Chapter 4

Quality Assurance Framework for Component Based Software Development

The only liberty I mean, is a liberty connected with order; that not only exists along with order and virtue, but which cannot exist at all without them.

E. Burke

The last two chapters ‘Component Based Software Engineering: a state of art’ and ‘Survey on Quality Assurance in CBSE’ are mainly an overview of CBSE and Quality Assurances literature. Through this study we have come to this conclusion that quality is a main aspect in CBSE and that is required to be improved. We realize that Quality Assurances should not be considered only within the aspect of reliability, security, performance, maintainability etc but development cycle is also one of its important factor which cannot be ignored. Component Based Software quality is a controversial subject for research community. With the large application developed by component based software engineering, and the increasing recognition of quality risks involved with using third-party components, need arises for separate quality models in component based software systems. A third party component applied in different domains using numerous types of environments plays various roles. The quality provided by a third-party component does not usually address the quality
requirements of all possible application types in all environments. It also depends on the attributes of the component quality and environment where they are used.

The quality attributes and quality model have not been of primary interest and have not been explicitly addressed by the research community. They have instead been treated separately from the component based software engineering.

At the same time one question arises in our mind that 'How! At what level the quality evaluation will satisfy the developer, integrator and user. For this we have to discuss its scope and goal.

4.1 Scope and Goal

In this study, our intention is to demonstrate the diversity of quality attributes and the different assumptions which should be considered for constructing quality model attributes from the requirement of actors involved i.e, developer, integrator and user. Measuring the degree of complexity of a system is one technique for evaluating the quality. Quality evaluation in component based software systems is becoming more and more desirable for component integrators and users. This study tries to address this problem by using matrix structure. In this study, we propose a new matrix that shows the quality based complexity of Component Based Software. To make it possible, we first identify all type of relationship between quality characteristics of component based software system and then present a dependence based representation called the quality matrix.
This study lists the basic quality attributes and quality models. It is interesting to know which quality attributes are suitable for inclusion in quality model. By keeping these concepts in our mind we propose a new Quality model known as a Component Based Quality Model (CBQM). It incorporates ten new characteristic. Our model is divided into quality attributes required by different actors. In this study, we have also discussed a number of activity areas that form a life cycle framework for component-based software development. In this context we are proposing a life cycle model for component based software engineering called Umbrella Life Cycle Development Model (ULCDM). It provides a component based software engineering life cycle model perspective on testing or verification in different phase.

4.2 Quality Framework

Day by day software applications are getting more and more complex. Developers have to manage the trade-off between short development life and user needs stable and reliable software systems.

The situation gets worse when applications are used by people who do not have technical knowhow. Technicians often find logical and stable solutions when they use them. They are aware of the processes in the background, the relations between them and the resulting operations. Users generally do not have technical background; they often face problems with the systems.

Component-Based Software Development (CBSD) approaches are based on software reuse by selecting appropriate components and then assemble them. It can significantly reduce development cost, time-to-market; simultaneously improve
maintainability, reliability and overall quality of software systems. However, quality assurance technologies for Component-Based Software System must address two inseparable questions:

1) Component quality measurement.

2) Measurement of quality of software systems based on components.

To answer these questions, models should be developed to define the overall quality of components and systems; metrics should be found to measure the size, complexity, reusability and reliability of components and systems [Gill2003]. Component requirement analysis involves the process of requirement gathered from users, select or construct best one according to requirement, design specification, validating and managing the requirements for a component.

Software developed with component based technologies has unique feature that make it different from traditional software process. The traditional quality models are inadequate for Component Based Software process, for instance, the dependence between the state of the component, interface between them and the behavior of that component in specified environment [Gill 2004]. A component may prove fulfilling requirement in one application in a particular operating environment, but the same component may not be considered at all in a completely different application. But before analyzing these aspects we have to go through from the component quality characteristics.

There are many desirable software quality attributes. Broadly software quality can be classified into two groups [Ghezzi1991]
(i) External Versus Internal

The external qualities are visible to the users of the system and internal quality is related with the developers.

(ii) Product Versus Process

Here product means end product of process. In the software engineering product means 'software' and process mean various phases of software life cycle.

4.3 Definition

In general, there is no consensus yet on how to define and categorize software component quality characteristics. The American Heritage Dictionary defines quality as "a characteristic or attribute of something" as an attributes of an item, quality refers to measurable characteristics.

According to the IEEE 610.12 standard, software quality is a set of attributes of a software system and is defined as [IEEE1990]:

(1) The degree to which a system, component, or process meets specified requirements.

(2) The degree to which a system, component, or process meets user's needs or expectations.

(3) Quality comprises all characteristics and significant features of a product or an activity which relate to the satisfaction of given requirements.

*Putting the Pieces Together In Reference of Component Based Software Engineering*  85
ISO 8402 has defined Quality Model [ISO1986] as: “The set of characteristics and the relationships between them which provide the basis for specifying quality requirements and evaluating quality”

We can identify that there are two major attributes or factors while discussing the meaning and definition of (software) quality [Hoyer 2001]:

(i) **According to specification:**

Quality is defined as a matter of systems and services whose measurable characteristics must conform to defined specification.

(ii) **According to user’s requirements:**

Quality, which is identified independent of any measurable characteristics, must be according to customer expectations – explicit or not.

Here we will try to follow as much as possible a standard terminology, in particular the one defined by ISO/IEC 9126. The ISO/IEC 9126 is a generic software quality model and it can be applied to any software product by tailoring it to a specific purpose. The main drawback of the existing international standards is that they provide very general quality models and guidelines, and are very difficult to apply to specific domains such as component based software development (CBSD).

A quality characteristic is a set of properties of a software system by which its quality can be described and evaluated. A characteristic may be refined into multiple levels of sub-characteristics.
A Quality model is the set of characteristics and sub-characteristics, as well as the relationships between them that provide the basis for specifying quality requirements and for evaluating quality.

Different quality models are exiting with identification of different perspective. Through these quality models different quality characteristics have been identified according to its actor's priority.

4.4 Existing Software Quality Models

4.4.1 McCall Model

Jim McCall produced this model for the US Air Force and the intention was to bridge the gap between users and developers. He tried to map the users view with the developer's priority. McCall identified three main perspectives for characterizing the quality attributes of a software product [McCall1977]. These perspectives are:-

(a) Product revision, (b) Product transition and (c) Product operations

(a) Product revision

The product revision perspective identifies quality factors that influence the ability to change the software systems, these factors are:-

- Maintainability

It can be defined as the ability to find and fix a defect.

- Flexibility

Flexibility can be defined as the ability to make changes required as dictated by the business.
• Testability

It can be as the ability to validate the software requirements.

(b) Product transition

The product transition perspective identifies quality factors that influence the ability to adapt the software to new environments:

• Portability

Portability can be defined as the ability to transfer the software from one environment to another.

• Reusability

It can be defined as the ease of using existing software components in a different context.

• Interoperability

Interoperability in the system can be defined as the extent, or ease, to which software components work together.

(c) Product operations

The product operations perspective identifies quality factors that influence the extent to which the software fulfills its specification:

• Correctness

Correctness of the system can be defined as matching of functionality with the specifications.
Chapter 4

- **Reliability**

It can be measured as the extent to which the system fails.

- **Efficiency**

It is ratio of the system resource (including cpu, disk, memory, network) usage with the output.

- **Integrity**

It can be defined as the protection from unauthorized access.

- **Usability**

Usability of the system can be defined as the ease of use.

In total McCall identified the 11 quality factors broken down into three perspectives detailed as given below:

**Table 4.1 McCall Quality Model**

<table>
<thead>
<tr>
<th>Product Operations</th>
<th>Product Transition</th>
<th>Product Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Correctness</td>
<td>• Portability</td>
<td>• Maintainability</td>
</tr>
<tr>
<td>• Reliability</td>
<td>• Reusability</td>
<td>• Flexibility</td>
</tr>
<tr>
<td>• Efficiency</td>
<td>• Interoperability</td>
<td>• Testability</td>
</tr>
<tr>
<td>• Integrity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Usability</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

McCall quality model consider software system as a product like other engineering product. The major advantage of this model is the relationship between quality attributes. The main drawback of this model is its silence for other important
quality attribute like ‘Functionality’. The other limitation of this model is its only consideration of eleven characteristics. This model is also not applicable with respect to the IEEE Standard for a Software Quality Metrics.

### 4.4.2 Boehm Model

Barry W. Boehm [Boehm 1978] also defined a hierarchical model of software quality characteristics, described software quality as a set of attributes and metrics. At the highest level of his model, Boehm defined three primary uses, these three primary uses are:

- **Maintainability**: ease of identifying what needs to be changed as well as ease of modification and retesting.

- **Portability**: ease of changing software to accommodate a new environment.

- **Utility**: the extent to which the as-is software can be used.

These three primary uses had quality factors associated with them, representing the next level of Boehm’s hierarchical model. Boehm identified seven quality factors, namely:

- **Portability**: the extent to which the software will work under different computer configurations.

- **Reliability**: the extent to which the software performs as required, i.e. the absence of defects.

- **Efficiency**: optimum use of system resources during correct execution.
Chapter 4

- **Usability**: ease of use.

- **Testability**: ease of validation, that the software meets the requirements.

- **Understandability**: the extent to which the software is easily comprehended with regard to purpose and structure.

- **Flexibility**: the ease of changing the software to meet revised requirements.

These quality factors are further broken down into Primitive constructs that can be measured. As with McCall's Quality Model, the intention is to be able to measure the lowest level of the model.

**Table 4.2** Boehm Quality Model

<table>
<thead>
<tr>
<th>Maintenability</th>
<th>Utility</th>
<th>Portability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testability</td>
<td>Reliability</td>
<td>Flexibility</td>
</tr>
<tr>
<td>Understandability</td>
<td>Efficiency</td>
<td></td>
</tr>
<tr>
<td>Modifiability</td>
<td>Human Behavior</td>
<td></td>
</tr>
</tbody>
</table>

Major drawback of Boehm Model is its silence in respect of measurement of quality attributes. On the other hand Boehm quality model considers human nature, this is its positive aspect.

### 4.4.3 FURPS/FURPS + Model

The FURPS model was originally developed by Robert Grady in Hewlett Packard and later extended by Rational Software -now IBM Rational Software.
FURPS, the following characteristics are identified [Grady1992]:

- **Functionality**

The F in the FURPS+ acronym represents all the system-wide functional requirements. These usually represent the main product features that are familiar within the business domain of the solution being developed. The functional requirements can also be technically oriented. Functional requirements that may consider to be architecturally significant. Each of these may represent functionality of the system being developed and they are each a system-wide functional requirement.

- **Usability**

Usability includes looking at, capturing, and stating requirements based around user interface issues, things such as accessibility, interface aesthetics, and consistency within the user's interface.

- **Reliability**

Reliability includes aspects such as availability, accuracy, and recoverability.

- **Performance**

Performance involves throughput, system response time, recovery time, and startup time.

- **Supportability**

Supportability includes a number of other requirements such as testability, adaptability, maintainability, compatibility, configurability, installability, scalability, localizability, and so on.
The FURPS define quality attributes as of two different types: Functional and Non-functional. These attributes can be used as both software requirements as well as in the assessment of software quality.

### 4.4.4 ISO 9126 Software Quality Model

ISO 9126 is an international standard for the evaluation of software quality. The standard is divided into four parts which addresses, respectively, the following subjects: quality model; external metrics; internal metrics; and quality in use metrics. ISO 9126 Part one, referred to as ISO 9126-1 is an extension of previous work done by McCall, Boehm, FURPS, and others in defining a set of software quality characteristics. The ISO 9126-1 software quality model [ISO 2001] identifies 6 main quality characteristics.

<table>
<thead>
<tr>
<th>Table 4.3 ISO 9126 Quality Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functionality</strong></td>
</tr>
<tr>
<td>Suitability</td>
</tr>
<tr>
<td>Security</td>
</tr>
<tr>
<td>Accuracy</td>
</tr>
<tr>
<td>Interoperability</td>
</tr>
<tr>
<td>Compliance</td>
</tr>
</tbody>
</table>

These characteristics are broken down into sub characteristics. The main characteristics of the ISO9126-1 quality model, can be defined as follows:-
(a) Functionality

Functionality is expressed as a totality of essential functions that any software systems provide. It is also important to note that the presence or absence of these functions in software systems can be verified as either existing or not. In other terms, it expresses the ability of a component to provide the required services, when used under specified conditions;

- Suitability

This is the essential Functionality characteristic and refers to the appropriateness of the functions of the software.

- Accuracy

This refers to the correctness of the functions.

- Interoperability

A given software component or system does not typically function in isolation. This sub characteristic concerns the ability of a software component to interact with other components.

- Compliance

This sub characteristic addresses the compliant capability of software.

- Security

This sub characteristic relates to unauthorized access to the software functions.
(b) Reliability

Reliability can be defined as the ability of the component to maintain a specified level of performance, when used under specified conditions.

- Maturity

This sub characteristic concerns frequency of failure of the software.

- Recoverability

To bring a failed system, to full operation.

- Fault tolerance

Recovery of software from its failure.

(c) Usability

Usability can be defined as the ability of a component to be understood, learnt, used, configured, and executed, when used under specified conditions.

- Learnability

Learning effort for different users, i.e. novice, expert, casual etc.

- Understandability

Determines the ease of which the systems functions can be understood.

- Operability

Ability of the software to be easily operated by a given user in a given environment.
(d) Efficiency

Express the ability of a component to provide appropriate performance, relative to the amount of resources used.

- **Time behavior**

Characterizes response times for a given throughput, i.e. transaction rate.

- **Resource behavior**

Characterizes resources used, i.e. memory, CPU, disk and network usage.

(e) Maintainability

The ability to identify a fault within a software component. Express the ability of a component to be modified. Maintainability is impacted by code readability or complexity as well as modularization. Anything that helps with identifying the cause of a fault and then fixing the fault is the concern of maintainability.

- **Stability**

Characterizes the sensitivity to change of a given system that is the negative impact that may be caused by system changes.

- **Analyzability**

The ability to identify the root cause of the failure.

- **Changeability**

Characterizes the amount of effort to change a system.
• Testability

Characterizes the effort needed to verify (test) a system change.

(f) Portability

Portability can be defined as how well the software can adapt to changes in its environment or with its requirements.

• Installability

Characterizes the effort required to install the software.

• Conformance

Similar to compliance for functionality.

• Replaceability

Characterizes the plug and play aspect of software components, that is how easy is it to exchange a given software component within a specified environment.

• Adaptability

Adaptability can be defined as the ability of the system to change to new specifications or operating environments.

One of the major advantages of this model is identification of the internal and external quality attributes of software. On the other hand, it does not show how these attributes can be measured.
4.4.5 Component Quality Model (CQM)

The model is based on ISO/IEC 9126, contains marketing characteristics and some relevant component information's which is not supported in other component quality models. One can argue that ensuring the quality of component-based systems is much more difficult than is the case with any products. The problem is that the components may well be of uncertain quality and their uses and behavior may be only partially known. There are many potential problems that may affect the assessment of the quality properties of components and of component-based systems. One way of categorizing these problems is as follows:

(i) Related to atomic components.

(ii) Related to component interoperability.

(iii) Related to component compositions.

(iv) Related to acquisition risks.

4.5 Component Based Quality Model Framework

There are several existing quality model used to evaluate software systems in general, however, none of them is dedicated to component based systems. Component based software systems quality can be determined from several different perspectives, including the level of satisfaction from the users, developers and integrators side.

An analytical study has been carried out on several existing software quality models, namely: McCall's, Boehm, ISO 9126 and FURPS. Looking to these
aspects which has been discussed at Para 4.2/N to Para 4.4/N. A new quality model for component based software system known as CBQM has been proposed which is discussed at Para 4.6/N to 4.7/N.

First of all we have developed the quality attributes relationship model of the component based systems using the matrix form.

4.6 Quality Matrix

We propose a matrix based approach to define dependency between different quality attributes. From this matrix we easily find most of attributes are interlinked, that means each attributes directly or indirectly affects others. The quality of component based software system not only depends on quality of individual components but also the interface or interdependency among them. Since quality of component based software system depends heavily on the application context in which they are used, setting quality matrix of each component may be difficult task. For this reason, Dependency Relation Quality Matrix of Component Based Software System should be constructed based on the quality attributes [Dixit2008c].

4.6.1 Why Formal Approach

In software engineering, formal methods refer to mathematically based techniques for the specification, analysis and development of software systems. Indeed, correctness of software systems can be improved by formalizing different products and processes in the life-cycle. In this section we propose software quality dependency matrix. In
the first matrix we define relationship between user, developer and integrator for quality interest. In the second matrix we construct relationship with quality attributes.

4.6.2 Matrix Representations

If the property of a component \( c_i \) is dependent on another component \( c_j \), then \( \text{Matrix}[i, j] = 1 \). More formally, the values of all elements in \( AM_{n \times n} = (d_{ij})_{n \times n} \) are defined as follows:

\[
\begin{align*}
    d_{ij} &= 1 \text{ if } v_i \neq v_j \\
    &= 0 \text{ if otherwise.}
\end{align*}
\]

Component based software development paradigm involves three actors' i.e, Developer, integrator and User. Everyone has their own interest, which is represented by matrix structure.

4.6.3 Level of Interest

(i) Integrator ('I')

Integrator's point of view, the components are like a black box with a very well defined interface.

(ii) Developer ('D')

Developer's point of view, the components are just like a function or module, according to the user's requirement or generic need it is developed.

(iii) User ('U')
According to the user components are just like others software, he is not interested to know how it is developed and where it is developed, he is only interested in functionality shows in desired environment.

\[
\begin{array}{ccc}
D & I & U \\
D & 1 & 0 & 0 \\
I & 1 & 1 & 0 \\
U & 1 & 1 & 1 \\
\end{array}
\]

**Figure 4.1** Matrix Representation of Quality Interest

Here: D, U and I denote Developer, User and Integrator respectively in the Component Based Software Systems. In the above mentioned level, developer is only concerned with the development of that specific component. He may not know that where this component will be used and for what purpose. On the basis of required functionality, he develops the component. It means developer has no idea about the interest of user as well as integrator. This essence is shown in 1st Row of our matrix by using 1 for developer interest and his nil interest for integrator and user shown in second and third column of matrix. The integrator selects the required component from component repository, it means integrator has interest in individual component’s quality created by different developers and concerned with his own creation. The same is defined by using 1’s in first two column of second row.

The last row shows user’s concern with both contributions of integrator as well as developer which is defined by using 1’s in all three column of third row. In the Component Based Software Development process generally we use third party component which is developed by those persons in most cases who never use final
system. It means developer only interested in fulfilling the required general functionality. In the other part integrator is only interested in specification provided by the user of systems.

Through this matrix we can also define the relationship exists among various quality attributes.

4.6.4 Quality Attributes Relationship Matrix

Here F, M, U, E, R and P are respectively used for Functionality, Maintainability, Usability, Efficiency, Reliability and Portability attributes of Component Based Software Systems.

<table>
<thead>
<tr>
<th></th>
<th>F</th>
<th>M</th>
<th>U</th>
<th>E</th>
<th>R</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>M</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>ud</td>
</tr>
<tr>
<td>U</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>1</td>
<td>ud</td>
</tr>
<tr>
<td>R</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>1</td>
</tr>
<tr>
<td>P</td>
<td>1</td>
<td>ud</td>
<td>1</td>
<td>ud</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

**Figure 4.3 Qualities Attributes Matrix**

(i) In this matrix we get dependent relation between quality attributes of software.

(ii) Here 1’s represent these is some relation may be direct or indirect between quality characteristics.
(iii) Ud shows unknown

(iii) X represents undefined

These matrix relationships can also be represented by six faces of hexagon.

**4.6.5 Hexagonal Relationship of Quality Attributes**

![Hexagonal Relationship between Quality Attributes](image)

**Figure 4.3** Hexagonal Relationship between Quality Attributes

Hexagon all the six edge work as a six attributes of the quality model and relationship with the others attributes. F, R, U, E, M and P are denotes quality attributes i.e, functionality, reliability, usability, efficiency, maintainability, and portability respectively.

Apart from the functional aspect’s evaluations of software component, component’s quality evaluation is one of the most concerning issue of research. We realize the need to make correction in already existing quality model as they were mainly concerned with the traditional software systems. Now it is required to involve more characteristic in particular component based quality model [Dixit2009e].

---

*Putting the Pieces Together In Reference of Component Based Software Engineering* 103
4.7 Component Based Quality Model (CBQM)

Most of the research dedicated to software components is focused on their functional aspects i.e. component specification, component development, component tests, etc. In our ongoing research, we are concerned with the evaluation of components quality. This evaluation should be performed using a component quality model. However, there are several difficulties in the development of such a model, such as:

(1) Which quality characteristics should be considered

(2) How we can evaluate them

(3) Who should be responsible for such evaluation?

There is a lack of an effective assessment of software components quality. Besides, the international standards like ISO, IEEE that address the software products’ quality issues have shown to be too general for dealing with the specific characteristics of software components. While some of their characteristics are appropriate to the evaluation of software components, others are not well suited for that task.

Component based software engineering mainly involves three groups of involvement:

1. User
2. Integrator
3. Developer

All the three groups have different quality interest. Previous developed quality models concerns with the traditional software development. Component based
software development process is much different from traditional software development. In the traditional software development process, software is developed from the scratch i.e, software developer take requirement from user and according to his requirement he starts to do so. Component based software development offers focus of software reuse. Component based software systems is developed by three different categories of component

1. External Component libraries

(a) Open Source Components or Pre Built Component generally known's as a third party component.

(b) Commercial off-the-shelf (COTS) Components

2. In House Component, component constructed from scratch.

3. Mining Components from legacy systems.

First category of component is much used in component based software systems. In the component based software development process, developer likes to construct component for generic user’s requirement. It means component does not develop for any particular requirement. According to user’s requirement integrator combine these prebuilt components well known as a third party component. Therefore quality definition and quality attributes are different for all the three actors i.e, user, developer, and integrators.

This makes a need for making correction in various quality models proposed for traditional software engineering. As a result of this analysis, a new quality model has
been built that supports quality characteristics suitable for all the three actors associated with component based development. The new model defines sets of new characteristics and avoids some of the limitations found in the existing models. Our proposed Component Based Quality Models (CBQM) tries to solve different quality interests.

Table 4.4 CBQM (Component Based Quality Model)

<table>
<thead>
<tr>
<th>User</th>
<th>Developer</th>
<th>Integrators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Certification</td>
<td>Extendibility</td>
<td>Modularity</td>
</tr>
<tr>
<td>Simplicity</td>
<td>Correctness</td>
<td>Compatibility</td>
</tr>
<tr>
<td>Performance</td>
<td>Reliability</td>
<td>Performance</td>
</tr>
<tr>
<td>Economic</td>
<td>Portability</td>
<td>Complexity</td>
</tr>
<tr>
<td>Functionality</td>
<td></td>
<td>Maintainability</td>
</tr>
<tr>
<td>Efficiency</td>
<td></td>
<td>Portability</td>
</tr>
<tr>
<td>Usability</td>
<td></td>
<td>Reliability</td>
</tr>
</tbody>
</table>

These are new characteristics added to proposed quality model which have not been incorporated in the previous models [Dixit2009e].

Certification, Simplicity, Performance, Compatibility, Modularity, Extendibility, Correctness, Robustness, Economic, Complexity
1. Simplicity

Simplicity is the ability to model real world problems in intuitive software structures.

2. Modularity

Modularity is the ability to decompose an application in single modules, which communicates only over defined interface.

3. Economic

Economic is the ratio of financial benefits with its cost.

4. Complexity

Complexity is software metric that provides a quantitative measure of the program.

5. Correctness is the exact representation of a software product with respect to its specification.

6. Robustness is the ability of software to respond appropriately to abnormal or modified conditions.

7. Extendibility is the ease of extending software products to new changes of specification or other problem domains.

8. Compatibility is the ease of combining software elements with others.

9. Performance is the capability to predict the time exposure of an application depending from the software architecture, amount of processed data and
hardware and in following internal factors.

10. **Certification** is the guarantee provided for software quality by third party.

    As previously discussed quality of component based software system can be assured when their development phase which is termed as life cycle is considered. Only considerations of reliability, security, performance etc are not enough to assure the quality of any software.

### 4.8 Component Based Software Life Cycle Framework

As in the other engineering activities software process also go through a series of development phases. The series of phases (progresses) through which the software undergoes from requirement exploration to maintenance is termed as a software life cycle. Software systems come and go, through a series of phases or activities that starts from the Requirement collection, Initial prototype, Testing, Maintenance, Operation, Upkeep, and Retirement. The process provides interaction between user of software and developer as the medium for communication, each new round of interaction eliciting more useful knowledge from the users. Building computer software is an iterative learning process, and the outcome is called software systems.

In the early days of software construction process was followed in sequential flow known as waterfall model but in late 70s a drastic change has been seen in software industry and number of variations are introduced. A software life cycle model can be chosen based on the nature of application and environment. The sequential models define a sequence of activities in which one activity follows after a completion of the
previous one. Evolutionary models allow performance of several activities in parallel without requirements on a stringent completion of one activity to unable to start with another one. CBSE has received considerable attention in research community recently since it promises to bring to software development in less cost and time. During component based software development the software goes through a series of phases i.e., requirement, specification design, selection, retrieval, testing and maintenance etc...

Many software developers experience difficulties with integrating third party components into systems. This is particularly the case for 'binary' components with not accompanying source code. In practice, components delivered by component providers only include specification of the interface. To test software component is a problem since the source code of the software component under test may not be available for the third party user. This imposes significant difficulties in adequate testing of integrated component based systems, without source code, many testing techniques will not be applicable. Much research focus on component specification to achieve the advantage of the component based approach in theory. The previous research works pay least attention to component implementation, which is the realization of the specification to construct a reliable component based system. However, in the real world, the component implementation is not that easy to achieve and testing of this implementation till now is open task for research community. A software life cycle is usually defined as a vague concept. The life cycle phase of every software paradigm is different. Some process will spend large time and cost as compared to other phases. Software development process is an expensive activity.
The development process or life cycles usually start with requirement gathering from the users of system. All the traditional software life cycle models are sequential type it means each phase must be completed before next phase is started. Well known example of sequential models is waterfall model, or V model, and of evolutionary models, iterative and incremental development, or spiral model.

Component Based Software Engineering is integration-centric approach, emphasizing on the selection, acquisition and integration of components from third party or in-house sources. How to verify software component characteristic is a great challenge for research community. A lot of research efforts have been devoted to the analysis and design methods for Component Based Software. There are differences between component based software life cycle and traditional software development life cycle. Majority of component based application are developed by third party component. It means component based software life cycle start with retrieval not from construction. In the component based systems, component just like binary object are provided by third party without source code. Therefore a need arises to verify specification given by third party [Capretz2004].

In every software development life cycle process, some activities are important these are considered as the foundation steps in Quality Assurance. In this study we describe the life cycle model intended to support basic component Based Software Engineering activities [Crnkovic 2005].

4.8.1 Prerequisite for Umbrella Life Cycle Models

- Where should be testing done?
The testing is not a separate phase, but rather an activity that takes place all the way through software production. The requirements have to be tested, the specification document has to be tested, and the design has to be tested, and so on. There are times in the process when testing is carried out to the almost total exclusion of other activities. This occurs towards the end of each phase (verification) and is especially true before the product is handed over to the client (validation). Although there are time when testing predominates, however, there should never be time when no testing is being performed. If testing is treated as a separated phase, then there is a real danger that testing will not be carried out constantly through every phase of the product development and maintenance process.

Prescriptive models are used as guidelines or frameworks to organize and structure how software development activities should be performed, and in what order. The success of any software depends upon good design. Like any other product process software systems also need some steps to assure quality of software. Generally these phase are used in every software life cycle model [Pressman1997].

- Software Process Model

More precise and formalized descriptions of software life cycle activities – a networked sequence of activities, objects, transformations and events

- Requirement

Requirement analysis is the first phase in most of software life cycle models. It is a process after a feasibility study has been performed. In the requirement analysis phase we decide user's desire in terms of functionality, performance, behavior and
interfacing. Exact requirement of the system is documented. In this phase user of the system or customer and developer are closely interact to decide all these things.

- Specification

Once the developer understand the user's requirements in next phase specification document is drawn. In the specification phase as compared to informal requirement phase a detailed specification document is constructed which describe the functionality of the software. This phase is sometimes split into two sub phases:

(a) Architectural

(b) Detailed design.

- Design

In the design phase we translate requirements into a representation of the software.

- Implementation

In the implementation phase design is translated into a machine readable form.

- Verification

Once code is prepared, program verification begins.

- Maintenance

Once a software system passes all the verification and validation, it is delivered to the user and enters in the maintenance cycle. Any modification made to the system after initial delivery is part of this phase.
Before proposing Umbrella model first of all we need to clarify limitations of exiting development models. These are also our motivational factors for introducing Umbrella concepts.

4.8.2 Existing Software Life Cycle Models

4.8.2.1 Waterfall model

The waterfall model [Royce1970] was first introduced by Royce in 1970. It is linear model where various phase are interconnected so that output of one phase become subsequently input for next phase. The Waterfall model 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.

It is first published model for software life cycle.

Challenges

1. Major drawback of Waterfall Model is its inflexible division of phase. Today requirements are changing very fast so that all phases are overlapped.

2. In the waterfall model, outputs are verified in last stage.

3. The waterfall model is good for such situation where requirement are well defined.
4.8.2.2 Incremental Model

To overcome drawback of waterfall model a cycle model know as incremental model was developed. In the incremental model we combine the property of sequential model and iterative characteristic of prototype life cycle model. In the incremental model partial implementation of system is constructed after refining these implementations. After that required functionality can be achieved. Rapid application
development (RAD) is an incremental software development process model that emphasizes an extremely short development life cycle.

**Challenges**

It is time consuming process. User of the system is involved in the whole process.

**4.8.2.3 Evolutionary model**

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 uncontrollable hacking.

**4.8.2.4 Prototyping model**

Generally customer of the software system initially define broader requirement of the systems. In this case a quick prototype can be constructed. Prototyping provides constructive feedback to designers and potential users so that the system requirements can be clarified and refined early during software development. This prototype is evaluated by the customer and refines his requirement.

**4.8.2.5 Spiral Model**

The spiral model [Boehm1988] was first developed by Boehm, is an evolutionary software life cycle model. In the spiral life cycle model different phase are represented as a spiral rather than sequence of activities. The Spiral model makes software development more flexible and has been proposed mainly to speed up
software development through prototyping. In the spiral model, software is developed in a series of incremental release. The spiral model is divided into a number of regions, typically these are between three to six.

4.7.2.6 V Model

In the V model the process starts in a usual way by requirement engineering and requirement specification, followed by system specification. In a non-component-based approach the process would continue with the unit design, implementation and test. Instead of performing these activities which are often time and efforts consuming, we simply select appropriate components and integrate them in the system.

4.7.2.7 Y Model

A strictly top-down or bottom-up strategy to software production is not quite appropriate. The Y model preaches a top-down or bottom-up fashion for software creation, taking into consideration the knowledge that a software engineer has about the application domain. This knowledge naturally determines the prevailing strategy to software development [Capretz2005].

4.7.2.8 A Model

The A shape Model developed for supporting parallelism and evolutionary design of the application and of the test plans too [Szabó2007].

Challenges

This model does not provide a notation or a set of patterns to increase reusability.
Here R stand for requirement analysis ; D for Design : P for Planning : L for low level H for high level : I for Implementation and T for testing.

Life cycle models as narrated at Para 4.7/N deal with traditional software development process. They have no relevance with Component Based Software Engineering (CBSE) as its development sequences are totally different from that of traditional processes. A new life cycle model termed as Umbrella Life Cycle Model (ULCM) has been proposed which has been illustrated under Para 4.9/N.

4.9 ULCM: Umbrella Life Cycle Model

4.9.1 Component-Based Approach

The main idea of the component-based approach is building systems from already existing third party components. This assumption has several consequences for the system lifecycle;

Putting the Pieces Together In Reference of Component Based Software Engineering 117
• Separation of development processes.

The development of components are separated from software life cycle. The components should already have been developed and possibly have been used in other products when the system development process starts.

• A new phase

Component selection from component repository appears in the component-based software development life cycle.

• Development processes

The activities in the component-based software development life cycle are different from the activities in traditional software development life cycle. In the component-based software development life cycle, selection and verification for component reuse will be the main concern. The life cycle of components based software consists of three stages:

• Selection phase: components are selected from repository or designed, defined and constructed

• Integration phase: when components are integrated with other components

• Deployment phase: Components are instantiated and executed in the running system.

4.9.2 A Software Component Life Cycle Model should define

• What sequence components based software systems follows.
• Why need of these sequences.

• How to compose components.

Over the past three decades, several component based software development methodologies have appeared. Such methodologies address some or all phases of the software life cycle ranging from requirements to maintenance. These methodologies have often been developed in response to new ideas about how to cope up with the inherent complexity of software systems.

4.9.3 Component Based Software Reuse Approach:

Typical steps:

1. Requirements specification

2. Component analysis: finding the right component for the specification if no suitable component found goto step 3 otherwise step 4

3. Requirements modification (to suit components found)

4. System design

5. Integration

6. System validation

4.9.4 Step Involve In Component Based Software Life Cycle:

• Requirement Gathering (User->Developer)

• Requirement Analysis(Developer)

• Design Component Specification
Chapter 4

- Component Selection (Iterative)
- System Design / Integrations
- Modification

A number of integration risks can often be resolved by selecting the ‘right’ set of components. A group of components that compose a system may have overlaps and gaps in required functionality. A gap represents a lack of functionality; an overlap can cause a confusion of responsibility and degrade nonfunctional properties. We have found that the information required evaluating those components using those quality models and metrics is not usually available in the existing commercial software repositories. In any component selection method, it is unrealistic to expect a perfect match between components needed and components available. The search for component from component repository is not simple task. It leads one of the possible outcomes.

- An initial exactly match occurs with required functionality and specification provided by the third party.
- Find some closing match component, then try for adaptation.
- Modify Requirement to get component from repository.
- If no component find or match then construct component according to user’s requirement.
- **Requirement Phase Testing**
It is essential that the requirement be carefully checked by both user and integrator to be certain that it reflects their current needs. There is always a possibility that forces to make changes from both sides.

- **Specification Testing**

A major source of faults in delivered software is faulty specification design. Any mistake occurred in specification design can be detected after software gone into operational mode. Before the specification phase is deemed to finish it is duty of Quality Assurance peoples to verify the entire document with regarding contradictions and ambiguities. In the specification testing phase every statement in the specification document is traced with the statement made by the client team. Another way of checking the specification document is by means of a review. The aim of the review process is to determine whether the specifications are correct.

- **Selection Testing**

A number of components are required to have a reasonable chance of finding matching components for a given task. Selection of component has been identified as a major problem in component based development process. To support the development of high quality component-based systems, component selection processes need to address the problem of Functional and Non-Functional requirement evaluation of component-based software systems. Selection testing is the main quality control activity that is carried out in the early design phase of component based design, it includes component specification verification with user 's requirement.
Figure 4.6 Umbrella Life Cycle Development Model
• Integration Testing

The purpose of integration testing is to check that the components combine correctly to achieve desired functionality which satisfies its specification. Integration testing verification amounts to assemble the software from the set of component that were developed and tested separated. In this particularly we pay attention to testing component interface. After integration testing has been completed, we perform system testing [Dixit2009d].

CASES STUDY

In order to validate the effectiveness of the proposed quality and life cycle model in component based software engineering and to compare its precision in the respect of other quality models, we approached to software professional and academicians directly or indirectly involved in component based software development. We received more than 95% positive feedback. This is indeed a motivation for this research.