Component-Based Software Development

Virendra Kumar Sharma[+], Narendra Prakash Gupta[++]

[+] Faculty of Engineering, BIT, Bhagwantpuram, Muzaffarnagar -UP, India
[++]School of Computer Science, Bhawant University , Ajmer – India

Summary
Component-based software development (CBSD) or component-based software engineering (CBSE) is concerned with the assembly of pre-existing software components into larger pieces of software. Underlying this process is the notion that software components are written in such a way that they provide functions common to many different systems. Borrowing ideas from hardware components, the goal of CBSD is to allow parts (components) of a software system to be replaced by newer, functionally equivalent, components

Key words:
API - Application Programming Interface
CBD - Component Based Development
CBSE - Component-based Software Engineering
COM - Component Object Model
CORBA - Common Object Request Broker Architecture
COSE - Component-oriented Software Engineering
COTS - Commercial Off-the-Shelf
UML - Unified Modeling Language

1. Introduction

Component-based software engineering (CBSE) is an approach to software development that relies on software reuse. It emerged from the failure of object-oriented development to support effective reuse. Single object classes are too detailed and specific. Components are more abstract than object classes and can be considered to be stand-alone service providers. Apart from the benefits of reuse, CBSE is based on sound software engineering design principles Components are independent so do not interfere with each other. Component implementations are hidden. Communication is through well-defined interfaces. Component platforms are shared and reduce development costs.

2. Component

A software component is a program element with the following two properties:
• It may be used by other program elements, or clients.
• The clients and their authors do not need to be known to the component's authors.
•A component is a non-trivial, nearly independent, replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.

Manuscript received November 5, 2010
Manuscript revised November 20, 2010

3. Software Development

Component based software development encompasses two processes
a. Assembling software systems from software components and
b. Developing reusable components.

The activity of developing systems as assemblies of components may be broadly classified in terms of four activities.
3.1Component qualification,
3.2 Component adaptation,
3.3 Component assembly, and
3.4 System evolution and maintenance.

Although the process differs from traditional software development, there are still problems common to both CBSD and traditional methods. For example, the problem of managing change and maintenance in developed software. In the following subsections a brief description of each of the above activities is given.

3.1 Component Qualification

Qualification is the process for determining the suitability of a component for use within the intended final system. When a marketplace of competing products exists, it also involves the selection of the most suitable component. Selection is dependent on the condition that measures exist for comparing one component against another and evaluating the fitness of use of components. It is during this activity that the issues of trust and certification arise. The process of certification is two-fold (Bachman et al. 2000):
i). To establish facts about a component and to determine that the properties a component possesses is also conformant with its published specification; and
ii). To establish trust in the validity of these facts, perhaps by having a trusted third-party organization attest

the truth of this conformance and to provide a certificate to verify this.
The motivation for component certification is that there is a causal link between a component's certified properties and the properties of the end system. If enough is known about the (certified) components selected for assembly then it may be possible to predict the properties of the final assembled system. Accuracy in prediction is founded on the degree of trust in a component and also how well the glue that joins the components is understood. For many of components in the marketplace prediction is difficult because of lack of information about a component's capabilities and lack of trust in this information.

Conventional software doctrine states that component specifications should be sufficient and complete, static—writable once and frozen, and homogenous. However, full specifications may be impractical: some components may exhibit (non-functional) properties which are infeasible to document, let alone to document in a homogenous notation (some practitioners go as far to say that reusable components do not exist yet). One method for addressing this issue is to use credentials (Shaw 1996) —knowledge-based specifications that evolve as more is discovered about a component.

3.2 Component Adaptation

Individual components are written to meet different requirements, each one making certain assumptions about the context in which it is deployed. The purpose of adaptation is to ensure that conflicts among components are minimized. Different approaches to adaptation depend upon the accessibility of the internal structure of a component.

White-box components may be significantly rewritten to operate with other components.

Grey-box components provide their own extension language or application programming interface (API).

Black-box, or binary, components have no extension language or API.

Ideally, a component is a black box, its services are only accessible through some well-defined interface. However, as shall be seen in the sequel, there is nothing to stop us from considering white or grey box components.

3.3 Component Assembly

Assembly is the integration of components through some well defined infrastructure, which provides the binding that forms a system from disparate components. COTS components, for example, are usually written to some component model defined by e.g., Enterprise JavaBeans, COM, CORBA, or, more recently, .NET.

3.4 System Evolution and Maintenance

Because components are the units of change, system evolution is based around the replacing of outdated components by new ones, or, at least, ideally. The treatment of components as plug-replaceable units is a simplistic view of system evolution. In practice, replacing a component may be a non-trivial task, especially when there is a mismatch between the new component and old one, triggering another stage of adaptation with the new component.

4. Software Components

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies. An interface is a set of named operations that can be invoked by clients. Context dependencies are specifications of what the deployment environment needs to provide, such that the components can function.
Most definitions converge to some degree but differences arise in the fact that there are different conceptions about what a component can be. Traditionally, a component is considered to be an implementation-level unit of deployment. However, it is not unheard of for some to consider abstract specifications or even design documents to be components in their own right. Reusable artifacts may occur at any level of the development cycle, in particular there is an increased interest in business-level components which are independent of any specific implementation or middleware technology. This has prompted the distinction between component-based systems engineering and component-based software engineering, where the former refers to the process of reusing requirements and specifications. There, changes are made at the requirements or specification level and code where they may be validated and the resulting code may either be generated automatically or manually.

5. CBSD Research Areas

As alluded to in the preceding sections, component-based software development encompasses many areas of research, some of which have been active research areas in traditional software development paradigms. Below are descriptions of some of these research topics.

5.1 Component Modeling and Specification

The Unified Modeling Language (UML) has become the de facto standard for nearly all application modeling and

has been employed in CBD methods such as the Catalysis method Prior to UML version 2.0. However, there has been a need to extend UML to allow for more appropriate descriptions of a component's dependencies and also of its interface specification. These issues have been addressed in UML Components.
Besides UML, however, there is a large body of work on how the component-based development process can be approached formally. Specification-wise this work includes the specification of a component's interface — its operations' functional specifications or the specification of its interaction protocols, i.e., constraints on how and when an operation may be invoked.

5.2 Retrieval Techniques and Specification Matching

The issue of how to retrieve reusable artifacts has long been a research area in software reuse. Much of the work on retrieval has focused on what forms component descriptions should take in order that components can be retrieved from repositories while specification matching techniques are employed to search for components based on functional or behavioral criteria.

5.3 Generative Approaches to Component Development

Generative approaches are concerned with the generation of software from specifications. These techniques are employed within the context of component-based systems engineering described above. A brief mission statement on generative and component-based software engineering and associated research areas see the home page of the Working Group on Generative and Component Based Software Engineering.

5.4 Adaptation Techniques

As mentioned previously, different adaptation techniques can be employed for black, grey, or white box components. Research on adaptation can range from wrapping techniques to more sophisticated methods such as identifying appropriate adapters for specification matching. Related work on adaptation techniques includes how the reusability of a component can be improved by considering how freely they can be adapted while they are being developed.

5.5 Coordination and Composition Languages

In the absence of a defined component infrastructure, e.g., COM and CORBA, coordination and composition languages may be used to describe the wiring or glue for component assembly. These languages may also be used to define the ways in which software may be composed within some given framework or how components interact across systems.

5.6 Verification, Testing, and Certification

CBSD implies that a component undergoes two test phases. The first phase occurs during development, verifying whether a component meets its specification and fulfils its functional requirements. A component may be certified by a third party according to how the component performs during this round of testing. The second phase is concerned with testing how the component integrates with others during the development of a component-based system. Different testing strategies may be employed according to the visibility of the component's internal structure. Usually, for COTS components, black-box techniques are employed. For source code components, white-box techniques may be used.

6. Conclusion

Component-based approach can increase software-building performance with effective component repository. This implementation can provide interoperability between two different systems. With this approach maintenance cost is expected to be reduced since components are designed to be independent.

Acknowledgment

The authors would like to express their cordial thanks to S. P. Gupta for his valuable advice.

References

[1] PRESSMAN Roger, Software Engineering, McGraw Hill.
[2] SCHACH Stephen R., Classical and Object Oriented Software Engineering,4th Edition, McGraw Hill.
[3] SZYPERSKI Clemens, Component Software – Beyond Object-Oriented Programming, Addison Wesley.