

# Component Based Software Engineering

*A look at reusable software components*

<http://www.smb.uklinux.net/reusability>

## **What is CBSE?**

You purchase a "stereo system" and bring it home. Each component has been designed to fit a specific architectural style - connections are standardized, a communication protocol has been pre-established. Assembly is easy because you don't have to build the system from hundreds of discrete parts. Component-based software engineering (CBSE) strives to achieve the same thing. A set of pre-built, standardized software components are made available to fit a specific architectural style for some application domain. The application is then assembled using these components, rather than the "discrete parts" of a conventional programming language.

Making applications from software components has been a dream in software engineering community since its very early time. Programmers have relied on the reuse of code and data structures from as far back as 1968, when M.D. McILROY published a paper titled "Mass-Produced Software Components".

In the Nato conference, in 1968, McIlroy wrote, *"My thesis is that the software industry is weakly founded, in part because of the absence of a software components subindustry. ... A components industry could be immensely successful"*. However, wide spread reuse of software components over the industry has not come true.

Why? A number of obstacles have been identified. The most fundamental problems are lack of mechanisms to make components interoperable and lack of "really reusable" components.

Over the years, software reusability has become a research area in its own right – and more recently has emerged into what is now known as CBSE.

CBSE is a process that aims to design and construct software systems using reusable software components. Clements describes CBSE as embodying *"the 'buy, don't build' philosophy"*. He also says that *"in the same way that early subroutines liberated the programmer from thinking about details, [CBSE] shifts the emphasis from programming to composing software systems"*.

If one is familiar with Object oriented programming (OOP), it can be useful to think of CBSE in a similar way. In OOP, code is reused in the form of objects. These objects are often contained in vast libraries of reusable code. Frameworks take the process even further, providing more robust and disciplined systems of reuse. By obtaining and reusing parts of systems which have already been 'tried and tested', we can exploit the principal advantages of object-oriented programming techniques over procedural programming techniques, which is that they enable programmers to create modules<sup>1</sup> that do not need to be changed when a new type of object is added. A

---

<sup>1</sup> In software, a module is a part of a program. Programs are composed of one or more independently developed modules that are not combined until the program is linked. A single module can contain one or several routines.

programmer can simply create a new object that inherits many of its features from existing objects. This makes object-oriented programs easier to modify. In the same way, in CBSE, by reusing an existing component you cut out a lot of the hard work with establishing the usefulness and in testing that component – although, as we will see, some testing will still be required.

### ***Why CBSE?***

The goal of component-based software engineering is to increase the productivity, quality, and time-to-market in software development thanks to the deployment of both standard components and production automation. One important paradigm shift implied here is to build software systems from standard components rather than "reinventing the wheel" each time. This requires thinking in terms of system families rather than single systems.

CBSE uses Software Engineering principles to apply the same idea as OOP to the whole process of designing and constructing software systems. It focuses on *reusing* and *adapting* existing components, as opposed to just coding in a particular style. CBSE encourages the composition of software systems, as opposed to programming them.

Early approaches to reusability have been adhoc. There hasn't been any standards adopted by the industry as a whole, and without standards, it has been difficult for developers to be motivated to design with reusability in mind.

Increasingly, complex systems now need to be built with shorter amounts of time available. The "buy, don't build approach" is more important and popular than ever.

Over the past decade, many people have attempted to improve software development practices by improving design techniques, developing more expressive notations for capturing a system's intended functionality, and encouraging reuse of pre-developed system pieces rather than building from scratch. Each approach has had some notable success in improving the quality, flexibility, and maintainability of application systems, helping many organizations develop complex, mission-critical applications deployed on a wide range of platforms.

Despite this success, any organization developing, deploying, and maintaining large-scale software-intensive systems still faces tremendous problems, especially when it comes to testing and updating the systems. Unless carefully designed, it can be very expensive (in time as well as cost) to add further functionality to a system effectively, and then to test whether the addition has been successful. Furthermore, in recent years, the requirements, tactics, and expectations of application developers have changed significantly. They are more aware of the need to write reusable code – even though they may not always employ this practice.

Two important aspects of the question "why CBSE now?" are:

First, several underlying technologies have matured that permit building components and assembling applications from sets of those components. Object oriented and Component technology are examples – especially standards such as CORBA (discussed later).

Second, the business and organizational context within which applications are developed, deployed, and maintained has changed. There is an increasing need to communicate with legacy systems, as well as constantly updating current systems. This need for new functionality in current applications requires technology that will allow easy additions.

CBSE should, in theory, allow software systems to be more easily assembled, and less costly to build. Although this cannot be guaranteed, the limited experience of adopting this strategy has shown it to be true. The software systems built using CBSE are not only simpler to build and cheaper – but usually turn out to be more robust, adaptable and updateable.

CBSE allows use of predictable architectural patterns and standard software architecture leading to a higher quality end result.

### ***What is a component?***

Brown and Wallnau describe a component as “*a nontrivial, nearly independent, and replaceable part of a system that fulfils a clear function in the context of a well-defined architecture*”. In some ways, this is similar to the description of an object in OOP. Components have an interface. They employ inheritance rules. But the definition is taken even further. Components are defined to offer some level of service (a Service Level Agreement, perhaps). In the case of COTS (Commercial off-the-shelf components), little or nothing is known about the internal workings of the component by the software engineer. Instead, the software engineer is given only a well-defined external interface from which he must work. The level of service offered is therefore crucial and must be accurate if the integration of the component into the software system is to be successful. Brown and Wallnau describe a software component as “*a unit of composition with contractually specified and explicit context dependencies only*”. Unlike objects in OOP, components are usually built up of many software “objects” (although are not confined to OOP) and provide a coherent unit of functionality. These so-called “objects” all work together to perform a specific task at a particular level of service.

Components can be characterized based on their use in the CBSE process:

As mentioned above we have *commercial off-the-shelf* (or COTS) components. These are components that can be purchased, pre-built, with the disadvantage that there is usually no source code available, and so the definition of the use of the component given by the manufacturer, and the services which it offers, must be relied upon or thoroughly tested, as they may or may not be accurate. The advantage, though, is that these types of components should (in theory) be more robust and adaptable, as they have been used and tested (and reused and retested) in many different applications.

In addition to COTS components, the CBSE process yields:

- qualified components
- adapted components
- assembled components

- updated components

definitions of which can be found in Pressman, P741.

### ***The CBSE Process***

CBSE is in many ways similar to conventional or object-oriented software engineering. A software team establishes requirements for the system to be built using conventional requirements elicitation techniques. An architectural design is established. Here though, the process differs. Rather than a more detailed design task, the team now examines the requirements to determine what subset is directly amenable to *composition*, rather than *construction*.

For each requirement, the team will ask:

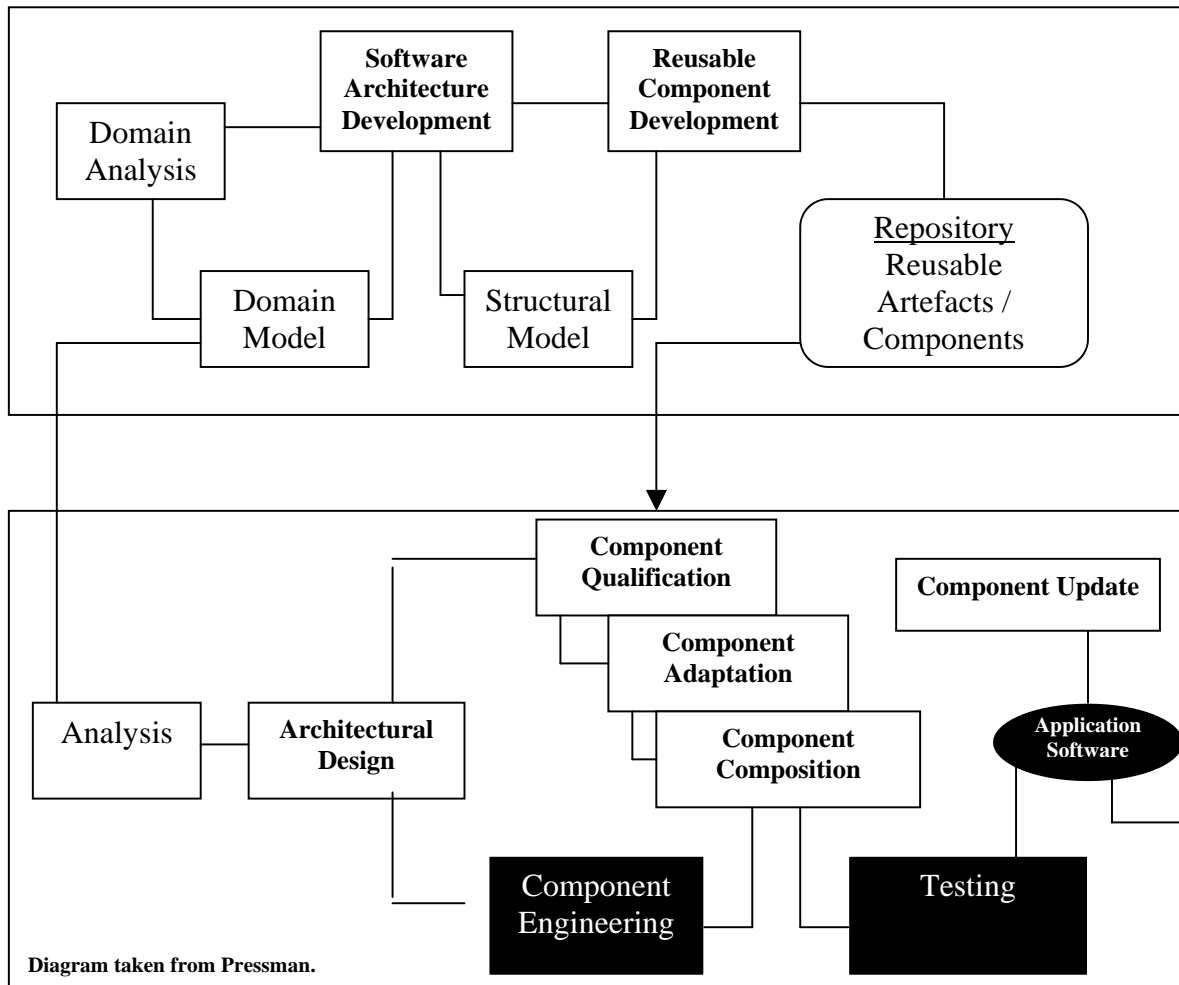
- Are commercial off-the-shelf (COTS) components available to implement the requirement?
- Are internally developed reusable components available to implement the requirement?
- Are the interfaces for available components compatible within the architecture of the system to be built?

The team will attempt to modify or remove those system requirements that cannot be implemented with COTS or in-house components. This is not always possible or practical, but reduces the overall system cost and improves the time to market of the software system. It can often be useful to prioritise the requirements, or else developers may find themselves coding components that are no longer necessary as they have been eliminated from the requirements already.

The CBSE process identifies not only candidate components but also qualifies each components' interface, adapts components to remove architectural mismatches, assembles components into selected architectural style, and updates components as requirements for the system change.

As shown in the diagram below, two processes occur in parallel during the CBSE process. These are:

- Domain Engineering
- Component Based Development



### ***Domain Engineering***

This aims to identify, construct, catalogue, and disseminate a set of software components that have applicability to existing and future software in a particular application domain. An application domain is like a *product family* – applications with similar functionality or intended functionality. The goal is to establish a mechanism by which software engineers can share these components in order to reuse them in future system.

*“Domain Engineering is about finding commonalities among systems to identify components that can be applied to many systems, and to identify program families that are positioned to take fullest advantage of those components”.*

**Paul Clements**

Examples of application domains are:

- Air traffic control systems
- Defence systems
- Financial market systems

Domain engineering begins by identifying the domain to be analysed. This is achieved by examining existing applications and by consulting experts of the type of application you are aiming to develop. A *domain model* is then realised by identifying operations and relationships that recur across the domain and therefore being candidates for reuse. This model guides the software engineer to identify and categorise components, which will be subsequently implemented.

One particular approach to domain engineering is *Structural Modelling*. This is a pattern-based approach that works under the assumption that every application domain has repeating patterns. These patterns may be in function, data, or behaviour that have reuse potential. This is similar to the pattern-based approach in OOP, where a particular style of coding is reapplied in different contexts. An example of *Structural Modelling* is in Aircraft avionics: The systems differ greatly in specifics, but all modern software in this domain has the same structural model.

### ***Component Based Development***

There are three stages in this process. These are Qualification, Adaptation and Composition (all in the context of components).

Component Qualification examines reusable components. These are identified by characteristics in their interfaces, i.e. the services provided, and the means by which consumers access these services. This does not always provide the whole picture of whether a component will fit the requirements and the architectural style. This is a process of discovery by the Software Engineer. This ensures a candidate component will perform the function required, and whether it is compatible or adaptable to the architectural style of the system. The three important characteristics looked at are *performance, reliability and usability*.

Component Adaptation is required because very rarely will components integrate immediately with the system. Depending on the component type (eg. COTS or in-house) different strategies are used for adaptation (also known as *wrapping*). The most common approaches are:

*White box wrapping* – here, the implementation of the component is directly modified in order to resolve any incompatibilities. This is, obviously, only possible if the source code is available for a component, which is extremely unlikely in the case of COTS.

*Grey box wrapping* – This relies on the component library providing a component extension language or API that enables conflicts to be removed or masked.

*Black box wrapping* – This is the most common case, where access to source code is not available, and the only way the component can be adapted is by pre / post – processing at the interface level.

It is the job of the software engineer to determine whether the effort required to wrap a component adequately is justified, or whether it would be “cheaper” (from a software engineering perspective) to engineer a custom component which removes these conflicts. Also, once a component has been adapted it is necessary to check for compatibility for integration and to test for any unexpected behaviour which has emerged due to the modifications made.

Component Composition integrated the components (whether they are qualified, adapted or engineered) into a working system. This accomplished by way of an infrastructure which is established to bind the components into an operational system. This infrastructure is usually a library of specialised components itself. It provides a model for the coordination of components and specific services that enable components to coordinate with one another and perform common tasks.

There are many mechanisms for creating an effective infrastructure, in order to achieve component composition, but in particular there is the *Data Exchange Model*, which allows users and applications to interact and transfer data (an example of which is *drag-and-drop* and *cut-and-paste*). These types of mechanisms should be defined for all reusable components. Also important is the *Underlying object model*. This allows the interoperation of components developed in different programming languages that reside on different platforms. Common methods of achieving this is by the use of technologies such as *Corba* which allows components to communicate remotely and transparently over a network, *Java Beans* and *Microsoft COM* which allows components from different vendors to work together within Windows. Sometimes it is necessary to bridge these technologies (eg. COM/Corba) to facilitate integration of software developed by different vendors, or for integration with legacy systems.

It must be noted that as the requirements for a system change, components may need to be updated. Usually, some sort of *change control procedure* should be in place to facilitate configuration management, and to ensure updates to the components follow the architectural style of the system being developed.

In summary,

- if reusable components are available for potential integration, they must be qualified and adapted.
- if new components are required, they must be engineered.

The existence of reusable components does not guarantee their integration easily and / or effectively.

### ***Use of CBSE in practice***

A Company working in CBSE basis enjoys safer code – the code will be refined as its reused. So long as it is retested and reapplied in an appropriate context.

Problems arise when many components are modified for a particular project or new components are created, as it then needs to be re-established whether the component is suitable for its intended purpose, and whether it is fulfilling the task it is in place to achieve.

An organisation that is developing new components needs to develop them with reusability in mind. This requires more careful design. If not considered carefully, this can lead to the loss of huge amounts of unanticipated development times.

The following paragraph describes the trend which organisations seem to be adopting:

"The use of commercial off-the-shelf (COTS) products as elements of larger systems is becoming increasingly commonplace. Shrinking budgets, accelerating rates of COTS enhancement, and expanding system requirements are all driving this process. The shift from custom development to COTS-based systems is occurring in both new development and maintenance activities. If done properly, this shift can help establish a sustainable modernization practice."

#### **SEI COTS-Based Systems (CBS) Initiative**

Filing components is also a consideration. In some domains there are too many components – some of which are irrelevant and undocumented. For a developer searching these libraries, it is difficult to find what you want. New components should be backed up with appropriate use and context information when submitted to the reuse library (this is part of domain engineering – explain ways in which this is done).

#### ***Classifying and retrieving components***

Although this topic will not be covered in any depth here, it is necessary to consider. Consider a large component repository, with tens of thousands of reusable components. A software engineer will need some way of searching for the desired component. To do this, the components will need to be described in unambiguous, classifiable terms.

One way of doing this is described by Tracz, called the 3C Model – concept, content, and context.

Whittle describes the *concept* of a software component as "*a description of what the component does*". Here, the interface of the component is fully described and the semantics are identified. The concept communicates the intent of the component. The *content* of a component describes how the concept is realised. It is information hidden from the casual user and important only to those who intend to modify the component.

The *context* places a reusable software component within its domain of applicability. This enables a software engineer to find the appropriate component to meet application requirements.

The above and further information on translating the 3C model into a concrete specification can be found in Pressman, Chapter 27.

#### ***Summary***

CBSE introduces major changes into design and development practices, which introduces extra cost. Software engineers need to employ new processes and ways of thinking – this, too, can introduce extra cost in training and education. However, initial studies into the impact of CBSE on product quality, development quality and cost, show an overall gain, and so it seems likely that continuing these practices in the future will improve software development. It is still not clear how exactly CBSE will mature, but it is clear that it aids in the development of today's large scale systems, and will continue to aid in the development of future systems, and is the perfect platform for addressing the requirements of modern businesses.

## *References*

### **Software Engineering, A Practitioners Approach**

*Roger S. Pressman.*

### **“Mass-Produced Software Components”**

*M.D. McIlroy.*

<http://cm.bell-labs.com/cm/cs/who/doug/components.txt>

### **“From Subroutines to Subsystems: Component Based Software Development”**

*Clements, P.C.*

American Programmer, vol. 8, No. 11, November 1995.

### **“Engineering of Component Based Systems”**

*Brown, A.W. and K.C. Wallnau.*

Component-Based Software Engineering, IEEE Computer Society Press, 1996, pp. 7-15.

### **“Developing a Handbook for Component-Based Software Engineering”**

Program: 1999 International Workshop on Component-Based Software Engineering.

<http://www.sei.cmu.edu/cbs/icse99/>

### **“The Current State of CBSE”**

*Alan W. Brown, Sterling Software.*

*Kurt C. Wallnau, Software Engineering Institute.*

IEEE Software, September/October 1998.

<http://www.computer.org/software/so1998/extras/s5037.htm>

### **“The Role of Formal Methods in Component-Based Software Engineering”**

*P.S.C. Alencar (palencar@csg.uwaterloo.ca)*

*D.D. Cowan (dcowan@csg.uwaterloo.ca)*

<http://www.sei.cmu.edu/cbs/icse99/papers/40/40.htm>

### **“Where Does Reuse Start?”**

*W. Tracz.*

Proc Realities of Reuse Workshop, Syracuse University CASE Center, January 1990.

### **“Models and Languages for Component Description and reuse”**

*B. Whittle.*

ACM Software Engineering Notes, vol. 20, no. 2, April 1995, pp. 76-89.

This document was created with Win2PDF available at <http://www.daneprairie.com>.  
The unregistered version of Win2PDF is for evaluation or non-commercial use only.