CHAPTER 4

CHALLENGES AND NEW DIRECTIONS FOR CBSR

4.1 Introduction........................................................................................................63
4.2 Component-Based Software Reuse.................................................................63
4.3 Component-Based Software Reuse and Software Engineering Concepts ....65
4.4 Challenges in Component-Based Software Reuse.........................................67
4.5 Factors for Success of Component-Based Software Reuse.........................71
4.6 Failure of Component-Based Software Reuse................................................73
4.7 New Directions for Component-Based Software Reuse.............................75
4.8 Summary.............................................................................................................77
4.1 INTRODUCTION

In 21st century, information society and users of different software are more aware of the opportunities of information technology and require constantly new and extra user friendly features for software systems with best quality. This demand of information society is completed with the help of Component-Based Software Engineering (CBSE). CBSE increase the quality and add extra functionality due to reusability. The importance of reusability increase with the CBSE approaches. The basic reasons of software reusability are to increase the productivity and quality of software since programming began but programming also faces challenges like component functionality, programming languages, component generality, encoding scheme and data format, time and efforts, algorithm and data structure, software system evolution, compatibility and maintenance to implement reusability. Successful implementation of reusability depends on the development of reusable component. Reusable component make reusability successful if component are developed with the software engineering concept like structured engineering, information hiding, documentation and good interface in mind. This chapter provides the foundation of Component-Based Software Reuse (CBSR) with software engineering concepts. This chapter introduces the various challenges with the few factors responsible for success and failure of CBSR and new directions to make CBSR successful.

4.2 COMPONENT-BASED SOFTWARE REUSE

In simple way, reuse is a matching between new and old context software and in other way, CBSR is a measure of the ease with which previously acquired concept, objects and components can be used in new contexts (Peters and Pedryz, 2000). The need of CBSR has become urgent as the size and complexity of software has increased tremendously. CBSR is considered as most important to many of the complex problems in software development. Jones identified ten different reusable aspects of software projects - architectures, source code, data, designs, documentation, human interfaces, plans, requirements, and test cases (Frakes and Terry, 1996). Reusable components are not only fragments of code.
but can also be design structures, module-level implementation structures, specifications, documentation, transformations, and so on. Software components are also defined as a unit of composition with contractually specified interface and a binary unit of independent production, acquisition, and deployment that interact to form a functioning system (Szyperski, 1997). A reusable component (Marko, 2002) is a unit of a software development process that can be used as part of a software unit and is documented in such a way that the documentation supports all reuse activities, namely, component creation, brokering and consumption. Reusable components are not only reused within organizations to which the components' developer belong, but are also distributed in the form of an object code via the internet and reused in other environments (Aoyama and Yamashita, 1998). Component reusability is playing important role in software industry for improving software quality and productivity by reducing cost and time to market. The major goals of CBSR are the provision of support for the development of systems as assemblies of components, the developments of reusable entities and the maintenance (Heineman and Councill, 2001). If new software is each time to be developed from scratch, the goal of reducing cost and time cannot be achieved. Effective reuse of software components requires effective means for documentation, compatibility, generality, functionality, and maintenance. In component-based technology, reuse is achieved by composing a new larger system from existing reusable component. CBSR is of interest because people want to develop software systems that are more reliable, less expensive and that are delivered on the time. Traditional software reuse feels that CBSR can provide a better way of developing complex software. Reusable component require several dimension for successful reusability e.g. support for the development of components on different platforms, support for the development of different variants of components for different software, support for maintenance, enhancement and independent development of components, support for how component programmed, how combination and composition of component
possible and lastly, support for the adaptation of selected component in the development process of Component-Based Software Development (CBSD).

### 4.3 COMPONENT-BASED SOFTWARE REUSE AND SOFTWARE ENGINEERING CONCEPTS

CBSR is the process of creating software systems from existing reusable and testable rather than building them from scratch. One of the main motivations behind component technology was reusability. If researchers and practitioners had a collection of reusable software components, they could build applications by simply plugging existing components together. In CBSD, a complex system is accomplished by assembling simpler pieces obtained in various manners. The research efforts have been made to make reusability process of Component-Based System (CBS) more effective, more predictable and less expensive in comparison to simple software reusability by using different approach e.g. object-oriented approach. CBSD and CBSR are not similar to traditional engineering domain and both finally provide solution to all complex problems. The goal is the use of reusable components as building blocks in new systems with modifications occurring in a controlled way with CBSE. CBSE produces a set of reusable components that can be combined to obtain a high level of reuse while developing members of an application family. Typically, (Griss et al., 2003) one first performs a domain analysis and then a layered modular architecture is defined that specifies key layers and core components, key subsystems and interaction mechanisms. High-level specifications and interfaces are defined for 'pluggable' components. Effective reuse of software components requires effective means for documenting and communicating several kinds of related information among many stakeholders in the reuse process of those components. Parallax study says reuse benefits have been reported from industry as improvements in quality, productivity, performance, reliability and interoperability. Reusable software is obtained by keeping various software engineering concepts in mind. Some of these are briefly summarize below.
4.3.1 Structured Programming
Structured programming related to few well understood control structures. The basic idea is to give careful thought to the structure of software as early as possible in the software development cycle. Its aim is to obtain software components that are relatively easy to construct, easy to reuse, correct and maintain.

4.3.2 Information Hiding
Information hiding related to the separation of concerns. The underlying idea is to make those parts of a code invisible that are of no concern to the user or environment, making the total software package a set or hierarchy of black boxes, each performing a specified function.

4.3.3 Grouping
Grouping related software pieces into modules help in developing software logically. The idea is to group logically related parts (data and operations) to ease construction and maintenance. Functional abstraction is a key concept for development of software components.

4.3.4 Standardization
Standardization of naming and interfaces is an important condition for software reuse. This allows the gluing of software components to construct a larger software system. If software with good interface is available at the time of development it helps in fast development of component-based software.

4.3.5 Platform-Independence
Platform-independence helps in software implementation on separate environments. Platform-independence is essential for software to be used on hardware platforms and in software environments other than the ones on which it has been developed. Software implementations are achieved by machine-specific software instructions, but they prohibit the reuse of software in heterogeneous environments.
4.4 CHALLENGES IN COMPONENT-BASED SOFTWARE REUSE

The CBSR focuses on building large software system by integrating previous existing software component. The idea of constructing modular software has long been recognized as advantageous within software community since its early days. CBS, and component reusability has many advantages and important one include (Brown, 2000 and Crnkovic et al., 2003) more effective management of complexity, reduced time to market, increased productivity and quality with greater degree of consistency and wide range of usability. Despite their importance, this study introduces few challenges.

4.4.1 Component Functionality

Component functionality is the most serious challenge of an existing component that cannot be reused without change. Existing component and a component to be newly developed match according to their functions. The new component to be developed may require some changes to corresponding functions in the existing component, or may require additional functions. Suppose the existing component contain excessive function, but to increase the performance most of the excess function may need to be eliminated (Won, 2005). In the digital age where software of one device can be used in another with or without any change e.g. software of digital camera and pen derive may be used in cell phones. Component developed to provide one particular major function for one system and this function of reusable component may be reused in another system.

4.4.2 Programming Language

A programming language is a tool in software development. At present, researchers are using modern programming language such as C++, Java, C#, Visual programming languages like VB and VC++ etc and these programming languages have changed even the programming style of the software developers depending upon application domains. Suppose the new software must be developed in JAVA, an existing component written in, say C#, cannot be reused without change, even if it satisfies all other requirements such as the component
CHAPTER-4 Challenges and New Directions for CBSR

qualification, component specifications, functionality, operating environment, system architecture, etc. sometimes cross compilation is not possible when reusable component is developed in one language and reused in another software which developed in another language.

4.4.3 Component Generality
The components must be sufficiently general to cover the different aspects of their use because software reuse principles place high demands on reusable components. Reusable component must be concrete and simple enough to serve a particular requirement in an efficient way (Szyperski, 1998). Developing a simple component requires three to four times less resources than developing a reusable component.

4.4.4 Encoding Scheme and Data Format
There are different types of data format and encoding scheme in software development e.g. file system and Data Base Management System (DBMS) are two main data format which is mainly used to manage data. Software components that interface with file systems are significantly different from those that interface with DBMS. Different countries use different formats to store such data as date, time, currency and some software encodes data in American Standard Code for Information Interchange (ASCII), while others use Extended Binary Coded Decimal Interchange Code (EBCDIC) (Won, 2005). Software components that explicitly deal with such data formats and encoding schemes cannot be reused without change in software that uses different data formats and encoding scheme.

4.4.5 Time and Efforts
More time and effort is required for the development of reusable components. It is difficult and time-consuming to fully understand an existing component that appears to closely match the requirements of the part of the new software for which the component may be reused with change. Then it takes effort to determine those parts of the component that require changes, and to actually make those changes. Suppose once the changes are made, the modified component must
be thoroughly tested and documented, and the entire software that includes the modified component. The development of a reusable component requires multiple times more than simple component. Time and effort require more in analyzing and modifying an existing component in comparison to develop a component from scratch.

4.4.6 Algorithms and Data Structure

Different type of algorithms and data structure is used to implement key functions in reusable component of software systems. Algorithms for developing reusable component include sorting, searching, and message routing on a network etc. Data structure e.g., linked list, hashing, binary tree; heap is used to implement the functions. One algorithm may be simple and quick to implement, while it may result in low performance, low level of reliability, and low level of security. One algorithm may be good to support the routing of a large number of small messages, while it may be unacceptable for routing a mixture of a small number of large messages and a large number of small messages (Won, 2005). Software components that implement algorithms techniques and accompanying data structures, in software under certain considerations cannot in general be reused without change in different software in different considerations.

4.4.7 Software System Evolution

Component-Based System (CBS) evolution is required due to competitive market situation where every user demand for continually improved system performance. The advancement of technology in the different fields in which software is used requires improved software. The improvements may require a completely new approach or new functions in software. CBS may seem relatively easy to evolve and upgrade, since components are the unit of change. To repair an error, an updated component is swapped for its defective equivalent, treating components as plug-replaceable units. Replacement of one component with another is time consuming and arduous task, since the new component will never be identical to its predecessor and must be thoroughly tested, both in isolation and in
combination with the rest of the system (Gill, 2003). Evolution in computer hardware and software technology is so fast that an organization manufacturing long-life and complex products must expect significant technology changes during the product life cycle (Crnkovic et al., 2002). From the reliability and risk point of view, such organizations prefer not to use the latest technology, but because of the demands of a highly competitive market, are forced to adopt new technology as it appears. Manufacturers are faced with a dilemma to adopt the new technology and possibly improve the development process at the risk of short term higher costs or to continue using the existing technology and thereby miss an opportunity to lower development costs in the long run.

4.4.8 Component Compatibility

In CBSD the most important factors for successful reusability of component is the compatibility between different versions of the components. A component can be replaced easily or added in new parts of a system if it is compatible with its previous version. The compatibility requirements are essential for running software system, for many years. Compatibility issues are relative simple when changes introduced in the software systems are of maintenance and improvement nature only. Using appropriate test plans, including regression tests, functional compatibility can be tested to a reasonable extent. More complicated problems occur when new changes introduced in a reusable component which eliminate the compatibility (Crnkovic, 2003). To solve such problem, additional software must be written which can help in managing the compatibility of both versions.

4.4.9 Component Maintenance and Enhancement

Component maintenance and enhancement are two main factors in reducing cost of CBSD. The component maintenance cost can be very high since the component must respond to the different requirements of different applications running in different environments, with different reliability requirement and require different level support. The maintenance process is also complex, because it must be handled on different levels - (Crnkovic et al., 2002) on the system level, product level and on component level. On system level customers report their problems,
on the product level, errors detected in a specific product version are reported, and finally on the component level, where the fault is located. The modification of the component can have an impact on other components and other products, which can lead to an explosion of new versions of different software systems. Maintenance is really the most important for reusable components, developers and service people. In enhancement, perfective and adaptive enhancement (Lientz et al., 1978; Schash and Mellor, 1996) is also requiring in reusable component. In perfective, component may get user enhancement and improved documentation and in adaptive, changes to data input and file system is required if the reusable component is used in another software which is again time consuming and increase the cost of software.

4.5 FACTORS FOR SUCCESS OF COMPONENT-BASED SOFTWARE REUSE

Component-Based Software Reuse (CBSR) is playing important role in software industry for improving software quality and productivity by reducing cost and time to market. (Heineman et al., 2001). The major goals of CBSR are the provision of support for the development of systems as assemblies of components, the developments of reusable entities and the maintenance. Research in CBSR has observed that most successful reuse efforts have been achieved using collections of components within well-defined and well-understood domains. According to (Tracz, 1995) “Biggerstaff’s Rules of Three” states that unless three real systems have been built in the domain, it is unlikely that the details required for successful reuse can be derived. In many ways this is unfortunate as tool builders are providing the largest degree of support for those problems that require the least amount of support. If the domain is mature, then people and artifacts exist that can supply the necessary background for solving the problem, reducing the need for sophisticated support models. A potentially larger return on investment lies in support for problems those have not yet matured, but for which some isolated solutions have been explored. Another factor which is helpful in CBSR is component architecture and software architecture. CBSR is closely related to
component architecture and software architecture. Software architecture provides a context for the development of reusable building blocks. Conversely, software architecture provides a skeleton into which new building blocks can be incorporated and it allows implementations of existing building blocks to be replaced by new implementations. Attention to architectural issues is one of the prime criteria for successful software reuse (Gill and Tomar, 2006). Reusable component require several dimension for successful reusability e.g. support for the development of components on different platforms, support for the development of different variants of components for different software, support for maintenance, enhancement and independent development of components, support for how component programmed, how combination and composition of component possible and lastly, support for the adaptation of selected component in the development process of CBS. Success of software depends on reusable software and component reuse which help in enhancing the following factors which increase the overall quality of CBS.

4.5.1 Productivity
The plug-in a collection of ready-made parts will be of great help increasing production speed and minimizing cost/effort. There is a reduction in the need for software design, implementation, testing, and documentation if researchers and practitioners reuse reusable software and component. So they are really help in increasing the productivity within time during development.

4.5.2 Software Quality
A component and reusable software with well-designed, well-tested, and well-documented code that has proven correct and well-understood behavior is very helpful in increasing quality of software.

4.5.3 Reliability
Components exercised in working systems are very helpful in increasing the reliability because they are already tested and easy to fit in another software as it is or after some modification.
4.5.4 Reduced Process Risk
By using CBSR researchers and practitioners can easily reduce the process risk. Less uncertainty in development costs is also reduced by CBSR.

4.5.5 Effective use of Specialists
Reuse components and experts for component developments instead of people. Reusable component is very important in case of software development to produce the software in time. So use always experienced person who can easily use a reusable component as a good expert.

4.5.6 Standards Compliance
Embed standards in reusable components to increase the use of reusable component in the real and whole world.

4.5.7 Accelerated Development
CBSR avoid original development and hence speed-up production. Because when researchers and practitioners use already developed code of reusable software and component they help in accelerated the development of software.

4.6 FAILURE OF COMPONENT-BASED SOFTWARE REUSE
Component reusability is playing important role in software industry for improving software quality and productivity. But the failure of CBSR is really affect the software industry due to the following non-technical factors - psychology of the programmer, lack of tool support, inadequate component-behavior models, language selection issues, platform selection issues, and difficulty of finding reusable software.

4.6.1 Non-Technical Aspects of Component-Based Software Reuse
This study introduces the some non-technical aspects of component which is really helpful in success of CBSR. Although a lot of software reuse is happening and many techniques exist that facilitate the creation and publication of reusable software, it is not always as prolific as one would like. Software engineering is not only concerned with technical aspects but with people and other
environmental aspects as well. By being embedded in a society and being controlled by human beings, software engineering is influenced by these. Non-technical issues like economical issues, management issues, support structure, and psychological issues are intertwined with technical ones. Arguments for or against reuse vary with the role someone plays in the software development process.

4.6.1.1 Economical Issues

Economics plays an important role in making decisions about CBSR. As mentioned earlier, a reusable program requires changes in the way software is developed, and these changes are often not for free. The production of reusable software typically requires an extra initial investment which only pays off after a certain period of time. In a commercial environment, such extra investments require the approval from the management and this is subject to the market situation etc. Parallab study says software becomes a capital good, with higher initial costs coupled to profit returns over a longer period. Code licensing is an example of an economical factor that influences CBS reuse. When proprietary reusable components are used in constructing a system, there are costs and legal issues concerning intellectual property, copyright, and scope of use.

4.6.1.2 Management Issues

The management also plays a key role in decisions about and success of software reuse. The management is largely responsible for planning the software development process. If reuse is not a clear objective from the start, then reuse will remain accidental. When programmers write the code and may only realize at a late stage that there are similarities between other or successive problems. Such unplanned approach is known as code scavenging or code salvaging.

4.6.1.3 Support Structure

A support structure needs to be put in place that is responsive to users’ problem reports and that is able to repair deficiencies in a swift manner. Reusable software and reusable component is not likely to be reused by others if there do not exist any recognized contact points for support. It is not acceptable that the success and
reputation of a commercial application is hindered by the software quality of embedded third-party software. In a commercial environment where the performance of the end-result is a major worry, the management may require the inspection of the source code of third-party reusable software to assure that the specifications meet the requirements.

4.6.1.4 Psychological Issues
The psychology of the programmer is another aspect in software reuse. Software reuse means that a programmer has to adapt, incorporate, and work with software written by other programmers. A major stumbling block is whether the programmer is willing to do so. Writing the code yourself can often be more fun and/or be a sign of craftsmanship especially if it is in a field that is of high interest to the programmer. But if the programmers want to develop the software with low cost and within time than psychology pays really an important role in this matter.

4.7 NEW DIRECTIONS FOR COMPONENT-BASED SOFTWARE REUSE
In this section study briefly reviews some of the factors to give some new directions which are helpful to make software available for wider reuse. Making software available happens at various stages of the development process. These factors are documentation and specification, components libraries, and generative programming.

4.7.1 Documentation and Specification
Documentation that accompanies a specification should include an overview of the design considerations. These have to focus on the problems encountered and choices made so as to allow software engineers to compare their situation with the one described in the documents. In addition, examples, test results, and component measurements should be included. Parallab study says results (actual or theoretical) of test results, and component measurements are often important criteria for users to adopt software, especially if there software exists with similar functionality. In case the software is not black-box software, the documentation
should aid re-users in adapting the components, and it should aid maintenance persons in the problem tracing and solving process. Finally, documents should include some installation guides, trouble shooting, FAQs, etc. Simply stated, a specification is a protocol or language that defines a set or hierarchy of basic components, each of which has an interface - what valid input are, and how it responds as it receive input and produce output. For each valid input to a component, the output must be well-defined and documented. Specifications are typically accompanied by tutorials, checkers, and examples. Basic components can for example be data types, subroutines and objects. The choice of abstraction at the interfaces is important and has a large impact on the reuse and acceptance of the software/specification. In order to understand the behavior of component it is important for users to be able to distinguish between what a specification prescribes and what it does not.

4.7.2 Components Libraries
Components libraries can be developed in a hierarchical manner, that is, one library can make use of functionality that is defined is another library. In addition, a library can be used in more than one other library for reuse of code. Practically anything worth reusing can be made publicly available in a library. Code libraries have probably been around longest, and they are an accepted part of the software development process. Marko Forsell, 2002 study says most libraries consist of source code or pre-compiled components. A library’s Application Programming Interface (API) is the means by which an application developer accesses and uses the parts of a library. Often, not all components of a library need be visible to the application developer, and then only the ‘public’ components or members are made accessible to the outside world.

4.7.3 Generative Programming
Generative programming is about designing and implementing software modules which can be combined to generate specialized and highly optimized systems with specific requirements. Code generation is the process by which some semi-
automated tool - the code generator is used to turn higher input into lower level output. The output of the code generator often needs further processing before it is complete. Automatic code generation is enabled by generative programming. A main goal of generative programming is to replace manual search, adaptation, implementation, and assembly of components with on-demand automatic generation of these components. Generative programming focuses on maximizing the automation of application development - given a system specification, generators use a set of reusable components to generate a concrete system.

4.8 SUMMARY

Component-Based Software Reuse (CBSR) with reusable component not only reduces the time to market but also brings down the cost of development heavily. This chapter introduces the various challenges of CBSR and new direction to make CBSR successful. Development of software with reusable component has many advantages but at the same time there are also major challenges that need to be addressed perfectly. Reusable software components is obtained by keeping various software engineering concepts in mind viz. structured programming, segregation of design, grouping related software pieces, standardization of naming and interfaces, and platform-independence. Success of CBS depends on reusable software components which help in enhancing the factors, namely, productivity, software quality and reliability. To reduce process risk and challenges software organization requires effective use of specialists, standards compliance. This chapter also summarizes success and failure factor with non-technical aspects of CBSR. Lastly, the study gives some new directions after reviewing the following factors - regarding documentation and specifications, components libraries and generative programming for making successful use of CBSR.