+, EJB, and CORBA. The use and status of CBSE in different contexts of practice were illustrated by taking the different cases. The second collection of Crnkovic I. and M. Larsson [CRN2002] is mostly concerned with the emergent quality of the component-based software. This provides a detailed description of CBSE discipline in terms of challenges, objectives, requirements, concepts, etc. Is also contains an overview of modern component technologies. This describes the specification and verification of component semantics and component properties (both static and dynamic). Closely related to this part is a part about the integration of components and prediction of the properties of component compositions. The first one explains the principles and process of construction of product line architectures from components, and takes the Koala component model as an example [OMM2000]. The second one presents a detailed overview of all significant issues for designing of the real-time systems from components i.e. scheduling, verification, and the third one described the main aspects of software development for embedded systems and illustrates the basic principles via a case study by Estublier J. and Favre J. M [EST2002]. The aspects of interface and assembling of components were identified and the basic blocks of component models were developed in form of general component model.

The software components are used in two different contexts (i) using
components as parts to design a single executable (ii) each executable is treated as a component in a distributed environment. The components collaborate with each other using internet or intranet communication protocols for IPC (Inter Process Communications). IBM led the path with their System Object Model (SOM) in the early 1990s and after some time the Microsoft started his work on the way for deployment of component software by designing the first OLE COM [RAP2008].

1.4 SOFTWARE COMPONENTS UNIT IN COMPONENT BASED SOFTWARE ENGINEERING

1.4.1 Definition of Components

An independent individual software component is a software package that may have a separate web service, a web resource or a module that is a combinations and integration of related functions. All of the functions of system are placed into separate components so that all of the functions can be differentiated. On the basis of this logic the components are called modular and cohesive. As pre Bill Council and George T. Heinemann, the software component can be defined as “A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties [CLE2004]”

Now here are some few basic ideas and concepts regarding software components.

(1) It is a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well defined architecture [PHI1998].

(2) It is a dynamic bind able package of one or more programs managed as a unit and accessed through documented interfaces that can be discovered in run time [WOJ1998]

(3) Software component is a unit of composition with contractually specified and explicit context dependencies only [WOJ1998]
Business component is the software implementation of an autonomous business concept or business process [WOJ1998].

In addition to COTS components, the CBSE process yields qualified components. The software engineers ensures that not only functionality, but performance, reliability, usability, and other quality factors to meet out the requirements of the system or product to be built.

Adapted components are developed in such ways that are flexible for modification.

Assembled components are integrated into an architectural style and interconnected with an appropriate infrastructure that allows the components to be coordinated and managed effectively [KIR2015].

Updated components replace existing software as new versions of components become available.

In most engineering disciplines, systems are designed by composing existing components that are developed by a different developer or vendor. Initially, software engineering has been doing system development from scratch or new but it is now recognized for better software, quick availability and at lower cost. To achieve this goal, a design process that is based on principle of systematic reuse is adopted. So, typically, components are rather small but independent parts of a system. Components were considered as runtime entities [DEB2010]. They exist while the system is running. Components are not only used as design entities like classes in object oriented language. Components provide a service without regard to where the component is executing or independent of its programming language. A component is an independent executable application that can be made up of one or more executable objects. The component interface is published and standard. All interactions are done through this published interface. Components can have one function to entire application systems. Components communicate with each other via interfaces in the system. A component with standard interface offers services to the rest of the system and other components utilize these services and start own services. This interface is so designed that the client
does not need to know about the inner workings of the component and its implementation while using it in the system.

1.4.2 Characteristics of Software Component

Software components often take the form of objects or collections of objects in some binary or textual form adhering to some Interface Description Language (IDL). The component exists autonomously from other components in a computer. In other words, a component acts without changing its source code. Although, source code of component change based on the behavior of application and extensibility of component. One approach to creating component-based software using object-oriented programming is interface-based programming. However interface based programming does not inherently support distributed systems and most of the application of computer systems are inherently distributed now a days. Interface-based programming in the OOP approach may be extended to distributed systems with distributed component object models. The encapsulation principle of the components makes it popular among software developer. Another important attribute of components is the integration or detachability, so that a component can replace another component at design time or run-time. If the successor component fulfills the requirements of the initial component then components can be replaced with either an updated version or an alternative without stopping the service of the system in which the component operates. As a general rule of thumb for engineers substituting components, the new component can immediately replace old component if new component provides at least what old component provided. Software components often take the form of objects or group of objects in some binary or textual form, so that the component may exist autonomously from other components in software. In other words, a component can be used in system without changing its source code. Although, the behavior of the component’s source code may change depending on the environment. The components can be used on network or can be delivered on the network. Reusability is an important main feature of a software component while software development. Programmers should design and implement software components in such a way that it can be fitted in any type of system and other programs may use
them again. It takes significant effort and awareness to write a software component that is effectively reusable [BBA2010]. The component needs to be (1) fully documented (2) thoroughly tested (3) input-validity checking (4) proper exception handling.

![Component Diagram](image)

**Figure 1.4: Component Diagram**

The UML illustrations represent provided interfaces by a symbol attached to the outer edge of the component. However, when a component needs to use service of another component in order to function, it adopts an interface that specifies the services that it needs.

### 1.4.2.1 Encapsulation

Encapsulation is the packing of data and functions into a single component [DAO2015]. The features of encapsulation are supported using classes in most object oriented programming languages. It hides the properties in form of data and method in a code of object. In programming languages, encapsulation is used to refer to Pierce and Benjamin [PIE2002] that a language mechanism for restricting access to some of the object's components. In general, encapsulation is one of the four fundamentals of component encapsulation refers to the bundling of data with the methods that operate on that given data. Encapsulation is used to hide the values or state of a structured data object inside a class and preventing unauthorized user to direct access to them. Publicly accessible and protected accessible methods are generally provided in the class to access the values and other client classes can call these methods to get the values and modify the values of that the object.

### 1.4.2.2 Reusability

Systems are designed by composing existing components that have been used in other systems. Initially, software engineering has been doing system
development from scratch or new but it is now recognized that to achieve better software, more quickly and at lower cost. There is a need to adopt a design process that is based on principle of systematic reuse. So, typically, components are rather small but independent parts of a system. The development processes of component-based systems are separated from development processes of the components [SU2012]. The component is already developed and possibly can be used in other products when the system development process starts, a search and selection of proper components. The system development the emphasis will be on finding the proper components and verifying them, and for the component development, design for reuse will be the main concern [DAO2015]. The requirements and business ideas in these two cases basically follows different approaches] that Components are built, will be reused in many applications [HIL2002. Crnkovic, I. System [CRN2002b] focused on development with components identification of reusable entities and relations between them beginning from the system requirements and from the availability of components.

1.4.2.3 Productivity

When reusable components are applied throughout the software process, less time is spent creating the requirement plans, designing models, writing documents, code, and data that are required to create a deliverable system. It follows that the same level of functionality is delivered to the customer with less input effort. Hence, productivity is improved by reusing the old component and just assembling the available components. Although percentage productivity improvement reports are notoriously difficult to interpret, it appears that 30 to 50 percent reuse can result in productivity improvements in the 25–40 percent range [NPN2009].

1.4.2.4 Reduced Delivery Time

Component based approach is used by the software developer in industry to develop a large and complicated software systems using off the shelf component so that the time to build the software is minimum. The time saving by using this approach can be compared and analyzed using developing independent function point from scratch. In a non component- based
approach the process would continue with the unit design, implementation and test. Instead of performing this activity, the component based approach will consume time and efforts in selecting the appropriate components, writing only interface code and integrating them in the system [DAO2015]. The use of reuse of Component depends on the characteristics and behavior of components and its surrounding environment where this component can be reused. Component repository should be designed in such a way that it can store more software components and it must have an interface or catalog for retrieving them from heavy repository. The retrieving of component may be easy if all the information about the component including its signature is stored with repository. An inventory management system and library management system may be developed for handling the repository. An online software component retrieval system that enables a remote application to search and select components and services from the remote server is used for web components. Another issue in this approach is assembling the components in architecture. For this, there are CBSE tools available that support the integration of reused components into a faster design or implementation. By applying, the time to delivery is reduced to a margin level.

1.4.2.5 Reduced Complexity

Component based approach is used to design highly complex software systems using market available component so that the complexity in designing the software is reduced drastically. The simplicity of architecture of software system on the number of function point or input or output values [VBS2014]. The components are designed on the basis of a particular component model and implemented in different technology. It may be possible that one can use components in place of other components of different reliability implemented in different component technologies [IVI2006]. But actually, it is not easy to achieve interoperability between these different component models. Particular component model requires a particular architectural framework and the application is supposed to use this framework. The selection of components has great impact on architectural decisions. For example if the component model requires a client-server architecture style, it is obvious that the application will use that style. This will
put limitations on the system design and constraint of reusing the same component at every place even though they have same function defined into these. Similar to this, other properties of components may have a direct influence on the design like availability, flexibility and platform independency. For this reason the design process is tightly connected to the availability of the components. Software architecture represents design patterns that are composed of components, connections, and coordination among service providing components. In essence the architecture defines the design rules for all components, identifying modes of connection and coordination. In some cases, existing reusable components may be mismatched to the architecture’s design rules. These components must be adapted to meet the needs of the architecture or this may be discarded and replaced by other more suitable components for compatibility in this architecture.

1.4.2.6 Maintainability

The component based system is easy to maintain the system. The better or upgraded version can be replaced with the old component. Use and throw policy save time and new version of system can be prepared without stopping the service of all the parts of system. The maintenance process includes some steps that are similar to the integration process [YUM2013]: A new or modified component may be deployed into the system at any time. In most of the cases an existing component will be replaced by a modified or a new version of the same component, just by installing into the system. When systems are implemented with COTS components, it is assumed that update will be complicated because the component is imposition of a third party organization that developed this component may be outside the immediate control [MRL1996]. The third party organization is in constant touch of software developer or client and a standard are followed by them.

1.4.2.7 Reliability of Software Component

Software Reliability is defined as the probability of the failure free software operation for a specified period of time in a specified environment [MRL1996]. Unreliability of any product comes due to the failures or presence of error in the system. Software does not “wear out” or “age”, as a mechanical or an
electronic system does. The unreliability of software is primarily due to bugs or design faults in the software [HAM2001]. Reliability is a probabilistic measure that assumes that the occurrence of failure of software is a random phenomenon. Some error or fault detection techniques may largely improve software reliability. In real situations, it is not possible to eliminate all the bugs in the software. However, by applying sound software engineering principles software reliability can be improved to a great extent. The application of systematic, disciplined, quantifiable approach to the development operation and maintenance of software will produce economical software that is more reliable and works efficiently on real situations. The reliability of system depends on the part of system i.e. component. The quality of the software can be improved by improving the quality of the component. The retesting of single component test becomes easy and as a result there is improvement in the quality of that component. The CBSE process yields qualified components. The software engineers ensures that not only functionality, but performance, reliability, usability, and other quality factors to meet out the requirements of the system or product to be built. Henry and Faller [HEN95] reported a 35 percent improvement in quality. Although anecdotal reports span a reasonably wide spectrum of quality improvement percentages, it is fair to state that reuse provides a nontrivial benefit in terms of the quality and reliability for delivered software.

1.4.2.8 Parallel and Distributed Development

The traditional process of project was assumed started when a software team has completed requirements for the system using conventional requirements methods [HAI2007]. In CBSE, an architectural frame work design is established. But rather than moving immediately into more detailed design tasks, the team examines requirements to determine what subset is directly replaceable for composition instead of starting the task from zero level. The important fact is the need of component adoption and testing before it can be integrated into the system. The process of component based development is different from the traditional system development process in life cycle model, requirement analysis and specification phase one important activity is to
analyze the possibility of realizing the solutions that will meet these requirements.

In a component-based approach, requirement specification is prepared whether these requirements can be fulfilled by available components [XZH2003]. This means that the requirements engineers must be aware of components that can possibly be used. Since it may not be appropriate components, are always available in the list of in house repository of developer. There is a risk that the new components have to implement. To pace with component-based approach and its utilization that is to negotiate the requirements and modify them as the maximum use of the existing components. System requirements and architecture define the components that will be required. Reusable components whether COTS or in house are normally identified by the characteristics of their interfaces. The interface does not provide a complete picture of the degree to which the component will fit the architecture and requirements. So the documentation of component is supplied by the vendor to developer.

1.4.2.9 Easy Software Testing

Component approach helps the system to find its defect readily by testing the components. The standard test and verification techniques are used for testing the specific components by the specialized engineer. The specific problem for component-based approach is to find the location of error, especially when components are of black box type and delivered from different vendors. During the testing of system, a component can exhibit an error, but the cause of the malfunction lies in another component. The documentation of interface plays an important role in checking the proper input and output from components. These interfaces enable a specification of input and output and checking the correctness of data. This means that there is a need of testing by developer even though the component is well tested by the vendor. The system must be verified either formally or by simulation or by actually testing. Architectural style again plays a key role in which software components are integrated to form a working system. By identifying connection and coordination mechanisms i.e. run-time properties of the design, the architecture dictates the composition of the end product. Software
component that is developed for reuse would be verified to be correct and should contain no defects. In reality, formal verification is not carried out in routine so defects occur. However, with each reuse, defects are found and eliminated, and a component’s quality improves as a result. With the time, the component becomes virtually defect free. Component-Based Usability Testing (CBUT) is a testing approach which aims at empirically testing the usability of an interaction component. The latter is defined as an elementary unit of an interactive system, on which behavior based evaluation, is possible. For this, a component needs to have an independent, user perceivable and controllable state, such as a radio button, a slider or a whole word processor application. The CBUT approach can be regarded as part of component-based software engineering branch of software engineering. CBUT is based on both software architectural views such as Model View Controller (MVC), Presentation Abstraction Control (PAC), ICON and CNUCE agent models that split up the software in parts. Processes that operate on higher level layers are more abstract and focus on a person’s main goal, such as writing an application letter to get a job. The Layered Protocol Theory (LPT), which is a special version of Perceptual Control Theory (PCT), brings these views together by suggesting that users interact with a system across several layers by sending messages [FAR1999]. Users interact with components on high layers by sending messages, such as pressing keys to components, invoking the commands of lower layers, which in turn relay a series of these messages into a single high level message to a component on a lower layer. Components operating on higher layers, communicate back to the user by sending messages to components operating on lower level layers. Whereas this layered-interaction model explains how the interaction is established, control loops explain the purpose of the interaction. LPT note down the purpose of the behavior of user as the attempt to control their perception. This means that users will only act if they perceive the component to be in an undesirable state. As interaction with components takes places on several layers, interacting with a single device can include several control loops. The amount of effort put into operating a control loop is seen as an indicator for the usability of an interaction component. CBUT can be categorized according to two testing paradigms, the Single Version Testing Paradigm (SVTP) and
the Multiple Versions Testing Paradigm (MVTP). In SVTP only one version of each interaction component in a system is tested. The focus is to identify interaction components that might reduce the overall usability of the system. SVTP is therefore suitable as part of a software-integration test. In MVTP on the other hand, multiple versions of a single component are tested while the remaining components in the system remain unchanged. The focus is on identifying the version with the highest usability of specific interaction component. MVTP therefore is suitable for component development and selection. Different CBUT methods have been proposed for SVTP and MVTP, which include measures based on recorded user interaction and questionnaires [FAR1999]. Whereas by the MVTP report of Brinkman [BRI2007] the recorded data can directly be interpreted by making a comparison between two versions of the interaction component, in SVTP log file analysis is more extensive as interaction with both higher and lower components must be considered [BRI2008]. Meta-analysis on the data from several lab experiments that used CBUT measures suggests that these measures can be statistically more powerful than overall (holistic) usability measures [BRI2009]. In a study conducted at Hewlett Packard, Lim [LIM94] reported that the defect rate for reused code is 0.9 defects per KLOC, while the rate for newly developed software is 4.1 defects per KLOC. For an application that was composed of 68 percent reused code, the defect rate was found equal to 2.0 defects per KLOC a 51% improvement from the expected rate over the structured application.

1.4.2.10 Customer Satisfaction

The customer satisfaction is a big issue in traditional developing system. The set of the requirements that cannot be changed or deleted then conventional or object-oriented software engineering methods are applied and designed or develop new components to meet the requirements. But for those requirements that are addressed with available components, a different set of software engineering activities commences: Component Qualification, Component Adaptation, Component Composition and Component Update. Component-based development model was launched to illustrate how a repository of reusable components can be integrated into a typical
evolutionary process model. In the CBSE process, the component is not stored but it also improves the quality of component as well as system. The components are designed to adapt architectural mismatches, assemble components into a selected architectural style, and updates components as requirements with the system change.

The process model for component-based software engineering emphasizes parallel activity in which domain engineering occurs concurrently with component-based development. Domain engineering performs the work required to establish a set of software components that can be reused by the software engineer. Domain engineering creates a model of the application domain that is used as a basis for analyzing user requirements in the software engineering. Finally, after reusable components have been purchased, selected from existing libraries, or constructed, they are made available to software engineers during component-based development. In Component based system, the prototype model solves the issue.

1.4.2.11 Easy Project Management

The delay of delivery and control the schedule of activity of project are main cause of software system failure. Component-based system, an ideal case is to build an application by direct integration of components, i.e. directly connecting components. The only left for developer is the adaptation of the components and may be implementation of new functions. In an ideal case the components are assumed already built and tested. However the component tests in isolation are not sufficient. Often design units will be implemented as assemblies of several components and these assemblies must be tested separately. The integration process includes integration of standard components that are assembled on a component framework and the application software is developed components for that particular job. The integration of a particular component into a system is called a component deployment. In difference to the entire system integration a component deployment is a mechanism for integration of components and it downloads and registering of the component. The project manager can easily monitor the progress the development of system easily.
1.4.2.12 Easy Cost Estimation

There is a need to understand what actually can be reused in a software engineering context and then what the costs associated with reuse really are. As a consequence, it is possible to develop a cost versus benefit analysis for component reuse. The net cost savings for reuse are estimated by projecting the cost of the project as if it were developed from scratch. The actual cost of the software as delivered is calculated by adding the cost of components to the cost of working scenario and cost of developing the interface. The cost of component is decreased with its multiple uses by the developer. After some time, the component cost is reduced to almost zero. Lesser time to build the software and one time implementation or buying of component make easy cost estimation [PKS2007b].

1.4.2.13 Flexible Modification

Adapted components are developed in such a way that is flexible for modification. Assembled components are integrated into an architectural style and interconnected with an appropriate infrastructure that allows the components to be coordinated and managed effectively [KWA2003]. The updated components replace existing software as new versions of components become available. In most engineering disciplines, systems are designed by composing existing components that have been used in other systems. Initially, software engineering has been doing system development from scratch or new but it is now recognized that to achieve better software, more quickly and at lower cost, there is a need to adopt a design process that is based on principle of systematic reuse. So, typically, components are rather small but independent parts of a system. But a large system as a whole can be seen as a component. The components are designed such as that that can be replaced with the new one. The code of component is not available to the developer even then software developed from the components can be easily modified.

1.5 SIMULATION

Computer simulation was developed hand-in-hand with the rapid growth of the computer, following its first large-scale deployment during the Manhattan
Project in World War II to model the process of nuclear detonation. Computer simulation is often used as an adjunct to, or substitution for, modeling systems for which simple closed form analytic solutions are not possible. There are many different types of computer simulation; the common feature they all share is the attempt to generate a sample of representative scenarios for a model in which a complete enumeration of all possible states of the model would be prohibitive or impossible. Computer models were initially used as a supplement for other arguments, but their use later became rather widespread. Simulation is usually understood as the process of generating reality [BDR1987, MOR1984].

In its prevailing sense simulation is concerned with the representation of systems by suitably defined models and observing the operation of such models under particular set of conditions. Thus technically speaking, the following definition of simulation can be adopted. Simulation is the numerical technique for conducting experiments on digital computer, which involves logical and mathematical relationships that interact to describe the behavior and the structure of a complex real world system over extended period of time.

Computer simulation is the discipline of designing a model of an actual or theoretical physical system, executing the model on a digital computer, and analyzing the execution output. The use of simulation is an activity that is as natural as a child who role plays. Children understand the world around them by simulating (with toys) most of their interactions with other people, animals and objects. As adults, one loses some of this childlike behavior but recapture it later on through computer simulation [CIS1995]. To understand reality and all of its complexity, one must build artificial objects and dynamically act out roles with them. Computer simulation is the electronic equivalent of this type of role playing and it serves to drive synthetic environments and virtual worlds. A computer simulation is an attempt to model a real-life or hypothetical situation on a computer so that it can be studied to see how the system works. By changing variables, predictions may be made about the behavior of the system.

Simulation is the imitation of the operation of a real-world process or system
over time. Simulation involves the generation of an artificial history of the system, and the observation of that artificial history to draw inferences concerning the operating characteristics of the real system that is represented. Simulation is an indispensable problem-solving methodology for the solution of many real-world problems. Simulation is used to describe and analyze the behavior of a system. Both existing and conceptual systems can be modeled with simulation.

The term simulation is used in different ways by different people. As used here, simulation is defined as the process of creating a model of an existing or proposed system in order to identify and understand those factors which control the system and to predict the future behavior of the system. Almost any system which can be quantitatively described using equations and rules can be simulated. Simulation is used to predict the way in which the system will evolve and respond to its surroundings, so that you can identify any necessary changes that will help make the system perform the way that you want it to.

1.5.1 Need of Simulation

A simulator is a collection of hardware and software systems which are used to mimic the behavior of some entity or phenomenon. Simulators may also be used to analyze and verify theoretical models which may be too difficult to grasp from a purely conceptual level. As such, simulators provide a crucial role in both industry and academia.

Simulation has always played some role in image processing, computer vision, and control system engineering. Yet, until now, there has been no simulation environment available that could support the complexity of complete perception and control systems, and provide the required realism at the same time. Therefore, developing a comprehensive simulation system with capabilities that are critical for solving the complicated design and testing problems that are typical in computer vision [YAL1990].

Simulation allows extended testing in arbitrary environments, under changing environmental conditions, and in extreme situations that may be rarely encountered in reality but may be crucial for system operation [SHE2004].
The simulations can be run unsupervised, at any time of day, under any weather conditions, and on several computers in parallel. The increased number of situations and test cases explored in this way should improve system reliability dramatically [GEN2005].

1. Adaptation and learning at all the system levels can be performed efficiently and autonomously. Large sets of training examples can be processed without human intervention in both supervised and unsupervised modes.

2. Realistic testing of individual modules in a complete system environment is possible at a very early phase, even when the real robot or other parts of the system are unavailable. The evaluation of system design alternatives is possible without actual deployment on the robot.

3. The costs for designing, building, and running the robot can be reduced. Simulation results will generally influence the robot's design, thus avoiding costly redesigns later in the process. Further, the transition between testing and deployment of the system can be streamlined [MAS1990].

4. Processing hardware requirements and real-time capability can be estimated much earlier in the design process. Usually, hardware performance requirements are difficult to estimate before the complete system is running. Real-time behavior can be simulated even when the required high-speed hardware is still unavailable during system development.

Despite the increasing recognition of simulators as a viable and necessary research tool, one must constantly be aware of the potential problems which simulators may introduce during simulations. Many of the problems are related to the computational limitations of existing hardware platforms. The advance development in the area of hardware and controlled software process will resolve this issue. Simulators must adapt to increase in system complexity by permitting users to simulate a system at several conceptual levels. Unfortunately, the design and implementation of simulators is almost as complex as the systems being simulated. As a result, there has been a
great effort by the software community to apply the latest advancements in software technology in an attempt to counteract this ever increasing complexity. Their efforts have lead to simulators which are easy to maintain and extend while at the same time preserving their relative efficiency [BBH1994].

1.5.2 Simulation versus Modeling

Traditionally, forming large models of systems has been via a mathematical model, which attempts to find analytical solutions to problems and thereby enable the prediction of the behavior of the system from a set of parameters and initial conditions.

Computer simulations might use some algorithms from purely mathematical models [DIE2000]. The computers can combine simulations with reality or actual events such as generating input responses to simulate test subjects who are no longer present. Whereas the missing test subjects are being modeled or simulated, the system they use could be the actual equipment, revealing performance limits or defects in long-term use by these simulated users.

Computer simulation is broader than computer modeling, which implies that all aspects are being modeled in the computer representation. However, computer simulation also includes generating inputs from simulated users to run actual computer software or equipment with only part of the system being modeled. An example is flight simulators which can run machines as well as actual flight software. Computer simulations are used in many fields, including science, technology, entertainment, and business planning and scheduling.

1.5.3 Types of Simulation

Simulation models can be classified as static or dynamic, deterministic or stochastic and continuous or discrete. A static simulation model represents a system, which does not change with time or represents the system at a particular point in time. Static simulation is also sometimes called Monte-Carlo simulation [BRI1994]. On the other hand, dynamic simulation models represent systems as they change over time.

From the viewpoint of simulation the systems are usually classified into two
categories:
1. Discrete Event Systems
2. Continuous Event Systems
3. Fixed Time Step Models
4. Event To Event Models
5. Monte Carlo Simulation
6. Stochastic Simulation

1.5.3.1 Discrete Event Simulation

It concerns modeling of a system as it evolves over time such that state variables change only at a countable number of points [FIS1973]. The systems in which the state changes abruptly at discrete points in time are called discrete Systems.

1.5.3.2 Continuous Event Simulation

It concerns the modeling over time of a system by a representation in which state variables change involve one or more differential equations that give relationship for the rate of change of state variables with respect to time [LPH2000]. Systems in which the state of the system changes continuously with time are called continuous systems. If the differential equations are simple, they can be solved analytically. If analytical solution is not possible then numerical methods such as Runge Kutta method can be used to solve the differential equation [BAN2005]. In simulating any dynamic system- continuous or discrete, there must be a mechanism for the flow of time. For continuous system, the time is incremented by small intervals for as long as is needed. In simulation of discrete system there are two fundamental different models for moving a system through time.

1.5.3.3 Fixed Time Step Models

In a Fixed time step model [NAY1966] a timer for clock is simulated by the computer. The clock is updated by a fixed time interval and the system is examined to see if any event has taken place. All events that take place
during this period are treated as if they occurred simultaneously at the tail end of interval.

1.5.3.4 Event to Event Models

In event to event model the computer advances time to the occurrence of the next event. It shifts from event to event the system state does not change in between. Only those points in time are kept track of when something of interest happens to the system.

1.5.3.5 Monte Carlo Simulation

Monte Carlo simulation is a computerized mathematical technique that allows people to account for risk in quantitative analysis and decision making. The technique is used by professionals in such widely disparate fields as finance, project management, energy, manufacturing, engineering, research and development, insurance, oil & gas, transportation, and the environment. A Monte Carlo method is a technique that involves using random numbers and probability to solve problems. The term Monte Carlo Method was used by S. Ulam and Nicholas Metropolis initially in reference to games of chance, a popular attraction in Monte Carlo, Monaco [HOF1998] [MET1949]. The technique was first used by scientists working on the atom bomb; it was named for Monte Carlo, the Monaco resort town renowned for its casinos. Since its introduction in World War II, Monte Carlo simulation has been used to model a variety of physical and conceptual systems. Monte Carlo simulation is a method for iteratively evaluating a deterministic model using sets of random numbers as inputs [HAL1970]. This method is often used when the model is complex, nonlinear, or involves more than just a couple uncertain parameters. A simulation can typically involve over 10,000 evaluations of the model, a task which in the past was only practical using super computers [WIT2004]. During a Monte Carlo simulation, values are sampled at random from the input probability distributions. Each set of samples is called iteration, and the resulting outcome from that sample is recorded. Monte Carlo simulation does this hundreds or thousands of times, and the result is a probability distribution of possible outcomes. In this way, Monte Carlo simulation provides a much more comprehensive view of what
may happen. It tells you not only what could happen, but how likely how it is to happen.

By using random inputs, you are essentially turning the deterministic model into a stochastic model. Simple uniform random numbers are used as the inputs to the model. In a deterministic simulation, a system is simulated under well determined conditions. This kind of simulation is useful to observe the behavior of system in certain particular cases, to discover errors in the design or in the implementations, to build examples, etc. In this kind of simulations, only one run is needed and there is no truly random variable involved. To see the behavior of the system one needs to track the output on a file and later to see and analyze it in a textual or in a graphical form.

1.5.3.6 Stochastic Simulation

In a Stochastic simulation, system performance is measured at random time [NAY1966]. This is useful to see if the system has good response time under average conditions, to compare different implementations of the same system, or totally different systems that have the same output. It is useful to classify the system being simulated into two separate categories depending upon the degree of randomness associated with the behavior of the system in its simulated environment. A system that relies heavily upon random behavior is referred to as a stochastic system. The results generated from a stochastic system are typically analyzed statistically in order to make conclusions regarding the behavior of the system.

1.5.4 Advantages of Simulation

The use of simulation is rapidly increasing. Due to advances in software, simulation is incorporating in daily operations on an increasingly regular basis. For most companies, the benefits of using simulation go beyond just providing a look into the future.

Following is a list of advantages of simulation

1. One of the primary advantages of simulators is that they are able to provide users with practical feedback when designing real world systems. This allows the designer to determine the correctness and efficiency of a design before the system is actually constructed.
Consequently, the user may explore the merits of alternative designs without actually physically building the systems. By investigating the effects of specific design decisions during the design phase rather than the construction phase, the overall cost of building the system diminishes significantly [BDR1987].

2. Another benefit of simulators is that they permit system designers to study a problem at several different levels of abstraction. By approaching a system at a higher level of abstraction, the designer is better able to understand the behavior and interactions of all the high level components within the system and is therefore better equipped to counteract the complexity of the overall system. This complexity may simply overwhelm the designer if the problem had been approached from a lower level. As the designer better understands the operation of the higher level components through the use of the simulator, the lower level components may then be designed and subsequently simulated for verification and performance evaluation.

7. By compressing or expanding time simulation allows you to speed up or slow down phenomena so that you can thoroughly investigate them. You can examine an entire shift in a matter of minutes if you desire, or you can spend two hours examining all the events that occurred during one minute of simulated activity.

8. Designing new systems and redesigning existing systems. Interacting with all those involved in a project during the problem-formulation stage gives you an idea of the scenarios that are of interest. Then you construct the model so that it answers questions pertaining to those scenarios. What if a person is removed from service for an extended period of time? What if demand for service increases by ten percent? What if....? The options are unlimited [MOR1984].

9. Simulators can be used as an effective means for teaching or demonstrating concepts to students. This is particularly true of simulators that make intelligent use of computer graphics and animation. Such simulators dynamically show the behavior and
relationship of all the simulated system's components, thereby providing the user with a meaningful understanding of the system's nature.

10. Simulation often goes hand in hand with visualization. The results of changes that a student puts into a model are directly shown on the screen. This generally appeals to students [ROS2002].

11. Simulation can be very purposive and for certain students very useful, such as students who need some insight before they are able to learn and understand a new concept.

12. By mimicking the behavior of the designs, the circuit simulator is able to provide the designer with information pertaining to the correctness and efficiency of alternate designs. After carefully weighing the ramifications of each design, the best circuit may then be fabricated.

13. Using simulation to present design changes creates an objective opinion. You avoid having inferences made when you approve or disapprove of designs because you simply select the designs and modifications that provided the most desirable results, whether it is increasing production or reducing the waiting time for service. In addition, it is much easier to accept reliable simulation results, which have been modeled, tested, validated, and visually represented, instead of one person's opinion of the results that will occur from a proposed design.

14. Production bottlenecks give manufacturers headaches. It is easy to forget that bottlenecks are an effect rather than a cause. However, by using simulation to perform bottleneck analysis, you can discover the cause of the delays in work-in-process, information, materials, or other processes [BUR2003].

1.5.5 Limitation of Simulation

Despite the advantages of simulation presented above, simulators, like most tools, do have their drawbacks. Many of these problems can be attributed to the computationally intensive processing required by some simulators.
The results of the simulation may not be readily available after the simulation has started an event that may occur instantaneously in the real world may actually take hours to mimic in a simulated environment [HAL1970]. The delays may be due to an exceedingly large number of entities being simulated or due to the complex interactions that occur between the entities within the system being simulated. Consequently, these simulators are restricted by limited hardware platforms which cannot meet the computational demands of the simulator.

1. To meet the computational complexity is to employ the hierarchical approach to design and simulation so as to permit the designer to operate at a higher level of design. However, this technique may introduce its own problems as well. By operating at too high an abstraction level, the designer may tend to oversimplify or even omit some of the lower level details of the system. If the level of abstraction is too high, then it may be impossible to actually build the device physically due to the lack of sufficiently detailed information within the design. Actual construction of the system will not be able to occur until the user provides low level information concerning the system's subcomponents. Simulation is today the most common form of verification. One disadvantage of simulation is the excessive number of tests needed for complete coverage. However, as will be shown, the number of tests may be substantially reduced if test case generation is combined with a structural analysis. The resulting set of test cases for exhaustive simulation may be smaller than exponential, which might make exhaustive simulation feasible.

1.5.6 Problem Solving using Simulation

The Problem Solving process consists of a sequence of sections that fit together depending on the type of problem to be solved. The process by which the simulation is performed remains constant. The following briefly describes the basic steps in the simulation process:

1.5.6.1 Identify the Problem

Enumerate problems with an existing system and produce requirements for a
1.5.6.2 Formulate the Problem

Select the bounds of the system, the problem or a part thereof, to be studied. Define overall objective of the study and a few specific issues to be addressed. Define performance measures - quantitative criteria on the basis of which different system configurations will be compared and ranked. Identify, briefly at this stage, the configurations of interest and formulate hypotheses about system performance [BRA1987]. Decide the time frame of the study, i.e., wills the model be used for a one-time decision (e.g., capital expenditure) or over a period of time on a regular basis (e.g., air traffic scheduling). Identify the end user of the simulation model, e.g., corporate management versus a production supervisor. Problems must be formulated as precisely as possible.

1.5.6.3 Collect and Process Real System Data

Collect data on system specifications (e.g., bandwidth for a communication network), input variables, as well as Altered System, performance of the existing system. Identify sources of randomness in the system, i.e., the stochastic input variables [MON1997]. Select an appropriate input probability distribution for each stochastic input variable and estimate corresponding parameter(s).

1.5.6.4 Formulate and Develop a Model

Develop schematics and network diagrams of the system. Translate these conceptual models to simulation software acceptable form. Verify that the simulation model executes as intended. Verification techniques include traces, varying input parameters over their acceptable range and checking the output, substituting constants for random variables and manually checking results and animation.

1.5.6.5 Validate the Model

Compare the model's performance under known conditions with the performance of the real system. Perform statistical inference tests and get the model examined by system experts. Assess the confidence that the end user
places on the model and address problems if any. This not only ensures that the model assumptions are correct, complete and consistent, but also enhances confidence in the model.

1.5.6.6. Document Model for Future Use

Document objectives, assumptions, output variable and input variables in detail.

1.5.6.7 Select Appropriate Experimental Design

Select a performance measure, a few input variables that are likely to influence it, and the levels of each input variable. Document the experimental design.

1.5.6.8 Establish Experimental Conditions for Runs

Address the question of obtaining accurate information and the most information from each run. Determine if the performance of system is stationary over time or non-stationary over time. Generally, in stationary systems, steady-state behavior of the response variable is of interest. Ascertain whether a terminating or a non terminating simulation run is appropriate. Select the run length and appropriate starting conditions e.g., empty and idle. Select the length of the warm-up period, if required. Decide the number of independent runs each run uses a different random number stream and the same starting conditions by considering output data sample size. Sample size must be large enough to take a case of all possibility and to provide the required confidence in the performance measure estimates [NAY1966]. Alternately, use common random numbers to compare alternative configurations by using a separate random number stream for each sampling process in a configuration. Identify output data most likely to be correlated.

1.6 SOFTWARE PROJECT MANAGEMENT

There is no doubt that organizations today face more aggressive competition than in the past and the business environment they operate in is a highly turbulent one. This scenario has increased the need for organizational accountability for the private and public sectors, leading to a greater focus and demand for operational effectiveness and efficiency. Effectiveness and
efficiency may be facilitated through the introduction of best practices that are able to optimize the management of organizational resources. It has been shown that operations and projects are dissimilar with each requiring different management techniques. Hence, in a project environment, project management can: (a) support the achievement of project and organizational goals; and (b) provide a greater assurance to stakeholders that resources are being managed effectively.

A reasonably detailed project management methodology, as compared to a loose methodology, improves productivity by 20 to 30 percent [JMV2005]. Furthermore, the use of a formalized project management structure to projects can facilitate:

1. The clarification of project scope
2. Agreement of objectives and goals
3. Identifying resources needed
4. Ensuring accountability for results and performance
5. Encouraging the project team to focus on the final benefits to be achieved.

Moreover, the research indicates that 85-90% of projects fail to deliver on time, on budget and to the quality of performance expected [DIE2000]. The major causes identified for this situation include:

1. Lack of a valid business case justifying the project;
2. Objectives not properly defined and agreed;
3. Lack of communication and stakeholder management;
4. Outcomes and/or benefits not properly defined in measurable terms;
5. Lack of quality control;
6. Poor estimation of duration and cost;
7. Inadequate definition and acceptance of roles (governance);
8. Insufficient planning and coordination of resources.

It should be emphasized that the causes for the failure to deliver on time, on budget and to the quality of performance expected could be addressed by the application of project management practices [TAT2007]. Although, significant progress has been made in software development methodologies, software
project failures continue to exist. Software development remains a risky undertaking where decisions must be made without complete information. Another approach to risk management is to concentrate on those making decisions as agents of an organization rather than just the methods they use. One proposes that the behavior of decision makers affected by risk propensity and motivation is critical to the outcome of a software project.

In 1998, Peat Marwick found that about 35 percent of 600 firms surveyed had at least one runaway software project. It is worth noting that it is estimated that 80 percent of failures are never reported. Of all failures, 25% were due to hardware faults [CHA2005], overload, or natural forces, which are not of concern to us; the remaining 75% are software failures, including those that allowed [WU4] attacks from hackers. It is possible for data to be lost due to software bugs, or even just poor software design.

For example, a program might have a problem where it crashes upon saving a file. Software failure will cause more inconvenience. Software errors have even caused human fatalities. The causes have ranged from poorly designed user interfaces to direct programming errors.

Here are the few examples of software failures which caused a great loss in terms of money and life.

1. A Mars orbiter had been working properly for over nine years when some new software was uploaded to the spacecraft. Unfortunately, a coding error caused it to overwrite two memory addresses and stopped the solar arrays from turning to catch the sun’s energy. The safety device put the orbiter into a safe mode from which it could be recovered [CHA2007].

2. The pilot of KAL 007 mistakenly set his direction by a compass setting rather than inertial guidance. Gradually, so gradually that he did not notice, his plane drifted over Russian airspace instead of flying to Korea. When Russian fighter jets intercepted him and tried to contact him he took evasive maneuvers, presumably still believing he was in international airspace, and was shot down, killing all aboard [PEA1987].

3. A few years ago, the high tech guided missile cruiser
USS Yorktown suddenly lost power to its engines and much else aboard the ship and according to some press accounts was adrift at sea for three hours. The alleged cause is that an engineer was testing fuel tank levels, as a precaution, and mistakenly tried to divide by zero. The software program that he was using on Windows NT should have refused to accept his command but did not, and the program shut down the system [SME2005, SLA1998].

4. The pilot of an Airbus jetliner mistakenly added an extra zero to the glide rate he called for and the airplane suddenly dove so quickly that he could not avoid crashing into a mountain and killing all aboard [SMI2000].

5. A US soldier in Afghanistan was about to call in an air strike on a distant target with his handheld GPS Receiver communicator. After setting the coordinates for the strike, he got a warning message that the battery power was low. Presumably figuring that it might be too low to call in the strike he changed the battery, and then pressed fire. He did not realize that when the battery was changed, the coordinates reverted to his own position rather than the one he had entered. He and most of his platoon were killed [JAC2004].

6. In 2004 the air traffic control center in Palmdale California failed, disrupting 800 flights and causing at least five near midair collisions. There was a bug in the software, and after running for months, a countdown timer reached zero and shut down the system.

All these above mentioned cases have resulted in failures due to the software faults, which could not be traced well in time. This emphasizes the importance of better software project management.

1.7 LIMITATION AND DRAWBACK OF CBD FRAMEWORK.

1.7.1 Assumption of Client Environment.

It is very difficult to build the environment that is fitted to component. The component is developed at different place from the actual software developer, and then the integration with client system appears.
1.7.2 Standard and License

The standards and license are needed for the use of these components. But among the three standards available, only CORBA is language independent. Hence the comparison is not viable among the standards.

1.7.3 Heavy Repository

In Component based development system, the components are developed by the third party by keeping the view in mind that the different environment is available for client use. With the time, the repository of software becomes heavier.

1.7.4 Lack in Documentation

In Component based development system, the use or reuse, the testing or regression testing of software component require full documentation otherwise, the functionality of that cannot be used in complete.

1.8 TYPES OF SOFTWARE COMPONENT

1.8.1 Third Party Software Component

In computer programming, a third-party software component is a reusable software component developed to be either freely distributed or sold by an entity other than the original vendor of the development platform. The third-party software component market thrives because many programmers believe that component-oriented development improves the efficiency and the quality of developing custom applications. Common third-party software includes macros, bots, software or scripts to be run as add-on for popular developing software. Enterprise JavaBeans technology can be easily discovered. Enterprise JavaBeans [MON2001] is chosen and all information is gathered and well coordinated. Firstly Sun Microsystems’ definition of Enterprise JavaBeans is. The Enterprise JavaBeans architecture is component architecture for the development and deployment of component-based distributed business applications.

Applications written using the Enterprise JavaBeans architecture is scalable, transactional, and multi-user secure. These applications may be written once and then deployed on any server platform that supports the Enterprise
JavaBeans specification.

The original JavaBeans is also a component model, but it is not a server-side component model like EJB. In fact, other than sharing the name “JavaBeans,” these two component models are completely unrelated. In the past, a lot of the literature referred to EJB as an extension of the original JavaBeans, but this is a misrepresentation. The two APIs serve very different purposes, and EJB does not extend or use the original JavaBeans component model. JavaBeans is intended to be used for intra-process purposes, while EJB is designed for inter-process components. In other words, the original JavaBeans was not intended for distributed components. JavaBeans can be used to solve a variety of problems, but it is primarily used to build clients by assembling visual Graphical User Interface (GUI) and non-visual widgets [MON2001].

1.8.2 Web Components

Web Components are a set of standards currently being produced by Google engineers as a W3C specification that allow for the creation of reusable widgets or components in web documents and web applications. The intention behind them is to bring component-based software engineering to the World Wide Web. The components model allows for encapsulation and interoperability of individual HTML elements. Support for Web Components is present in some Web Kit-based browsers like Google Chrome and Opera and is in Mozilla Firefox (requires a manual configuration change). Microsoft’s Internet Explorer has not implemented any Web Components specifications yet [FAR1999]. Backwards compatibility with older browsers is implemented using JavaScript based polyfills. Web Components consist of 4 main elements which can be used separately or all together.

1.8.3 Custom Components

Most software developers work with open-source programs to meet their client’s requirements. It all starts with making contact with clients. A lot of techies hate having to market them. Thus, they either work for a larger agency or rely on their network of contacts. Once prospective clients are interested, it’s time to show them your custom software proposal example. Proposal
writing can be time-consuming but most developers rely on a template. There are times though when the template isn’t enough. This is particularly true if you are working with clients who want the individual components of the work broken down. Some companies also need you to create highly customized paperwork that suits their needs exactly. However, it is a good practice to have XI developments in a custom component, other people’s SWCVs (i.e. SAP’s) cannot be modified because there I no direct control over them.

1.9 SERVICE COMPONENT ARCHITECTURE

Service Component Architecture (SCA) is a software technology created by major software vendors, including IBM, Oracle and TIBCO. SCA provides a model for composing applications that follow service-oriented architecture principles. The technology encompasses a wide range of disparate technologies and as such is specified in various independent specifications in order to maintain programming language and application environment neutrality. The goal of SCA is simply stated that is to reduce IT complexity through a standardized framework for assembling disparate enterprise Service Oriented Architecture (SOA) components into a higher-level composite, thus simplifying development, deployment and management of enterprise applications. Standards such as SOA, Java Connector Architecture (JCA), Business Process Execution Language (BPEL), and Web services have gone a long way towards standardizing some fundamental aspects of application development. Any functionality built with them becomes a component of larger IT sub-systems, systems, and architectures [PAT2009].

Many times it uses an Enterprise service bus (ESB). An enterprise service bus (ESB) is a software architecture model used for designing and implementing communication between mutually interacting software applications in a service-oriented architecture (SOA). As software architectural model for distributed computing, it is a specialty variant of the more general client server model and promotes agility and flexibility with regard to communication between applications. Its primary use is in Enterprise Application Integration (EAI) of heterogeneous and complex landscapes. The SCA Assembly Model consists of a series of artifacts, which are defined by elements contained in XML files. An SCA runtime may have
other non-standard representations of the artifacts represented by these XML files, and may allow for the configuration of systems to be modified dynamically. However, the XML files define the portable representation of the SCA artifacts. The basic artifact is the Composite, which is the unit of deployment for SCA and which holds Services which can be accessed remotely. A composite contains one or more Components, which contain the business function provided by the module. Components offer their function as services, which can either be used by other components within the same module or which can be made available for use outside the module through Entry Points. Components may also depend on services provided by other components — these dependencies are called References. References can either be linked to services provided by other components in the same module, or references can be linked to services provided outside the module, which can be provided by other modules. References to services provided outside the module, including services provided by other modules, are defined by External Services in the module. Also contained in the module are the linkages between references and services, represented by Wires. A Component consists of a configured Implementation, where an implementation is the piece of program code implementing business functions. The component configures the implementation with specific values for settable Properties declared by the implementation. The component can also configure the implementation with wiring of references declared by the implementation to specific target services. Composites are deployed within an SCA System.

An SCA System represents a set of services providing an area of business functionality that is controlled by a single organization. As an example, for the accounts department in a business, the SCA System might cover all financial-related functions, and it might contain a series of modules dealing with specific areas of accounting, with one for customer accounts and dealing with accounts payable. To help build and configure the SCA System, Composites can be used as component implementations, in the same way as Java classes or BPEL processes. In other words, SCA allows a hierarchy of composites that is arbitrarily deep - such a nested model is termed recursive.
The capture and expression of non-functional requirements, such as security, is an important aspect of service definition, and has an impact on SCA throughout the lifecycle of components and compositions. SCA provides the Policy Framework to support specification of constraints, capabilities and Quality of Service (QoS) expectations, from component design through to concrete deployment [GEO2008].

1.10 RESEARCH OBJECTIVES

Following are the main research objectives identified for use of simulation techniques in CBSE:

1. To identify the areas of component based software engineering where simulation techniques can be applied.

2. To develop simulator/simulation model for estimating the reliability of a component based software.

3. To develop simulator for identifying the components in component based software that need to be tested more rigorously as compared to other components of the same system.

4. To develop model for searching suitable components from component libraries so that components can perform better in an application.

5. To develop simulator for identifying components in a system for the optimum distribution of efforts.

6. To develop simulator that can be used for identifying manpower requirements for implementation of user requirements and removal of defects from components.

1.11 RESEARCH TOOL

1.11.1 Simulator for Planning Software Project Management

Effective project management is difficult and complex. Tasks of various types must be assigned to resources with different characteristics, taking complex dependencies, constraints and uncertainties into account, attempting to meet goals related to costs and time. Keeping this in view an attempt has been made to develop a Stochastic Simulator which helps in decision making to
identify the critical activities that needs to be given due priorities during the development of Software Project in OCTAVE3.6.1.

The proposed stochastic simulator will be an asset in effort estimation to Project/Module Leaders to complete the assigned Software Development Task in stipulated time to enable the development house to release the product software in time to achieve the specific target by the consultancy unit. The Project Life Cycle refers to a logical sequence of activities to accomplish the project’s goals or objectives. There are four major activities for project management process, requires specific time period for their completion. The simulator has been designed to evaluate time estimation for various stages of project management process using normally distributed. This simulator will be an asset to affordably keep track of the time of phases during the process of project management.

1.11.2 Reliability of Component Based System

One of the metrics is not everywhere useful for the automation engineers for the quality of software. It is not easy for software developer to certify the goodness of any project and what is the actual accurate cost in developing the software. Even the monitoring system has no control on the scheduling of their project. Measurements such as quantity of defects rework in hours, retesting in hours, number of scope changes, etc., represent the quality of the work performed. By themselves they are not really interesting; they have been put in a more measurable unit such as a percentage ratio:

1. Quantity of defects per 1000 lines of code (or module configured)
2. Rework vs. work as a %
3. Retesting vs. testing as a %
4. Cost associated with scope changes vs. total cost per type of changes [ART1985].

Even bug fixes may be a reason for more software failures, if the bug fix induces other defects into software. For reliability upgrades [XZH2003], it is possible to incur a drop in software failure rate, if the goal of the upgrade is enhancing software reliability, such as a redesign or implementation of some modules using better engineering approaches, such as clean-room method.
Software failures are characterized by keeping track of software defect density in the system [VBS2014]. This number can be obtained by keeping track of historical software defect history. Defect density will depend on the following factors:

1. Software process used to develop the design and code (use of peer level design/code reviews, unit testing)
2. Complexity of the software
3. Size of the software
4. Experience of the team developing the software
5. Percentage of code reused from a previous stable project
6. Rigor and depth of testing before product is shipped.

Defect density is typically measured in number of defects per thousand lines of code (defects/KLOC). The reliability will be. Therefore, a developer should attempt to generate as few faults as possible (fault avoidance), should consider techniques to withstand faults (fault tolerance), and ensure that software is readily repairable when a fault is found (fault removal) [XZH2003].

To develop a system that is reliable, apart from performing any reliability analysis, a developer should consider those three elements. In the design process a general model can be used MATLAB R2008a. In a sequential design process a series of steps are followed consisting of a specification phase followed by a realization phase. Realizations become less abstract as the system is decomposed into more detailed specifications.

Since faults are the underlying cause of failures in software, controlling the number of faults introduced in each development step and the number of faults that propagate undetected to the next development step is important in managing product reliability [THA2013]. Faults must be managed across all phases of the life cycle. Many development practices affect fault management. A few of the more important ones are as follows [PAU1990]:

1. Practicing a development methodology: -Using a common approach in translating high-level design into code documentation is particularly
important for larger projects. It facilitates good communication between project team members to reduce faults.

2. Constructing modular systems: - A modular system is well defined, simple and independent parts, well defined interaction through well-defined interfaces. Small and simple modules are easy for designers or programmers to build and thus less prone to faults being introduced in the design process. The modular designs are also maintainable and further increasing the chance that detected faults are properly repaired in time.

3. Employing reuse: - The reuse of software components that have been well tested for an operational profile that is to be used in new system reduces fault introduction. The alternative is to develop new components, which will have to go through a complete fault introduction and removal process will be followed in most software projects.

4. Unit and integration testing: - Testing plays a major role in preventing faults from propagating to a next development step. Unit tests verify function in defined in that modules at low-level designs. Integration testing verifies the interaction of modules specified in the higher-level design.

5. Conducting inspections and reviews: - Inspections and reviews can be held on requirements, design documents, software code, user manuals, training materials and test documents. Both reviews and inspections use a small team to compare the output of a development step with what was specified as input to that step.

6. Controlling change: - Many failures result from change in the intermediate items produced as part of the completed product. Such intermediate items include the code of software components, design and requirements specifications, test plans, and user documentation. To reduce the occurrence of such failures, version control is required.

Also, an orderly procedure must be used to handle requested changes to items. Version and change control together are referred to as configuration
management and update the existing system with reliable component [PAU1990]. This chapter also discusses simulation and modeling, its advantages, limitations, its use in software process modeling and the way one should proceed while applying simulation techniques in any discipline. This chapter tries to explore the areas of Component Based Software Engineering where simulation techniques can be applied for getting better results. Some of the application areas that have been identified in this chapter are:

2. Components Requirement and Testing management.
3. Component Search, Selection and optimize repository.
4. Prediction of Reliability of Component Based Systems using Software Reliability Growth Model (SRGM) like Goel Okumoto Model
5. Component Based Software Project’s Management and Component Integration.
6. Rapid Application Development and scheduling of component based project of testing.

The main contribution of the present work starts with the identification of application areas of simulation in CBSE. Due to the ever increasing costs and risks associated with actual experiments, simulation techniques have been applied in various field of human life like space engineering, energy production, gaming, decision making, manufacturing engineering, aerospace engineering and even in policy and strategy making in management and marketing field. Software engineering in general and Component Based Software Engineering in particular is a sophisticated engineering discipline where simulation has not been used to the extent it has been used in other disciplines [NAV2005]. This may be due to the difficulty in modeling human and organizational behavior. Other reason may be that this is a relatively new discipline and application areas of simulation in software engineering have not been explored up to a great extent. But just like other fields of life here also application of simulation has great potential. In the presented research work potential of simulation [DIE2000] in Component Based Software Engineering has been explored and several simulators have been designed and
developed and their results studied in order to study the behavior of component based software. Simulators developed in the present thesis work will be great asset to the people working in Component Based Software Development while selecting and integrating software components from various component libraries’ for the development of Component Based Software in lesser time, with lesser efforts and using optimum resources. The simulators presented here can be used in development houses to quickly compose and launch their software products in market in order to compete with other players in market in this age where time to market is very short.

![Figure 1.5 Engineering process for Component-Based Software](SPA2000)
1.12 ENGINEERING PROCESS FOR COMPONENT-BASED SOFTWARE

Combining a group of reliable software components may not yield a highly reliable component-based software system. The key to success is the software process, “a framework for tasks that are needed to build the high-quality software” [PRE2001].

1.12.1 Process Model for Component-Based Software

Component-based software will rapidly evolve over its lifetime, so a processing model for component-based systems has to be an evolutionary model. The overall engineering process for component-based software can be divided into the following phases, as shown in Figure 1.5[SPA2000] [CHE2001], requirements, analysis and design, and implementation. These phases include component selection, customization and composition, testing, deployment, and maintenance.

1.12.1.1 Requirements

Requirements analysis will derive specifications for the system under development. During this phase, engineering approaches was followed by the UML use-case model and client developer interaction will finalize the specification sheet.

1.12.1.2 Analysis and design

In this phase, software components and their specifications will be determined. The interactions, dependencies, and structural relationships among these software components will be decided as well. The design of framework with its architecture will be decided and analyzed.

1.12.1.3 Component selection, customization, and development

Component selection is the process of finding matching components from the repository of reusable components. The component selection process might be able to find a group of candidate components, a direct adoption often requires a perfect match of the candidate components with component architecture, component specification, and external environment. Any mismatch in the above areas requires customization of components. If, after
the process of component selection, no qualified candidate is identified, a new component will be developed according to the component specifications.

1.12.1.4 Component composition

Component composition is the process of integrating all available components together, which includes all newly developed components, customized reusable and COTS components. To properly compose these components, component architecture and a specific component model needs to be taken into consideration.

1.12.1.5 Testing

The testing of component-based software should be viewed in two aspects: The testing of individual components and the testing of component-based software. In general, once each individual component has been adequately tested, the testing of component-based software will mainly focus on interactions among different components.

1.12.1.6 Deployment

Component-based software can be used as a component for another system. Therefore, it is necessary to provide packaging and deployment capacity.

1.12.1.7 Maintenance

Maintenance for component-based software may not be as easy as it looks, just plug-and-play. The maintenance for component-based software focuses on how to adequately model the various modification activities and how to determine how much effort is adequate. Once the reliability of a system has been determined, engineers are often faced with the task of identifying the component(s) that cause the most problems to the system in order to prioritize improvements in the design and channel resources and efforts of system improvement to the areas that will have the most impact on the system’s performance. In simple systems such as a series system, it is easy to identify the weak components.

1.13 Summary

In more complex systems, however, this becomes quite a difficult task. Identifying the weakest component is an exercise that is based on
understanding both the reliability of each component and the roles the components play reliability-wise in the system, which is determined by their location in the reliability block diagram (RBD). For complex systems, the analyst needs a mathematical approach that will provide the means of identifying and quantifying the importance of each component in the system. Project managers usually know the expected completion date of a project long before they plan and schedule the activities required to finish the project. To get your project finished on time, you need to schedule all the necessary project activities as quickly as possible. A well-designed sequence of events is essential to controlling every element of your project. Project scheduling begins with the management team identifying the activities that are required to finish the project on time. This process begins with a consideration of the stated project goals and may require significant brainstorming by the team to develop a list of potential activities. The team then prioritizes the activities and develops a schedule for the completion of each item. Project milestones are used later to develop the critical path for the project. Effective planning includes defining the relationship the activities have with each other. Regardless of project size, there are internal and external dependencies that must be identified. Internal dependencies are elements of the project over which the management team has control, and external dependencies are elements the team may have little control over. External elements can create uncertainties that require planners to build slack time into project plans and activity sequencing. The desired outcome of activity sequencing is smooth transition from one activity to the next.