

# Component-Based Principles in Role-Based Development

Jonas Wadsten  
Mathias Bartoll

*Department of Computer Science  
Mälardalen University  
Västerås, Sweden*

jonas@wadsten.nu  
mathias@mathiasbartoll.com

**Abstract:** (200 words)

This paper's first part covers the most fundamental concepts in component-based software development. We explain different principles functionality, how they interact with each other and what the specific part can contribute with in an object-oriented system. Component-based Software Engineering (CBSE) main ideas are to shorten development time and provide better quality products. It is a new and growing technique on the market.

In the second part of the paper, we introduce the role-based technology. Role-based technology is an extension to the traditional object-oriented technique. We discuss what it has for impact on a regular object-oriented system and how it can help to improve.

## 1 Introduction

In today's growing software industry, our demands on these products rise. We want the software to be reliable, have the ability to be adaptable, include a good usability and so on. When fulfilling these demands, the products become increasingly large which results in more complexity. Together with the complexity, the development process and the maintaining phase become more critical. To be able to meet the budget constraints and finish the project in time before the deadline we need a certain technology. The solution to the problem would be a component-based approach with reusability. Component-based software engineering provides support for the development of components as reusable entities. It also facilitates the maintenance and upgrading of systems. Still there are some drawbacks as we discuss in the next chapter.

In this report we will first discuss the main principles of CBSE, which will give an overview of the technology. Then we introduce role-based technology in the CBSE and examine what it means and if it is worth the effort. Will it help us to reduce the common problem with complexity and make the maintenance and reusability of the software easier?

We start in section 2 by examining the basic principles of CBSE and continuing with a more detail view of the software component. In section 3 we introduce role-based component engineering and describe how it can improve today's object-oriented systems. Finally, in section 4 we analyze in which manner the CBSE and role-based techniques differs from each other.

## 2 Basic Concepts in CBSE

In many areas of business, it is common to reuse components. It has been like this for many years, as motorcycle manufacturers reuse engines and other parts in new models. In the system engineering area, reusing is a new way of gaining time and effort while developing new systems

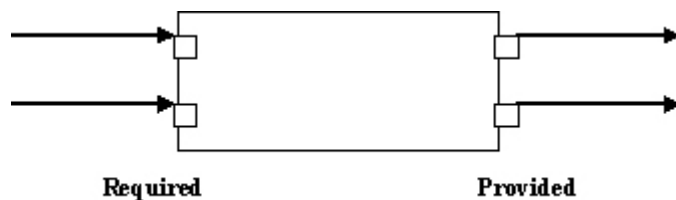
and it called Component-based software engineering. As a successful example from CBSE programming, Coulange [1] describes a project that took 15 289 hours to develop when reusing code. If they had done the programming from scratch, they would have needed 40 352 hours of developing time. With new code, they would have needed to find and correct 459 errors but with CBSE they found and corrected 184 errors.

In this chapter, we will go through the component and its parts to get an overview of the construction and functionality. We will finish with some problems and benefits of CBSE.

## 2.1 Components

Component-based technologies main parts are the components with a definition similar to objects but with the significant difference that components in CBSE should have the ability to be deployed independently. It should also be able to communicate through an interface.

The interface is constructed in such a way that it will connect to other parts regardless of how the implementation is made. The interface specifies the required in- and out-parameters as shown in figure 2.1.

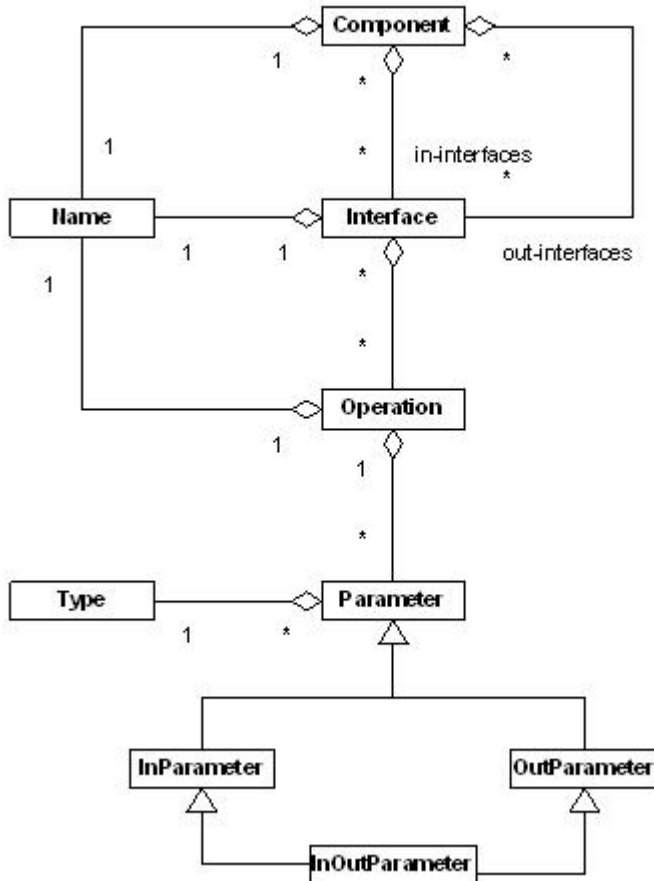


**Figure 2.1** An abstract model of a component and its interface.

To be able to describe a component you can use the list Crnkovic and Larsson describes [2] below:

- A set of interfaces for interaction with other components in the environment
- Code to be coupled to other components via the interface, which is executable.
- Provided and required specification of nonfunctional characteristic.
- Confirmation through a validation code for proposed connections.
- Documents for the documentation, specification requirement, design information and use cases.

In figure 2.2 a Unified Modeling Language (UML) class diagram describes the relationships between components and interfaces. Note that the interface can be implemented by several other components and an operation can be a part of several different interfaces. The implemented functionality in the component is hidden from the user. All method calls to the component goes through the specified interface, the description of which consists of in- and out-parameters.



**Figure 2.2** Relationships between components and interfaces.

To illustrate this example into code we show below a COM component generated in MS Visual Studio. The component has one interface with two operations and both these operations have in- and out-parameters. The interface inherits the IDispatch interface, which gives ActiveX components access, such as Visual Basic. HRESULT is a return value to indicate success or failure. The class includes the interface and consists of the implementation.

```

interface IShop : IDispatch
{
    HRESULT getNameByIndex([in] long index, [out] BSTR* pVal);
    HRESULT GetCostByName([in] BSTR name, [out] long *cost);
};

class CShop :
{
public:
CShop() { }
BEGIN_COM_MAP(CShop)
    COM_INTERFACE_ENTRY( IShop)
    COM_INTERFACE_ENTRY( IDispatch)
END_COM_MAP()
public:
    STDMETHOD(getNameByIndex)(/*[in]*/ long index, /*[out]*/ BSTR*
pVal);
    STDMETHOD(GetCostByName)(/*[in]*/ BSTR name, /*[out]*/ long
*cost);
};
  
```

## 2.2 Interfaces

One of the main features in CBSE is the separation of the interface from the implementation.

The interface is divided into two different types of interfaces, one is called exported interface, which describes the components services to the environment. The other one is called imported interface and it describes the services that the component requires from the environment.

## 2.3 Contracts

A contract in CBSE describes the specifications of the behavior for a single component. Furthermore, a contract could also specify interaction with groups of components in terms of the participating components, each components role through its contractual obligations, the maintained invariant for the component and the methods to initiate the contract.

## 2.4 Patterns

To provide an understanding of the component the patterns describe the relationships between mechanism and the deeper system structures. Another part of the patterns could be the description of the behaviors low-level implementation details and the components structure.

Crnkovic [2] illustrate patterns through three major categories: *architectural patterns*, *design patterns* and *idioms*. These three levels of patterns deal with different levels of abstraction for the documentation of the software solution but there could be problems using patterns. One issue is the unstructured knowledge containing many ambiguities in the design pattern. Developers could use design patterns to describe the behavior of the components inner parts or the composition of a component when designing a framework of several components.

## 2.5 Frameworks

A big part of the CBSE technique is to put components together in a framework and frameworks define the relations between the participant components that could be reused in a similar situation. Compared to patterns, Szyperski [3] concludes that they describe a larger unit of design, which is more specialized.

## 2.6 Problems with CBSE

Crnkovic [4] describe problems or risks with CBSE. One of them is the time and effort to build the systems of components where the effort to build an component is three to five times bigger than an normal unit for a particular purpose and the cost for the component are recovered after the fifth reuse of the component.

The main idea with CBSE is the reuse of components even in applications yet to be known. It is difficult to develop and create components for the future with unknown or no predicted requirements. At the same time, there could be a conflict between usability and reusability of the component. Components need to be sufficiently general, scalable and adaptable leads to components that are more expensive and more complicated to use.

The final disadvantages for the CBSE technique is that the maintenance cost decreases for the application but can increase for the component, depending on the complexity of the structure of the reliability requirements. The separate lifecycles together with the different requirements for both the component and application, point out that there could be problem with the requirements to the application. The component could hide important attributes for the developer of the applications.

## 2.7 Benefits with CBSE

A good start for succeeding with CBSE is to follow the two basic requirements for reusability [2]:

1. Reuse requires some modification of the object being used.
2. Reuse must be integrated into specific software development.

By following these requirements it is possible to benefit a shorter development time and better quality products. To be more specific you can even get reduced time to market, improved quality, a wider range of usability etc. To gain these benefits you need to establish methodologies and processes in practice and theoretically refine them so they are recognized by both industry and academia.

## 3 Role-Based Component Engineering

The role-based component engineering is an extension to the traditional object-oriented technique. The main idea is to divide the interface into smaller parts. Each of these parts represents a different role. When calling a component the user communicate with the smaller role interface instead of the full interface. By sorting out into smaller parts, it is possible to only involve the relevant part of the interface.

In this section, we will exemplify how roles can be used in different perspectives. To illustrate how roles affect the system, the six metrics is a way of measuring the complexity, to see whether the roles reduce or increase the complexity. Some techniques that make it possible to use roles in both design and implementation are listed and finally there is a part of roles in object-oriented frameworks.

### 3.1 Kemerer and Chidamber's Six Metrics

Chidamber and Kemerer created six metrics for object-oriented systems based on *cohesiveness* and *coupling* [6], which together make it possible to perform measurement on OO systems. Cohesiveness relates to the method similarity. A class with high cohesiveness has methods that operate mainly on the same properties in the class. Coupling is used if a method of one object is using instance variables or methods of another object.

1. *Weighted methods per class (WMC)*. This metric takes into consideration the number of methods and properties of a class. The metric value rise depending on the number of methods. A complex class with high WMC needs more time and effort to be developed and maintained. With many methods in a class, the influence is larger on the children, since children will inherit all the methods defined in the class. It also limits the possibility of reuse.
2. *Depth of inheritance tree (DIT)*. The depth of inheritance of the class is the DIT metric for the class. The deeper a class is in the hierarchy, the greater the number of methods is likely to be inherited classes and making it more complex to predict their behavior.
3. *Number of children (NOC)*. It measures how many sub-classes that are going to inherit the methods from the parent class. If the NOC number is high, the reuse is high, since inheritance is a form of reuse.
4. *Coupling between object classes (CBO)*. This metric counts the number of other classes, which the referring class is coupled to, i.e. one object acts on the other. It is useful to

determine how complex the testing of various parts of a design will be. A higher value demands a more rigorous testing.

5. *Response for a class (RFC)*. This metric measures how many methods that potentially can be executed in response to a message. A higher value represents a more complex class because the testing and debugging becomes more complicated.
6. *Lack of cohesiveness in methods (LCOM)*. The LCOM value describes the similarity of methods in a class. A class with high cohesiveness promotes encapsulations and if the class is noncohesive it should probably be separated into two classes.

With these metrics we can measure how roles affect ordinary OO systems. In general these metrics will improve when roles are used [2].

Roles will increase the DIT value since inheritance is the mechanism for providing roles on a component. The NOC value will rise because role interfaces are usually located at the top of the hierarchy. Despite the increased complexity the high values on these metrics gives an advantage to express a design on a higher and more abstract level.

Roles will reduce the complexity in the metrics: CBO, RFC, WMC and LCOM. The component relations are moved to a more abstract level and will simplify the implementation part.

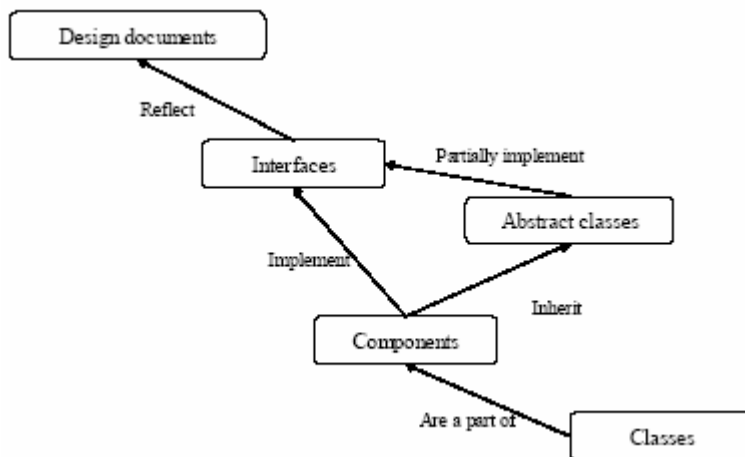
### **3.2 Role Technology**

Roles can be taken under considerations during the design phase and the implementation part. In the design level, both Catalysis and OORam uses UML metamodel to role-enable UML. When implementing, both Java and C++ have their benefits and weaknesses. Java has good support for interface extension and multiple inheritances of interfaces, which makes it very suitable for role-based component engineering. C++ solves the lack of interface support by constructing abstract classes with virtual methods. Together with the multiple inheritance of classes support, these abstract classes can be combined as in Java.

### **3.3 Frameworks and Roles**

The use of a framework avoids the repeated implementation of the same behavior and it hides a lot of the complexity of the interactions between the objects. An example of this is the phrase “don’t call us, we’ll call you” that is often used in frameworks.

A framework consists of components that only have to make sure that they can fulfill their role in the framework. The relations between different elements are shown in figure 4.1 and illustrated by labeled arrows [2]. The interface together with the abstract class is usually called a *white-box framework*. In order to use such a framework, the developer needs to extend classes and implement the interfaces, provided by the abstract class [7]. The *black-box framework* that consists of components and collaborating classes can, on the other hand, be instantiated and configured by developers.



**Figure 3.3** Framework elements.

As revealed in [5], when using multiple frameworks in an application there can be a number of problems such as; Frameworks are often assumed to be in control of the application and when two frameworks of that kind are composed, it can be difficult to synchronize their functionality. There can also be an overlap of framework functionality or that they do not cover the full application domain. These problems can be avoided to some extent by following a guiding principle.

Instead of using multiple frameworks, developers should state a set of roles based on the component collaborations identified in the design phase. The roles can then be used in implementation as abstract classes, components and as regular classes. Role interfaces should be highly cohesive and ought to be used instead of custom interfaces. Because role interfaces do not provide implementation the synchronize problem is avoided. The fully not cover problem can be solved by, whenever possible, reusing existing role interface in the white-box framework. Finally, the overlap problem can be resolved by creating wrapper components, which implement roles from both interfaces.

A framework problem that we do not cover in this report is how to minimize the coupling between components. Garp and Bosch discuss solutions about that in [2].

## 4 CBSE Principles in Role-Based Development

When designing and implementing a role-based system it differs on some principles from the component-based development. The most important effect of introducing roles into a system is that relations between components are no longer expressed in terms of classes instead they are called roles. The consequence of that is best measured with the six metrics explained in section 3.1. Furthermore, the standard OO interface is divided into smaller role interfaces, where each part represents a different role. By doing that, the component relations are moved up to a more abstract level.

Introducing roles in frameworks can improve structure and interoperability. Some of the common integration framework problems can also be solved as revealed in section 3.3.

## 5 Summary

In this paper, we have briefly discussed the basic concepts in CBSE and the function of role-based component engineering. To be able to develop component-based applications we have to understand how to use the basic principles. These are the main concepts and should be used in a

proper manner. Using CBSE in a project is a powerful tool, yet it is still a new technology and has some drawbacks as well. We stated both the benefits and the current drawbacks. The discussion then lead to what we would gain by introducing role-based technology in our object-oriented system. The results showed that by dividing the interface into certain roles, the complexity in larger software is reduced. In frameworks we could eliminate problems like synchronization, overlap of functionality or the fully not cover problem. Using role-based technology has shown that it can improve an object-oriented system.

## 6 References

- [1] Coulange B., *Software Reuse*. London: Springer, 1998
- [2] Crnkovic, I., and Larsson, M., *Building Reliable Component-Based Software Systems*, Norwood: Artech House, Inc, 2002
- [3] Szyperski C., *Component Software – Beyond Object-Oriented Programming*, New York: ACM Press, 2002
- [4] Crnkovic I., *Component-Based Software Engineering – New Challenges in Software Development*, Software Focus Winter 2001, volume 2 issue 4 page 127-133, John Wiley & Sons Ltd., 2002
- [5] Bosch, J., et al., “*Object Oriented Frameworks-Problems & Experience*”, in *Object-Oriented Application Frameworks*, Fayad, M. E., Schmidt, D. C., and Johnson, R. E., Eds., New York: John Wiley & Sons, 1999.
- [6] Chidamber, R. S., and Kemerer F. C., “*A Metrics Suite for Object Oriented Design*”, M.I.T. Sloan School of Management, Cambridge, 1993.
- [7] Roberts, D., and Johnson, R., *Patterns for Evolving Frameworks*, Reading, MA: Addison-Wesley, 1998.