ISSN No. 0976-5697
Volume 2, No. 1, Jan-Feb 2011
International Journal of Advanced Research in Computer Science
RESEARCH PAPER
Available Online at www.ijarcs.info

Understanding the Real Issues to Move towards a Systematic Reuse Based Approach

Kuljit Kaur* and Hardeep Singh
Deptt. of Computer Science and Engineering,
Guru Nanak Dev University,
Amritsar, India
kuljitchahal@yahoo.com

Abstract: The concept of reuse has been emphasized time and again. It was proposed as a solution to build large reliable systems in a controlled and cost-effective way by Mcllroy in his seminal paper in the NATO conference in 1968. Recently, on the onset of the recession which hit the global economy in 2008, the need has been again felt to make extensive use of the concept of reuse to improve the software cost structures. However, despite this the reuse based approach to software development is still adhoc in nature. A systematic reuse based software development approach is not followed. It may be due to the fact that tools and methods for successful implementation of software reuse are not available. Even the terminology related to the reuse concept is not uniform. This paper attempts to put together and analyse different views related to a popular reuse based software development paradigm – Component Based Software Engineering. It also discusses other issues that need to be looked into for successful implementation of the concept of reuse in the context of component based software engineering.

Keywords: Software Reuse, Component Based Software Engineering, Software Cost, Component Characteristics.

I. INTRODUCTION

The recent recession, which hit the globe on 15th September, 2008 (the day Lehman brothers filed for bankruptcy), has seriously impacted the business houses world-wide. In order to survive this worst recession, companies are doing every bit to save money or to reduce the costs. Software cost has been a major component of the costs that a company has to incur. So the onus further lies on the software companies to reduce the costs of their products.

Most software cost models are functions of five basic parameters: size, process, personnel, environment (available tools to automate the process), and required level of quality [1]. Cost is likely to be more for a large sized product (measured in LOC or function points). Keeping all other parameters constant, a reduction in size can help to reduce the cost of a product. Walker Royce defines size reduction as "to reduce the number of human-generated source lines" or "to reduce the amount of custom developed code" [1]. He suggests reuse, object oriented technology, higher order languages, and automatic code generators as some of the mature size reduction technologies. Capers Jones, also points out that a shift from custom development to reuse based development can help to improve the software cost structures [2].

Not only that, reuse based software development reduces the development cost, it also shortens the development cycle and thus the time to market. It increases the productivity of programmers. Rather than spending time and effort on mundane tasks, they can focus on more challenging aspects of the application and hence improve its level of quality [3.4]. In addition several other benefits of reuse have been reported: reduction of project planning overheads, improvements in support and maintenance, better use of resources, and better tackling of system complexity[5.6]. Mohageghi *et al.* review the industrial studies that link software reuse to quality, productivity, and economic benefits [7].

The idea of software reuse is not new. In 1968, Mcllroy suggested software reuse as a means for overcoming software crisis [8]. Software crisis is characterized by two major phenomenon: Lack of ability to produce software within budget and time constraints, and lack of quality in produced software [9]. Mcllroy pointed towards the effective use of reusable software libraries to build large reliable software systems in a controlled and cost effective way.

The concept of reuse has been emphasized time and again. But there is lack of tools and methods for successful implementation of reuse. Another issue is lack of uniformity in referring to different terms of reuse. This paper attempts to put together different views related to a popular reuse based paradigm – Component Based Software Engineering. Next section of the paper gives an overview of the reuse technologies. Third section details out the component based approach to software development. It presents different points of view regarding component based software development processes, the definition and characteristics of a component - the building block of a component based software system. Fourth section presents other issues and challenges in building successful component based software systems. Fifth section concludes the paper.

II. REUSE TECHNOLOGIES

Software reuse can be implemented in 3 different forms – Compositional reuse, Product Line Engineering (PLE), and Generator based reuse [10]. Compositional reuse refers to building an application by assembling already available reusable software components. Component Based Software Engineering (CBSE) paradigm adopts this form to produce software applications. PLE refers to creating a common set of core assets and then using them to create applications that satisfy the requirements of a specific domain [11]. Generator based development is applicable for more mature/narrow application domains in which the application developer specifies the variation through parameters and the generator generates the application according to these parameters. In the generator based reuse, domain knowledge is encoded into an application generator. For example, Lex and Yacc are the application generators in the UNIX environment.

To achieve an ideal level of reuse, the desirable trend is to move from compositional to generator based reuse.

III. COMPONENT BASED SOFTWARE ENINEERING

Component based software engineering is a systematic approach to develop software applications using already existing software components. The notion of 'developing an application program by writing code' has been replaced by 'building a software system with assembling and integrating existing software components'. It involves use of prefabricated pieces, perhaps developed at different times, by different people, and possibly with different uses in mind [12]. The components used in a component based software system may be in-house components or off-the-shelf components (which include open source components or commercial components also known as COTS -Commercial off the Shelf). Component developers develop software components keeping in mind their reuse value across product lines and organizations. These reusable components are reused as is or are adapted to meet the requirements of a different project in a context other than the one anticipated during their development.

This approach is different from the traditional way of software development. Here, the development process has two sides: Development of software components for reuse and development of software with reusable components as the building blocks (Figure 1). Main steps in development for reuse are [13]:

[a] Perform domain analysis
[b] Identify the components to be developed
[c] Develop the components
[d] Evaluate the components so that they can be added to the library
[e] Package the components and add to the library.

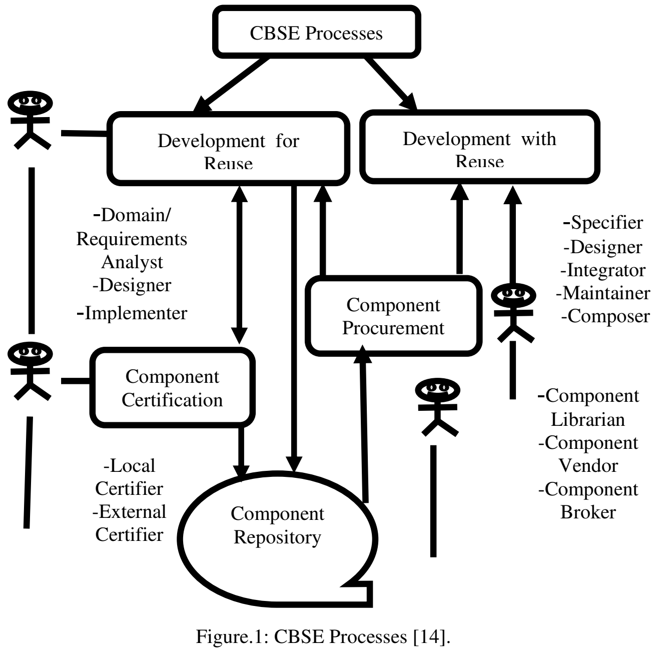

Figure.1: CBSE Processes [14].

Main steps in development with reuse [13] are as follows**:**

[a] Retrieve components from library (in house or third party) according to some need of the application under development,
[b] Evaluate the quality and appropriateness of the components.
[c] Adapt a component, if it cannot be reused as-is.
[d] Assemble the application
[e] Test the integrated assembly

It can be observed in the above discussion that component evaluation takes place at two stages: when components are added to the library of reusable components and when they are selected for use in an application. In the latter case, context of use is also important for evaluation.

A. Defining a Component

Both the researchers and practitioners in component based software engineering have not yet agreed upon the definition of a component. Different people perceive the concept of a component differently.

Hooper and Chester define a component simply as "anything which is reusable" [15]. Bertrand Meyer adds another dimension to it and separates a component user from a component developer [16]. He defines a component as "a software element that must be usable by developers who are not personally known to the component's author to build a project that was not foreseen by the component's author". Johannes Sametinger stresses on technical attributes of a software component [13]. According to him, software components are "Self-contained, clearly identifiable pieces that describe and/or perform specific functions, have clear interfaces, appropriate documentation, and a defined reuse status". Szyperski in his definition of a software component takes into account context of component deployment as well [17]. He defines a component as "a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties".

Heinemann and Councill's definition includes a reference to the component infrastructure (middleware) required for seamless integration of software components [18] They define a software component as "a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard".

In the context of object oriented paradigm, Valerio *et al.* define a component as "a homogeneous set of objects that collaborate to perform a feature or functionality and exposing a component interface that allows to integrate it in a system and make available to the external environment a set of services" [19] However components are not the same as classes or objects – the traditional object oriented artifacts. Classes are conceptual entities which form a part of the structure of a program. Once implemented as part of the program, they are not required to be accessible from outside. Components are the physical entities which are accessible and pluggable as per the requirements. Both can be assembled to build a new application, but the difference is that components are plugged, and objects are wired. Components generally provide complex functionality, where as objects provide limited functionality. Components offer explicit interfaces: required interfaces, and provided interfaces. But objects explicitly mention only their provided interfaces (set of public methods), their required interfaces are hidden in implementation.

B. Component Characteristics

Several attempts have been made to characterize the components in order to better understand and classify them which can further help in efficient component storage and retrieval, and component selection as well.

Szyperski identifies 4-tiers at which there are different reasons to use components in different forms [20]. In tier 1, organizations choose software components for economic reasons. Components at this level are the source components which include architectural, design, and source code artifacts. Tier 2 focuses on use of partial design and implementation information across multiple products of the same domain i.e. product lines. These components are known as build time components. In third tier, components are not consumed during development-time but during run-time. Such components are called deployable components. The fourth tier deals with the use of components to handle changes in the engineered solution in an open environment. Components are dynamically available as services at distributed remote locations which can be obtained on-demand, installed and integrated with the existing solution.

Bertrand Meyer characterizes components on the basis of four viewpoints: level of software process task, level of abstraction, level of execution, and level of accessibility [21]. At different levels of software life cycle, components exist in different forms. It may be a software requirements specification document in the analysis phase, a design pattern in the design phase, or an executable piece of code at implementation level. It may represent an abstraction of a function, or data with fine granularity as a class or coarse granularity as a cluster of classes or a complete system. A component is static if it is integrated into a system at compile/link time, and has to be recompiled after every modification. It is replaceable if it is static but its variants can be substituted dynamically. A component in the dynamic category can be integrated into the system at the time of execution. No source code is available mostly for components in the commercial category. Level of accessibility criteria distinguishes components, with source code available to component users, from components whose source code is not available or is available on demand only. Components in the former category are available in the open market and are called Open Source Software Components. Open source components are available free of cost under different types of licenses such as GNU, PDS (Public Domain Software) etc. Components in the latter category are available in the commercial market and are acquired for a fee. Such components are known as Commercial off the Shelf (COTS).

Other criteria to classify components can be age (level of maturity), level of reuse, context (application domain), technology/ infrastructure support, ability to plug and play with other components as well as with the underlying platform, role (active or passive- GUI v/s database component) [22].

Sametinger identifies component characteristics by means of different types of interfaces a component uses to communicate with the user, other components, or with the environment [13]. They include: type of user interface (command line or graphical), data interface (textual, file, or data base I/O), program interface (functional composition, object oriented composition, or open platform composition), and component platform (hardware, OS, and programming environment).

Morisio and Torchiano characterize software components on the basis of source (origin, cost, and property), customization (required modification, possible modification, and interface), bundle (packaging – static or dynamic, partial or total delivery, size), and role (functionality, and architectural level- support or core) [23].

The component characterization framework suggested by Sassi *et al*. groups characteristics into: general (cost, date of first release, and change frequency), structural (name and number of services), behavioral (pre/post-conditions and state-transition diagrams), architectural (component type and architectural style), quality of service (nonfunctional properties and possible modification), technical (conformance to standards), and usage (similar components and use cases) [24].

Kienle *et al.* present taxonomy to characterize software components as well as component based systems [25]. They use the following criteria to characterize software components: origin, distribution form, customization mechanisms, interoperability mechanisms, and packaging. Origin of the component specifies the source of the component i.e. in-house or off the shelf component. Distribution form is based on the availability and modifiability of source code – Black box (no source code available, no modification possible), white box (source code available, modifications possible), glass box (source code visible but no modifications possible). Sametinger specifies another distribution form i.e. Gray box in which limited source code is visible, and only that portion is modifiable [20]. Customization mechanisms are available at two levels: non-programmable and programmable. Non-programmable customization allows using command line switches, configuration files, or check boxes to customize a software component. In programmable customization, application programming interface (API) or scripting languages are used to modify or extend the behaviour of a component. Another characteristic of a component is its ability to interoperate with other components in the application. There may be no interoperability information available for a component or it may have programmable interfaces to enable interoperability with other components. Components may be packaged differently as standalone or non-standalone. Standalone components can be directly executed without any prior customization or integration. Whereas a non-standalone component has to be customized or integrated before it is executed.

IV. OTHER ISSUES AND CHALLENGES

CBSD is still not a very popular paradigm of development with the software developers. There are several obstacles to successful adoption of the CBSD in software development organizations. Kunda *et al.* have studied the human, social, and organizational factors responsible for making the CBSD application difficult in organizations [26]. Neumann elaborates on the risks of predictable compositions of software components. There are several problems related to composability of components including scalability, certification, quality assurance, incompatible policy matters, inadequate requirements specifications, poor software engineering practices etc. [27]. Voas has identified and analyzed five sources of headaches in dealing with library components [28]. Apart from technical risks, Hasselbring *et al.* talk about liability risks of using third party components [29]. From judicial point of view (in the European Union), the vendor providing a component based software solution to his customers is to be held legally responsible for malfunctioning in any component of the solution. The customer is not bound to localize the defect to a particular component of the solution. Application vendor has to pay the claims. The vendor can, in turn, ask for damages from the specific component supplier but for that he has to establish the fact that the supplier's component is problematic.

Component based software development brings with it its own issues and challenges. Several technical as well non-technical issues need to be addressed in order to have successful implementation of this paradigm [30]. Some of the issues are outlined below:

A. Creating tools, techniques, and well defined processes to support essential component activities such as component specification, component certification, component search and retrieval, component selection, component composition, component integration, and component version management.
B. Component repository management - A rich repository of reusable components is essential for successful implementation of component based development. Further the component users should be able to locate needed components easily and quickly. So there is need to design efficient algorithms for storing and retrieving components from a repository.
C. Risk analysis and management – Components acquired from external sources carry the risks of unpredictable quality, architectural mismatch, and uncertainty of future support from component suppliers [31]. In addition to this there are legal risks involved. So it is necessary to identify the risks, and manage them in advance so as to facilitate seamless integration of software components.
D. Support for evolving third party components – Successful software requires modification from time to time to accommodate changes in domain as well as in technology. As software components evolve, problems arise due to inadequate support from the component vendor, delay in identification of modification requirements and their implementation, conflicts between needs and priorities of different component users. So there is need to manage component evolution otherwise it may result in higher maintenance burden and lead to other quality issues such as reliability.
E. Establishing Trust in third party components – There is a need to define mechanisms to establish trust in third party software components. A component user has every reason to not to believe the component developer/supplier regarding component quality attributes till sufficient documentary proof is not made available.
F. Component quality assessment- Quality of existing components in general and of third party components in particular has been an issue of great concern. Bertrand Meyer suggests that foremost importance should be given to quality of software components especially acquired from third parties [16]. He stresses that quality of a component based application is equal to the quality of its worst component. Here issues that need to be explored include: component characterization, component documentation, availability of component related information, component testing, component certification, component quality models, and predictable assembly of components.
G. Software reuse metrics and models – Software metrics in the reuse context may be divided into five categories-

[a] Metrics which measure the extent of reuse within a software application,
[b] Metrics which measure the consequences (economic benefits) of reuse in an application,
[c] Reuse library (repository) metrics
[d] Metrics which measure the ability to use a software component in a context other than that for which it was originally developed, also known as reusability metrics.
[e] Metrics which measure the quality of a reuse based application.

There is need to define metrics based on the formal specifications so that they can be theoretically as well as empirically validated.

V. CONCLUSIONS

The idea of reuse is almost half a century old. But it is not yet a mature technology. One of the major reasons could be the non-availability of tools and processes to implement the concept in its true spirit. Another important issue is the lack of good quality reusable components in the market. Even the terminology related to the concept is not uniform across various design/implementation methodologies or among the researchers and practitioners. This paper details out different views on some of the important concepts of a reusable program in the context of a composition based reuse approach i.e. component based software engineering. It has been observed that there are several issues that need to be looked into for successful implementation of the reuse based approach for software development.

VI. REFERENCES

[1]. Royce, W. (1998). Software Project Management: A Unified Framework. Pearson Education.
[2]. Jones, C. (2009) Software Engineering Best Practices: Lessons from Successful Projects in the Top Companies, McGraw-Hill Osborne Media, 1st Edition, 2009.
[3]. Jacobson, I., Griss, M. and Johnsson, P. (1997). Software Reuse, Architecture, Process, and Organization for Business Success. Addison-Wesley.
[4]. Tracz. W. (1988). Sofware Reuse: Motivations and Inhibitors. Software Reuse: Emerging Technology. pp. 62-67. IEEE Computer Society Press.
[5]. Almeida, E., Alvaro, A., Garcia, V., Mascena, J., Burégio, V., Nascimento, L., Lucrédio, D. and Meira, S. (2007). Component Reuse in Software Engineering (C.R.u.i.S.E.). Reuse in Software Engineering (RiSE) Group, available at http://cruise.cesar.org.br/index.html last accessed on 18/12/09.
[6]. Llorens, J., Fuentes, J., Prieto-Diaz, R. and Astudillo, H. (2006). Incremental Software Reuse. Proceedings of 9th International Conference on Software Reuse

(ICSR2006), LNCS 4039, pp 386 – 389. Springer Berlin / Heidelberg.

[7]. Mohagheghi, P. and Conradi, R. (2007). Quality, productivity and economic benefits of software reuse: a review of industrial studies. Empirical Software Engineering 12: 471–516.

[8]. Mcllroy, D. (1968). Mass-Produced Software Components. Proceedings of the 1st International Conference on Software Engineering. pp 138–155. Garmisch, Germany.

[9]. Kim, H. and Boldyreff, C.(1996). An Approach to Increasing Software Component Reusability in Ada, Reliable software Technologies –Ada-Europe'96. pp. 89-100. Lecture Notes in Computer Science, Springer Berlin/ Heidelberg.

[10] Mili, H., Mili, A., Yacoub, S. and Addy, E. (2002). Reuse Based Software Engineering – Techniques, Organization, and Measurement, John Wiley & Sons.

[11] Pohl et al., 2005 Pohl, K., Linden, F. and Bockle, G. (2005). Software Product Line Engineering: Foundations, Principles, and Techniques. Springer.

[12] Clemente et al., 2008 Clemente, P. J., Herandez, J. and Sanchez, F.(2008). Extending Component Composition Using Model Driven and Aspect-Oriented Techniques, Journal of Software 3(1): 74-86. Academy Publishers.

[13] Sametinger, J. (1997). Software Engineering with Reusable Components, Springer, -Verlag New York, Inc., USA.

[14] Hutchinson, J. and Kotonya, G. (2006). A Review of Negotiation Techniques in Component Based Software Engineering, Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'06), pp 152-159. Cavtat/Dubrovnik, Croatia.

[15] Hooper, J. W. and Chester, R. O. (1991). Software reuse: Guidelines and Methods. Plenum Press, New York, 1991.

[16] Meyer, B. (1999). On To Components. IEEE Computer 32(1): 139–143.

[17] Szyperski, C. (1999). Component Software - Beyond Object-Oriented Programming, 2nd Edition. Addison-Wesley.

[18] Heineman, G.T. and Councill,W.T. (2001). Component-Based Software Engineering: Putting the Pieces Together, Addison-Wesley Professional.

[19] Valerio, A., Cardino, G. and Leo, V.(2001). Improving software development practices through components, Proceeding of the 27th Euromicro Conference 2001: A Net Odyssey (Euromicro01), pp 97-103. Warsaw, Poland.

[20] Szyperski, C. (2003). Component technology: what, where, and how? Proceedings of 25th International Conference on Software Engineering. pp 684–693.Oregon, USA.

[21] Meyer, B. (2003). The Grand challenge of Trusted Components. Proceedings of the 25th International Conference on Software Engineering, pp. 660-667. IEEE Computer Society. Portland, Oregon.

[22] Yacoub, S., Ammar, H. and Mili, Ali. (1999). Characterizing a Software Component, Proceedings of International Workshop on Component-Based Software Engineering, May 1999.

[23] Morisio, M., Ezran, M. and Tully, C. (2002). Success and Failure Factors in Software Reuse. IEEE Transactions on Software Engineering 28(4): 340-357.

[24] Sassi, S., Jilani, L. and Ghezala, H. (2003). COTS Characterization Model in a COTS-Based Development Environment. Proceedings of the 10th IEEE Asia-Pacific Software Engineering Conference (APSEC'03). pp. 352–361. Chiang Mai, Thailand.

[25] Kienle, H., Holger, M. and Muller, H. (2007). A Lightweight Taxonomy to Characterize Component-Based Systems, Proceedings of Sixth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS'07). pp 192-204. Alberta, Canada.

[26] Kunda D. and Brooks, L. (2000). Assessing Organizational Obstacles to Component-Based Development: A Case Study Approach. Information and Software Technology 42: 715–725.

[27] Neumann, P. (2006). Risks Relating to System Compositions. Communications of the ACM 49(7): 128.

[28] Voas, J.(1998b). The challenges of using COTS Software in Component-Based Development. IEEE Computer 31(6):44-45.

[29] Hasselbring, W., Rohr, M., Taeger, J. and Winteler, D. (2006). Liability Risks in Reusing Third-Party Software, Communications of the ACM 49(12): 144-145.

[30] Kalagiakos, P. (2003). The Non-Technical Factors of Reusability, Proceedings of the 29th EUROMICRO Conference "New Waves in System Architecture" (EUROMICRO'03). Pp 124. Belek-Antalya, Turkey

[31] Vitharana, P. (2003). Risks and Challenges of Component-Based Software Development. Communi cations of the ACM 46(8): 67-72.