Component Based Software Engineering

[1]Dimple Nagpal, [2]Iqbaldeep Kaur, [3]Ms. Apoorva, [4]Ms. Sheilly, [5]Ms. Sonali
[1]Research Scholar, [2]Associate Professor, [3]Assistant Professor, [4]Assistant Professor, [5]Assistant Professor
[1, 2 ,3, 4, 5]Department of Computer Science Engineering, Chandigarh Engineering College, Landran, Mohali , India

Abstract -This paper is about the overview of component based software engineering. In this the various study of what the component is, and how the components are used in the process are there,and pros and cons of CBSE are described here.The research issue to analyse or study the whole CBSE and understand whether technologies merge in both the domain in CBSE and give the efficient and effective results

Keywords - Component, Component based development, Component-based Software Engineering, Component-based Software Development.

I. INTRODUCTION

The concept of building software from components is not new. A 'classical' design of complex software systems always begins with the identification of system parts designated subsystems or blocks, and on a lower level modules, classes, procedures and so on[3]. Component-based software engineering strives to achieve the same thing. A set of prebuilt, standardized software components are made available to fit a specific architectural style for some application domain[1].The purpose of CBSD is to develop large systems,incorporating previously developed or existing components,thus cutting down on development time and costs. 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[2].

II. COMPONENT BASED SOFTWARE ENGINEERING

Component-based software engineering (CBSE) is a process that emphasizes the design and construction of computer-based systems using reusable software "components." CBSE embodies the "buy, don't build" philosophy[1].Basically CBSE aims to focus on component,that can further be used on different software thus yielding better results in an efficient and effective manner . So that the overall development of the software is improved by reusing the components.

CBSE shifts the emphasis from programming software to composing software systems. Implementation has given way to integration as the focus[1]. 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[2]. Therefore CBSE must address both the development of reusable components and the development application using the reusable components .component based development shows the perspective of software reuse[4].

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

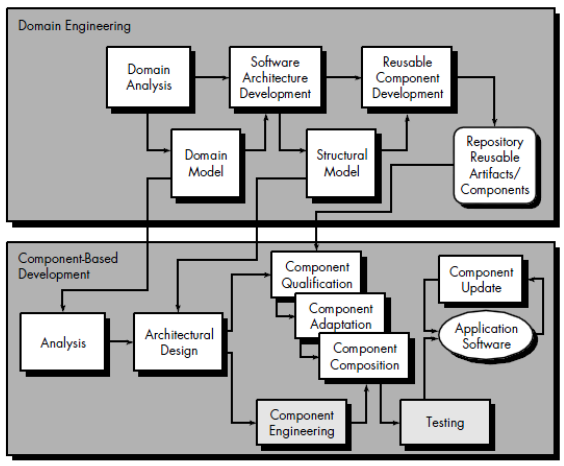
A component or system needs to be shown to have desired behavior, either through logical reasoning, tracing, and/or testing. The quality assurance approach must be modular to be scalable.

D. Maintainability

A software system should be understandable, and easy to evolve [2].

III. CBSE PROCESS

The CBSE process, however, must be characterized in a manner that not only identifies candidate components but also qualifies each component's interface, adapts components to remove architectural mismatches, assembles components into a selected architectural style, and updates components as requirements for the system change. The process model for component-based software engineering emphasizes parallel tracks in which domain engineering occurs concurrently with component-based development[1]. The intent of domain engineering is to identify, construct, catalog, and disseminate a set of software components that have applicability to existing and future software in a particular application domain. The overall goal is to establish mechanisms that enable software engineers to share these components—to reuse them—during work on new and existing systems[1]. So domain engineering team is responsible for producing, maintaining and cataloguing reusable assets[4].Domain Engineering aims at supporting application engineering which uses the domain models and architectures to build concrete systems. The emphasis is on reuse and product lines. The Domain-Specific Software Architecture (DSSA) engineering process was introduced to promote a clear distinction between domain and application requirements [11]. A Domain-Specific Software Architecture consists of a domain model and a reference architecture, and guides in reusing components[6].Domain engineering includes three major activities—analysis, construction, and Dissemination[1].



IV. THE DOMAIN ANALYSIS PROCESS

Domain analysis is the process of identifying, collecting, organizing, and representing the relevant information in a domain, based upon the study of existing systems and their development histories, and knowledge captured from domain experts[6].

The steps in the process were defined as:

1. Define the domain to be investigated.
2. Categorize the items extracted from the domain.
3. Collect a representative sample of applications in the domain.
4. Analyze each application in the sample.
5. Develop an analysis model for the objects.

It is important to note that domain analysis is applicable to any software engineering paradigm and may be applied for conventional as well as object-oriented development[1].

V. THE DOMAIN CONSTRUCTION PROCESS

It is sometimes difficult to determine whether a potentially reusable component is in fact applicable in a particular situation. To make this determination, it is necessary to define a set of domain characteristics that are shared by all software within a domain. A domain characteristic defines some generic attribute of all products that exist within the domain. For example, generic characteristics might include the importance of safety/reliability, programming language, concurrency in processing, and many others. A set of domain characteristics for a reusable component can be represented as {*Dp*}, where each item, *Dpi*, in the set represents a specific domain characteristic. The value assigned to *Dpi* represents an ordinal scale that is an indication of the relevance of the characteristic for component *p*. A typical scale might be

1: not relevant to whether reuse is appropriate
2: relevant only under unusual circumstances
3: relevant—the component can be modified so that it can be used, despite differences
4: clearly relevant, and if the new software does not have this characteristic, reuse will be inefficient but may still be possible
5: clearly relevant, and if the new software does not have this characteristic, reuse will be ineffective and reuse without the characteristic is not recommended[1].

VI. THE DOMAIN DISSEMINATION PROCESS

When domain analysis is applied, the analyst looks for repeating patterns in the applications that reside within a domain. *Structural modeling* is a pattern-based domain engineering approach that works under the assumption that every application domain has repeating patterns (of function, data, and behavior) that have reuse potential Structural models consist of a small number of structural elements manifesting clear patterns of interaction. The architectures of systems using structural models are characterized by multiple ensembles that are composed from these model elements. Many architectural units emerge from simple patterns of interaction among this small number
of elements[1].

VII. COMPONENT BASED DEVELOPMENT

Component-based development is a CBSE activity that occurs in parallel with domain Engineering [1]. The architecture of a software system defines that system in terms of components and interactions/connections among those components. It is not the design of that system which is more detailed. The architecture shows the correspondence between the requirements and the constructed system, thereby providing some rationale for the design decisions.

Different views on component-based software architectures may be distinguished:

1. Design-time: This includes the application-specific view of components, such as functional interfaces and component dependencies.
2. Compose-time: This includes all the elements needed to assemble a system from components, including generators and other build-time services; a component framework may provide some of these services.
3. Runtime: This includes frameworks and models that provide runtime services for component-based systems[6].

Once the architecture has been established, it must be populated by components that

(1) are available from reuse libraries and/or
(2) are engineered to meet custom needs.

Hence, the task flow for component-based development has two parallel paths. When reusable components are available for potential integration into the architecture, they must be qualified and adapted. When new components are required, they must be engineered. The resultant components are then "composed" (integrated) into the architecture template and tested thoroughly [1].

COMPONENT

A nontrivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture [1].

A Component is:

• An opaque implementation of functionality
• Subject to third-party composition
• Conformant with a component model

There are two motivations for the criterion that a component is an *opaque implementation*.

- First, we envision a commercial market in software components. Notwithstanding the suc-cess of Linux and "open source" software, the predominant and most successful businessmodel for software components has been based upon software as intellectual capital that must be protected from disclosure. Second, as is already a well-established precept in computer science, clients of software components should not come to rely upon implementation details that are likely to change. In computer science this has led to programming support for abstraction and information hiding; opaqueness serves the same purpose for components.
- The motivation for third-party composition is straightforward: the use of components should not depend upon tools or knowledge of the component that is in the possession of only the component provider. This criterion implies that a component-based system can comprise components from multiple, independent sources, and that a system can be assembled by a third party system integrator who is not also a component supplier. This criterion should hold true even if none of the components used in a system come from external suppliers.
- The last criterion, that a component is conformant with a component model, is what differentiates components from conventional COTS software products. Component models prescribe how components interact with each other, and therefore express global, or architectural design constraints. Conformance to component models transforms software implementations
into architectural implementations[5].

COMPONENT QUALIFICATION, ADAPTATION, AND COMPOSITION

Domain engineering provides the library of reusable components that are required for component-based software engineering. Some of these reusable components are developed in house, others can be extracted from existing applications, and still others may be acquired from third parties. Unfortunately, the existence of reusable components does not guarantee that these components can be integrated easily or effectively into the architecture chosen for a new application. It is for this reason that a sequence of component-based development activities are applied when a component is proposed for use[1].

COMPONENT QUALIFICATION

Component qualification ensures that a candidate component will perform the function required, will properly "fit" into the architectural style specified for the system,and will exhibit the quality characteristics (e.g., performance, reliability, usability) that are required for the application. The interface description provides useful information about the operation and use of a software component, but it does not provide all of the information required to determine if a proposed component can, in fact, be reused effectively in a new application. Among the many factors considered during component qualification are:

- Application programming interface (API).
- Development and integration tools required by the component.
- Run-time requirements, including resource usage (e.g., memory or storage), timing or speed, and network protocol.
- Service requirements, including operating system interfaces and support from other components.
- Security features, including access controls and authentication protocol.
- Embedded design assumptions, including the use of specific numerical or nonnumerical algorithms.
- Exception handling.

COMPONENT ADAPTATION

In an ideal setting, domain engineering creates a library of components that can be easily integrated into an application architecture. The implication of "easy integration" is that
(1) consistent methods of resource management have been implemented for all components in the library,
(2) common activities such as data management
exist for all components, and
(3) interfaces within the architecture and with the external environment have been implemented in a consistent manner.
In reality, even after a component has been qualified for use within an application architecture, it may exhibit conflict in one or more of the areas just noted. To mitigate against these conflicts, an adaptation technique called *component wrapping* is often used. When a software team has full access to the internal design and code for a component (often not the case when COTS components are used) white-box wrapping is applied. Like its counterpart in software testing, *white-box wrapping* examines the internal processing details of the component and makes code-level modifications to remove any conflict. *Gray-box wrapping* is applied when the component library provides a component extension language or API that enables conflicts to be removed or masked. *Black-box wrapping* requires the introduction of pre- and post processing at the component interface to remove or mask conflicts.

COMPONENT COMPOSITION

The *component composition* task assembles qualified, adapted, and engineered components to populate the architecture established for an application. To accomplish this, an infrastructure must be established to bind the components into an operational system. The infrastructure (usually a library of specialized components) provides a model for the coordination of components and specific services that enable components to coordinate with one another and perform common tasks Among the many mechanisms for creating an effective infrastructure is a set of four "architectural ingredients" that should be present to achieve component composition:

Data exchange model: Mechanisms that enable users and applications to interact and transfer data (e.g., drag and drop, cut and paste) should be defined for all reusable components. The data exchange mechanisms not only allow human-to-software and component-to-component data transfer but also transfer among system resources (e.g., dragging a file to a printer icon for output).

Automation: A variety of tools, macros, and scripts should be implemented to facilitate interaction between reusable components.

*Structured storage:*Heterogeneous data (e.g., graphical data, voice/video,text, and numerical data) contained in a "compound document" should be organized and accessed as a single data structure, rather than a collection of separate files. "Structured data maintains a descriptive index of nesting structures that applications can freely navigate to locate, create, or edit individual data contents as directed by the end user" .

Underlying object model: The object model ensures that components developed in different programming languages that reside on different platforms can be interoperable. That is, objects must be capable of communicating across a network. To achieve this, the object model defines a standard for component interoperability [1].

VIII. STANDARDS FOR SOFTWARE COMPONENTS

Because the potential impact of reuse and CBSE on the software industry is enormous, a number of major companies and industry consortia3 have proposed standards for component software:

OMG/CORBA:The Object Management Group has published a *common object request broker architecture* (OMG/CORBA). An object request broker (ORB) provides a variety on services that enable reusable components (objects) to communicate with other components, regardless of their location within a system. When components are built using the OMG/CORBA standard,
integration of those components (without modification) within a system is assured if an *interface definition language* (IDL) interface is created for every component. Using a client/server metaphor, objects within the client application
request one or more services from the ORB server. Requests are made via an IDL or dynamically at run time. An interface repository contains all necessary information about the service's request and response formats.

Microsoft COM: Microsoft has developed a component object model (COM) that provides a specification for using components produced by various vendors within a single application running under the Windows operating system. COM encompasses two elements: COM interfaces (implemented as COM objects) and a set of mechanisms for registering and passing messages between COM interfaces. From the point of view of the application, "the focus is not on how [COM objects are] implemented, only on the fact that the object has an interface that it registers with the system, and that it uses the component system to communicate with other COM objects".

Sun JavaBean Components: The JavaBean component system is a portable, platform independent CBSE infrastructure developed using the Java programming language. The JavaBean system extends the Java applet4 to accommodate the more sophisticated software components required for component-based development. The JavaBean component system encompasses
a set of tools, called the *Bean Development Kit* (BDK), that allows developers to
(1) Analyze how existing Beans (components) work,
(2) Customize their behaviour and appearance,
(3) Establish mechanisms for coordination
and communication,
(4) Develop custom Beans for use in a specific
application, and
(5) Test and evaluate Bean behaviour.

Which of these standards will dominate the industry? There is no easy answer at this time. Although many developers have standardized on one of the standards, it is likely that large software organizations may choose to use all three standards, depending on the application categories and platforms that are chosen[1].

IX. BENEFITS IN CBSD

Software developers create software components mainly with an intention of being reused in various software systems. Components are designed to interact with its environment through its well-defined interfaces but to encapsulate their implementation. Component-based software development brings the potential for

1. significant reduction in the development cost and time-to-market of enterprise software systems because developers can assemble such systems from a set of reusable components rather than building them from scratch,
2. increasing the reliability of enterprise software systems - each reusable component undergoes several review and inspection stages in the course of its original development and previous uses, and CBSD relies on explicitly defined architectures and interfaces,
3. improving the maintainability of enterprise software systems by allowing new, higher-quality components to replace old ones, and

4. enhancing the quality of enterprise software systems - application-domain experts develop components, then software engineers who specialize in Component software engineering assemble those components into enterprise software systems[7].

X. DIFFICULTIES IN CBSD

The disadvantages of Component Based Software Development are as follows:

1) Finding suitable components which fit the architectural design of the software to be developed may be sometimes difficult because gaps exist between the component features and the software requirements.

2) Component Based development has not been widely adopted in domains of embedded systems because of inability of this technology to cope with the important concerns of embedded systems like resource constraints, real time or dependability requirements.

3) Component Based Software Engineering is young; therefore long term maintainability is still unknown [8].

XI. CONCLUSION

Component-based systems are the result of structuring a system according to a particular design pattern. This design pattern relies upon a number of closely related technical concepts,including: *components*, *component models*, *component composition*,component *interfacesand so on.* CBSE focuses more on the reusability of components rather than rebuilding of components every time whenever a new software is being build.

XII. REFRENCES

[1]. Hasselbring, Wilhelm. "Component-based software engineering." (2002): 289-305.

[2]. Kaur, Iqbaldeep, Parvinder S. Sandhu, Hardeep Singh, and Vandana Saini. "Analytical study of component based software engineering." *World Academy of Science, Engineering and Technology* 50 (2009): 437-442.

[3]. Goel, Shivani Guide. "Designing A Framework for Handling Barriers to Software Reuse."

[4]. Book: software engineering: a practitioner’s approach by roger pressman.

[5]. BOOK: VolumeII software engineering.

[6]. Ishita Verma , International Journal of Computer Science & Communication Networks,Vol 4(3),84-88 87 ISSN:2249-5789 .

[7]. Internet Source