Analytical Study of Component Based Software Engineering

Iqbaldeep Kaur, Parvinder S. Sandhu, Hardeep Singh, and Vandana Saini

Abstract—This paper is a survey of current component-based software technologies and the description of promotion and inhibition factors in CBSE. The features that software components inherit are also discussed. Quality Assurance issues in component-based software are also catered to. The feat research on the quality model of component based system starts with the study of what the components are, CBSE, its development life cycle and the pro & cons of CBSE. Various attributes are studied and compared keeping in view the study of various existing models for general systems and CBS. When illustrating the quality of a software component an apt set of quality attributes for the description of the system (or components) should be selected. Finally, the research issues that can be extended are tabularized.

Keywords—Component, COTS, Component based development, Component-based Software Engineering.

I. INTRODUCTION

COMPONENT-based software development approach is based on the idea to develop software systems by selecting appropriate off-the-shelf components and then to assemble them with a well-defined software architecture. Because the new software development paradigm is very different from the traditional approach, quality assurance (QA) for component-based software development is a new topic in the software engineering community [2].

The purpose of CBSD is to develop large systems, incorporating previously developed or existing components, thus cutting down on development time and costs. It can also be used to reduce maintenance associated with the upgrading of large systems. It is assumed that common parts (be it classes or functions) in a software application only need to be written once and re-used rather than being re-written every time a new application is developed.

Iqbaldeep Kaur is Sr.Lecturer with Computer Science & Engineering Department, Rayat & Bahra Institute of Engineering & Bio-Technology, Sahauran, Distt. Mohali (Punjab)-140104 India (Phone: +91-92161-44321; e-mail: er_iqabldeep kaur@yahoo.com).

Parvinder S. Sandhu is Professor with Computer Science & Engineering Department, Rayat & Bahra Institute of Engineering & Bio-Technology, Sahauran, Distt. Mohali (Punjab)-140104 India (Phone: +91-98555-32004; e-mail: parvinder.sandhu@gmail.com).

Hardeep Singh is Professor with Computer Science the Electrical Engineering Department,,Guru Nnak Dev University, Amritsar (Punjab)-India.

Vandana Saini is Lecturer with Computer Science & Engineering Department, Rayat & Bahra Institute of Engineering & Bio-Technology, Sahauran, Distt. Mohali (Punjab)-140104 India.

II. COMPONENT BASED SOFTWARE ENGINEERING

CBSE embodies the "the 'buy, don't build' philosophy". CBSE is aiming at realizing long-waited software reuse by changing both software architecture and software process. Because of the extensive uses of components, the Component-Based Software Engineering (CBSE) process is quite different from that of the traditional waterfall approach. CBSE not only requires focus on system specification and development, but also requires additional consideration for overall system context, individual components properties and component acquisition and integration process. This work presents an indicative literature survey of techniques proposed for different phases of the CBD life cycle. The aim of this survey is to help provide a better understanding of different CBD techniques for each of these areas [3].

The CBSE generally embodies the following fundamental software development principles:

A. Independent Software Development

Large software systems are necessarily assembled from components developed by different people. To facilitate independent development, it is essential to decouple developers and users of components through abstract and implementation-neutral interface specifications of behavior for components.

B. Reusability

While some parts of a large system will necessarily be special-purpose software, it is essential to design and assemble pre-existing components (within or across domains) in developing new components.

C. Software quality

A component or system needs to be shown to have desired behavior, either through logical reasoning, tracing, and/or testing. The quality assurance approach must be modular to be scalable.

D. Maintainability

A software system should be understandable, and easy to evolve [6],[7].

III. COMPONENT

In defining a software component, this definition can be quoted:

"A component is a software element that conforms to a software model and can be independently deployed and

composed without modification according to a composition standard" [4].

CBSE is about creating a software package in such a manner as to be able to easily reuse its constituent components in other similar or dissimilar applications. It includes writing high level code that glues together pieces of pre-built functionalities or software building blocks called components. Component is one of the parts of the system that make up a system. It may be hardware, software or firmware and may be sub divided into other components [5].

The following figure (Fig. 1) illustrates the architecture when applying this approach. The Connection between components A and B is implemented is through the Connector C. The function of the additional modules, called wrappers is to adjust the component and this way it will better match to the requirements of the rest of the system [11],[12].
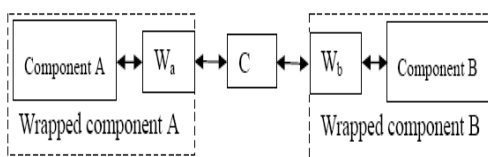

Fig. 1 Architectural approach for component-based systems
Wa – wrapper for component A
Wb – wrapper for component B
C – Connectors

IV. ABSTRACT SOFTWARE COMPONENT MODEL

This abstract model should be perceived as a reference framework for all current models with general definitions and not seen to be specific to any programming language or type of software. Many component-based software applications use such models with the following associated terms and attributes:

A. Syntax

It refers to the 'grammar' or the rules followed in the code as per the specific programming language. In a component model, the language used for the component is specified and the syntax in accordance to that is followed. In most software component models (SCM), the component is defined as a class of function.

B. Semantics

It refers to the actual meaning and view of the components A component is associated with a name, an interface and the body that includes the code.

i) The name of the component describes the use of component with some kind of naming convention. This makes it easier for developers working on large projects to identify existing components in databases, search engines and in the market place.

ii) The interface is the visible area through which information flows to (the input) and from (the output) the component. This information or data is required for the component to perform its operations and is accessible from outside. Usually, the input consists of input values and parameters. This interface requires documentation that contains all the necessary information to use the component.

This documentation also specifies the dependencies to match the required and provided services.

iii)The body or code implements the services that are required of the component and provides the output. This area is generally visible or accessible to the developer from outside.

C. Composition

This relates to the construction and working together of components. Components are subparts of a larger system. To make it simpler, they can be seen as the building blocks, these blocks all assembled, would complete the system.

So, the Software Component model contains a language to compose the system with suitable syntax and semantics to connect with the components. In many cases, with high end programming languages, there are used classifiers or connectors provided in the environment to handle this interaction. The Fig. 2 below shows some examples of component models available to be used.
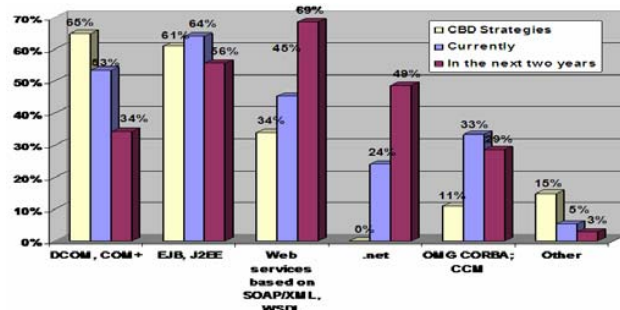

Fig. 2 Component Models

V. LIFE CYCLE

The term component-based software development (CBD) can be referred to as the process for building a system using components. CBD life cycle consists of a set of phases, namely, identifying and selecting components based on stakeholder requirements, integrating and assembling the selected components and updating the system as components evolve over time with newer versions [3].

According to [13], software architecture has four levels of abstraction:
• Internal functionality of components
• Interfaces, exported by a component to the rest of the system
• Interconnection of the architectural elements (components, connectors and Wrappers) in architecture
• Rules for the architectural styles

Component-based software systems are developed by selecting various components and assembling them together rather than programming an overall system from scratch, thus the life cycle of component-based software systems is different from that of the traditional software systems. The life cycle of component-based software systems can be summarized as follows [17, 18]:
• Requirements analysis

- Software architecture selection, construction, analysis, and evaluation;
- Component identification and customization
- System integration
- System testing
- Software maintenance.

VI. ROLES IN CBSE

The followings figure illustrates the percentage amount of roles played by different persons currently and in the next two years in CBSE (see Fig. 3).
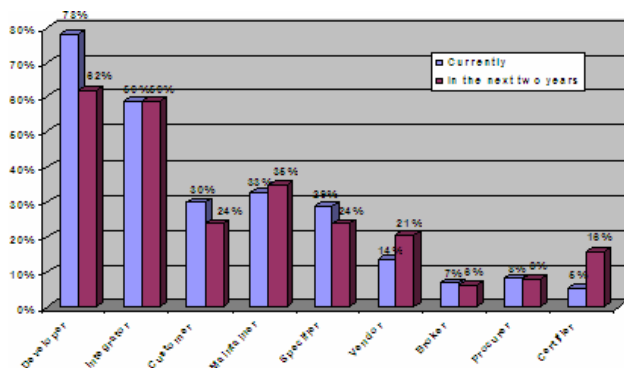

Fig. 3 Roles in CBSE

VII. COMPONENT BASED SOFTWARE DEVELOPMENT (CBSD)

Component-based software development approach is based on the idea to develop software systems by selecting appropriate off-the-shelf components and then to assemble them with a well-defined software architecture. The term component-based software development (CBD) can be referred to as the process for building a system using components [3].

This approach is based on the idea that software systems can be developed by selecting appropriate off-the-shelf components and then assembling them with a well-defined software architecture [17,18]. This new software development approach is very different from the traditional approach in which software systems can only be implemented from scratch. These commercial off-the shelf (COTS) components can be developed by different developers using different languages and different platforms. This can be shown in Fig. 4, where COTS components can be checked out from a component repository, and assembled into a target software system.
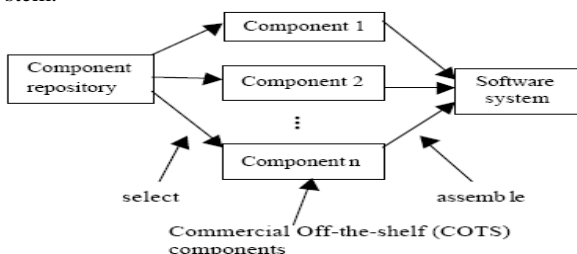

Fig. 4 Component-Based Software Development

VIII. COMPONENT SPECIFICATION EXAMPLE

The following example as mentioned in [7],[8] shows the specification of a *List* type in RESOLVE notation (reproduced from [8]). *List_Template* is a generic **concept** (specification template) which is parameterized by the type of entries to be contained in lists. To provide abstract mathematical explanations of the operations, an object of type *List* **is modeled by** an *ordered pair of mathematical strings* of entries. A string is similar to, but simpler than, a "sequence" because it does not explicitly include the notion of a position. The operator "*" denotes string concatenation; "<*x*>" denotes the string containing the entry *x*; and "|*s*|" denotes the length of *s*. Conceptualizing a *List* object as a pair of strings makes it easy to explain insertion and removal from the "middle". A sample value of a *List* of *Integer*s object, for example, is the ordered pair (<3,4,5>,<4,1>). Insertions and removals can be explained as taking place between the two strings, e.g., at the left end of the right string. The declaration of type *List* introduces the mathematical model and says that an object of type *List* initially (i.e., upon declaration) is "empty": both its left and right strings are empty strings. Each operation is specified by a **requires** clause (precondition), which is an obligation for the caller; and an **ensures** clause (postcondition), which is a guarantee from a correct implementation. In the postcondition of *Insert*, for example, *#s* and *#x* denote the incoming values of *s* and *x*, respectively, and *s* and x denote the outgoing values. *Insert* has no requirement, and it ensures that the incoming value of *x* is concatenated onto the left end of the right string of the incoming value of *s*; the left string is not affected. Notice that the postcondition describes how the operation **updates** the value of *s*, but the return value of *x* (which has the mode **alters**) remains unspecified.

Given this specification, students act as clients and use lists in problem solving within the first few weeks of their second quarter/second semester course. They use a specification-based "natural" or forward reasoning method to reason about correctness [8]. Only later they learn how to implement lists using pointer structures. In addition to classical examples such as Lists, students also see data abstractions that result from recasting classical algorithms as objects [9], and aspects of specification-based interface violation testing using wrapper components [10]. RESOLVE specifications use a combination of standard mathematical models such as integers, sets, functions, and relations, in addition to tuples and strings. The explicit introduction of mathematical models allows use of standard notations associated with those models in explaining the operations. Our experience is that this notation—which is precise and formal—is nonetheless fairly easy to learn to understand even for beginning computer science students, because they have seen most of it before in high school and earlier.

Concept List Template (**type** Entry)
Type List **is modeled by**
(left: **string of** Entry,
right: **string of** Entry)
exemplar s

initialization ensures
|s.left| = 0 **and** |s.right| = 0
Operation Insert (
alters x: Entry
updates s: List
)
ensures s.left = #s.left **and**
s.right = <#x> * #s.right
Operation Remove (
replaces x: Entry
updates s: List
)
requires |s.right| > 0
ensures s.left = #s.left **and**
#s.right = <x> * s.right
Operation Advance (
updates s: List
)
requires |s.right| > 0
ensures s.left * s.right =
#s.left * #s.right **and**
|s.left| = |#s.left| + 1
Operation Reset (
updates s: List
)
ensures |s.left| = 0 **and**
s.right = #s.left * #s.right
Operation Advance To End (
updates s: List
)
ensures |s.right| = 0 **and**
s.left = #s.left * #s.right
Operation Left Length (
restores s: List
): Integer
ensures Left Length = |s.left|
Operation Right Length (
restores s: List
): Integer
ensures Right Length = |s.right|
end List Templat

IX. BENEFITS IN CBSD

The obvious benefit in SR components is the "time-to-market", thus lowering the cost of developing the software. Shorter development cycles would save time as to developing a system from scratch. Developing software systems using CBSE offers many advantages e.g. Development costs are reduced since existing components are used to develop the systems. Reliability is increased since the components have previously been tested in various contexts Time to market is reduced since the components used already exist. Maintenance costs are reduced. Efficiency and flexibility is improved due to the fact that components can easier be added or replaced. The figure below illustrates the major goals and advantages. The main tangible benefits are shorter development life cycle and reduction in IT cost. There are also some intangible benefits like IT adaptability, improved business processes,

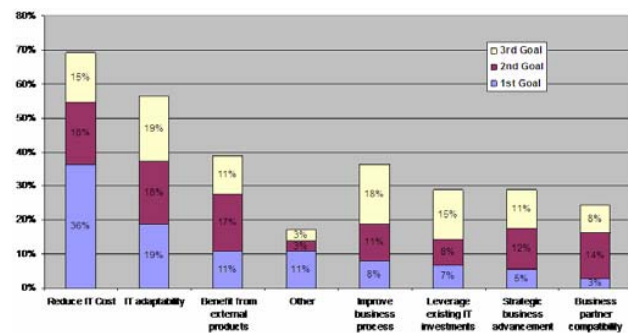benefit from external products and many more as shown in Fig. 5.

Fig. 5 Goals in CBSD

X. DIFFICULTIES IN CBSD

The factors that inhibit the use of components can be followed as shown in the two figures: Fig. 6 and 7 below. The second graph lists the difficulties from managerial and Technical Perspectives.
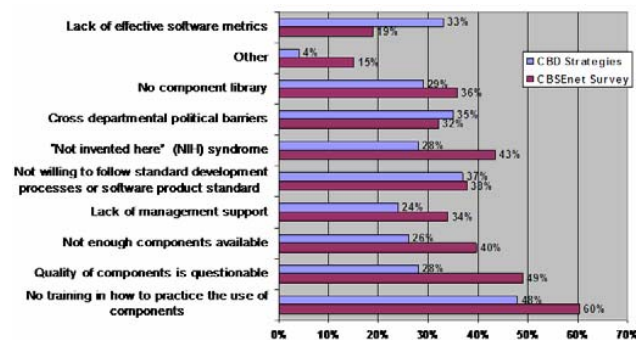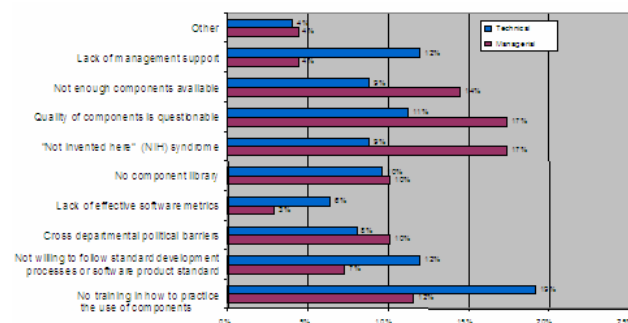

Fig. 6 Inhibitors to the Use of Component

Fig. 7 Inhibitors from Managerial and Technical Perspectives

XI. METRIC USED IN CBSD

The area of software measurement is one of the areas in software engineering where researchers are active from a long time. The area of software measurement is also known as software metrics.

According to IEEE [IEEE 93],"A software metric is a quantitative measure of the degree to which a system, component or process possess a given attribute".[14]

Software metrics are intended to measure the software quality and performance characteristics quantitatively encountered during the planning and execution of software development resource and effort allocation, scheduling and product evaluation. These can serve as measures of software products for the purpose of comparison, cost estimation, fault prediction and forecasting. Poulin [15] presents a set of metrics used by IBM to estimate the efforts saved by reuse. The study suggests the potential benefits against the expenditures of time and resources required to identify and integrate reusable software into a product. Study assumes the cost as the set of data elements like Shipped Source Instructions (SSI), Changed Source Instructions (CSI), Reused source Instructions (RSI) etc. Reuse Percentage measures how much of the product can be attributed to reuse and is given as:-

*Product Reuse Percentage = (RSI / (RSI + SSI)) * 100%*

Cho et al [16] proposes a set of metrics for measuring various aspects of software components like complexity, customizability and reusability. The work considers two approaches to measure the reusability of a component. The first is a metric that measures how a component has reusability and may be used at design phase in a component development process. This metric, Component Reusability (CR) is calculated by dividing sum of interface methods providing commonality functions in a domain to the sum of total interface methods. The second approach is a metric called Component Reusability level (CRL) to measure particular component's reuse level per application in a component based software development. This metric is again divided into two sub-metrics. First is CRLLOC, which is measured by using lines of code, and is expressed as percentage as given as:-

*CRL LOC (C) = (Reuse (C) / Size (C)) *100%*

XII. CURRENT COMPONENT TECHNOLOGIES

Comparison among current component technologies can be found in [17]-[23]. Here is simply a summarization of their different features in Table below.

	CORBA	EJB	COM/DCOM
Development environment	Underdeveloped	Emerging	Supported by a wide range of strong development environments
Binary interfacing standard	Not binary standards	Based on COM; Java specific	A binary standard for component interaction is the heart of COM
Compatibility & portability	Particularly strong in standardizing language bindings; but not so portable	Portable by Java language specification; but not very compatible.	Not having any concept of source-level standard of standard language binding.
Modification & maintenance	CORBA IDL for defining component interfaces, need extra modification & maintenance	Not involving IDL files, defining interfaces between component and container. Easier modification & maintenance.	Microsoft IDL for defining component interfaces, need extra modification & maintenance
Services provided	A full set of standardized services; lack of implementations	Neither standardized nor implemented	Recently supplemented by a number of key services
Platform dependency	Platform independent	Platform independent	Platform dependent
Language dependency	Language independent	Language dependent	Language independent
Implementation	Strongest for traditional enterprise computing	Strongest on general Web clients.	Strongest on the traditional desktop applications

Fig. 8 Comparison of current component technologies

XIII. APPLICATIONS

This emerging component development approach is being widely used in various distinct domains .As it can be observed from the bar graph shown (see Fig. 9), the approach is vividly applied in the field of software development /CASE.
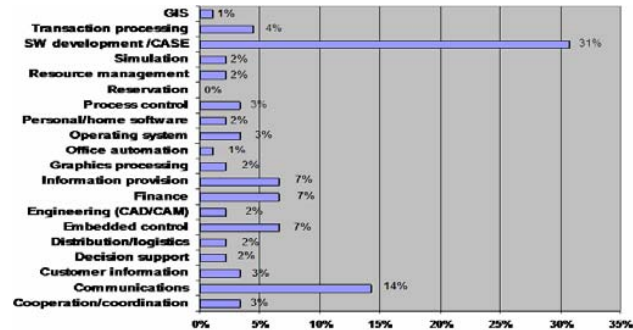

Fig. 9 Representative Domains

XIV. CONCLUSION

CBSD is an inevitable next wave solution that has potential to improve time –to-market and man power/cost trends that have been ongoing. CBSD is best implemented using more modern software technologies like:

- COM
- JAVA
- EJB
- CORBA
- ActiveX

There is little that would prevent virtually any technology from implementing CBD so long as component conforms to the basic requirements of its definition [5].

XV. SCOPE AND PROPOSED WORK IN CBSD

The above discussion leads to the following topics that can be worked upon in future:

1) Generation and adaptation of component-based systems
2) Components and model-driven development
3) Specification, verification, testing and checking of component systems
4) Compositional reasoning techniques for component models
5) Measurement and prediction models for component assemblies
6) Patterns and frameworks for component-based systems
7) Extra-functional system properties of components and component-based systems
8) Static and execution-based measurement of system properties
9) Assurance and certification of components and component-based systems
10) Components for service-oriented architectures, web services and grid systems

11) Development environments and tools for building component-based systems

12) Components for real-time, secure, safety critical and/or embedded systems

13) Case studies and experience reports

REFERENCES

[1] CBSE Network, "Component based software engineering workshop", Budapest April 3-4

[2] APSEC2000, "Software Engineering Conference", Proceedings, Seventh-Asia-Pacific, 2000.

[3] www.scitation.aip.org/getabs.

[4] George T. Heineman and William T. Councill, "Component-Based Software Engineering Putting the Pieces Together", Addison-Wesley, Boston, MA ,880, June 2001.

[5] Sajan Mathew, "Software Engineering", Edition 2nd S.Chand.

[6] M. Sitaraman and B. W. Weide , "Special Feature Component-Based Software Using RESOLVE", ACM SIGSOFT Software Engineering Notes 19, No. 4, 21-67, October 1994.

[7] Murali Sitaraman, Timothy J. Long ,E. James Harner. Bruce W. Weide, "A Formal Approach to Component-Based Software Engineering Education and Evaluation", In ICSE 2001: Proceedings 23rd International Conference on Software Engineering}, pp. 601-609, 2001.

[8] Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B. W., Long, T. J., Bucci, P., Heym, W., Pike, S., and Hollingsworth, J. E., "Reasoning About Software Component Behavior", Proceedings Sixth International Conference on Software Reuse, Springer Verlag LNCS 1844, 266-283, 2000.

[9] Sitaraman, M., Weide, B. W., Long, T.J., Ogden, W. F., "A Data Abstraction Alternative to Data Structure/Algorithm Modularization", Volume on Generic Programming, LNCS 1766 608, 102-113 Springer-Verlag, 2000.

[10] Edwards, S., Shakir, G., Sitaraman, M., Weide, B. W., and Hollingsworth, J., "A Framework for Detecting Interface Violations in Component-Based Software", Proceedings of the Fifth International Conference on Software Reuse, IEEE Computer Society Press, Victoria, Canada, pp. 46-55, June 1998.

[11] Aleksandar Dimov and Sylvia Ilieva, "System level modeling of component based software systems", International Conference on Computer Systems and Technologies - CompSysTech- II.7-1,2004.

[12] Dean, J. and M. Vigder, "System Implementation using Commercial-Off-The-Shelf (COTS) Software",1997. URL: http://seg.iit.nrc.ca/papers/NRC40173.pdf.

[13] N. Medvidovic, R. Taylor, and E. Whitehead, "Formal Modeling of Software Architectures at Multiple Levels of Abstraction", In Proceedings of the California Software Symposium 1996, Los Angeles, CA, pp. 28-40., April 1996.

[14] Arun Sharma, Rajesh Kumar, and P. S. Grover, "A Critical Survey of Reusability Aspects for Component-Based Systems" , Proceedings of world academy of science engineering and technology ,volume 21, January 2007.

[15] J. Poulin, J Caruso and D Hancock, "The Business Case for Software Reuse", IBM Systems Journal, 32(40),567-594, 1993.

[16] Eun Sook Cho et al., "Component Metrics to Measure Component Quality ", Proceedings of the eighths Asia-Pacific Software Engineering Conference, 1530-1362,2001.

[17] Xia Cai, Michael R. Lyu, Kam-Fai Wong Roy Ko ,"Component-Based Software Engineering Technologies Development Frameworks and Quality Assurance Schemes", The Chinese University of Hong Kong Hong Kong Productivity Council.

[18] G. Pour, "Component-Based Software Development Approach: New Opportunities and Challenges", Proceedings Technology of Object-Oriented Languages, TOOLS 26.,pp. 375-383,1998.

[19] A.W.Brown, K.C. Wallnau, "The Current State of CBSE", IEEE Software ,Volume:15 5, pp. 37-46.,Sept.-Oct. 1998.

[20] G. Pour, "Enterprise JavaBeans, JavaBeans & XML Expanding the Possibilities for Web-Based Enterprise Application Development", Proceedings Technology of Object-Oriented Languages and Systems, TOOLS 31, pp.282-291. 1999.

[21] G.Pour, M. Griss, J. Favaro, "Making the Transition to Component-Based Enterprise Software Development: Overcoming the Obstacles – Patterns for Success", Proceedings of Technology of Object-Oriented Languages and systems, pp.419 – 419, 1999.

[22] G. Pour, "Software Component Technologies JavaBeans and ActiveX", Proceedings of Technology of Object-Oriented Languages and systems, pp. 398 – 398, 1999.

[23] C. Szyperski, "Component Software: Beyond Object- Oriented Programming", Addison-Wesley, New York, 1998.

[24] Roger S. Pressman, Software Engineering, "A Practitioner's Approach",Sixth Edition,Tata McGraw Hill.

[25] Rajiv Mall, "Software Engineering" ,2nd edition , PHI.