Chapter – 1
Introduction and Literature Review

1.1 Overview

During the last few years a significant change has occurred in the paradigm of software development and dissemination. Modern software's are large scale and complex. Many organizations consider implementing such software's using software components with the expectations that software components can significantly lower development costs, shorten development life cycle, producing high reliable and high stable product (Tran et al., 1997). Component based software engineering has emerged as a separate discipline which ultimately leads to software system that requires less time to specify, design, test and maintain and yet establishes high reliability requirements (Raje et al., 2001; Kallio and Niemela, 2001; Preiss et al., 2001; Szyperski, 1998). In the current chapter a sound base for the research work is developed. The organization of the chapter is as follows: Section 1.2 provides transition of software development structure from development centric to procurement centric approach. Sub-section 1.2.1 provides the description of the need and importance of the research study. Terminologies and definitions related to component paradigm have been provided in sub-section 1.2.2. Sub-section 1.2.3 covers research objectives of the work and the brief structure to accomplish those objectives. Organization of the thesis is presented in sub-section 1.2.4. In section 1.3, critical literature review is performed. This review (sub-section 1.3.1 to sub-section 1.3.4) is classified under – System Approach, Software Components Classification, Quality Modeling and Evaluation and Decision Approach for Software Components Selection. Research gaps and motivation are discussed in section 1.3.5. Finally section 1.4 provides concluding remarks of the chapter.

1.2 Introduction

In traditional software development approach, software organizations concentrated on the development of systems using conventional software engineering process models such as Waterfall, Spiral, or Iterative from scratch. These process models provided them control over all or most of the pieces of software system. To create fully functional software product no matter which process model an organization used, it performed requirements, design, architecture, construction, and integration and test activities. However, the concept of using pre-fabricated, reusable and tested software components for creating software system has changed the focus of the development centric approach assumed in traditional software development (i.e., custom development) by a procurement centric approach (Brownsword et al., 2000; Kotonya and Hutchinson, 2007).

Table 1.1 provides a general overview of this fundamental paradigm shift of software development. This change also has had an impact on nature, timing, and order of the activities and the processes performed during the life cycle of the software development. Furthermore, it can be seen that utilizing software components concept for the creation of software system is not merely a technical matter for system designers and integrators, but many other changes must be posed. To adopt component paradigm numerous technical, organizational, management, and business activities must be adapted to deal with the challenges and risks of efficiently using software components and exploiting their benefits (Voas, 1998; Moraes et al., 2007). The requirement engineering activity must also support simultaneous consideration of the system requirements and the marketplace. The development effort must be less and design quality evaluation must not be too complex.

Traditional Software Lifecycle	**Custom Development**	**Software Component Based System Development**
Requirement	Identification and creation of software system.	Identification and creation of a set of flexible requirements according to existing search pattern and software components market place information that best fit these requirements.
Design	Analysis of requirements to create structural elements, constraints and rationale to provide basis for system functionality construction.	Analysis of existing software components and integration feasibility to meet system requirements. It implies an iterative trade-off process of requirements analysis, architecture, software components availability, prioritization and negotiation.
Construction	To implement the system requirements, coding of the design is accomplished.	Requirements functionalities that are not addressed by software components are created in house in terms of glue code or wrappers. Bridges or adaptors are also created or utilized to smoothen incompatibilities in the component interfaces.

Continued...

Traditional Software Lifecycle	**Custom Development**	**Software Component Based System Development**
Testing	Based on finite set of test cases, integration and evaluation of the product quality is achieved	Less testing is required as individual software components are already tested by their respective vendors.
Maintenance	Modification of code and associated documentation due to a problem or need for improvement.	Software Components Based systems undergo a technology refresh and renewal cycle that has many implications, due to maintenance effects because of multi vendor support.

Table 1.1 Paradigm shift of software development

The academia, practitioners and researchers have shown an interest in component based development as it shortens development life cycle, reduces costs while delivering high quality complex and distributed systems. The notion of component was coined by McIlllory (1968) at the NATO workshop. Component technology today is one of the fastest growing technologies in the world as component industry is expected to grow by an average of 49 percent which is much higher than an average 14.5 percent growth rate for the software industry during the corresponding period (Weyuker, 1998). This is because developing large and complex industrial software systems with very high reliability and quality requirements entails enormous costs and the use of software components to develop such systems offers the following benefits:

- Software components designed for reuse can significantly lower development costs and shorten development cycles; and
- Using them will ultimately lead to software systems that require less time to specify, design, test and maintain, yet satisfy high quality requirements.

Many organizations see the arrival of platforms like C#(C Sharp), EJB (Enterprise Java Beans), CORBA (Common Object Request Broker Architecture) and VB.NET (Visual Basic .NET) for the specific purpose of implementing software components.

1.2.1 Need and Importance of the Study

Software system designers and developers view components as building blocks that can be easily incorporated into a software design and system to provide specified functionality. However, despite all the advantages such as - lower development cycle and costs, high quality and stable product, just-in-time development, market edge etc., (Allen, 2002; Henry and Faller, 1995; McMahon, 1995) the software component technology may introduce risks and failures in the process and system because component technology involves the acquisition and assembly of software components from different vendors and therefore it might lead to failure when unreliable components or low quality components are used in the development process. It is to be noted that this happens because a component based system is largely dependent on the quality of each component and the manner they interact with each other that comprises the system. Following case studies support this claim:

1) **Case study I** (Dowson, 1997; Nuseibeh, 1997; Le, 1997)**:** The Ariane5 rocket disaster resulted from a failure of the software component controlling the horizontal acceleration of the Ariane5 rocket. That particular component contained a small computer program that converts a 64-bit floating point real number, related to the horizontal velocity of the vehicle, to a 16 bit signed integer. The software was tested and used for the Ariane4 project without any problems. However, Ariane5 was a much faster vehicle than Ariane4 and the 16 bits allocated for the converted signed integer was no longer sufficient. This overflow error confused the control system and caused it to determine that a wrong turn had taken place. As a result, an abrupt course correction that was not needed was triggered and the disaster happened.
2) **Case Study II** (Councill, 1999)**:** An award winning design and management tool vendor (and component consumer) incorporated in its application architecture an ORBIX daemon for client to database communications. The daemon worked flawlessly in the application's first version. The producer then recommended that the daemon's consumers upgrade to a more efficient version. Trust had developed between the producer and consumer and the vendor implemented the new daemon. Customers were made aware of the upgrade and they awaited the increased

performance. The new daemon was exceptionally defective and the application's mean time between failures at some customer's sites decreased from months to hours.

3) **Case Study III** (Voas, 1998)**:** An automatic shut down of photoshop3.0 happened because Adobe team forgot to remove a time bomb that automatically shut down the program. The time bomb was an overlooked remnant of the beta test cycle.

The first case study points out to the fact that the component usage may have catastrophic results without involvement of high quality component. Also, case studies 2 and 3 identify the increased level of customer dissatisfaction by ignoring minor (important) details and the reliance on some assumptions. Therefore, quality of software components and their collaborations affect the overall quality of the system. The building of CBSS involves simultaneous consideration of many aspects such as: performing domain engineering in order to identify functional, behavioural and data components that are candidate for reuse, selection of architectural style (propagated from structural model) as per component based requirements, selection, qualification, adaptation, modification and integration of components to form sub-systems and the application as a whole. This process involves decision making for each aspect at each step. Thus there is a dire need of unified approach which takes into account all aspects concurrently leading to an effective (quality) solution. The approach should also be capable of considering decisions at all levels. Using such approach quality of software components, their collaborations and their placements in architecture can be considered concurrently and in totality. A composite quality index can be developed utilizing unified (concurrent) approach for software components and component based software system and designs. There are many mathematical models and decision techniques available in the literature such as systems approach, graph theoretic models, multi-attribute decision making models, genetic algorithms, fuzzy logic, neural networks etc. In this thesis a unified approach is developed utilizing graph theoretic models, systems approach, decision models, concurrent engineering principles and fuzzy approach. Using such unified approach effective analysis, evaluation, optimization, and selection of software component and component based software design can be achieved.

1.2.2 Component Paradigm

Component based software development (CBSD) is an approach in which systems are built from well defined independently produced pieces known as components. Some definitions emphasize that components are conceptually coherent packages of useful behaviour, while some others state that components are physical, deployable units of software which are executed within a well defined environment. Researchers have proposed several definitions for a component. Some of these are as:

- 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 interface. While a component may have the ability to modify a database, it should not be expected to maintain state information. A component is not platform constrained nor is it application bound (Sparling, 2000).
- A software component is a unit of composition with contractually specified interface and explicit context dependencies only. A software component can be deployed independently and is subjected to composition by third parties (Szyperski, 1998).
- A software component is a unit of packaging, distribution or delivery that provides services within a data integrity or encapsulation boundary (Sharma et al., 2007).
- A software component is a coherent package of software implementation that can be independently developed and delivered. It has explicit and well specified interfaces for the services it provides and for the services it expects from the others. Also, it can be composed with other components, perhaps customizing some of their properties, without modifying the components themselves (D'Souza and Willis, 1998).
- A COTS acts as pre-existing software products, sold in many copies with minimal changes; whose customers have no control over specification, schedule, and evolution; access to source code as well as internal documentation is usually unavailable; complete and correct behavioural specifications are not available (Vigder and Dean, 2000).
- A COTS product is also defined as been sold, leased, or licensed to the general public; offered by a vendor trying to profit from it; supported and evolved by the vendor, who retains the intellectual property rights; available in multiple, identical copies; and used without source code modification (Oberndorf, 1997).
- A COTS product is a commercially available or open source piece of software that other software projects can reuse and integrate into their own products (Torchiano and Morisio, 2004).

- A component is any piece of independently executable binary code written to a specification, which can only be accessed via a set of well published interfaces and which can be integrated into any kind of software application irrespective of language/platform. A component always offers a set of services via its interfaces and may be encapsulated inside a container depending on the kind of middleware technology used to develop the component (Kalaimagal and Srinivasan, 2008a).

It is to be noted that a black box component encapsulates services behind well defined interfaces which are restricted in some way due to the fact that their plug and play functionality is supported by component frameworks rather than by application domain entities. Components are meant to be reused and collaborate with others to achieve some objective thus primarily they are not used in isolation and also their composition may be governed by some specific rules of the frameworks and technology. Component technology allows software modules written in different technologies to be integrated with one another, with the help of middleware technologies such as CORBA, COM, DCOM, JavaBeans and EJB. These middleware technologies are simply a set of specifications or rules in the form of functions, which when incorporated into the code allows the software to be integrated with software developed using other platforms/languages.

Component Based Software System (CBSS) is considered as a computer based application that integrates one or more software components, while Component Based System Development (CBSD) is treated as the processes that lead to the development of a CBSS.

1.2.3 Scope and Research Objectives

In this work, the main emphasis is to develop effective methodological framework to model, analyze, and evaluate component based software systems; effective software component classification framework; quality model and evaluation methodological framework to design, and evaluate software components; and effective decision framework for software component selection. The objectives of the present research work are:

- ***To develop system model by identifying complete component based software system, sub-systems, sub-sub-systems up to component level and studying interactions among them and finally evaluating the overall component based software system.***

A systematic approach based on graph theoretic methodological framework is developed to establish unique system characteristic expression to identify influential parameters for the complete system analysis. Elements and concepts necessary to establish concrete framework to model, analyze and design component based software system are exploited and utilized by limiting composition of typical component based web application. Exhaustive modeling and analysis of typical component based web application is not considered in the current research work.

- ***To develop software component classification framework in order to explore, learn, assess, compare and evaluate software components.***

  A six dimensional classification framework is developed for the identification of appropriate software components (as per the requirement) available from the software components market place and the same approach is used in decision making process.

- ***To develop component specific quality model and evaluative methodology to evaluate software component considering characteristics concurrently.***

  A component specific quality model is developed that facilitates component (re)users to assess components. The model is for black box software components, though the model can easily be adopted by modifying certain characteristics and notions for other form of components such as open source software components. A concurrent framework based on graph theory is also developed which provides systematic approach to design and evaluate component based on quality standards. Concurrent evaluation of quality characteristics such as *reliability*, *usability* and *maintainability* of components are considered based upon developed software quality model. Reliability evaluation at the architectural level made up of heterogeneous styles such as: pipes and filters, call-back, fault tolerant are considered. There are other styles such as peer-2-peer style that are not explored in the current research work. Failure modes identified to evaluate failure index of a typical component based web application are not exhaustive but are used to demonstrate the applicability of the approach in calculating failure index. *Usability* evaluation and design of a component is performed, based upon the concept of *usability* as mentioned in the developed component quality model, by calculating *usability* index. Similarly, *maintainability* evaluation and design of a component is also performed, based upon the concept of *maintainability* as mentioned in developed component quality model, by calculating *maintainability* index.

- ***To develop concurrent decision framework for software component selection.***

  An integrated (concurrent) decision framework is developed for software component selection both for non-fuzzy and fuzzy environment.

The values used for calculating *quality, reliability, failure, usability* and *maintainability* indices and for software component selection were based on experts' subjective and objective knowledge. The values and interactive complexity might be changed based upon experts' decisions.

1.2.4 Outline of Thesis

The thesis is divided into nine chapters, each of which explores a specific topic comprehensively. Each chapter begins with a short overview of the chapter. A brief description of the chapters is as follows:

- ***Chapter 1, Introduction and Literature Review,*** provides introduction, objectives, scope and outline of the thesis. It also presents an overview of the state-of-art and state-of-practice of software components quality modeling, classification, evaluation, design and selection.
- ***Chapter 2, Modeling and Analysis of Component Based Software Systems,*** provides methodological framework based on systems engineering approach, concurrent engineering approach and graph theory to model and analyze component based software systems. Concepts of coefficient of similarity and dissimilarity have been also provided by which two or family of component based software system designs can be compared and evaluated. A case study of typical component based web application has also been discussed to demonstrate the applicability and validity of the methodological framework.
- ***Chapter 3, Software Component Classification Model,*** reviews existing classification models for software components and attempt Six Dimension Classification Strategy Framework (*SDCS*) which overcomes the shortcomings of the existing ones and acts as a reference model to learn, assess, evaluate and compare

software components with the other existing ones. The validity of the framework is done by performing an extensive survey on group of researchers, academicians and practitioners.

- ***Chapter 4, Concurrent Usability Evaluation and Design of a Software Component,*** provides a methodology based on digraph and matrix approach to provide in depth analysis of a component considering *usability* characteristic. Digraph and (permanent) matrix is utilized to analyze component *usability* by considering all sub-characteristics and attributed factors along with all the levels of interactive complexity (inter-intra) based on the concurrent approach. The concept of formation of hypothetical maximum (best) *usability* and hypothetical minimum (worst) *usability* index is also discussed. Based upon these, users can take decisions on selection, evaluation and ranking of potential candidates and wherever possible attain improvements in the component design and development as per *usability* point of view.
- ***Chapter 5, Concurrent Maintainability Evaluation and Design of a Software Component***, describes a methodology based on digraph and matrix approach for developing the *maintainability* (characteristic) index of a software component. Sub-characteristics and associated attributes of a component, which characterize *maintainability* are identified and modeled in terms of *maintainability* digraph. The nodes in the digraph represent the maintainability sub-characteristics and edges represent the interactive complexity among the sub-characteristics. A detailed procedure for the *maintainability* analysis of *component* is suggested through a *maintainability* function. The *maintainability* index (I_m) is obtained from *VPF - m* (i.e. permanent of the matrix) by substituting the numerical values of the sub-characteristics and their interactions. Concept of *hypothetical best maintainability index* and *hypothetical worst maintainability index* is also attempted which will help system developers to identify relative comparison of candidates from *hypothetical best maintainability index* and *hypothetical worst maintainability index*. Designers and developers can improve the component *maintainability* characteristic (considering critical attributed factors) by performing sensitivity analysis**.** A higher value of the *VPF - m* implies better *maintainability* of the component.
- ***Chapter 6, Concurrent Reliability Evaluation and Design of a Software Component,*** presents a graph theoretic approach for concurrent failure modes and effects analysis (*CFMEA*) of component based software systems (CBSS). Failure Modes and Effects

(*FMEA*) digraph, derived from the structure of the CBSS**,** models the effect of failures modes of the system. As failures in the systems are not independent thus the approach takes in to account CBSS structural and functional interactive characteristics complexity. *CFMEA* function (*VPF – cfmea*) is computed from permanent matrix which represents the characteristics of CBSS failure mode and effects. This leads to the identification of the various structural components of failure mode and effects. The procedure is useful not only for the analysis, but also for identification, comparison and evaluation of failure modes and effects. To evaluate and rank failure mode and effects of the system (CBSS), failure modes and effects analysis index (I_{cfmea}) is developed which is derived from *VPF – cfmea*. The method is useful and applicable both at design and operational stage. It permits analysis, identification and comparison of CBSS based on *FMEA* and provides the user directions for minimizing the failure mode and effects leading to the improvement of the CBSS *reliability*. A methodology is also developed to compute reliability of CBSS when reliabilities of its constituent elements are known.

- ***Chapter 7, Concurrent Quality Modeling and Evaluation of a Software Component,*** presents a software component quality model (*SCQM*) by overcoming shortcomings of existing quality models. The chapter also discusses a methodology for the quality evaluation of a component using digraph and matrix approach. The quality index (I_Q) is obtained from 'variable permanent quality function' obtained from 'quality digraph' which is used to evaluate and rank the quality of alternative components. Based upon these users can take decisions on selection, evaluation and ranking of potential candidates and wherever possible attain improvements in the component design and development. The validity of proposed component quality model and methodological framework to evaluate the quality of a component is performed by conducting surveys. Case study demonstrates the applicability of the framework by considering concurrent evaluation of '*reliability' 'usability'* and *'maintainability'* characteristics.
- ***Chapter 8, Concurrent Decision Approach for Software Component Selection,*** discusses the concurrent decision approach for effective selection of software components from the available pool of alternatives. Case studies have been discussed to describe the utility and dimension of the approach.

- ***Chapter 9, Conclusions and Future work,*** presents the overall conclusion of the research work by presenting the contributions made, salient features, usefulness of the methodological framework, models and approaches. It also discusses the future directions of the current research work.

The list of tables, list of figures and list of abbreviations are presented after table of contents. The references are cited in the text by author(s) name(s) with year of publication in parenthesis. In the reference section, the references are listed alphabetically by author's names, followed by initials, year of publication, title of the article, name of the journal/conference/book (abbreviated according to standard practice), volume number, and number of first and last pages. The list of publications is shown after the reference section. Appendices are labeled as A, B, C,…, etc., in the order of appearance. The brief biography of the candidate (student) and the supervisor is given in the last two pages.

1.3 Literature Review

In this section a systematic literature review is done to provide a sound basis for the current research work. The literature review is broadly classified into the following areas:

- System Approach
- Software Component Classification schemes
- Quality model and Evaluation
- Decision Approach for Software Component Selection

1.3.1 System Approach

In this sub-section literature is reviewed under following category: *systems modeling, graph theory, concurrent engineering and tools and techniques.*

1.3.1.1 System Modeling

In the recent years the growing impact of the *component technology* paradigm can be noticed, in relation to the development of customizable, cost effective, just-in-time and reusable large scale and complex software systems (Tran et al., 1997). The context of *component* based development has become very important in industry and research (Ivica et al., 2006). It is widely accepted that the component based software system (CBSS) development requires a different way of thinking than the conventional development. A CBSS is an organized collection of cooperative components representing real world entities.

The quality of CBSS is affected greatly by the complexity of the component structure in the system (Upadhyay et al., 2009; Woit, 1997). Researchers have identified that the quality of any system is a function of its basic architecture (i.e. layout and design). Systems engineering has been evolved as a novel approach to model software architectures by focusing on exogenous and endogenous interactions and dependencies of systems/sub-systems (Saradhi, 1992). In order to estimate the contribution of different attributes of the quality of the CBSS it is necessary to understand CBSS architecture. Software architecture describes the structure of software at an abstract level (Garlan and Shaw, 1993). The structure of every component, sub-system and system as a whole that is denoted by the geometry and topology, decides the quality of CBSS under any given situation. It consists of a set of *components, connectors* and *configurations*. An architectural style (Dutton and Sims, 1994; Garlan, 1995) is considered as a repeatable pattern that characterizes the configurations of components and connectors of software architectures. Many architectural styles have been identified (Shaw, 1993; Tracz, 1995) and with the need and technology many new styles are continuously emerging (Medvidovic et al., 1996; Medvidovic et al., 1997; Taylor et al., 1996). A method or model to evaluate the quality of a CBSS based on *components* and its placement in architecture/structure can certainly provide a means through which designers can configure the architecture that best fits their quality demands.

1.3.1.2 Graph Theory

One of the interesting features of the study of graphs lies in the geometric or pictorial aspect of the subject. Graphs play a significant role in solving a rich class of problems. A graph can be used to represent almost any physical or real world situation involving discrete objects and a relationship among them. The intrinsic simplicity of graph theory has rendered its applicability in numerous fields such as engineering, linguistics, physical-social, biological-sciences etc. The Königsberg bridge problem, evolved by Leonhard Euler, can be cited as one of the best example in graph theory (Biggs, 1986; Deo, 2004).

Graph theory is also considered to understand molecules in chemistry (Balaban, 1976) and physics (Enting, 1978). By gathering statistics on graph-theoretic properties related to the topology of the atoms the three dimensional structure of complicated simulated atomic structures can be studied quantitatively in condensed matter physics. In chemistry a molecule can be represented as graph, where vertices represent atoms and edges stand for bonds. In statistical physics, local connections between interacting parts of a system, as well as the dynamics of a physical process on such systems can be represented through graphs. Graph theory has also been applied to sociology (Barnes, 1969) to study behavior pattern of individual in social networks.

Graph theory has also numerous applications in the areas of mechanism and machine theory. Ambedkar and Agrawal (1987) utilized graph theory based min codes to identify kinetic chains, mechanisms, path generators and functions generators. Agrawal and Rao (1989) attempted the graph theory and matrix approach in the identification, classification and isomorphism of kinetic chains. Gandhi et al. (1991) developed an evaluative methodology to evaluate system reliability, and to evaluate and analyse the system wear (Gandhi and Agrawal, 1994). Using digraph and matrix methods Gandhi and Agrawal (1996) developed a methodology to analyse failure cause of a system. Failure Mode and Effect Analysis of mechanical and hydraulic systems was studied utilizing digraph and matrix approach (Gandhi and Agrawal, 1992). Venkatasamy and Agrawal (1996) utilized graph theoretic analysis for analyzing automobile vehicles. Wani and Gandhi (1999) developed a maintainability index for mechanical systems; Sehal et al. (2000) presented a reliability evaluation and selection of rolling element bearings; Garg et al. (2006) developed a methodology evaluate and compare power plants; Venkata and Padmanabhan (2006) developed a procedure for industrial robots selection, identification and comparison.

Graph theory is widely used in computer science to represent network of communications (Yegnanarayanan and Umamaheswari, 2010), data organizations (Deo, 2004), computation services (Tamura et al., 1996), complexity measures (Watson and McCabe, 1996, Pressman, 2005), precedence (Upadhyay, 2004) etc. In Stickney (1978) work selection of software test data is demonstrated with the help of graph theory. Alexander et al., (2006) showed that object oriented systems deal with the analysis, design and implementation of systems employing classes as modules that can be represented as directed graphs. They also mentioned that study of graph properties is valuable in many ways for understanding the characteristics of the underlying software system. Pressman (2005) used graph theory to show concurrent execution of activities in spiral model. Guo et al. (2009) discussed estimation of reusable software component by utilizing component assembly graph. The work also demonstrated the optimization of the architecture from the component level. Jenkins and kirk (2007) have analyzed software architecture graphs as complex networks. Using the graph approach they developed software metric to quantify the evolution of the stability vs maintainability of the software through various releases. Beizer (1990) described state graph as a useful way to think about software behavior and testing. Chow (1978) presented approach of testing combinations of action called “switch cover” in finite state machines. One can use *de Bruijn* sequences to generate the appropriate actions to get switch cover (Gross and Yellen, 1998). McCabe’s structural testing methodology (Watson and McCabe, 1996) based on graph theoretical complexity measuring technique became the widely used method in the complexity of the code analysis, independent logical path testing, and integration test planning and test coverage estimation in different industries. Eric et al. (2010) have discussed cognitive model for assessing complexity of software architecture. Their model is based on cognitive science and system attributes that have proven to be indicators of maintainability in practice (McCabe & Associates, 1999). Alexander et al. (2006) have reviewed applications of graph theory by looking at the identification of God classes, clustering, detection of design patterns and scale-freeness of object oriented systems.

From the literature it is clear that study of graphs and their properties is a classical subject of interest in the area of computer science. A graph structure can be extended by assigning weight to each edge of the graph. Graphs with weights, or weighted graphs, are used to represent structures in which pairwise connections have some numerical values. For example if a graph represents a web based system, the weights at each edge could represent the page retrieval time. A digraph with weighted edges in the context of graph theory is called a network. Network analysis has many practical applications, for example, to model and analyze traffic networks. Applications of network analysis split broadly into three categories:

- First, analysis to determine structural properties of a network.
- Second, analysis to find a measurable quantity within the network,
- Third, analysis of dynamical properties of networks.

It has been recognized that diagrammatic representation of software designs are important for analyzing and evaluating its overall performance characteristic (Peterson and Davie, 2004). Graph as a visualization tool puts the focus on relationships between nodes (entities such as components), for component based software system, while hiding details. Quality of software system depicts the quality of the architecture which consists of its constituent elements, their relationships and rationale for their selection. A CBSS design or architecture represented through UML diagrams (Sheldon, 2001; Coombs and Coombs, 1998) can easily be mapped to one or more graphs.

At present, there is no effective mathematical model applied to study CBSS in a comprehensive manner which could take into account all its sub-systems, sub-sub-systems up to component level concurrently. One approach having such a capability reported in the literature is Graph Theory (Harary, 1969; Deo, 2004), a systems approach. This approach has extensively been applied to a number of disciplines. It serves as a mathematical model of the system reflecting the system characteristics. Its usefulness to provide concepts, representation and methods for the system analysis has led to successful results in many fields. It has numerous advantages over all other methods. The system modeling, analysis and design of CBSS, which comprises of many systems and sub-systems, has not been attempted till now in the literature using graph theoretic systems approach.

1.3.1.3 Review of Tools and Techniques

The selection of software components is a complex task. According to the observations and findings, many decision makers and designers select software components according to their experience and intuition using subjective approach. The weakness of this approach is addressed in several research studies (Hwang and Yoon, 1981; Saaty and Vargas, 2001; Traintaphyllou and Mann, 1989; Finnie et al., 1993; Hong and Nigam, 1981; Kontio, 1996). A good software component selection strategy during designing phase of component based software system (CBSS) plays a significant role in developing final quality product. Multi Criteria Decision Making (*MCDM*) approaches were evolved in order to overcome the drawbacks of subjective approaches. This approach is based on ranking or selecting one or more software components from a pool of available alternatives with respect to multiple criteria established by stakeholders' requirements and systems/business constraints. Priority based, distance-based, outranking, weighted and mixed methods could be considered as the primary class of the current methods (Pomerol and Romero, 2000; Hwang and Yoon, 1981; Saaty and Vargas, 2001). It is almost impossible to decide which one is the best decision method. Traintaphyllou and Mann (1989) addressed a virtual paradox to judge the relative effectiveness of the *MCDM* methods. The analytic hierarchy process (*AHP*), one of the latest and most talked about *MCDM* techniques can efficiently handle the tangible as well as intangible attributes. Many research studies have shown the usage of *AHP* for software component selection during the design phase (Finnie et al., 1993; Hong and Nigam, 1981;

Kontio, 1996; Min, 1992). The BAREMO approach (Adolfo and Asuncion, 2002) explains in detail how a decision can be made based on the *AHP* (Saaty, 1996; Saaty, 1999) method. Kontino et al. (1996) suggest creating hierarchy which addresses the evaluation criteria based on *MCDM* approach. The LusWare (Morisio and Tsoukis, 1997) is a two phase approach which addresses the formal selection process and quality requirements during the evaluation process using *AHP*. All the above mentioned approaches fail to identify what to do when there are many criteria and alternatives. In *AHP* the number of pair-wise comparisons in a decision problem having *m* alternatives and *n* criteria is expressed by the following equation (Triantaphyllou, 1999):

$$\frac{n(n-1)}{2} + n\frac{m(m-1)}{2} \tag{1.1}$$

However, this becomes highly unmanageable if the criteria and alternatives are very large.

The Analytic Hierarchy Process (*AHP*) (Saaty, 1990 and Saaty, 1994) has been widely used to solve multiple-criteria decision-making problems. A hierarchical criteria is established first then pair-wise comparison matrices using a nine point-scale is created which then upon synthesis results into selection of alternatives. The pair-wise comparison converts the human preferences as equally, moderately, strongly, very strongly or extremely preferred. The uncertainty associated with the preferences i.e. decision maker’s judgments cannot be described with the help of discrete scale of *AHP* (Ayağ, 2005). The priority of one decision variable over other and construction of fuzzy pair wise comparison matrices in fuzzy-*AHP* **(***FAHP*) is accomplished by using triangular fuzzy numbers (Chan and Kumar, 2005; Ghodsypour and O’Brien, 1998).

One of the most widely adopted methods of multi-attribute decision making (*MADM*) is Technique for Order Preference by Similarity to Ideal Solution (*TOPSIS*) which is applied to varied disciplines (Agrawal et al., 1992, Bhangale et al., 2004, Jee and Kang, 2000, Prabhakaran et al., 2006, Satapathy and Bijwe, 2004, Tong et al., 2003, and Wang et al., 2000). To consider impreciseness in the decision making the method has been extended to fuzzy environment. Fuzzy multi-attribute decision making (*FMADM*) has been developed for handling the problem of inherent uncertainty and imprecision in human decision-making processes involving multiple attributes (Yeh et al., 1999).

1.3.1.4 Concurrent Engineering

Concurrent Engineering (Yeh, 1992; Rosenbalt and Waston, 1991) is a discipline which deals with concurrent processing of activities, tasks, actions and associated states to accomplish certain goal. The integration of Concurrent Engineering (*CE*) principles in software projects has made a significant contribution both in cycle time reduction and quality improvement (Sprague et al., 1991). In a *CE* environment all constraints and requirements from all disciplines are satisfied concurrently as the design progresses. This development process results in optimal design solution because the team working in parallel can rapidly verify multiple options. Each new requirement deals with quality concerns. Designing software for these concerns is a complex task. It has been argued that the software component selection for "Criteria/ *X-bilities*" is an intertwined process (Lawlis et al., 2000; Maiden and Ncube, 1998a; Chung and Cooper, 2001) for designing and building component based software systems. Huang et al. (1999) proposed a web-based product and process data modeling in concurrent "design for X-bilites". Lowenstein et al. (1990) addressed various issues concerning the implementation of concurrent engineering.

1.3.2 Software Component Classification

The selection of software component is one of the most critical activities in the CBSD as a wrong selection will increase risks of project failure drastically (Basili and Boehm, 2001; Vitharana et al., 2003a; Bhuta and Boehm, 2007). Various studies show that the selection process consists of set of different phases and strategies (Finkelstein et al., 1996; Oberndorf and Brownsword, 1997; Kunda and Brooks, 1999; Mohamed et al., 2007) and the selection process is mainly driven by two main activities:

- Searching software component candidates from the marketplace,
- Evaluating them with respect to the system requirements for taking the final decision.

It is to be noted that there is an ever growing marketplace of software component with many vendors providing several solutions with the help of software component (Vitharana et al., 2003b). These software components have partial or lack of information which creates difficulty for customer to compare and select proper software component for their system's use (Li et al., 2006; Bhuta and Boehm, 2007). To address the solution for the above problem several researchers have proposed selection techniques (Kontio et al., 1995; Morisio and

Tsoukis, 1997; Lichota et al., 1997; Tran et al., 1997; Maiden and Ncube, 1998b; Kunda and Brooks, 1999; Ochs et al., 2000; Chung et. al., 2001; Phillips and Polen, 2002; Franch and Carvallo, 2003). However, most of them concentrated on the evaluative aspect of software component selection leaving aside the searching and building of software component repository in the marketplace. It is to be noted that this lack of support affects the whole selection process. Even if the evaluation is very good it leads to high risks of failure of projects if evaluation is performed on wrong repository of software component (Neubauer and Stummer, 2007).

Software components are inherent reusable. In order to reuse software components, re-users must be able to find and understand the components that best fit their needs. If the process fails, reuse cannot happen (Frakes and Pole, 1994), or even worst it may result into the selection of some erroneous components causing critical problems to the software development project. In this context, how to classify software component so that they can be found and understood are the two important issues in enabling their efficient reuse (Bass et al., 2000; Ravichandran and Rothenberger, 2003). Software reusability deals with two aspects: firstly, developing for reuse which means developing of components so that they can be made available for reuse and secondly, developing with reuse which refers to the building of system using reusable components.

To build reuse framework we have to understand the reuse environment which enables the re-users to look for best fit components. To be identified and selected by the re-users, component should be properly classified or indexed and stored in a software repository. Using the classification or indexing re-user can search the repository for components and if they meet requirements, they can be incorporated into new applications. The whole environment can be depicted in Figure 1.1 which enables re-users to obtain good results.

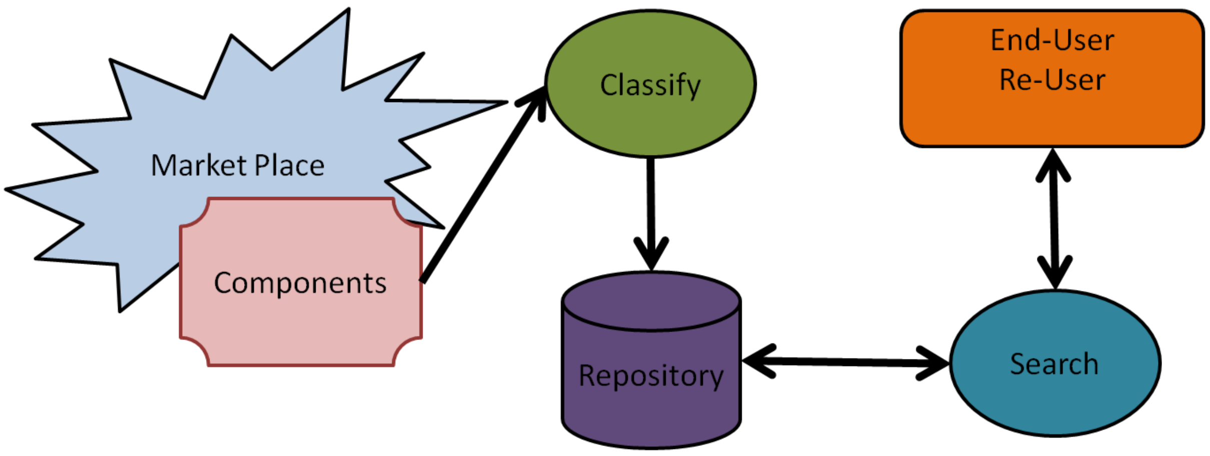

Figure 1.1 Reuse environment

Classification (indexing) is central to the software reuse practice thus critical for CBSD (Prieto-Díaz, 1987). A well-defined classification structure is essential to the design of an effective storing, searching and retrieval mechanism. The advantages of classification in CBSD are many and some of them are mentioned below:

- It provides an easy to use mechanism for the organization and identification of software component.
- It provides filtering of appropriate software component for Just-in-time development by improving the information retrieval systems' classical measures such as- *Recall* and *Precision*. *Recall* is the number of relevant items retrieved over the number of relevant items in the database. *Precision* is the number of relevant items retrieved over the number of all items retrieved.
- It enables end users to understand technology and creates knowledge map for better understanding of software components.
- It also serves as a comparison tool to compare different software components that fall under homogeneous structure.

In the literature various researchers have proposed several classification mechanisms in order to achieve efficient and reliable selection process. Some of the research works have been concentrated on the classification schemes for the reusable components e.g., (Prieto-Díaz, 1991; Glass and Vessey, 1995). The area of software component classification has recently emerged. Thus it becomes a cornerstone of CBSD and several recent works arrange software component by means of attributes for identifying relationships between characteristics of products and their impact on CBSD.

An initial attempt to classify software component was done by Carney and Long (2000). They used a bi-dimensional cartesian space. The dimensions that they defined are *origin* and *modifiability*. They also reported some examples that populated this space. The origin dimension represented the way the product is produced. For that they proposed following values: *independent commercial item*, *special version of commercial item*, *component produced by contract*, *existing components from external sources*, *components produced in-house*. The *modification* dimension described the scope of product to be modified by the system developer that uses the component. This attribute comprised of five possible values: *extensive reworking of code*, *internal code revision*, *necessary tailoring and customization*, *simple parameterization*, *very little* or *no modification*. Two of them assumed access to code (*extensive reworking*, *internal code revision*), to (*necessary tailoring*, *parameterization*) implied some mechanisms built into the software component to modify its functionality. One of the major shortcomings of their work is that no distinction can be found between what needs to be modified in order to make a product work and what can be modified in order to better integrate it into the delivered system.

A more complex classification of software component was presented by Morisio and Torchiano (2002) which was the extension of the work proposed by Carney and Long (2000). In their work they emphasized that different research works often adopt different implicit definition of software component, thus making difficult comparing them and evaluating the applicability of proposed approaches. The purpose of their proposal was twofold: firstly, it was a tool to precisely define the meaning of a software component software component and secondly, it represented a way of distinguishing different sub-classes of products in order to characterize them better. They proposed ten attributes, grouped into four areas: *source*, *customization*, *bundle*, and *role*. Though their work was extensive but it could not relate to architectural context, domain taxonomy and product functionality related features. This proposal is similar to (Torchiano et al., 2002; Jaccheri and Torchiano, 2002) which emphasized the assessment of the reuse of attributes.

The software component Acquisition Process (*CAP*) (Ochs et al., 2000) provides a more general framework for product characterization. The main aim of *CAP* is to reduce effort needed for characterization and provide the basis for reusing the information acquired during the process. The process is composed of three main aspects: *initialization*, *execution* and *reuse*. *COCOTS* models (Abst et al., 2000) could also be used as a driver for software component classification. To understand marketplace some of its cost drivers could be used to identify software component categories: product maturity, supplier willingness to extend product, product interface complexity, supplier product support, and supplier provided training and documentation. It is to be noted that these factors are though important but are not related to technology. In CBSD the integration problem of software component is addressed by several researches; in particular the work by Yakimovich et al. (1999) is most important as in order to estimate integration effort they proposed a set of criteria for classifying software architectures. Using same characteristics classification of both components and systems is possible. Egyed et al. (2000) proposed a methodology for evaluating the architectural impact of software components. Utilizing such methodology the selection of components and architectural styles become possible. The key point of the methodology is the identification of architectural mismatches. Following criteria have been used to compare the most relevant classification schemes and the result is shown in Table 1.2.

- Domain Specific: It represents whether the approach is addressing a specific domain or it is used for general domains or not.
- Characterization Schema: It represents whether the approach is describing the attributes used to classify the components or not.
- Classification Schema Evolution: It represents whether the approach is addressing effective mechanisms to evolve the classification schema to deal with the constant growing and evolution of the software component marketplace.
- Reuse: It represents whether the approach is addressing reusability aspect of software component.

From Table 1.2, it can be realized that the proposals cited before do not fully resolve the problems of classifying software component neither for performing efficient searching and retrieval mechanisms, nor for reusing knowledge gained about software component.

Research work	**Domain Specific**	**Characterization Schema**	**Classification Evolution**	**Reuse**
Yakimovich et al. (1999)	×	Not clearly defined	×	×
Carney and Long (2000).	×	Origin and Modifiability	×	×
Egyed et al., (2000)	×	Not clearly defined	×	×
Morisio and Torchiano (2002)	×	Categories of source, Customization, Bundle and Role	×	×
Abst et al., 2000	×	Not clearly defined	×	×
Torchiano et al., 2002;	×	Set of general attributes similar to ISO 9126	×	Partial
Jaccheri and Torchiano, 2002	Partial	Kind, Architectural, level and phase	×	Partial

Table 1.2 Classification schemes analysis

It can be clearly seen that though the existing approaches have shown various ways of representing and understanding software component, they lack in accounting the users requirements, architectural context, and evolution of the domain and trends of the marketplace. The most important point in learning about software component and its evolution in ever growing market place is to define a proper set of attributes and then to collect information about these attributes. The selection of software component in an industrial context is clearly depends upon project specific goals. The time spent on learning about software component in an industrial context is quite expensive. Thus the motivation for research work is to develop software component classification framework which will be useful for learning and understanding its usage. Using the framework end user can get the benefit of assessing, comparing, evaluating and learning software components as per project goals.

1.3.3 Quality Modeling and Evaluation

Quality as perceived by both acquirers and end-users identify the business value of a software product. Therefore, quality is very critical to the product, since its absence results in dissatisfied users and financial loss, and may even endanger lives (Garvin, 1984). Software development organizations, in general, are not best equipped to deal with it as they do not have at their disposal the quality related measurement instruments that would allow (or "facilitate") the engineering of quality throughout the entire software product life cycle, even less when CBSD is used (Carvallo et al., 2007).

The state of art literature yet does not provide a well established and widely accepted description scheme for assessing the quality of software products (Behkamal et al., 2009). Various quality models have been evolved since 1976 and each of these quality models consists of a number of quality characteristics (or factors as called in some models). These quality characteristics could be used to reflect the quality of the software product. In the literature various definitions exists for the term “quality” in relation to software:

- The degree to which a system, system component, or process meets specified requirements and customer (user) needs (expectations) (IEEE Std 610.12, 1990).
- A set of characteristics and sub-characteristics, as well as the relationships between them that provide the basis for specifying quality requirements and evaluating quality (ISO 9126-1, 2001).
- Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software (Pressman, 2005).
- The totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs (ISO/IEC 14598-1, 1999).
- The existence of characteristics of a product which can be assigned to requirements (Petrasch, 1999).

McCall’s quality model (McCall, 1977) for software product is one of the well known quality models in the literature of software engineering. It originated from US air-force electronic system decision (*ESD*), the Rome air development center (*RADC*) and General Electric (*GE*) and primarily aimed towards the system developers and the system development process. McCall’s model combines eleven criteria around product operations, product revisions, and product transitions (Fizpatrick, 1996). One of the major contributions provided by this model is the consideration of relationships between quality characteristic and metrics. There has been a criticism that not all metrics are objective and the issue of product functionality is not considered. Boehm model (Boehm et al., 1978) is similar to the McCall model in that it represents a hierarchical structure of characteristics, each of which contributes to the total quality. In Boehm model the main emphasis is on the *maintainability* aspect of a software product. Boehm's model looks at utility from various dimensions, considering the various users who are expected to work with the system once it is delivered. Boehm's notion includes user's needs, as McCall does. It also adds the hardware yield characteristics not encountered in the McCall model. However, Boehm's model contains only a diagram and does not elaborate the methodology to measure these characteristics. Dromey (1995) proposed a quality evaluation framework taking into the consideration relationship between the attributes (characteristics) and the sub-attributes (sub-characteristics) of the quality. Dromey model is the product based quality model in which the quality evaluation differs for each product and a more dynamic idea for modeling the process is needed to be wide enough to apply for different systems. This model suffers from lack of criteria for measurement of software quality and it is difficult to see how it could be used at the beginning of the lifecycle to determine the user needs. The FURPS originally presented by Robert Grady (Khosravi and Guehneuc, 2004), was extended by IBM Rational Software (Jacobson et al., 2000; Krutchen, 2000) into FURPS+, where the '+' indicates requirements such as design constraints, implementation requirements, interface requirements and physical requirements (Grady, 1992). In this quality model, two different categories of requirements were identified:

- *Functional requirements* (*F*)*:* Defined by input and expected output.
- Non-functional requirements (*URPS*): Usability, reliability, performance and suitability.

One disadvantage of the *FURPS* model is that it fails to take into account the software product's *maintainability*, which may be important criterion for application development especially for component based software systems (CBSS). The Bayesian belief network (*BBN*) is a special category of graphical model which represents quality as root node. Root node is connected to other quality characteristics via directed arrows (Stefani et al., 2003;

Stefani et al., 2004). Similarly, each characteristic is connected to sub-characteristics. This model is useful to manipulate and represent complex quality model that cannot be established using conventional methods. However, this model fails to evaluate fully software product due to lack of criteria involvement. Different perspectives of software quality can be represented by star model (Khosravi and Guehneuc, 2004). Even though it considers various viewpoints of quality it does not evaluate fully software product due to lack of criteria involvement. The ISO 9126 (2001) is a part of ISO 9000 standard, which is the most important standard for quality assurance. In this model, the totality of software product quality attributes is classified in a hierarchical tree structure of characteristics and sub-characteristics. The highest level of this structure consists of the quality characteristics and the lowest level consists of the software quality criteria. The model specifies six characteristics including *functionality, reliability, usability, efficiency, maintainability and portability;* which are further divided into 21 sub-characteristics. These sub-characteristics are manifested externally when the software is used as part of a computer system, and are the result of internal software attributes.

Bertoa's model (Bertoa and Vallecillo, 2002a) is a well known initiative to define the attributes that can be described by software component vendors (no matter whether they are external or internal providers). The model defines the characteristics and sub-characteristics in the change context of component based software systems. The characteristics were discriminated into local characteristics, global characteristics, runtime characteristics and product characteristics. Bertoa's model is just a mere adoption of ISO 9126, the only difference being that the *portability* and *fault tolerance* characteristics disappear together with the *stability* and *analyzability* sub-characteristics. Two new sub-characteristics: *compatibility* and *complexity* have been added. Although the research work presents a good description on quality characteristics, sub-characteristics and their measurements, it fails to perform any empirical evaluation of the attributes on any application, thus leaving the proposed work as incomplete. Also, *portability* and *fault tolerance* which have been eliminated are very significant to software components. The purpose of Alvaro's model (Alvaro et al., 2005a) is to determine which quality characteristic should be considered for the evaluation of software component. This model also follows ISO 9126 model. A few sub characteristics have been added and some exiting sub-characteristics have been removed. According to (Alvaro et al., 2005b), *scalability* is relevant in order to express the ability of the component to support major data volumes. *Self-contained* is an intrinsic property of a component and must be analyzed. *Configurability* becomes essential for the developer to analyze if the component can be easily configured. *Reusability* is important for the reason that software factories have adopted component based approaches on the premise of resue. The *maintainability* and *analyzability* sub-characteristics have been removed from ISO 9126. A high level characteristics '*Business'* have also been added with following sub-characteristics: *development time, cost, time to market, targeted market* and *affordability.* The model is similar to Bertoa's model but provides better footprints as the model has introduced a number of components specific quality characteristics or sub-characteristics like *self-contained, configurability* and *scalability*. Though Alvaro's model can be treated as a major step for component quality model but it also has some drawbacks. Firstly, *reusability* has been treated as quality attribute rather than quality factor. Secondly, the definition of *scalability* is ambiguous. *Scalability* as defined in their model indicates only about data volume and not the maximum number of components. In it the component can interact with other components without reducing performance. In Rawedah's model (Raweshdah and Matalkah, 2006) standard set of quality characteristics suitable for evaluating software components along with newly defined sets of sub-characteristics associated with them are identified. The sub-characteristics *fault tolerance*, *configurability*, *scalability* and *reusability* have been removed. New characteristic *manageability* with sub-characteristics *quality management* has been added. The model also attempts to match the appropriate type of stakeholders with the corresponding quality characteristics. It can be clearly seen that no improvement has been made from the previous models. The sub-characteristics that have been removed are significant to components. Also this model does not explain how the attributes belonging to various characteristics and sub-characteristics will be measured to finally evaluate the quality of the system.

Following sub-sections provide literature review on evaluation and design of *usability*, *maintainability* and *reliability* aspects of a software component which will be utilized in chapter 4, chapter 5 and chapter 6 respectively.

1.3.3.1 Usability Evaluation and Design

The existing literature offers several definitions of *usability*:

- The capability of the software product to be understood, learned, used and be attractive to the user, when used under specified conditions (ISO/IEC 9126, 2001).

- The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use (ISO 9241-11, 1998).
- The ease with which a user can learn to operate, prepares inputs for, and interprets outputs of a system or component (IEEE Std. 610.12, 1990).
- Usability of a software product is the extent to which the product is convenient and practical to use (Boehm et al., 1978).
- The probability that the operator of a system will not experience a user interface problem during a given period of operation under a given operational profile (Fenton and Pfleeger , 1998).

Existing literatures have not much discussed about the *usability* aspect of a component's quality. Bertoa et al. (2005) definition of *usability* is adapted for software components as "*the capability of software component to be understood, learned, used and be attractive to the user, when used under specific conditions".* According to them under specific conditions are similar to "*context of use*" of ISO/IEC quality model. This leads to the fact that *usability* as a quality characteristic is intrinsically dependent upon the kind of "*use*" that is expected and the kind of user that will use the component. Thus in order to evaluate *usability* of a component its characteristics or dimensions should be able to consider the relationships between component, user and use aspects, as the level of *usability* depends upon each of them. The use aspects are included in *usability* evaluation because *usability* is not a unique property of a component in isolation. That means not only the users who use a component but also the use aspects (tasks) that the users perform with a component should also be analyzed in the *usability* evaluation. It is to be noted that the three elements (*component, user and use aspects*) are the principal factors of the *usability* evaluation.

1.3.3.2 Maintainability Evaluation and Design

Maintainability is a broad concept and thus needs further specification for the proper understanding. Several existing quality models (McCall et al., 1997; Boehm et al., 1978; ISO/IEC-9126, 1991; Dromey, 1995; Sedigh-Ali et al., 2001a; ISO 9126, 2001) supports *maintainability* characteristic of a software application but none of them addresses the concerns of component based software systems (specifically to components) directly in a detailed manner. The existing literature offers several definition of maintainability:

- The level of easiness to understand, modify and retest the software product (Boehm et al., 1978).
- The capability of a software product to be modified. Modification may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications (ISO 9126, 2001).
- Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment (IEEE Std 1219, 1998).

Conventional measure of *maintainability* deals with mean time to *repair* (MTTR) which is inefficient to satisfy maintenance demands of CBSS due to the unavailability of the visible code. It is difficult to identify whether the problem is in the component itself or in the system or may be interaction among components. The primary reason for this is that the evolution and upgrades for the individual software components are outside the direct control of system developers and acquisition organizations (Vigder and Dean, 2000). It is to be noted that if due to new requirements system needs to be upgraded, the compatibility of the new system with the existing one may vanish. This results into ripple effect on other components which leads to the upgradation or deployment of new components with the desired compatibility. Voas (1998) has highlighted several issues regarding software component based systems such as – frozen functionality, incomplete upgrades (such as added features or bug fixes that, while independently reliable, are incompatible with the host system); defective or unreliable software component; and complex or defective middleware (such as wrappers). The paper also suggests some guidelines regarding CBSS in order to minimize overhead in *maintenance* activities:

- Avoid building mini-systems rather build up large scale and complex systems where reusability is a major concern.
- Keep detailed requirements documentation on each component for better understandability.
- Keep up to date the information repository for storing suitable components.
- Keep two similar components in the repository if competing applications share a component but cannot tolerate changes the other might need.

It can be seen that Voas study on maintenance of CBSS**,** though theoretical, provides an insight to understand and measure software components.

Judith et al. (1998) also address challenges for maintenance of components and CBSS:

- Incompatible upgrades due to the modification or replacement of existing components (based upon new requirements). Incompatibility that can also arise with the underlying hardware. This will lead to substantial amount of effort to modify system in order to achieve desired functionality.
- The required changes for the modified component such as tools and languages may not be supported by the software system.
- There can be overhead on performance and security measures due to modification of a component.

It is to be noted that in their study the major issue is about changeability of a component. The concentration is on the ability of a component to be upgraded, backward compatible, adaptable and testable.

Vigder and Dean (1998) throw light on various other challenges in order to effectively maintain and manage software component based system:

- Reconfiguring of components – addition, deletion and replacement, results in an extensive maintenance activity.
- Customizing of components in order to achieve required functionality depends upon vendor supported tailoring techniques (parameterization) for the products and inclusion of glue code.

Existing literatures have limited scope on the *maintainability* aspect of component's quality (Alvaro et al., 2005a; Alvaro et al., 2005b; Simão and Belchior, 2003; Bertoa and Vallecillo, 2002a; Bertoa and Vallecillo, 2002b; Goulao et al., 2002a, Goulao et al., 2002b). More or less every emergent component quality model is an adaptation of existing quality model (ISO/IEC) considering quality aspects – *functionality, efficiency, reliability, usability, maintainability and portability.* ISO 9126 is a generic model for any software product and caters to the need of software industry to standardize the evaluation of software products in a more promising and suitable way. Thus the model should be customized to cater to the needs of component oriented user as per *maintainability* point of view.

1.3.3.3 Reliability Evaluation and Design

The reliable assurance activities are becoming as an integral part of design process of the software systems. Customers are placing increased demand on software community for just-in-time, stable and reliable product (Tran et al., 1997). This challenge can be realized through the use of component based software development approach, which promotes compositional development and component reuse (Medvidovic et al., 1997). Reliability assessment provides an insight into the key areas of the software system and highlights potential problem areas that can be dealt with at the design stage of software development life cycle (LaCombe, 1999; Teng and Ho, 1996; Hawkins and Woollons, 1998). It allows comparison to be made among the alternative competing designs. Reliability of a CBSS is defined as the probability of performing its intended function satisfactorily under given condition for desired period of time. Reliability of systems made of software component is affected greatly by their complexity that is influenced directly by the number of components and their interactions in the system (Woit, 1997). The structure of every component, sub-system, and system as a whole that is denoted by the topology and geometry, decides the reliability of a real system under any given situation. Reliability of complex systems is generally obtained by splitting the system into a set of series and parallel systems, as available methods are capable of calculating the reliability of series and parallel systems only (Lyu, 1996). In the present age of advanced technology and increased complexity it is not possible to represent a real system into simple sets of series and parallel components. Software system made of large number of components and sub-systems can be represented by a general graph (Upadhyay et al., 2009). It means reliability of a real CBSS depends upon the reliability of all sub-systems forming complete CBSS.

Software designers are motivated to integrate software components for rapid software development. To ensure high *reliability* for such applications using software components it is necessary to assess the *reliabilities* of such systems by investigating the architectures, the testing strategies, and the component reliabilities (Lyu, 1996 ; Musa et al., 1987; Musa, 1998). In the field of software *reliability* modeling (Musa et al., 1987; Musa, 1994) the validity of the execution time theory and operational profile was investigated first. In (Gokhale and Trivedi, 2002; Gokhale, 1998) authors assumes the failure behaviour of each component as time-dependent failure intensity. The total number of failures is obtained and the *reliability* is estimated via the enhanced non-homogeneous Poisson process (ENHPP). Everett (1999) uses the extended execution time (EET) reliability growth model and several test cases to model the *reliability* growth of each component. Yacoub et al. (1999) proposes a reliability analysis technique called the *scenario-based reliability analysis* (SBRA), which is based on execution scenarios to derive a probabilistic model for the analysis of a component-based system. It has been identified that the *reliability* of a component-based software system is evaluated using *reliabilities* of its components (Krishnamurthy and Mathur, 1997).

Identification of critical sub-systems, components, failure mode, failure mode and effects etc., at early stage, helps in realizing the reliable product. Various tools are available that the designers can use to attain the objective such as failure effect analysis (*FEA*), failure mode and effect analysis (*FMEA*), fault tree analysis (*FTA*), event tree analysis (*ETA*), cause effect analysis (*CEA*), cause consequence diagrams (*CCDs*), etc. The predominant among all the tools is *FMEA* which is used to systematically identify and investigate potential system weaknesses (Arnzen, 1967; Collaxcott, 1977; Jüttner et al., 2000; Kara-Zaitri et al., 1991; Ormsby et al., 1991; Stamatis, 1995). It consists of a methodology for examining all the modes in which a system failure can occur and it also studies the seriousness of these effects. To automate the analysis process various researchers have also used the matrix form of *FMEA* and have applied it to various applications (Barbour, 1977; Legg, 1978; Jordan, 1972; Kumamto et al., 1981; Reifer, 1979). Thus the main research thrust in this area is to identify tools and techniques for the early detection and estimation of failure and/or reliability of software system.

1.3.4 Decision Approach for Software Component Selection

The methodology suggested by Brownstein and Lerner (1982) in order to evaluate software conducts three main activities - firstly, review of the planning guidelines in order to assess user requirements, secondly, identification of necessary activities and thirdly, estimation of the resources needed for the evaluation. Talley (1983) discussed general guidelines for the selection and evaluation of administrative software. Criteria development and partitioning as a main technique is addressed in the work of Edmonds and Urban (1984). Various works have taken care of different methods and techniques for software evaluation and selection such as - generic domain methodology (Frankel, 1986), experts rating of candidate software as inputs (Anderson, 1989), classification method based on the products' quality (Eskenasi, 1989), usage of automated tools (Meier and Williamson, 1989), creation of evaluation criteria and alternatives assessment (Subramanian and Gershon, 1991), criteria and alternatives evaluation (Williams, 1992), initial product screening and benchmarking (Adeli and Wilcoski, 1993), and quality checklist driven evaluation (Jeanrenaud and Romanazzi, 1994). Kontio et al. (1995) present the Off-The-Shelf-Option (*OTSO*) method for reusable component selection. The IusWare (IUStitia softWARis) (Morisio and Tsoukis, 1997) approach tried to formalize the selection process, and to address quality requirements during the evaluation process. It relies on multi-criteria decision aid (*MCDA*) (Ncube and Dean, 2002) to select software components. The PRISM (Portable, Reusable, Integrated, Software Modules) (Lichota et al., 1997) approach proposed a generic component architecture that can be used during the software component evaluation process.

Tran et al. (1997) have proposed the software component based integrated system development (*CISD*) model for the selection of multiple homogeneous software component. It is based on waterfall approach. Maiden and Ncube (1998a) suggested a template approach known as procurement oriented requirements engineering (*PORE*) to define requirements that depend on evaluating software components and usage of *AHP*. Feblowitz and Greenspan (1998) presented a scenario based technique to manage the task of making software component selection decisions by considering enterprise-level factors. Kunda and Brooks (1999) in their approach, *STACE* (Social-Technical Approach to COTS Evaluation), emphasized the importance of non-technical issues when defining the evaluation criteria and conducting the evaluation process. The *CRE* (*COTS*-based RE) approach emphasized the importance of non-functional requirements (*NFR*) frameworks (Chung et al., 1999) as decisive criteria when comparing software component alternatives. Lai et al. (1999) have discussed the selection of multimedia authorizing systems (*MAS*) to facilitate the group-decision making with the applicability of *AHP* in problem solving. Merad and Lemos (1999) have described game theoretic solution for the problem of the optimal selection of software components with respect to their non-functional attributes. Jung and Choi (1999) in order to develop modular systems have proposed two optimization models for selecting the best software component among alternatives for each module.

Ochs et al. (2000) have proposed the COTS acquisition process (*CAP*) which emphasizes the concept of a “tailorable evaluation process” for software components. Teltumbde (2000) have presented a structured methodology incorporating participatory learning and decision-making processes based on Nominal Group Technique (*NGT*) and the evaluation methodology adopting the *AHP* to evaluate enterprise resource planning (*ERP*) projects. COTS-Aware Requirements Engineering (*CARE*) approach (Chung and Cooper, 2001; Chung and Cooper, 2002; Chung and Cooper, 2004a; Chung and Cooper, 2004b) uses a flexible set of requirements based on different agents' views. Lawlis et al. (2001) have proposed a requirement driven formal process for evaluating software components. Alves and Castro (2001) have presented the COTS based requirements engineering (CRE) method, which emphasizes on requirements to assist the software component selection process. The *PECA* (Plan, Establish, Collect, and Analyze) approach (Dorda et al., 2002) describes a detailed tailorable software component selection process. The BAREMO approach explains in detail how a decision can be made based on the *AHP* (Saaty, 1990; Saaty, 1994) method. The storyboard approach (Gregor et al., 2002) relies on use-cases and screen-captures during the requirements engineering phase to help customers understand their requirements, and thus acquire more appropriate software components.

The combined selection approach (Burgues et al., 2002) selects multiple software components by two way method- local scale and global scale. Sedigh-Ali et al. (2001a) suggest a metrics-based approach that employs cost of quality and capability maturity models for cost-benefit analysis of software component based systems. The *CEP* (Comparative Evaluation Process) approach (Phillips and Polen, 2002) have introduced the use of the so-called confidence factor (CF) for software component selection. Morera (2002) have emphasized on methodologies govern by decision making techniques to evaluate software component. Lai et al. (1999) have proposed usage of *AHP* to support the selection of a multi-media authorizing system in a group decision environment. Sahay and Gupta (2003) software component evaluation method is based upon software solution merit index (*SMI*). The WinWin spiral model (Boehm et al., 2003) which is a risk-driven approach uses a decision framework to provide guidance for the software component based development decisions, e.g. make-or-buy, software component selection, software component tailoring, glue-coding, etc. Erol and Ferrell (2003) have suggested the use of fuzzy theory to quantify qualitative data, and then to use optimization techniques to determine optimal (or near optimal) solutions from a finite number of alternatives.

Based on the work done (Franch and Carvallo, 2003), the DesCOTS (Description, evaluation and **s**election of software component) system was developed (Grau et al., 2004) which includes a set of tools that can be used to evaluate software component based on quality models (ISO/IEC9126, 2001). Wei and Wang (2004) have presented a comprehensive framework based upon both objective and subjective data to select a suitable *ERP* project. Colombo and Francalanci (2004) have described a hierarchical ranking model based on *AHP* to help the selection of customer relationship management (*CRM*) packages based on their functional and technical quality. Cil et al. (2005) developed a Web-based collaborative system framework for knowledge management and decision making on a special organizational problem. Wei et al. (2005) have presented a comprehensive group decision based framework for selecting a suitable *ERP* system. Various potential gaps such as - inability to measure uncertainty, lack of control over understanding and sharing information/knowledge and stakeholders involvement and decision support tools are also identified in most of the selection approaches (Wanyama and Far, 2006). Some new developments in multiple attribute decision making (*MADM*) and multiple criteria decision making (*MCDM*) methods have been highlighted for software selection and investment decisions (Hu and Tsai, 2006; Bernroider and Stix, 2006). Lin et al., (2007) have proposed a fuzzy based approach for software component selection.

Table 1.3 summarizes the pertinent gaps in the software selection approaches:

- None of the approaches has taken in account concurrent engineering (*CE*) principles i.e. in totality consideration of criteria along with their interactions/interdependencies at all levels.
- Very few have taken care of uncertainty but again rely on single decision technique thus becomes cumbersome to use when criteria and alternatives are large in number.
- There is a lack of support for the creation/evolution of critical selection factors/criteria for the selection of software component**.** All the approaches select criteria on the basis of intuition, perception and experience.
- Group decision technique has been addressed in a very few approaches only.

Features / Selection Approaches	Environment		Concurrent Engineering	Decision Technique Type(s)	Single Decision Technique	Multiple Decision Technique	Result		Group Decision
	Crisp (c)	Fuzzy (f)					Crisp (c)	Fuzzy (f)	
(Brownstein and Lerner, 1982)	√			WSM	√		√		
(Talley, 1983)	√			Heuristic	√		√		
(Edmonds and Urban, 1984)	√			WSM			√		
(Franke, 1986)	√			Heuristic	√		√		
(Anderson, 1989)	√			Heuristic	√		√		
(Eskanasi, 1989)	√			Heuristic	√		√		
(Meier and Williamson, 1989)	√			Heuristic	√		√		
(Subramanian and Gershon, 1991)	√			Electre I	√		√		
(Williams, 1992)	√			Heuristic	√		√		
(Adeli & Wilcoski, 1993)	√			WSM	√		√		
(Jeanrenaud and Romanazzi,1994)	√			Heuristic	√		√		
(Kontio et al., 1995)	√			AHP	√		√		
(Morisio and Tsoukias, 1997)	√			MCDM	√		√		
(Lichota et al., 1997)	√			---	√		√		
(Tran and Liu, 1997)	√			First-fit	√		√		
(Maiden and Ncube, 1998)	√			MCDM	√		√		
(Feblowitz and Greenspan,1998)	√			Heuristic	√		√		
(Lai et al., 1999)	√			AHP	√		√		
(Merad and Lemos, 1999)	√			Game theory	√		√		
(Jung and Choi, 1999)	√			0/1 Integer programming	√		√		
(Kunda and Brooks, 1999)	√			AHP	√		√		
(Phillips and Polen, 2002)	√			----	√		√		
(Chung et al., 1999)	√			WSM	√		√		
(Ochs et al., 2000)	√			AHP	√		√		
(Teltumbde, 2000)	√			AHP	√		√		
(Lawlis et al., 2001)	√			Heuristic	√		√		
(Altes and Castro, 2001)	√			WSM	√		√		

Continued…

Features / Selection Approaches	Environment		Concurrent Engineering	Decision Technique Type(s)	Single Decision Technique	Multiple Decision Technique	Result		Group Decision
	Crisp (c)	Fuzzy (f)					Crisp (c)	Fuzzy (f)	
(Sedigh-Ali et al., 2001 (a-b))	√			Heuristic	√		√		
(Chung and Cooper, 2001, 2002, 2004 (a-b))	√			-----	√		√		
(Morera, 2002)	√			AHP	√		√		
(Lai et al., 2002)	√			AHP	√		√		√
(Dorda et al., 2002)	√			-------	√		√		
(Adolfo and Asunction, 2002)	√			AHP	√		√		
(Gregor et al., 2002)	√			--------	√		√		
(Burgues et al., 2002)	√			------	√		√		
(Sahay and Gupta, 2003)	√			Heuristic	√		√		
(Erol and Ferrel, 2003)	√	√		Fuzzy OFD, Pre-emptive 0-1 goal programming		√	√	√	
(Boehm et al., 2003)	√			MCDM	√		√		
(Wei and Wang, 2004)	√	√		Fuzzy set theory	√		√	√	
(Colombo & Francalanci, 2004)	√			AHP	√		√		
(Grau et al., 2004)	√			AHP	√		√		
(Cil et al., 2005)	√			MCDM	√		√		
(Wei et al., 2005)	√			AHP	√		√		
(Bernroider and Stix, 2006)	√			MADM	√		√		
(Hu and Tsai, 2006)	√			AHP	√		√		
Lin et al. 2007	√	√		MCDM	√		√	√	

Table 1.3 Comparative analyses of software component selection techniques

Above mentioned points indicate that not a single available work deals with integration of decision techniques and concurrent engineering principles for the software component selection process comprehensively for fuzzy and non-fuzzy environments.

1.3.5 Gaps and Motivation

In a critical literature review it is seen that though component paradigm has offered several benefits but still its proper usage is in infant stage as proper quality standard is not available. Most of the standards are generic and cannot be applied "*as is*" to component paradigm. Thus first motivation of the thesis is to establish quality standard for software component. Component oriented paradigm based upon *part* and *connector* approach helps to design, develop, evaluate, assess, rank and select architectural designs, components, software systems and strategies. A system engineering approach is well fitted in designing and developing component based software. The approach is apt to develop a unique system characteristic by which component based software system or family of component based software systems can be compared and appropriate decision can be taken by performing sensitivity analysis at the early stage of life cycle. This process can be followed to compare component and/or family of components. As the component market supports components from different vendors and in different versions applicable to different domains from generic to specific, it is difficult to identify components for the specific requirement. This forms the second motivation for the creation of classification framework in order to assess, evaluate and learn component and associated technologies.

Some of the characteristics of a component provide a clear distinction of component standard from conventional quality standard such as *usability*. Thus, this research thesis emphasizes on the evaluation and the design of three important characteristics such as – *usability, maintainability* and *reliability* of a software component thereby creating good quality component based software system. In addition to this, since the selection of a component has a critical effect on the overall project, a wrong component selection may increase the risks and may lead to disaster. This provides third motivation to create decision framework for the selection of software component in non-fuzzy and fuzzy environment. Human nature and decisions are often vague thus to avoid vagueness fuzzy theory is utilized to incorporate fuzziness in decision making. Accomplishment of activities, tasks and actions together reduces development time, costs and provide interdisciplinary approach to tackle the problem. This becomes an important motivation to include concurrent engineering principles in component paradigm for modeling, analyzing and designing component based software system. This approach facilitates simultaneous functioning of different concerns at the same time.

A systematic graph theoretic approach is developed to create system characteristics expression. The approach is utilized to evaluate and design individual characteristics of CBSS and also to evaluate and design multiple characteristics concurrently. The thesis illustrates functioning of each approach with appropriate case studies. It is believed that current research work provides an effective approach to model, analyze, design, evaluate and optimize software component and component based software systems.

1.4 Concluding Remarks

In this chapter, introduction of the research work is given by mentioning component related terminologies, impacts and benefits. This is followed by research scope and objectives necessary for doing the current research work. Later critical literature review has been done to identify the research gaps and motivations to do current research work. The critical literature review has been categorized under four significant aspects: *System Approach, Software Component Classification*, *Quality modeling* and *Evaluation* and *Decision Approach for Software Component Selection.*

In the next chapter, modeling, analysis and evaluation of CBSS is described. Coefficient of similarity and dissimilarity is also developed by which two or family of CBSS can be compared and evaluated. A case study of typical component based web application is considered to demonstrate the application of the developed methodological frameworks.