CHAPTER 2

STUDY AND REVIEW OF LITERATURE

2.1 Introduction .................................................................................................................. 13
2.2 Analysis of Software Models ...................................................................................... 13
2.3 Study of Component-Based Development .................................................................. 18
2.4 Review of Software Reuse ......................................................................................... 24
2.5 Study of Component-Based Software and Component Reuse ................................. 29
2.6 Review of Software Testing ....................................................................................... 32
2.7 Study Component-Based Software and Component Testing .................................... 34
2.8 Summary ..................................................................................................................... 37
CHAPTER-2  Study and Review of Literature

2.1 INTRODUCTION

Software engineering is concerned with practical application of scientific knowledge in the design and development of software and the associated documentation required to develop, operate, and maintain them (Boehm, 1976). Furthermore, software engineering must accommodate human, economic, and programming concerns, and these concerns must be satisfied from both the product and process sides of the software engineering (Boehm, 1981; Pressman, 1992). Software engineering involves a number of different disciplines - computer science, management, psychology, design and economic, among other things (Freeman and Gaudel, 1991) to solve problems in building software. In software engineering the concern is, on one hand, to build economically usable software systems, and, on the other, to manage the development process (Boehm, 1981). Software engineering seeks a way to improve the quality, pace, and utilization of resources in software development. One specific way to increase productivity and quality in software development is to reuse existing reusable and testable software components. Reusability and testability are two main important factors which help in developing better component-based software with the help of software model and testing process. Moving this direction this study presents study and review of literatures regarding the software models, Component-Based Development (CBD), software reuse and testing, component-based software, component reuse and testing. The study and literature review in this chapter is designed to be comprehensive and to discuss the significance of the research aims, provide a critical review of the relevant literature, identify knowledge gaps and address the relationship of the literature to the research work.

2.2 ANALYSIS OF SOFTWARE MODELS

A software model depicts the significant phases or activities of a software project from conception until the product is retired. It specifies the relationships between project phases, including transition criteria, feedback mechanisms, milestones, baselines, reviews, and deliverables. Typically, a life cycle model addresses the phases of a software project - requirements phase, design phase,
implementation, integration, testing, operations and maintenance. This study presents analysis of classifications of different models in software engineering.

2.2.1 Prescriptive Software Models
A prescriptive model prescribes how a new software system should be developed. Prescriptive models are used as guidelines or frameworks to organize and structure how software development activities should be performed, and in what order. Typically, it is easier and more common to articulate a prescriptive life cycle model for how software systems should be developed. This is possible since most such models are intuitive or well reasoned. Thus, of course, should raise concern for the relative validity and robustness of such models when developing different kinds of development settings, using different programming languages, with differentially skilled staff, etc. Prescriptive models are used to package the development tasks and techniques for using a given set of software engineering tools or environment during a development project.

2.2.2 Descriptive Software Models
A descriptive model describes the history of how a particular software system was developed. Descriptive models may be used as the basis for understanding and improving software development processes, or for building empirically grounded prescriptive models (Curtis et al., 1988). Descriptive software models characterize how particular software systems are actually developed in specific settings. As such, they are less common and more difficult to articulate for an obvious reason - one must observe or collect data throughout the life cycle of a software system, a period of elapsed time often measured in years. Also, descriptive models are specific to the systems observed and only generalizable through systematic comparative analysis. Therefore, this suggests the prescriptive software life cycle models will dominate attention until a sufficient base of observational data is available to articulate empirically grounded descriptive life cycle models.
2.2.3 Traditional Software Models

Traditional models of software evolution have been with us since the earliest days of software engineering. The classic software life cycle and stepwise refinement models are widely instantiated in just about all books on modern programming practices and software engineering. The incremental model is closely related to industrial practices. The progressive steps of software evolution are often described as stages, such as requirements specification, preliminary design, and implementation; these usually have little or no further characterization other than a list of attributes that the product of such a stage should process. Further, these models are independent of any organizational development setting, choice of programming language, software application domain, etc. In short, the traditional models are context free rather than context sensitive.

2.2.4 Classical Software Models

The classic software model is often represented as a simple prescriptive waterfall software model, where software evolution proceeds through an orderly sequence of transitions from one phase to the next in order (Royce, 1970). Such models resemble finite state machine descriptions of software evolution. However, these models have been perhaps most useful in helping to structure, staff, and manage large software development projects in complex organizational settings, which was one of the primary purpose (Royce, 1970; Boehm, 1976). Alternatively, these classic models have been widely characterized as both poor descriptive and prescriptive models.

2.2.5 Stepwise Refinement

In this approach, software systems are developed through the progressive refinement and enhancement of high level system specifications into source code components (Wirth, 1971; Mili at el., 1986). However, the choice and order of which steps to choose and which refinements to apply remain unstated. Instead, formalization is expected to emerge within the heuristics and skills that are acquired and applied through increasingly competent practice. This model has
been most effective and to teach individual programmers how to organize their software development.

2.2.6 Incremental Development

Developing systems through incremental development requires first providing essential operating functions, then providing system users with improved and more capable versions of a system at regular intervals (Basili, 1975). This model combines the classic software life cycle with iterative enhancement at the level of system development organization. It also supports a strategy to periodically distribute software maintenance updates and services to dispersed user communities. This in turn accommodates the provision of standard software maintenance contracts. It is therefore a popular model of software evolution used by many commercial software prototyping tools and techniques, which more directly provide support for incremental development and iterative release for early and ongoing user feedback and evaluation (Graham, 1988).

According to (Mathiassen et al., 1998) there are different approaches that can be divided in two groups; specification based and exploratory based. Of the basic specification based approach, the waterfall model (Royce, 1970) is the prime example whereas the exploratory approach is exemplified by the prototype approach (Gomaa, 1983). Third approach of software development is the mixture of the two, which could be called evolutionary approach (Sommerville, 1996). Example of this third type include the spiral model (Bohem, 1986) and incremental models (Grahms, 1988; Gilb, 1988).

Initially the term lifecycle was used as a reference to the approaches according to (Davis et al., 1988) and (Acuna et al., 1999), but in the late 80’s works by (Osterweil, 1987) and (Humphrey, 1989), among others, drew attention to the process aspect of software development. While the life cycle and model approach is concerned with the products that are produced during the development effort, the process view focuses on the production process. Explicit models of software evolution date back to the earliest projects developing large software systems in
the 1950's and 1960's (Hosier, 1961; Royce, 1970). Overall, the apparent purpose of these early software life cycle models was to provide a conceptual scheme for rationally managing the development of software systems. Such a scheme could therefore serve as a basis for planning, organizing, staffing, coordinating, budgeting, and directing software development activities. Since the 1960's many descriptions of the classic software life cycle have appeared (e.g., Hosier, 1961; Royce, 1970; Boehm, 1976; Distaso, 1980; Somerville, 2001). Royce 1970 originated the formulation of the software life cycle using the familiar model known as “waterfall” model. As software is growing increasingly complex, so too is the effort required to produce it, on this account numerous software life cycle models have been proposed. Their main utility is to identify and arrange the phases and stages involved in software development and evolution, so it is appropriate to generally examine different software life cycle models and point out their strengths and weaknesses before an alternative one is put forward. The Waterfall model (Royce, 1970) has been long used by software engineers and has become the most prevalent software life cycle model. This model initially attempts to identify phases within software development as a linear series of actions, each of which must be completed before the next is commenced. The Waterfall model is marked by the apparently neat, concise and logical ordering of the series of obvious phases, which must be followed in order to obtain the final software product. Refinements to this model consider that completion is seldom and that iteration back to a previous stage is likely to happen, but it takes no account of bottom-up development and prototyping. The Spiral model (Boehm, 1986) makes software development more flexible and has been proposed mainly to speed up software development through prototyping. Prototyping is the process of building an incomplete piece of software that exhibits some of the most relevant aspects of the final software system. Prototyping provides constructive feedback to designers and potential users so that the system requirements can be clarified and refined early during software development. Evolutionary prototypes provide incremental software development, so that software systems may be
gradually developed and tested, allowing major errors to be exposed and corrected early, which means that they are often cheaper to fix, but without effective management to control iterations, this process can degenerate into uncontrollable hacking. V model is widely used in many organizations for building complex software. The Y model, (Luiz, 2005) has been proposed as a viable alternative to address software reusability during Component-Based System (CBS) production. The creation of software is characterized by change and instability; hence the diagrammatic representation of the Y model considers overlapping and iteration where appropriate. So this study proposes a new X component-based model to avoid overlapping and iteration with software reusability and testing.

2.3 STUDY OF COMPONENT-BASED DEVELOPMENT

Component-Based Development (CBD) is assumed to have many advantages. These include more effective management of complexity, reduced time to market, increased productivity, improved quality, a greater degree of consistency and a wider range of usability (Brown, 2000; Crnkovic et al., 2003). It also brings many challenges, because it involves various stakeholders and roles, such as component developers, application developers, and customers. Different stakeholders and roles have different concerns (Brown et al., 1998), and face different issues and risks (Berereton, 2000; Padmal, 2003). CBD differs from traditional development, where the usual approach is for stakeholders to agree upon a set of requirements and then build a system that satisfies these requirements from scratch. CBD builds application by reusing existing components. Available components may not be able to satisfy all the requirements. Therefore, component-based projects must have flexibility in requirements, and must be ready to (re) negotiate the requirements with the customer. Moreover, component are intended to be used ‘as-is’. If some additional functionality is required, ‘glue-code’ is needed to be built to meet the differences between the requirement and component functionality. Another important feature of CBD is the strong focus on the quality attributes and related testing. CBD and Component-Based Software Engineering (CBSE) are often used indistinguishably, but some literature
distinguishes between these two. *Bass et al., 2001* write that CBD involves technical aspects for designing and implementing software components, assembling systems from pre-built components and deploying system into target environment. CBSE involves practices needed to perform CBD in a repeatable way to build systems that have predictable properties.

CBD is playing an increasing role in the software industry (*Bass et al., 2001*) (*Williams, 2000*). There is an economic push to such growth - the claim is that CBD allows the reduction of cost and time to market, while increasing software quality, through reuse (*Szyperski et al., 2002*). The Software Engineering Institute (SEI) defines a component as “an opaque implementation of functionality subject to third party composition and conformant to a component model” (*Bachman et al., 2000*). With an increasing percentage of component-based architecture relying on black-box software components, the quality of such architectures depends, to a large extent, on the quality of those components and on the interactions among them. Therefore, components evaluation should be integrated in CBD (*Brownsword et al., 2000*).

CBD is an approach to the old problem of handling the complexity of a system by decomposing it. Already in 1976, *David Parnas* wrote about the benefits of decomposing a system into modules, such as shorter development time since modules can be developed by separate groups, increased product flexibility and ease of change, and increased comprehensibility since modules can be studied one at a time (*Parnas, 1976*). He also wrote that the main criteria for modular decomposition should be information hiding. Modules such as in procedural languages and objects in object-oriented design are example of previous attempts at decomposition. Software reuse has also been discussed for decades. So what is new in CBD? The answer is the focus on software architecture as a guideline to put pieces together, viewing components as independent units of development and deployment, and on component models. Developing components so that they become reusable is called developing for reuse, while developing
systems of reusable components is called developing with reuse (Karlsson, 1995). CBD facilitates reuse by providing logical units for assembly and makes systematic reuse possible by demanding that components should adhere to a component model. There are two distinct activities in CBD (Ghosh, 2002) - Development of components for CBD and CBD process itself, which includes assembly. Bachman et al. list advantages of CBD as (Bachman et al., 2000) - reduced time to market - even if component families are not available in the application domain, the uniform component abstractions will reduce overall development and maintenance costs. CBD is about building composable components and building systems from these components. Important aspects are therefore reuse, autonomy of components and composition. Researchers and practitioners faces challenges are due to immaturity or lack of software engineering methods, processes and tools in all these aspects. Bass et al. mention inhibitors in CBD as lack of available components, lack of standards for component technology, lack of certified components and lack of engineering methods (Bass et al., 2000). Crnkovic lists the challenges of CBSE as - specification, component models, life cycle, composition, certification and tools (Crnkovic et al., 2002). The present focus is on analysis, design and development of models by avoiding the challenges in each development (Ghosh, 2002; Jacobson et al., 1997).

Study on CBD and reuse started in 1969. McIlroy first introduced the idea of systematic reuse as the planned development and widespread use of software components in 1968 (McIlroy, 1969). Many software organizations around the world have reported successful reuse programs such as IBM, Hewlett-Packard, Hitachi and many others (Griss, 1993). Following the success of the structured design and object-oriented paradigms, Component-Based Software Development (CBSD) has emerged as the next revolution in software development (Padmal, 2003). More and more IT companies have started to reuse code by encapsulating it into components. Whitehead defines a component as - A software component is a separable piece of executable software, which makes sense as a unit, and can
interoperate with other components, within some supporting environment. The component is accessible only via its interfaces and is capable of use ‘as-is’, after any necessary installation and configuration procedures have been carried out (Katharine, 2002)

Terms and definitions for a reusable part of component vary greatly (Sametinger, 1997). It may be called an asset (Lim, 1998), artifact (Jacobson et al., 1999) or work product, among others. Even using these terms the authors want to emphasize the fact that a component can be more than code component. Frakes and Terry, 1996 identified ten different reusable aspects of software project - architectures, source code, data, design, documentation, estimates, human interfaces, plans, requirements, and test cases. Kruger (Kruger, 1992) states that the types of reusable artifacts are not only fragments of code but can also be design structures, module-level implementation structures, specification, documentation, transformations and so on. Bosch (Bosch, 2000) defines a software component as a unit of composition with explicitly specified provided; required and configuration interfaces and quality attributes. Szyperski (Szyperski, 2000) sees software components to be binary units of independent production, acquisition and deployment that interact of form a functioning system.

Components are defined and classified in multiple ways. Definitions vary based on the life cycle phase for component identification (e.g. logical abstractions vs. implementation units), origin (in-house, bought or free software), or roles a component can play in a system (e.g. process components, data components etc.). A few of these definitions are given here, before discussing what is important for reuse. A reusable component is a product of a software development process that can be used as part of a software product other than it was originally designed for. Additionally, a reusable component is documented in such a way that the documentation supports all reuse activities, namely component creation, brokering and consumption. This definition means that a component is part of the end product of the software development process, for example, the defined
analysis phase products. Study also emphasizes systematic approach by requiring that component’s documentation must support reuse activities. Study does not regard a component reusable unless these three aspects are satisfactorily taken into account. Reusable components are created in a software development. This means that components are a result of a software development effort.

This study concentrates on CBD, so it is important to define the term component for the purpose under consideration - A software component is a unit of composition with explicitly specified provided, required and configuration interfaces, plus quality attributes. This definition is based on the well known definition form ECOOP'96 (Szyperski, 1998), that defines a component as unit of composition with contractually specified interfaces and context dependencies only, that it can be deployed independently, and that it is subject to composition by third parties. From this it becomes apparent that components are basically built upon the same fundamental principles as object technology. The principles of encapsulation, modularity, and unique identities are all basic object-oriented principles that are subsumed by the component paradigm (Atkinson, 2002). Brown and Wallnau describe a component as “a nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture”. Components have an interface. Brown and Wallnau describe a software component as “a unit of composition with contractually specified and explicit context dependencies only” (Brown and Wallnau, 1998). Unlike objects in Object-Oriented Programming (OOP), components are usually built up of many software “objects” and provide a coherent unit of functionality. These so-called “objects” all work together to perform a specific task at a particular level of service.

There is still much debate about the meaning of the term component. There are many definitions of components used throughout the software industry. Moreover, a good software component is one that has been designed in such a manner that it can be used effectively as a black box, with its interfaces and
behaviors documented in a manner that the user can easily understand the services offered and can rapidly integrate the component. So at this point, it is necessary to analyze existing popular definitions of software component and then to derive some more general definition of a software component characterizing all its possible attributes. The following definitions of software component are commonly cited throughout the literature (Brown et al., 1998; Hopkin, 2000):

- A component is a language neutral, independently implemented package of software services, delivered in an encapsulated and replaceable container, accessed via one or more published interfaces (Sparling, 2000).
- A software component is a coherent package of software artifacts that can be independently and delivered as a unit and that can be composed, unchanged with other components to build something larger (D'Souza, 1998).
- A software component is a nontrivial, nearly independent, and replaceable part of a system that fulfils a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces (Brown and Wallnau, 1998).
- A software component is a physical packaging of executable software with a well-defined and published interface (Hopkin, 2000).
- A runtime software component is a dynamically bindable package of one or more programs managed as a unit and accessed through documented interface that can be discovered at runtime.

Abstraction level of the components that decides the phase of the development process wherefrom the components can be used (Lenz et al., 1987). When talking about reuse of components two different processes can be separated - production of components and use of them (Taivalsaari, 1993), (Sommerville, 1996), (Biggerstaff and Richter, 1987). In addition to this, some researchers see the maintenance of the components repository as a separate task (Lim, 1998), (Prieto, 1991), (Devanbu et al., 1990), (Caldiera and Basili, 1991), (Ostertag et al., 1992). Moreover, some researcher have considered abstraction as a means to
find a components (Ostertag et al., 1992) and (Karlson, 1995). Rine (Rine et al., 1998) researched the fact that presently there is no set of success factors which are common across organizations and have some predictable relationship to software reuse. Object-oriented and procedural development approaches produce software with high commonality between applications, and have at least reasonably mature processes. Despite the apparent potential for success, about one-third of the projects failed. Morisio et al., 2002 identified three main causes of failure - not introducing reuse specific process, not modifying non-reuse processes, and not considering human factors. Morisio et al. highlight that the root cause was a lack of commitment by top management, or non-awareness of the importance of these factors, often coupled with the belief that using the object-oriented approach or setting up a repository is all that is necessary to achieve success in reuse. Active areas of reuse research in the past twenty years include reuse libraries, domain engineering methods and tools, reuse design, design patterns, domain specific software architecture, componentry, generators, measurement and experimentation, and business and finance. Jacobson et al.’s book describes the reuse-driven software engineering business to manage business, architecture, process and organization for large-scale software reuse (Jacobson, 1997). Morisio et al. and Rine et al. summarize many reuse cases and discuss reuse success factors (Morisio et al., 2002; Rine et al., 1998). One of the recent books on software reuse is (Mili et al., 2002), describing technological, organizational, and management or control aspects.

2.4 REVIEW OF SOFTWARE REUSE

Reuse in software engineering was proposed at the same conference as the term software engineering was introduced (Krueger, 1992; Bauer, 1976) and (Mellroy, 1976). So reuse and software engineering have co-existed for thirty years. Researchers and practitioners recognized the need to build a library of mathematical subroutines in 1949 to avoid having to rewrite them (Tracz, 1995). In 1968 Mellroy (Mellroy, 1976) introduced the idea of reusable components that are used in computer manufacturing. Mellroy’s components would have been
built by dedicated component factories. His components were source code components and were to be used in their original form in other programs. Unlike in other engineering disciplines, software engineering has only succeeded moderately, at the best, in reuse (Prieto Diaz, 1994).

The basic approach of reusability is to configure and specialize pre-existing software components into viable application systems (Biggerstaff, 1984; Neighbors, 1984; Gogune 1986). Such source code components might already have associated specifications and designs associated with their implementations, as well as have been tested and certified. However, it is also clear that software domain models, system specification, designs, test case suites, and other software abstractions may themselves be treated as reusable software development components. These components may have a greater potential for favorable impact on reuse and semiautomated system generation or composition (Neighbors 1984). Therefore, assembling reusable software components is a strategy for decreasing software development effort in ways that are compatible with the traditional life cycle models.

The assumption in software engineering is that people use some sort of method to construct the software development (Yourdon, 1979). Software development method is a predefined and organized collection of techniques and a set of rules which state by whom, in what order, and in what way the techniques are used to achieve or maintain some objectives (Tolvanen, 1998). Most of the methods exclude, for example, maintenance, and configuration control. One big parts that (Boehm, 1976) stresses in software engineering are the management of the process. Systematic software reuse is the purposeful creation, management, support and reuse of reusable components (Jacobson et al., 1997). There exists some reuse process models (Jacobson et al., 1997), (Lim, 1998), (STARS, 1992), (Karlsson, 1995) which address the features for reuse process. In the Lim’s model reuse-process is divided in four tasks - managing the reuse process, producing assets, brokering, and consuming assets. One problem with reuse models and
components and the ratio of reused components to total components will determine the reuse benefits (e.g. improved productivity and quality) (Frakes, 1990; Jeffery, 1997). Software reuse can take many different forms, from ad-hoc to systematic (Kruger, 1992). In the broad definition of reuse, it includes reusing everything associated with software projects, such as procedures, knowledge, documentation, architecture, design and code.

According to Biggerstaff and Richter reuse in software engineering can be divided in composition and generation technologies (Biggerstaff and Richter, 1987). Composition techniques base reuse on the idea that component should be understood as atomic units and be reusable more or less unchanged. An external agent combines these components are design patterns (Gamma et al., 1995), code skeletons or frameworks (Fayad et al., 1999), software architectures (Tracz, 1995; Bosch, 2000) and of course binary components (Szyperski, 1997), among other things. Generative technologies weave patterns or components into a program generator. These patterns are of two main types - patterns of code and patterns in transformation rules (Biggerstaff and Richter, 1987). Usually, in these techniques, software is defined in a higher level language and from this definition an application generator creates the required program. Patterns that are used to generate the software are modified to fulfil requirements. In compositional reuse some portion of the products from the software development process are reused. This means that the project that creates the reusable part is not the same as the project which uses the component.

According to (Frakes et al., 1995), a key concept of systematic reuse is the domain, which may be defined as an application area or, more formally, a set of systems that share design decisions. In this context, "systematic software reuse is a paradigm shift in software engineering, from building single systems to building families of related systems". Software reuse has been practiced since programming began. Reuse as a distinct field of study in software engineering, however, is often traced to Mellroy's paper which proposed basing the software
industry on reusable components. Other significant reuse research developments include Parnas' idea of program families and Neighbors' introduction of the concepts of domain and domain analysis.

Mili et al. define reusability as a combination of two characteristics (Mili et al., 2002) - usefulness, which is the extent to which an asset is often needed and usability, which is the extent to which an asset is packaged for reuse. Frakes et al., 1996 define software reuse as "The use of existing software knowledge or artifacts to build new software artifacts", a definition that includes reuse of software knowledge. Morisio's definition is closer to what is meant by "software reuse" i.e. reuse of building blocks in more than one system (Morisio et al., 2002). Reuse of software knowledge such as domain knowledge, or patterns may happen without reuse of building blocks and is captured in domain engineering. Frakes et al. have investigated sixteen questions about software reuse using a survey in twenty nine organizations in 1991-1992 (Frakes et al., 1995) and presents a report that most software engineers prefer to reuse rather than to build from scratch. They also did not find any evidence that use of certain programming languages, Computer Aided Software Engineering (CASE) tools or software repositories promote reuse. On the other hand, reuse education and a software process that promotes reuse have positive impact on reuse. Frakes et al. also found that the telecom industry has higher levels of reuse than some other fields.

2.5 STUDY OF COMPONENT-BASED SOFTWARE AND COMPONENT REUSE

Component-based software, software components and software reuse complement each other perfectly. Using software components to build CBS almost automatically leads to software reuse and trying to reuse software almost automatically evolves in the composition of software out of components. Software reuse can not be done without the involvement of any components. Reusing software has a much broader influence on software engineering than one might initially think. Not only does it influence the construction process, it fundamentally affects organizational structures and project structures, and it
influences legal and economic issues of software engineering. For software reuse to become a matter of fact software life cycles have to be adapted accordingly, important new activities like domain analysis come into the scene. The reuse of legacy code poses new challenges. Maintaining it is hard enough, reusing it and in building new software systems. Requirements on software systems change constantly. Rebuilding new systems every time requirements change considerably is neither feasible nor economical. This study must be able to incorporate old components of systems, split them into useful artifacts, and combine them with new developments. It is the wrong approach to build gigantic monolithic systems that nobody fully understands and that are hard if not impossible to adapt to new environments and situations.

Due to the ever changing and increasing requirements on software systems, researchers and practitioners have been struggling with the software crisis for decades. Software must be composed of components that can be reused and replaced. Instead of replacing a whole system every twenty years, researchers and practitioners have to continually add, remove and replace components to adapt a system to changing requirements. After twenty years everything in the system may be different, but this will have happened gradually with small changes that are manageable. Software reuse and software components will not solve all problems encountered in software engineering, but they will contribute to an important step towards more flexible software systems that are constantly evolving and adapting. Reusable, adaptable software components rather than large, monolithic applications are the key assets of successful software companies.

To build CBS, researchers can reuse many things, for example, algorithms, designs, requirements specifications, procedures, modules, applications, design patterns, architectures. Components are artifacts that clearly identify in our software systems. They have an interface, encapsulate internal details and are documented separately. In this context it is required that components be easily combined with each other, especially without knowing from each other's
built by dedicated component factories. His components were source code components and were to be used in their original form in other programs. Unlike in other engineering disciplines, software engineering has only succeeded moderately, at the best, in reuse (Prieto Diaz, 1994).

The basic approach of reusability is to configure and specialize pre-existing software components into viable application systems (Biggerstaff, 1984; Neighbors, 1984; Gogune 1986). Such source code components might already have associated specifications and designs associated with their implementations, as well as have been tested and certified. However, it is also clear that software domain models, system specification, designs, test case suites, and other software abstractions may themselves be treated as reusable software development components. These components may have a greater potential for favorable impact on reuse and semiautomated system generation or composition (Neighbors 1984). Therefore, assembling reusable software components is a strategy for decreasing software development effort in ways that are compatible with the traditional life cycle models.

The assumption in software engineering is that people use some sort of method to construct the software development (Yourdon, 1979). Software development method is a predefined and organized collection of techniques and a set of rules which state by whom, in what order, and in what way the techniques are used to achieve or maintain some objectives (Tolvanen, 1998). Most of the methods exclude, for example, maintenance, and configuration control. One big parts that (Boehm, 1976) stresses in software engineering are the management of the process. Systematic software reuse is the purposeful creation, management, support and reuse of reusable components (Jacobson et al., 1997). There exists some reuse process models (Jacobson et al., 1997), (Lim, 1998), (STARS, 1992), (Karlsson, 1995) which address the features for reuse process. In the Lim’s model reuse-process is divided in four tasks - managing the reuse process, producing assets, brokering, and consuming assets. One problem with reuse models and
tools is that most of them describe only the use or creation of components and further, reuse processes are not usually integrated to software development process (Forsell et al., 2000). At present most of the reuse texts emphasize that reuse must be systematic, i.e. it must be planned, defined and managed (Lim, 1998). The software reuse community has presented domain analysis as a way to foster reuse in software development (Tracz, 1995). Domain analysis takes place prior to actual development of software. A domain may be studied as a group of similar programs or as a business area (Wartik and Prieto-Diaz, 1992). With domain analysis, software developers try to identify general features in the domain in order to make them also pay attention to documenting it. Documentation must be done in such a way that other developers in other projects can find and use the component.

Despite the fact that reuse has traditionally meant reuse of code, it is necessary to realize that reuse can-and should-be extended too many other artifacts, too. Like Horowitz and Munson (Horowitz and Munson, 1989) state, reuse can happen at several levels and the best benefits can be achieved when, in addition to the reuse of code, requirement analysis, designs, testing plans etc are reused. Caldiera and Basili (Caldiera and Basili, 1991) argue that experience is also an important reusable resource. Reuse of experience enables other types of reuse. Lanergan and Grasso (Lanergan and Grasso, 1989) report highly successful reuse of the software's logic structures. Once software reuse provides a competitive advantage, there is little incentive to share lessons learned across organizations. There has been little research to determine if an individual organization's software reuse success factors are applicable to other organizations. Moreover, Rine (Rine et al, 1998) highlights that the majority of current information available on software reuse comes from literature, which contains unproved theories or theory applied in a limited way to a few pilot projects or case studies within individual application domains.

The basic dilemmas encountered with reusable software component include - acquiring, analyzing and modeling a software application domain, how to define
an appropriate software part, collecting or building reusable software components, configuring or composing library. In turn, each of these dilemmas is mitigated or resolved in practice through the selection of software component granularity. The granularity of the components (i.e., size, complexity, and functional capability) varies greatly across different approaches. Most approaches attempt to utilize components similar to common data structures with algorithms for their manipulation; small-grain component in and of itself does not constitute a distinct approach to software development. Other approaches attempt to utilize components resembling functionally complete systems or subsystems. The reuse of large components guided by an application domain analysis and subsequent mapping of attributed domain objects and operations onto interrelated components does appear to be an alternative approach to developing software systems (Neighbors, 1989). According to Dusink and Katwijk (Dusink and Katwijk, 1995) software reuse is the systematic application of existing software artifacts during the process of building a new software system, or the physical incorporation of existing software artifacts in a new software system. Here the term systematic means that the process of reuse is explicated and that the reusable elements are designed to be reused. More programs should be created in shorter time, with smaller costs, with higher quality, and with fewer people. In the current economic situation the model of a software product is shortening, i.e., new versions should be presented more often (Bosch, 2000).

Systematic reuse is generally recognized as a key technology for improving software productivity and quality (Lee et al., 1997). With the maturity of component technologies, more and more companies have reused their software in the form of components. Component reuse consists of two separate but related processes. The first deals with analysis of the application domain and development of domain-related components, i.e, development for reuse. The second process is concerned with assembling software system form pre-fabricated components, i.e., development with reuse. These two processes are closely related, especially in reusing in house built components. The number of
components and the ratio of reused components to total components will
determine the reuse benefits (e.g. improved productivity and quality) (Frakes,
1990; Jeffery, 1997). Software reuse can take many different forms, from ad-hoc
to systematic (Kruger, 1992). In the broad definition of reuse, it includes reusing
everything associated with software projects, such as procedures, knowledge,
documentation, architecture, design and code.

According to Biggerstaff and Richter reuse in software engineering can be
divided in composition and generation technologies (Biggerstaff and Richter,
1987). Composition techniques base reuse on the idea that component should be
understood as atomic units and be reusable more or less unchanged. An external
agent combines these components are design patterns (Gamma et al., 1995), code
skeletons or frameworks (Fayad et al., 1999), software architectures (Tracz,
1995; Bosch, 2000) and of course binary components (Szyperski, 1997), among
other things. Generative technologies weave patters or components into a program
generator. These patterns are of two main types - patterns of code and patterns in
transformation rules (Biggerstaff and Richter, 1987). Usually, in these techniques,
software is defined in a higher level language and from this definition an
application generator creates the required program. Patterns that are used to
generate the software are modified to fulfil requirements. In compositional reuse
some portion of the products from the software development process are reused.
This means that the project that creates the reusable part is not the same as the
project which uses the component.

According to (Frakes et al., 1995), a key concept of systematic reuse is the
domain, which may be defined as an application area or, more formally, a set of
systems that share design decisions. In this context, “systematic software reuse is
a paradigm shift in software engineering, from building single systems to building
families of related systems”. Software reuse has been practiced since
programming began. Reuse as a distinct field of study in software engineering,
however, is often traced to Mellroy’s paper which proposed basing the software
industry on reusable components. Other significant reuse research developments include Parnas' idea of program families and Neighbors' introduction of the concepts of domain and domain analysis.

Mili et al. define reusability as a combination of two characteristics (Mili et al., 2002) - usefulness, which is the extent to which an asset is often needed and usability, which is the extent to which an asset is packaged for reuse. Frakes et al., 1996 define software reuse as "The use of existing software knowledge or artifacts to build new software artifacts", a definition that includes reuse of software knowledge. Morisio's definition is closer to what is meant by "software reuse" i.e. reuse of building blocks in more than one system (Morisio et al., 2002). Reuse of software knowledge such as domain knowledge, or patterns may happen without reuse of building blocks and is captured in domain engineering. Frakes et al. have investigated sixteen questions about software reuse using a survey in twenty nine organizations in 1991-1992 (Frakes et al., 1995) and presents a report that most software engineers prefer to reuse rather than to build from scratch. They also did not find any evidence that use of certain programming languages, Computer Aided Software Engineering (CASE) tools or software repositories promote reuse. On the other hand, reuse education and a software process that promotes reuse have positive impact on reuse. Frakes et al. also found that the telecom industry has higher levels of reuse than some other fields.

2.5 STUDY OF COMPONENT-BASED SOFTWARE AND COMPONENT REUSE

Component-based software, software components and software reuse complement each other perfectly. Using software components to build CBS almost automatically leads to software reuse and trying to reuse software almost automatically evolves in the composition of software out of components. Software reuse can not be done without the involvement of any components. Reusing software has a much broader influence on software engineering than one might initially think. Not only does it influence the construction process, it fundamentally affects organizational structures and project structures, and it
influences legal and economic issues of software engineering. For software reuse to become a matter of fact software life cycles have to be adapted accordingly, important new activities like domain analysis come into the scene. The reuse of legacy code poses new challenges. Maintaining it is hard enough, reusing it and in building new software systems. Requirements on software systems change constantly. Rebuilding new systems every time requirements change considerably is neither feasible nor economical. This study must be able to incorporate old components of systems, split them into useful artifacts, and combine them with new developments. It is the wrong approach to build gigantic monolithic systems that nobody fully understands and that are hard if not impossible to adapt to new environments and situations.

Due to the ever changing and increasing requirements on software systems, researchers and practitioners have been struggling with the software crisis for decades. Software must be composed of components that can be reused and replaced. Instead of replacing a whole system every twenty years, researchers and practitioners have to continually add, remove and replace components to adapt a system to changing requirements. After twenty years everything in the system may be different, but this will have happened gradually with small changes that are manageable. Software reuse and software components will not solve all problems encounter in software engineering, but they will contribute to an important step towards more flexible software systems that are constantly evolving and adapting. Reusable, adaptable software components rather than large, monolithic applications are the key assets of successful software companies.

To build CBS, researchers can reuse many things, for example, algorithms, designs, requirements specifications, procedures, modules, applications, design patterns, architectures. Components are artifacts that clearly identify in our software systems. They have an interface, encapsulate internal details and are documented separately. In this context it is required that components be easily combined with each other, especially without knowing from each other's
existence. The primary intention in reusing components is that takes a component and integrates it into a software system e.g., take a procedure and use it for some computations, reuse an algorithm. But cannot simply take the algorithm and integrate it into a software system, it is necessary to implement it first. Thus reuse the idea that is described in some pseudo code and tells how can solve the problem. But to solve the problem ourselves using a specific programming language and dealing with the special characteristics of this language. This work focuses on component reuse because components are a field that promise a rich harvest in productivity through reuse.

2.5.1 Component-Based Reuse

The benefits of the Component-Based Reuse (CBR) and component approach can only be fully utilized if there is a sufficient supply of high quality software components and if they are being actually reused. Thus one of the key success factors in CBSE is the reuse of software components. The establishment of the component technologies has led to the development of a number of component marketplaces (ComponentSource). These marketplaces offer a large number of commercial and non-commercial components. Furthermore open source community is another large provider of reusable software components. There are over 60000 Open Source Software (OSS) located on (SourceForge.net), the largest website for OSS. But despite of the component technologies and marketplaces the level of reuse in software developing companies is mostly not satisfactory (Dietzsch, 2001). In the following paragraphs this study will therefore present a number of obstacles to software reuse (Oliver et al., 2004).

- Some developers have some kind of researcher attitude e.g. "do-it-yourself.
- In many cases there's a lack of motivation to provide reusable software components to other developers, since reusable components require more development effort than normal software and the developers are under high time pressure in their own projects.
• Lack of qualification i.e. there’s not enough knowledge about CBSE among the developers and project managers.

• A lot of legal problems are not ultimately solved yet, e.g. who is responsible if a system fails due to the misinterpretation of the functionality of third party components.

• The basic concepts of the business processes in different companies have not yet been unified and maybe won’t be in the near future which makes cross-organizational software reuse much more difficult.

These obstacles can be overcome by using incentive systems to encourage software reuse by providing additional education training on the topic of CBSE.

2.6 REVIEW OF SOFTWARE TESTING

Software testing process is the set of activities needed to check bugs and functionality according to the users’ requirement. The general intent of a software process is to coordinate individual activities so that they achieve a common goal (Cook and Wolf, 1998). Software testing process includes descriptions about how to found bugs, and how to manage associated development and managerial activities (Koskinen, 2000). Example of software testing processes includes Verification and Validation (V & V), testing documentation and testing strategies. Testing process in software engineering is one of the oldest forms of verification.

Thus, numerous testing techniques and tools have been used by software developers to help them increase their confidence that the software has various qualities. The ultimate goal of software testing processes is to help developers construct systems with high quality. Testing is thus used by developers of all types of systems. As technology has improved, it has become possible to apply testing techniques to larger systems. However, widespread use of systematic testing processes is not common in industry. For example, although a number of testing processes to build the testable components have been developed. For example, although the retesting of software after it is modified can be automated, many practitioners still perform this task manually. Testable components based on a well-defined test interface to the test supporting framework. Gao et al.
considered each testable bean has the following parts - A component test interfaces supporting test operations, built-in test code supporting the interaction between component Application Programming Interfaces (APIs) and test interface, components tracking interface for monitoring component operations and behaviors (Gao et al., 2001). It is our belief that a revision of the development process is a necessary precondition for the effective success of a component-based production, and this revision must also concern in particular the testing stage. In (Bertolino et al., 2002) study have highlighted that the traditional three phases of the testing process (unit, integration and system test) can be revisited in terms of three corresponding new phases, respectively referred to as component, deployment and system test.

Researchers have extended existing testing techniques for use by component providers. For example, Doong and Frankl describe techniques based on algebraic specifications (Doong and Frankl, 1994). Murphy and colleagues describe their experiences with cluster and class testing (Murphy et al., 1994), and Kung and colleagues present techniques based on object states (Kung et al., 1994). Other researchers have extended code-based approaches for use by component providers for testing individual components. For example, Harrold and Rothermel present a method that computes definition use pairs for use in class testing (Harrold et al., 1994). These definition use pairs can be contained entirely in one method or can consist of a definition in one method that reaches a use in another method. Buy and colleagues present a similar approach that uses symbolic evaluation to generate sequences of method calls that will cause the definition use pairs to be executed (Buy et al., 1999).

Researchers have considered ways that component users can test systems that are constructed from components. Rosenblum proposes a theory for test adequacy of CBS (Rosenblum, 1997). His work extends Weyuker's of axioms that formalize the notion of test adequacy (Weyuker, 1986), and provides a way to test the component from each subdomain in the program. Devanbu and Stubblebine
present an approach that uses cryptographic techniques to help component users verify coverage of components without requiring the component developer to disclose intellectual property (Devanbu et al., 1990). For example, Eickelmann and Richardson (Eickelmann and Richardson, 1996) consider the ways in which architectural specification can be used to assess the testability of a software system. Bertolino and colleagues consider the ways in which the architectural specification can be used in integration and unit testing (Bertolino et al., 1997); Harrold presents approaches for using software architecture specification for effective regression testing (Harrold et al., 1998); and Richardson, Stafford, and Wolf present a comprehensive architecture-based coverage criteria, architectural testability and architecture slicing (Richardson et al., 1998). Software system should go through at least three stages of correctness testing, namely, unit testing, integration testing, and system testing. David Garlan, Robert Allen, and John Ockerbloom presented a case study detailing the problem that arose as they built a system from reusable components. Performance problems resulted from both the system’s massive size and its complexity, which was frequently in appropriate for the tasks performed. The authors reported that they have trouble fitting selected components together; in some cases it took significant reengineering to make them interoperate properly. Maintaining the synthesized systems also proved difficult in the absence of low-level understanding.

2.7 STUDY OF COMPONENT-BASED SOFTWARE AND COMPONENT TESTING

Testing remains a fundamentally important way to check that a software program behaves as required, but a weakness of testing is that successful testing only leads to informal quality statements. Even where quantitative methods are employed, it is not clear how the objective statements (e.g. 100% code coverage has been achieved) relate to the statements that are really useful such as “the software is correct,” or “the software is reliable”. According to Dijkstra’s famous comment “Program testing can be used to show the presence of bugs, but never to show their absence!” (Dijkstra, 1972). It is a belief that a revision of the development
process is a necessary pre-condition for the effective success of a CBS production, and this revision must also concern in particular the testing stage. Briefly, in the component test phase, the component is tested in the component developer environment, to ascertain the conformance of the component to the developer specifications. However, as many authors recognized (Weyuker, 1998), the tests performed by the developer are clearly inadequate to guarantee the dependable use of the component in the final application environment, both for the component as a single element and when integrated with the other interacting components. Therefore the deployment test phase performed by the component user results composed of two sub-phases - in the first phase a selected component is evaluated directly invoking the provided Application Programming Interfaces (API); in the second phase, the component is inserted in a subsystem and then integration tested as an element of the subsystem. The last stage of system test does not show notable difference with respect to the traditional one.

2.7.1 Component-Based Testing

Component-Based Testing (CBT) is used to verify that the software conforms to its specification. In a CBS line each component of a CBS has to be tested individually. However, it is not enough to test the variants when they are first created. Each time the component variant is deployed in a new product, it has to be tested again as the component may fail to provide its services due to, faults that were already in the component gets executed due to the different user profile, the new environment might not use the component as intended by the developer and the component might depend on some services that the new environment does not provide. If the component variant is made configurable, the same will hold each time that the component is being reconfigured. Hence, the final testing cannot be done until the component has been configured and taken in use in the final product. This is challenging, since the testing then has to be performed by other people than those who developed the component. The variant typically have low testability since they are black-box entities that can be seen as state machines. As with configurability, the testability has to be built into the components. One way
to increase the testability is to use built-in tests (Binder, 1994). Built-in test have been used in electronic components in order to make complex integrated circuits self-testable. Since built-in tests are put inside the components they are able to detect errors. This is important since errors can be hidden for long time before they are exposed as failures on the normal interface of the component. Techniques that can be used to find errors in software components include assertions (Meyer, 1997; Binder, 1999) and control flow checking (Mahmood, 1998; Miremadi, 1992). Early work on applying built-in tests to software components suggested that the complete test suite could be put inside the components (Wang et al., 2000). This way the tests were constantly present and reused with the component. While this strategy might seem attractive at first sight, it is generally not a feasible solution. Many test cases are needed in order to thoroughly test a component. Having them all built-in would add too much overhead to the component. Significantly less overhead has to be added to the component if the test cases are placed outside the component and the built in tests are only used to provide the test cases with information needed in order to test the component. Examples of such built-in tests are methods for setting and reading the component’s state. Since most components are state machines, they are typically tested using CBT. Doing this effectively without being able to set and read the states is a practical impossibility (Binder, 1994). If these built-in tests are provided through well-defined interfaces, as in the component + architecture, they significantly increase the testability of the components.

From the above discussion, this study involves in at least two kinds of work. The first is the Component-Based Software Reuse (CBSR), and the second is Component-Based Software Testing (CBST). On the basis of the specification, develops test cases that can be used to ascertain the conformance of the found selected components. The vision of CBD is to allow software vendors to avoid the overheads of traditional development and testing methods by assembling new applications from high-quality, pre-fabricated components, it is expected that the overall time and costs involved in application development will be reduced, and
the quality of the resulting application will be improved. This expectation is based
on the implicit assumption that the effort involved in integrating components
during configuration and deployment is lower than the effort involved in
developing and validating applications through traditional techniques. However,
this does not take into account the fact that when an otherwise fault-free
component is integrated into a system of other components, it may fail to function
as expected. Current component technologies can help to verify the syntactic
compatibility of interconnected components, but they do little to ensure the
semantic compatibility of interconnected components, so that the individual parts
are assembled into meaningful configuration.

2.8 SUMMARY
In order to satisfy the study, this chapter deals with publications which replicate
some of the literature that subsequently gets cited in the individual
papers/manuscripts. This chapter deals with the study and review of literature of
software models, CBD, software reuse and testing, component-based software,
component reuse and testing for critical analysis and study. In this chapter review
of literature is used to review the critical points of software models and reuse
approaches on CBSE with testing. Literature reviews are secondary sources, and
as such, do not report any new or original experimental work. Study and literature
review of this study is discussed to proceed towards the research proposal by
reviewing the various approaches of researchers. Ultimate goal of this chapter is
to bring the researchers up to date with past and current literature on a topic and
forms the basis for research goal, such as future research that may be needed in
the area.