CHAPTER-3 Evolution of Component-Based Software Engineering

3.1 INTRODUCTION

The history of software development began in UK in 1948 (Ezran et al., 2002). In that year, the Manchester "Baby" was the first machine to demonstrate the execution of stored-program instructions. Since that time, there has been a continuous stream of innovations that have pushed forward the frontiers of techniques to improve software development processes. From subroutines in the 1960s through to modules in the 1970s, objects in the 1980s, and components in the 1990, software development has been a story of a continuously ascending spiral of increasing capability and economy battled by increasing complexity. This necessity is a consequence of software projects becoming more complex and uncontrollable, along with problems involving schedules, costs, and failures to meet the requirements defined by the customers. In the 1960's Mcllroy (1969) introduced the idea of software development based on components. Currently, the notion of a software component may include, apart from plain code, other artifacts of the software development process as well - for instance, requirements, design documents, test scripts, and user manuals (Freeman, 1983; Krueger, 1992 and Lim, 1998). The whole idea of Component-Based Software Development (CBSD) thus pursues an efficient and effective reuse process; i.e., the activities of managing, producing, brokering, and consuming all kinds of software components (Lim, 1998 and STARS, 1993), beyond any single software development process. Software industries are striving for new techniques and approaches that could improve software developer productivity, reduce time to market, deliver excellent performance and produce systems that are flexible, scalable, secure, and robust. Only software components can meet these demands. Component-Based Software Engineering (CBSE) has emerged, which has generated tremendous interest in software development community. The paradigm shift to software components and component reuse appears inevitable, necessitating drastic changes to current software development and business practices (Stritzinger, 1996). Reuse of prefabricated hardware building blocks has proven to be an extremely useful production in numerous industrial domains. Electric switches, controllers, motors,
gears, pumps, etc. are usually not designed for a single application, but for many potential uses. Therefore, the desire to assemble or compose software products from pre-fabricated, reusable parts is also natural and rather old. The invention of Object-Oriented Programming (OOP) was an important step in the direction of greater reusability. The simplest and most often practiced reuse in OOP is reuse of concrete classes. This kind of reuse presupposes concrete implementations of well-defined, elaborated concepts. On other hand, CBSD is getting popular and is being considered as both efficient and effective approach to build large software applications and systems, potential risks within component-based practicing areas such as quality assessment, complexity estimation, performance prediction, configuration, and application management should not be taken lightly. In the existing literature there is lack of systematic work in identifying and assessing these risks. In particular, there is lack of a structuring framework that could be helpful for related CBSD stakeholders to measure these risks. The CBSD is close related to reuse. The idea about reusing pieces of software originates from early sixties when the term software crisis was mention first time. The basic idea is simple - when developing new systems use components that are already developed. When researchers and practitioners develop specific functions they need to improve other system and they develop it in that way any body can use this function in other software in the future.

3.2 MOTIVATION FOR COMPONENT-BASED SOFTWARE ENGINEERING

Software are becoming increasingly complex and providing better functionality. Organization often uses component-based technologies instead of developing all the parts of the system from scratch to produce cost-effective systems. The motivation behind the use of components was initially to reduce the cost of development, but it later became more important to reduce the time to market, to meet rapidly emerging consumer demands. At present, the use of component is more often motivated by possible reductions in development costs. By using components it is possible to produce more functionality with the same investment
CHAPTER-3 Evolution of Component-Based Software Engineering

of time and money. CBSE provides methods, models and guidelines for the developers of Component-Based System (CBS). In software engineering several underlying technologies have matured that permit building components and assembling applications from sets of those components with object-oriented and Component technology e.g. Common Object Request Broker Architecture (CORBA). Business and organizational context within which applications are developed, deployed, and maintained has changed. There is an increasing need to communicate with legacy systems, as well as constantly updating current systems. This need for new functionality in current applications requires technology that will allow easy additions. There hasn't been any standards adopted by the industry as a whole, and without standards, it has been difficult for developers to be motivated to design with reusability in mind. Increasingly, complex systems now need to be built with shorter amounts of time available. Software engineers are faced with a growing demand for complex and powerful software systems, where new products have to be developed more rapidly and product cycles seem to decrease, at times, to almost nothing. Some advances in software engineering have contributed to increased productivity, such as OOP, Component-Based Development (CBD), domain engineering, and software product lines, among others. These advances are known ways to achieve software reuse. Some studies into reuse have shown that 40% to 60% of code is reusable from one application to another, 60% of design and code are reusable in business applications, 75% of program functions are common to more than one program, and only 15% of the code found in most systems is unique and new to a specific application (Ezran et al., 2002). According to Mili et al., 1995 and 1998 rates of actual and potential reuse range from 15% to 85%. With the maximization of the reuse of tested, certified and organized assets, organizations can obtain improvements in cost, time and quality.

Building software from existing software components is not a fresh idea but development of software from component is getting more and more popularity in the software organization, due to the short development life cycle and intense
marketing pressure of software. CBSE focuses on the process for the rapid assembly of system by reusing software component with high quality and it is based on the aspects given by Bachmann et al., 2000. First aspect is, the property of software component must be predictable because researchers and practitioners break complicated problem into small pieces and integrate the solution of all the parts of the problem which are the fundamental concept of most mature engineering disciplines, and it also applies to CBSE. So to build a software component with high quality, the property of the software component must be predictable, just like that there is no way of building a computer with high quality without knowing anything about the parts. Second aspect is, the software component should be ready for rapid assembly in the present software market, and time is a critical factor for success. According to this study rapid assembly has always been one of the advantages of CBSE due to the new CBS technology and reuse-oriented development environment. Reuse-oriented development environment is claimed to be the healthy direction for software technology development. Reuse-oriented development environment can increase software quality and reliability because the reusable software component has been used and tested before. Software reuse can increase software productivity and shorten the software release time to market, reduce maintenance cost (Krueger, 1992).

The work done in this study is to explore a practical model to develop a CBS by using two different approach viz. development of reusable and testable software component and development with reuse of software component. CBSD is receiving increasing amounts of interest in the software engineering community. The aim of CBSE is to create a collection of reusable and testable components that can be used for CBD. CBD becomes the selection, adaptation and composition of components rather than implementing the application from scratch. But what are the main causes to use the software component for software development in comparison to traditional software approaches? OR why researchers and practitioners cannot be satisfied with the current state of software development? The answer is, unfortunately, a buzzword in software industry - the
CHAPTER-3 Evolution of Component-Based Software Engineering

software crisis. This study presents problems of traditional software development and software crisis.

3.2.1 Software Crisis

The development of component and CBS is a young discipline. Peoples are surrounded by technical equipment that contains different types of software. Software has given more advanced machines, but at the same time has made more dependent on reliable software. For a number of years researchers and practitioners have wrestled with the so-called "software crisis" which Professor Boehm used in his work (Boehm, 1976) at first time, crisis means here, that the quality of software is generally unacceptably low and that deadlines and cost limits are not being met. Software is produced in a very individual manner. Software is always produced from scratch and when it is delivered it is often poorly documented, and much of the knowledge surrounding it is still sitting in the programmers head. Researchers and practitioners face major problems of software development - poor software quality, and low productivity, high cost and time.

3.2.1.1 Poor Quality Software and Low Productivity

The rapid development of software with the rapid changes in the requirements increases the development time and complexity of software systems. This problem is worsened by the fact that most of the programming languages currently are used for software development, does not promote working in the desired way. The traditional way of driving program development led to many problems. Those who order to develop software often find it difficult to explain to programmers what they want. As lead times become shorter and shorter, documentation and testing suffer. When at last the product is delivered, sometimes very late, it is often of poor quality in terms of lack of documentation and testing techniques. Documentation is the main process during the development of any software or components. Good documentation of any software and components support reusability.
3.2.1.2 **High Software Cost and Time**

The economic aspect of the software crisis is aggravated by the fact that the relative cost of software compared with the cost of hardware has increased greatly by a factor of 10:1 at present. A significant reason for this is that whereas hardware prices have fallen dramatically, software development costs and time cannot match this reduction because of its labor-intensive nature. Software support costs, which are made up primarily of labor costs, account for about 75% of the total software costs. Due to the non-availability of good programmers, developers and regular changes in the software requirements during software development affects the software deadlines and increase the time limit to deliver the software.

Facing the software crisis, people have made great improvement in last 30 years. Reusability and testing are maybe the two effective approaches to response the software crisis. Reuse of software components is becoming more and more important in a variety of aspects of software engineering. Recognition of the fact that many software systems contain many similar or even identical components that are developed from scratch over and over again has led to efforts to reuse existing components. Structuring a system into largely independent components has several advantages. It is easy to distribute the components among various engineers to allow parallel development. Maintenance is easier when clean interfaces have been designed for the components, because changes can be made locally without having unknown effects on the whole system. Software reuse and testing have a major influence on the development of software systems if components’ interrelations are clearly documented, tested and kept to a minimum, it becomes easier to exchange components and incorporate them into new system. The main things that should be solved by component reuse and testing are:

- Components that can be easy to search and integrate.
- Reusability and testability of existing code is easy and fast.
- A specific problem only has to be solved once, and the solution should be put into components and can be used by anyone.
Easier modification and maintenance of existing components.
Components are modular and it is quite easy to use.

3.3 CBSE AND CBSE PROCESSES
Today component engineering is gaining substantial interest in the software engineering community. Although a lot of research effort has been devoted to analysis methods and design strategies of CBS, a few papers address the CBSE and CBSE process of CBS. This chapter identifies and classifies the design, development and analysis issues of software components and CBS. This study proposes a model and component testing process. Finally, it shares our observations and insights on development and testing of CBS. CBSE is a process which helps in design and development software by using reusable and testable software components. Clements (Clements et al., 2003) describes CBSE as embodying “the ‘buy, don’t build’ philosophy”. He also says that “in the same way that early subroutines liberated the software developer from thinking about details shifts the emphasis from programming to composing software systems”.

The goal of CBSE is to increase the productivity, quality, and time to market in software development. Important paradigm shift of CBSE is to build software systems from standard components rather than from scratch. Over the past decade, many software developers, computer scientists and researchers have attempted to improve software development practices by improving design techniques, testing techniques, developing more expressive notations for capturing a system’s intended functionality, and encouraging reuse of pre-developed system pieces rather than building from scratch. Each approach has had some notable success in improving the quality, flexibility, and maintainability of application systems, helping many organizations develop complex, applications deployed on a wide range of platforms. Despite this success, any software industries developing, deploying, and maintaining large-scale software-intensive systems still faces tremendous problems, especially when it comes to testing and updating the systems. Unless carefully designed, it can be very expensive in term of time as well as cost, to add further functionality to a system effectively, and
then to test whether the addition has been successful. Furthermore, in recent years, the requirements, tactics, and expectations of application developers have changed significantly. They are more aware of the need to write reusable code—even though they may not always employ this practice. CBSE should, in theory, allow software to be more easily assembled, and less costly to build. Although this cannot be guaranteed, the limited experience of adopting this strategy has shown it to be true. The software systems built using CBSE are not only simple way but usually turn out to be more robust, adaptable and updateable.

Conventional software reuse and CBSE although object-oriented technologies have promoted software reuse, there is a big gap between the whole systems and classes. To fill the gap, many interesting ideas have emerged in object-oriented software reuse for last several years. CBSE takes different approaches from the conventional software reuse in the following manner (Aoyama, 1998):

- Component should be able to plug and play with other component so that component can be composed at run-time without compilation.
- Component should separate the interface from the implementation and hide the implementation.
- Component is designed on a pre-defined architecture so that they can interoperate with other component.
- Component interface should be standardized so that they can be developed by multiple developers and widely reused across the corporations.
- Component can be acquired and improved by getting feedback from users and end users.

CBSE is in many ways similar to conventional or Object-Oriented Software Engineering (OOSE). A software team establishes requirements for the system to be built using conventional requirements elicitation techniques. Rather than a more detailed design task, the team now examines the requirements to determine what subset is directly amenable to composition, rather than construction. The CBSE process identifies not only candidate components but also qualifies each components' interface, adapts components to remove architectural mismatches,
assembles components into selected architectural style, and updates components as requirements for the system change. Two processes occur in parallel during the CBSE process. These are domain engineering and CBD processes.

3.3.1 Domain Engineering Process
The main aim of domain engineering process is to identify, construct, catalogue, and disseminate a set of software components that have applicability to existing and future software in a particular application domain. The goal is to establish a mechanism by which software engineers can share these components in order to reuse them in future system. Examples of application domains are - railway management systems, defence control systems, financial management systems, and air traffic control system. Domain engineering begins by identifying the domain to be analysed. This is achieved by examining existing applications and by consulting experts. A domain model is then realised by identifying operations and relationships that recur across the domain and therefore being candidates for reuse. This study guides the software engineer to identify and categorise components, which will be subsequently implemented. One particular approach to domain engineering is structural modeling. This is a pattern-based approach that works under the assumption that every application domain has repeating patterns. These patterns may be in function, data, or behaviour that has reuse potential. This is similar to the pattern-based approach in OOP, where a particular style of coding is reapplied in different contexts.

3.3.2 Component-Based Development Process
There are three stages in this process. These are qualification, adaptation and composition. Component qualification examines reusable components. These are identified by characteristics in their interfaces, i.e. the services provided, and the means by which consumers access these services. This does not always provide the whole picture of whether a component will fit the requirements and the architectural style. This ensures a candidate component will perform the function required, and whether it is compatible or adaptable to the architectural style of the
system. The three important characteristics looked at are performance, reliability and usability. Component adaptation is required because very rarely will components integrate immediately with the system. Depending on the component type different strategies are used for adaptation. The most common strategies are - White Box Wrapping (WBW), Grey Box Wrapping (GBW), and Black Box Wrapping (BBW). Component composition integrated the components into a working system. This accomplished by way of an infrastructure which is established to bind the components into an operational system. This infrastructure is usually a library of specialised components itself. It provides a model for the coordination of components and specific services that enable components to coordinate with one another and perform common tasks.

3.4 CBSD AND CBSD PROCESSES
The CBSD consists of two separate but related processes. The first is concerned with the analysis of application domains and the development of domain i.e. development for reuse. The second process is concerned with assembling software systems from prefabricated component i.e. development with reuse (Voas, 1998; Vigder et al., 1996; Voas, 1999; Harrold et al., 1999).

3.4.1 Development for Reuse
The component development process is concerned with developing generic and domain-specific components. To achieve successful software reuse, commonalities of related systems must be discovered and represented in a form that can be exploited in developing similar systems. Domain commonalities are used to develop models or software components that can be used to develop system in the domain. Once reusable components are created, they can be made available within organizations or on the open market as commercial components. If an organization wants to achieve the benefits of sharing and reusing components a successful reuse strategy must include the following - (Kotonya et al., 2003) A reuse method that is consistently applied by component and
application developers. The method that supports reuse contains the following steps.

- Define and classify client requirement with the help of application, interface, and support.
- Search repository for reusable and testable components that support the requirements.
- Analyse candidate component to ensure there is an acceptable match between them and the requirements.
- Develop CBS on standard component model.
- Incorporate the reuse methodology into system development life cycle.

3.4.2 Development with Reuse

Development with reuse develops some of the early ideas on CBD (Brown et al., 1998) to provide a scalable process with a clear separation of concerns. The negotiation phase attempts to find an acceptable trade-off amongst multiple competing development attributed. The planning phase sets out a justification, objectives, and strategies. The development phase implements the agenda set out in the planning phase. Essential issues to be addressed in phase include.

- The process of defining requirement for a CBS.
- The process of partitioning software requirements into logical “components” or “sub-systems” (Sommerville, 2001).
- The process of replacing abstract design components with “concrete” off-the-shelf components.
- The verification process is intended to ensure that there is an acceptable match between the software component used to build the software and the software being built.

Component raises the level of abstraction to that which can be easily used by a domain expert who is not necessarily an expert programmer. They allow software vendors to build visual development environments in which the concept of
plugging together these "software" forms the basis of any new development. The writing of actual code is kept to a minimum - scripting can be used to glue together components or to tailor existing behavior. A typical development effort using components would be importing the components of interest and customizing each one it without explicit coding and finally wiring together the components to form an application. The advantages are immediately obvious (www.objs.com).

- Increased productivity gained by reuse of design and implementation;
- Increased reliability by using well tested code;
- Lower maintenance costs because of a smaller code base;
- Minimizes effects of change since Black Box Programming (BBP) tends to rely on interfaces as compared to explicit programming;
- Components provide a well encapsulated mechanism to package, distribute and reuse software.

3.5 COMPONENT-BASED SYSTEMS

Component-Based Software Engineering (CBSE) has two different perspectives. One is based on the relationship to fundamental principles of system design. In this view, the key features of CBSE could have developed simply as inevitable consequences of applying general "system thinking" to software systems. Another is based on observing the history of two related branches of software engineering research. Human-engineered physical devices such as cars and appliances and computers, as well as naturally occurring physical objects such as forests, galaxies and nerve bundles, often are called "system." Both this word and the thinking that underlies it are crucial features of modern software engineering as well. Software systems, however, are purely symbolic entities that have meaning and intellectual interest even without a physical manifestation. CBS in physical engineering and in software engineering therefore have many common features, yet differ in important and sometimes subtle ways. Examples of CBS are Microkernels, Client/Server Architectures. CBS is any part of the CBSE in which someone
wants to treat as an indivisible unit and develop system with high quality (Weide, 2001). CBSE has received considerable attention among software vendors and information technology organizations. Software component concepts have been evolving very rapidly and a marketplace for software components has emerged. Components have changed the way that programmers develop major applications (Bill and George, 2000; Maurer, 2000). Developing and using various software components as building blocks can significantly enhance CBSD and use that is why both the academic and commercial sectors have shown tremendous interest in CBSD indeed, much effort has been devoted to define and describe the terms and concepts involved. Interpretation of component and CBD varies from vendor to vendor. Some package vendors use the term component to refer to what might be regarded as a sub-system whereas others use the term component as a synonym for the distributed object. Few others define component as reusable deliverable whereas Unified Modeling Language (UML) defines a component as a physical executable, irrespective or its attributes (Grover and Gill, 2003).

In order to make a system easy to develop and maintain, the component programming technique is often used. A component is an instance of some class of components. Components of the same class may be parameterized differently and therefore behave in different ways. In component-wise programming, system functionalities are decomposed into small components with each implementing a small functionality. Components can be glued together by some binding mechanism to realize the system functionality. There are several advantages to component-based design (Xiaoming, 2001). Component realizes a small functionality, which is easy to implement and debug. Managing the system as a set of components is easier. Many practical systems are constructed from components. adaptive systems can be easily made out of such reconfigurable CBS. Such system can adapt their functional and non-functional behaviors in four ways.

- Some components being used in system composition can adjust their behaviors by adjusting their parameters. Self adjusting components have
been used for a long time. In multimedia systems, a server can adjust its playback rate according to receiver’s consuming rate in order to achieve better playback quality (Cowan et al., 1995).

- People want to build systems that can solve problems under as many conditions as possible. It is hard to come up with a single algorithm that is optimal in every possible situation.

- A component can be added to the system to introduce new functionality. Similarly, a component can be deleted to save system resources. In many systems, what types of components are required is best decided at runtime. For example, in a distributed conference application, if all the participants are behind the same firewall, they may trust each other. Otherwise, an encryption mechanism is needed for security; message can be processed differently depending upon whether they are going over an insecure link or not (Van et al., 1998).

- Components interact with each other. A distributed system or application works best if the distributions of its components are mapped well onto its underlying network or system.

3.6 COMPONENT-BASED MODELS AND TECHNOLOGIES

In CBSE there are three major component models. These three component models are COM, JavaBeans, and CORBA and all of them have different levels of service for the application developers. These models in component software are the one which are proposed by the Object Management Group (OMG), Microsoft, and Sun. They are - Common Object Request Broker Architecture (CORBA Component Model (CCM)) (OMG, 1999) CORBA is release of CORBA standards, and it proposes the new CORBA Component Model (CCM). CCM is an extension of Enterprise Java Beans (EJB). CORBA is a container-based specification, that is, every component instance is inside a container. Component Object Model (COM+) (Eddon, 1999), COM+ is an extension of COM, the foundation of Microsoft’s component platform. COM+ 2.0 has been used for the .NET Framework. COM+ separates declarative attributes about infrastructure
services from the code of components. EJB (Sun Microsystems, 2001) is a Java’s component model based on container. The container will implement the runtime environment for the enterprise bean, which includes security, concurrency, life cycle management, transaction and other services. This study discusses here following component-based models and technologies e.g. COM, EJB and CORBA according to the (Crnkovic et al., 2001).

3.6.1 Component Object Model
The Component Object Model (COM) provides a model for designing component that have multiple interfaces with dynamic binding to other component. COM is an open standard, which has been implemented on many different platforms, but the main platform is of course Microsoft Windows for which it was first developed. Components exposed themselves only through interfaces. The interfaces are binary which makes it possible to implement the component in a verity of programming languages such as C++, Visual Basic and Java. A COM component can implement and expose multiple interfaces. A client uses COM to locate the server component and then it queries for the wanted interfaces.

3.6.2 Enterprise JavaBeans
Enterprise JavaBeans (EJB) is component architecture for server-side component used to build distributed system with multiple clients and servers. A Java Bean is a reusable component that support persistency and can inter-operate across all platforms supported by java. EJB uses JavaBeans but it is a lot more then a component model. EJB provides support for transaction and security over a neutral object. Which gives the user benefits to implement the application on top of a protocol of their choice? EJB (Sun Microsystems, 1997) is a part of the Java 2 Platform Enterprise Edition (J2EE) which includes many other technologies - Remote Method Invocation (RMI), Java Naming and Directory Interface (JNDI), Java Database Connectivity (JDBC), Java Server Pages (JSPs) and Java Messaging Services (JMS).
3.6.3 Common Object Request Broker Architecture

Common Object Request Broker Architecture (CORBA) is used to develop large heterogeneous systems that use the services of distributed components. These distributed objects can be viewed as components and they provide services based on calls made to the methods they implement. In this model, composition is simply the acquisition of an object reference by another object. CORBA is a standard that has been developed by the OMG in the beginning of the nineties. The OMG provides industry guidelines and object management specifications to supply a common framework for integrating application development. Primary requirement for these specifications are reusability, portability and interoperability.

To understand how component technology helps to decompose a system, it is necessary to know which are the main pieces used to build the whole systems, components. The term component is very vague in software engineering and it can have different meanings, thus in (Szyperski et al., 2002), three different but complementary, definitions of components software are given each one adopting a different level abstraction.

- "Software component are binary units of independents production, acquisition and deployment that interact to form a functioning system".
- "A component is a unit of compositions with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties". This is given at the 1996 European Conference on Object-Oriented Programming (ECOOP) as one of the results of the Workshop on Component-Oriented Programming (COP).
- "A component is a set of normally simultaneously deployed atomic components. An atomic component is a module and a set of resources."

The definition of a concept, it also can be given using its characteristic properties, thus Szyperski states that a software component is a unit of independent deployment, is a unit of third party composition and has no observable state.
CHAPTER-3 Evolution of Component-Based Software Engineering

According to the taxonomy proposed in (Voelter, 2003) technical components are technical building blocks to assemble applications. These kinds of components evaluate their own features by means of a tool known as container, which provides a runtime environment for the component.

3.7 OBJECT-ORIENTED PROGRAMMING

Object-Oriented Programming (OOP) (http://en.wikipedia.org) is an important programming paradigm that was born in the 1960s. The concept of OOP is that a program is composed of objects. The basic type in OOP is called a class and objects are instances of classes. OOP’s core features include encapsulation, inheritance and polymorphism - encapsulation is a language feature that accumulates methods and fields into classes, whose accessibility is controlled by modifiers; inheritance enables a class to reuse the interface or implementation of other classes; polymorphism makes possible the same operation applying on different types of data. All these language features provide high quality information hiding and reusability for software development. Consequently, programming in OOP significantly improves the modularity of software implementation by encapsulating related data and behaviors together.

The idea in OOP is that software should be written according to a model of the actual or imagined object it represents. OOP and the related discipline of the object-oriented design and object-oriented analysis focus on modeling real-world interaction. OOPs techniques promote a new approach to software engineering in which reliable, open application can be largely constructed, rather than programmed, by reusing “frameworks” (Deutsch, 1989) of plug compatible software components. According to Mellroy, 1969

- Modern applications are increasingly open in terms of development, platform and evolution, and so the need for a component-oriented approach to development is even more acute than in past;
CHAPTER-3 Evolution of Component-Based Software Engineering

- Objects provide an organizational paradigm for decomposing large application into cooperating objects as well as a reuse paradigm for composing applications from pre-packaged software components.

Despite the contributions of object-oriented technologies, there are several open research problems that must be resolved to reach the goal of effective component-oriented approach for composition and reuse must be cleanly integrated with other features, such as concurrency, persistence and distribution. Reuse of software presupposes and retrieval of components according to application requirements and the development running applications from components. The design of reusable frameworks is an iterative, evolutionary process, so it is necessary to manage software and software information in such a way that designs and implementations can evolve gracefully. Finally, present object-oriented methodologies do not explicitly address the design of reusable frameworks. Not only the languages and tools, but the methods and culture of software development must ultimately adapt to a new evolutionary software life-cycle if realize the benefits of large-scale software reuse (Cox, 1990; Wegner, 1984).

A series of ongoing research projects at the University of Geneva that address Component-Oriented Software Development (COSD) at the levels of languages, tools and frameworks, in particular, first is the integration of object-oriented language features that support software composition with features concerned with other issues, like concurrency, second one is application development tools to support composition and reuse, and third one is the development of reusable application frameworks particular domain. If one is familiar with OOP, it can be useful to think of CBSE in a similar way. In OOP, code is reused in the form of objects. These objects are often contained in vast libraries of reusable code. Frameworks take the process even further, providing more robust and disciplined software of reuse. By obtaining and reusing parts of software which have already been ‘tried and tested’, researchers and practitioners can exploit the principal advantages of OOP techniques over procedural programming techniques, which is that they enable programmers to create modules (www.smb.uklinux.net) that do
not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits many of its features from existing objects. This makes object-oriented programs easier to modify. In the same way, in CBSE, by reusing an existing component researchers and practitioners cut out a lot of the hard work with establishing the usefulness and in testing that component, but some time it is found that testing will still be required at the time of integration on the user site. CBSE uses software engineering principles to apply the same idea as OOP to the whole process of designing and developing CBS. It focuses on reusability of existing components, as opposed to just coding in a particular style. CBSE encourages the composition of CBS, as opposed to programming them. CBSE, by contrast, makes no such assumptions, and instead states that software should be developed by gluing prefabricated components together much like in the field of electronics or mechanics.

3.8 COMPONENT-ORIENTED PROGRAMMING

The idea of software components has been raised by Mellroy. Everybody has their own understanding of an objects and OOP, component and Component-Oriented Programming (COP) until the late 1980s. The idea of assembling software systems out of existing components instead of building software systems from scratch was first developed by Mellroy (Mellroy, 1969). Component software is primarily concerned with reuse of software artifacts in a centralized setting, where a single software vendor has complete control. Component software is primarily concerned with the extensibility of software systems in a distributed setting, where any interested party can develop extensions which can be acquired and integrated at any time (Nierstrasz et al., 1992).

Components were originally considered operations in structured programming and modules that contain operations in modular or OOP. In modular programming, components are modules that encapsulated a collection of related operations. In OOP, components are types (i.e. classes), that again encapsulate a collection of related operations. For the emerging paradigm of COP, a
combination of all these abstraction mechanisms has been suggested instead (Szyperski et al., 2002) - Components are modules that encapsulate a collection of operations as well as a collection of types. As the essence of COP, distributed extensibility should be able to explain the necessary features of an appropriate programming language. As an organizational paradigm, COP is concerned with the composition and interaction of components and frameworks through interfaces; it is not concerned with their insides in any way. Systems that support COP and the programmers that use them adhere to a set of core principles. The most important of these include - separation of interface and implementation, binary compatibility, language independence, location transparency, concurrency management, version control, component-based security. Component programming requires both systems that support the approach and programmers that adhere to its discipline. As the supporting technologies become more powerful, no doubt software engineering will extend its understanding of what constitutes COP and generates new ideas.

3.9 SUMMARY
This chapter was intended to begin a history and foundation of the study and preliminary studies in CBSE. This chapter starts with the motivation for CBSE and presents software crisis with main factors e.g. poor quality, low productivity, high software cost and time. This chapter summarizes in detail processes of CBSE and CBSD with component-based model and technologies. This chapter also summarizes how and why researchers and practitioners shift from OOP to COP. This chapter helps in attempting only to state an idea to develop software models and CBS test processes.