Compositional Structured Component Model:

Handling Selective Functional Composition

Hamdan Msheik, Alain Abran, Eric Lefebvre
Electrical Engineering Department, École de Technologie Supérieure,
1100 Notre-dame Ouest, Montréal, Québec
Canada H3C 1K3
hamdan.msheik.1@ens.etsmtl.ca, aabran@ele.etsmtl.ca, elefbvre@ele.etsmtl.ca

Abstract

Software component technology has been promoted as an innovative means to tackle the issues of software reuse, software quality and, software development complexity. Several component models (CORBA, .Net, JavaBeans) have been introduced, yet certain issues and limitations inherent to components still need to be addressed. As software components with hosts of functionalities tend to be coarse to large-grained in size and since the set of functionalities required by an application varies according to the particular application context, an excessive number of unwanted functionalities might be generated by such components within the application. In this paper, we present the Compositional Structured Component Model (CSCM) designed to handle the issue of unwanted component functionalities and to provide a flexible approach for easier customization, adaptation, and reuse. The CSCM model is designed to handle this issue via component functional composition using metadata composition instances which allow selective composition of a component's required functionalities.

1. Introduction

Software components represent a major step in the evolution of computing technology. Peter Maurer describes the process of software construction using components as another computing revolution on a par with those of stored programs and programming languages [1]. However, the idea behind software components is not new: it first appeared in a NATO conference on software engineering in the late 1960's [2].

Software components have been defined in many different ways [1, 3-6]. From these definitions we have selected the following characteristics which are common to all software components: a) they have interfaces and interface implementations used in interconnecting with other components; b) their behaviour is almost independent; and c) their from is binary so that it can be treated as black boxes. Heineman's definitinon [6] goes a little further, in that a software component is required to comply with a component model which defines the component's interactions and composition standards.

The technology of software components is aimed among other at addressing the issues of better software reuse, better quality and easier development in the context of growing numbers of functionalities, large-scale software complexity and requirements mutability. Inappropriate handling of these issues often leads to software applications which are monolithic [7, 8] less flexible, more complex, difficult to reuse and costly to develop and maintain.

In an effort to reduce the complexity of software applications, increase the potential for software reuse and enhance software distribution and interoperability, several component models (CORBA, .Net, EJBs, JavaBeans) have emerged. While these component models represent significant technological improvements, they still have several limitations. For instance, these models were originally introduced to satisfy particular aspects, such as applications interoperability, object distribution [9], and rapid GUI (Graphical User Interface) construction.

The unwanted functionalities exhibited by software components in particular application contexts are caused by the tendency of the size of software components to be coarse to large-grained. In such components, the set of useful and required functionalities provided by a particular component varies according to the particular software application context. Typically, during the development of software application families, considerable effort is expended on the wrapping, adaptation, and customization of the

functionalities of components shared by the various constituent applications.

In this paper, we present our Compositional Structured Component Model (CSCM) designed to:

- Handle the issue of unwanted component functionalities; and
- Provide an easier and more flexible software customization, adaptation, and reuse approach for application development.

The idea behind this model is to develop components with physically disjoint functional fragments called compositional interfaces. At composition time, the application developer selects the functionality fragments needed to form the basis for a new software component.

We start Section 2 by presenting background information on component-based software engineering and component-based software construction. In Section 3, we present the CSCM model. We present the CSCM component-based software construction process in Section 4. In Section 5, we discuss a number of CSCM Java inheritance issues. This is followed by Section 6 in which we present related work. Section 7 briefly presents current and future work. Finally, Section 8 terminates with a summary and a discussion.

2. Background

2.1 Component-based software engineering (CBSE)

Software engineering processes have evolved through several programming paradigms: from the structured paradigm of the late 1960's and the 1970's, moving to the object-oriented paradigm of the early 1980's to the more recent component-oriented paradigm of the first half of the 1990's.

CBSE is divided into two distinct processes [10, 11]: component engineering and application engineering. The first deals with the analysis and development of domain-generic and domain-specific components, while the latter deals with software application development by assembly, composition, integration, and plugging of components such as COTS (commercial off-the-shelf) and other in-house developed components.

The Software Engineering Institute at Carnegie Mellon University [12] uses the term CBSD to refer to the process of software development by the assembly and integration of software components. Essentially, the terms CBSE and CBSD both more or less refer to the same process. The focus of CBSE is the development of software by assembling and integrating COTS and other existing types of components with an emphasis on composition rather than on programming [13]. It assumes that certain software parts are common to several software applications; therefore, it would be advantageous to reuse them for several reasons [12]:

- Better quality and diversity of COTS.
- Pressure to reduce development and maintenance costs.
- Use of standards, open systems, and the emergence of integration mediators such as CORBA ORBs (Object Request Brokers).
- Increase in enterprise inventory of potentially reusable software components.

2.2. Component-based software construction

Software construction can be considered as a sub-process which matches the implementation phase in the software development life cycle. Software construction per se is a software engineering act which encompasses the activities of software coding, validation and unit testing. According to the SWEBOK (Software Engineering Body of Knowledge) Guide [14], this sub-process must be instantiated taking into account four general principles (*reduction of complexity, anticipation of diversity, structuring for validation and the use of external standards*) as well as the tools used by this subprocess such as compilers, code generators, and development tools.

As software design breaks software down into smaller parts for construction, those parts are expected to comply with the general principles of software construction. Interestingly, component-based software construction meets these four general principles. For instance, components can reduce the complexity of an application since they offer modular reusable parts which can be bought from specialized suppliers instead of being developed in-house. In addition, components are reusable and replaceable parts, thus they meet the anticipation of diversity principle. Furthermore, by breaking down a software application into modular components, the validation of these components will be easier. Finally, software components generally conform to a component model, and therefore they make use of standards.

The SWEBOK Guide [14] identifies three styles of software construction: linguistic, visual and formal. These styles are general and are applicable to almost any software development process. The process of software construction by component assembly and composition may use any style or a combination of these styles.

One of the goals of software engineering is to transfer the construction process to higher levels of automation [14] in order to reduce software complexity and achieve better reuse. Coincident with this is the

goal of using component assembly approaches and compositional languages for software construction. The type of construction languages used in software construction by assembly and composition of components are configuration, compositional, scripting and general-purpose programming languages. The choice of construction language is dictated by various factors which can be related to the granularity of the software components used in the construction process as well as to other aspects such as the simplicity and flexibility, and the expressive power of the construction language itself.

3. Compositional Structured Component Model

The Compositional Structured Component Model (CSCM) (see Figure 1) is designed to construct software components through selective functional composition based on component metadata composition descriptor instances. This selective composition property provides flexibility for the adaptation and customization of components, as well as for facilitating software maintenance and helping to more readily achieve software reuse.

The CSCM model can be seen as an extension to the object oriented model which provides compositional capabilities. Consequently, a CSCM component instance is an object with an enhanced capability allowing selective functional composition of disjoint compositional parts.

A compositional part is a method implemented independently outside of the component implementation. We call such a part a compositional interface for being independently implemented and physically disjoint from the component's implementation. CSCM component's compositional interfaces are methods and not types as they are Java interfaces.

The proposed CSCM model is generic and can be implemented in various programming languages. From an object oriented perspective, CSCM components instances can be considered as objects since they possess and exhibit similar properties and characteristics to those of objects. CSCM components support inheritance; however, they are provided with a powerful composition and retrogression mechanism which allows CSCM component to either include or exclude compositional interfaces according to the information provided by the component metadata instance.

CSCM components mechanism of composition and retrogression is based on:

- An extension to the syntax (see Table 1) of an object oriented programming language to support compositional members.
- The use of the composition principle to selectively include the required functionalities suitable for a software application.
- The use of the delegation principle which permits the dispatching of the component methods' calls to the compositional interfaces of the component.

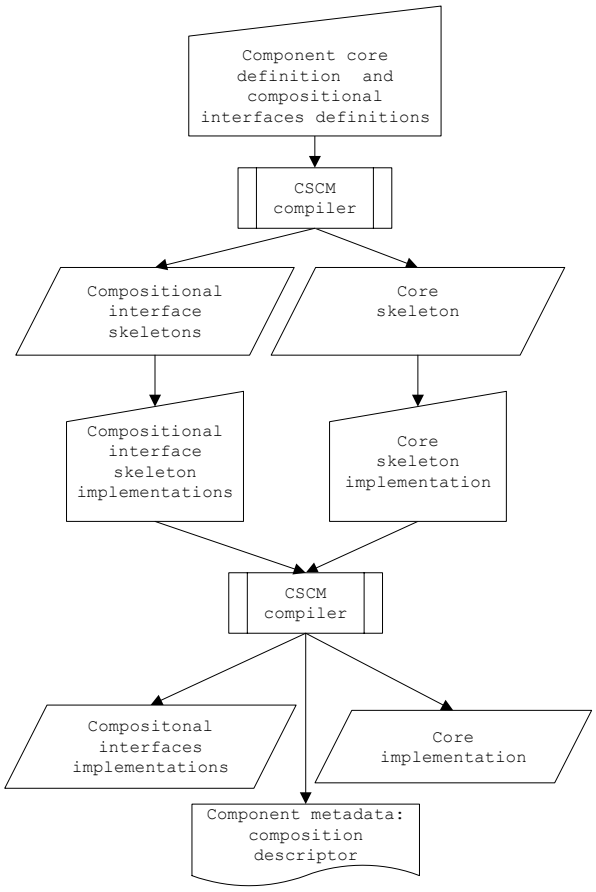

Figure 1. CSCM component construction process

A CSCM component compositional interface differs from CSCM ordinary methods in that they are selectable via the component metadata composition descriptor instance. In other words, the same component in two different applications might have different subsets of compositional interfaces, depending on the functional requirements needed by the hosting application.

The CSCM model does not handle the aspects of distribution, synchronization, and interoperability. Therefore, CSCM components rely for these aspects on the underlying mechanisms provided either by other component models or by host programming languages

in which CSCM components are implemented. Indeed, the current scope of our research is focused on providing a solution for the issue of components having an excess of unwanted functionalities and on finding an easier approach to components composition, customization, adaptation, and reuse.

3.1. CSCM component structure

A CSCM component is a software part possessing compositional interfaces, and a composition descriptor which captures metadata information specifying the various aspects, characteristics, dependencies and properties necessary for functional composition. The structure of a CSCM component (see Figure 2) is composed of three logical parts: a definition part, a metadata part and an implementation part.

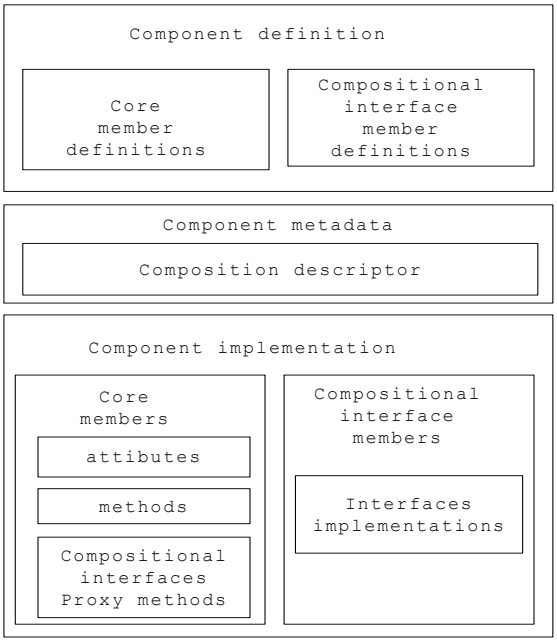

Figure 2. CSCM component structure

3.1.1 CSCM component definition. The component definition part includes the definition of two distinct categories of members: core members and compositional interface members.

Core members are composed of method and attribute members acting as the component core for any CSCM component instance. Compositional interface members are selectively available for composition through CSCM component instances. CSCM components instances behave almost like objects. Without their compositional members (interfaces) CSCM component instances are indistinguishable from ordinary objects.

The definition of a CSCM component has to be done using the host programming language in addition to the syntax constructs of Table 1. The definition of core members and the definitions of compositional interface members are necessary to generate the component core skeleton as well as the compositional interface skeletons.

As illustrated in Table 2 a CSCM component definition has to be defined in a file similar to the way object oriented source classes are defined. This definition defines the component `"Compressor"` with one core method and two compositional interfaces.

Table 1. Reserved words of CSCM components

Reserved words	Role
`component`	Declares the beginning of a CSCM component
`compositional`	Declares a compositional interface member

Table 2. Component definition

```
import java.io.File;

Component Compressor {
   public String displayUsage(){…}
   // compositional members
     // zip algorithm
   Compositional public File zip(File f,
                          String oper){}
     // gzip algorithm
   compositional public File gzip(File f,
                          String oper){}
}
```

3.1.2 CSCM component metadata. The component metadata part contains the composition descriptor which captures in XML format the descriptive metadata information of the component. In particular, the composition descriptor provides all the necessary information on a component's members and their dependencies so that at composition-time the selection of an interface will also result in load-time selection of all compositional members on which the interface depends. Furthermore, this metadata information can also be used to specify useful information on other aspects of CSCM components such as, constraints, licensing, cataloging and indexation. A partial sample of a composition descriptor for the component shown in Table 2 is illustrated in Table 3.

3.1.3 CSCM component implementation. The CSCM component implementation part contains one core implementation class for the component core members

and a different class for each compositional interface. Though logically related, a component's compositional interface implementations are physically disjoint, thereby providing compositional capabilities, flexibility, and an easier software development, maintenance, and reuse method.

The core class is connected to the classes of compositional interfaces via a composition relationship as illustrated in Figure 3.

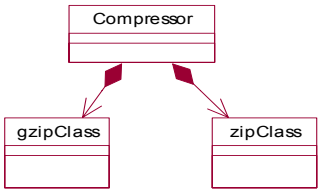

Figure 3. Relationship between the component core class and the component compositional interfaces

Besides the implementation of core members, the component core class also provides a proxy delegate method for each compositional interface.

The implementation of the CSCM composition mechanism is based on the composition and delegation principles. Furthermore, the implementation of compositional interfaces as separate classes helps also realizing the implementation of this mechanism.

Typically, object methods can access one another. This is also true for CSCM compositional interfaces; they can access each others through the delegation mechanism which uses the core instance to dispatch access requests to the concerned interface members by calling their proxy methods.

4. CSCM component-based software construction process

The CSCM component-based software construction process is divided into two processes: the component construction process and the application construction process.

4.1. Component construction process phases

The construction process of CSCM component passes through four phases as illustrated in Figure 4:

- Component core definition and compositional interface definitions.
- Compilation and generation of core skeleton and compositional interface skeletons.
- Core implementation and compositional interface implementations.
- Compilation and metadata composition descriptor generation.

Table 3. A partial illustration of the composition descriptor elements for the component defined in Table 2

```
<component name”Compressor”>
  <import-list>
    <package>java.io.File<package>
  </import-list>
  <core-members>
    <methods>
      <method name=”displayUsage”
                return-type=”String”/>
        <modifiers-list scope=”public”/>
        <parameters/>
        <dependency-list/>
          <methods>…</methods>
          <attributes>…<attributes>
        </dependency-list>
      </method>
    </methods>
   </core-members>
   <compositional-members>
    <attributes/>
    <interfaces>
       <interface name="zip" selected=
           "true" return-type="File"/>
         <modifiers-list scope="public"/>
         <parameters>
          <parameter Type="File" name="f">
          <parameter Type="String"
                          name="oper">
         </parameters>
         <dependency-list/>
           <methods>
             <method name="displayUsage"
                    return-type="String"/>
             <modifiers-list
                        scope="public"/>
             <parameters/>
             </dependency-list>
         </dependency-list>
       </interface>
       <interface name="gzip" selected=
           "true" return-type="File"/>
         <modifiers-list scope="public"/>
         <parameters>
           <parameter Type="File"
                              name="f">
           <parameter Type="String"
                              name="oper">
         <parameters/>
         <dependency-list/>
           <methods>
             <method name="displayUsage"
                    return-type="String"/>
             <modifiers-list
                    scope="public"/>
             <parameters/>
             </dependency-list>
           <methods>
         </dependency-list>
       </interface>
     </interfaces>
  </compositional-members>
</component>
```

During the first phase, the component core members and compositional interface members must be defined by the software constructor in a similar way to that in which object-oriented classes are defined. The output of this phase is the component definition source code as illustrated in Table 2.

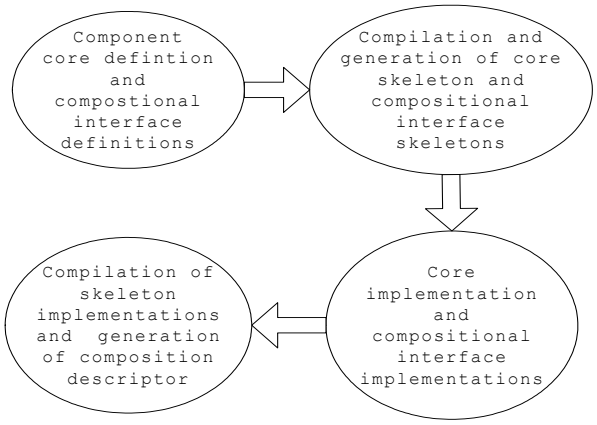

Figure 4. CSCM component construction process phases

During the second phase, the component definition source code is passed to the CSCM compiler for compilation. The outputs of this phase, as illustrated in Figure 1, are the core skeleton as well as the skeletons for each compositional interface.

The implementation of the component skeletons has to be done by the software constructor in the third phase. The output of this phase (see Figure 1) is the component implementation source code which includes the implementation of the core skeleton as well as the implementation of each compositional interface skeleton.

During the fourth phase, the component compositional interface skeleton implementations and core skeleton implementation are passed again to the CSCM compiler (see Figure 1) to generate the composition descriptor as well as the source and binary code of the components implementation. The output of this phase is a CSCM component ready for composition, inheritance and instantiation.

Illustrations of the source code implementation for the core skeleton and compositional skeletons for the component shown in Table 2 are presented respectively in Table 4 and Table 5.

4.2. Software application construction process

The process of software application construction using CSCM components requires the completion of two tasks: first, the software constructor has to select the component's compositional interfaces needed (to satisfy the application requirements) via the components' composition descriptors. Second, the software constructor has to use and manipulate the components as if it were casual object oriented class.

Whenever an instance of a CSCM component is to be created with a different combination of functionalities, this instance must be provided with the appropriate composition descriptor instance. For instance, to select a particular interface implementing a required functionality, the developer has to set the value of the parameter `"selected"` of the compositional interface element to `"true"` in the component composition descriptor as shown in Table 3. The name of the file containing the composition descriptor instance must be passed at instantiation as a parameter to the component's constructor.

Table 4. Component core implementation code

```
public class Compressor {
  /* this code is generated by CSCM
     compiler */
  java.util.Hashtable
               compositionalInterfaces;

  Compressor(new File(
               "compositonDescriptor"){
     initializeCompositionals();
  }
  Public void initializeCompositionals(){
   compositionalInterfaces = new
              java.util.Hashtable();
    /* for every selected interface in the
       composition descriptor create an
       instance of the interface and store
       in the hash table
       compositionalInterfaces */
  }
   // core method
  public String displayUsage(){
     return new String();
  }
  // compositional interfaces
  Private File gzip(File f,String oper){
     return((gzipClass)
         compositionalInterfaces.get(
              "gz")).gzip(f, oper);
  }
  private File zip(File f, String oper){
     return((zipClass)
         compositionalInterfaces.get(
              "z")).zip(f, oper);
  }
}
```

5. CSCM Java inheritance issues

CSCM component are object oriented types equipped with a composition mechanism which after transformation by the CSCM compiler are mapped to ordinary Java components.

In this section, we explain how the object-oriented inheritance mechanism is handled by a Java implementation of CSCM model.

Table 5. Compositional interfaces skeletons implementation code

```
Public class zipClass {
  /* The code is generated by the CSCM
     Compiler */
   Compressor __This;
   zipClass(Compressor comp){
   __This = comp;
   }
   public File zip(File f, String oper) {
     __This.displayUsage();
     return new File("testTextFile.txt");
   }
}
Public class gzipClass{
  Compressor __This;
  gzipClass(Compressor comp){
     __This = comp;
  }
   public File gzip(File f, String oper){
     __This.displayUsage();
     return new File("testTextFile.txt");
    }
}
```

5.1 CSCM and Java class inheritance

CSCM components may inherit other CSCM components as well as casual Java classes using the same syntax rules of Java class inheritance; i.e., using the reserved word "extends". Consequently, the same rules that apply to class inheritance apply also to CSCM components inheritance with minor differences as will be shown below.

When a CSCM component inherits another base CSCM component, not only it inherits the composition descriptor of the base component, but also its compositional interfaces.

However, when a CSCM component inherits a casual Java class, the members of the inherited base class will be available to other CSCM components as core members only.

5.2 CSCM and Java interface inheritance

CSCM components may inherit Java interfaces using the Java `"implements"` keyword. Similar to the way in which Java classes implement interfaces, CSCM components have to fulfill the contract dictated by the interface they implement. Moreover, the inherited members will be available as core members only.

6. Related work

Several software construction approaches have emerged to enhance software reuse, flexibility and maintenance. For instance, *Mixin* and role oriented-programming [15] allows the reduction of redundancy in different classes by sharing their common behavior through roles. View-oriented programming [16] allows object evolving over time to attach new views depending on new requirements.

Aspect-oriented programming [17], which is the most popular among these approaches, allows for automatic cross-cutting static waving of aspects such as logging, failure handling, etc. across objects [18]. Consequently, these aspects which are separate code chunks injected across objects yield therefore less entangled code [19].

Although these approaches represent significant enhancement and extension to the object oriented paradigm, they have been designed to tackle different issues from the main issue addressed by the CSCM model.

Unlike Aspect-oriented programming, CSCM model CSCM model tackles dynamic composition of existing functions of components. In other words CSCM model does not inject code chunks, but provides a flexible mechanism for selective composition of existing functionalities.

The difference between the CSCM model and the approaches mentioned above is that the CSCM model is designed to allow for the construction of software components with a specified variable list of functionalities. The functionalities are selected in a composition descriptor at runtime. The modification of the composition descriptor is a configuration task which does not lead to modification of the source code nor even its presence. It is this property of dynamic selection of the functions of CSCM components which tackles the issue of excessive unwanted functionalities.

Similar work conducted by Al-Hatali and Walton [20] on the issue of excessive unwanted functionalities suggests the use of compositional wrappers to hide a component's unwanted functionalities, thereby remedying this behavioral side effect. However, such an approach does not completely solve the problem because even it is hidden the excessive functionalities code persists inside the component's code. Furthermore, considerable efforts must be expended to devise the wrapping code, which is not the case when CSCM component are used instead.

7. Current and future work

Currently, we are concentrating our efforts on the development of the CSCM compiler targeted to the Java programming language. Future work will explore and address distribution and interoperability through integration of the CSCM with other component model such as CORBA. Even though attribute composition is

an interesting issue, it is not currently addressed for implementation efficiency reasons and could be addressed later. Once the implementation is ready, validation with a variety of components and applications will be conducted. Furthermore, an empirical study to validate and measure the efficiency of the model will be the subject of a subsequent research work. Also, we intend to explore venues for optimization in order to reduce the computational overhead that might be incurred. Furthermore, possibility of implementing the CSCM in C++ will be considered.

8. Summary and discussion

In this paper, we have presented the CSCM model designed to allow the construction of software components with variable lists of functionalities selected according to components' composition descriptor instances at runtime. The capability offered by CSCM component to select the required functionalities tackles the issue of excessive unwanted functionalities. Furthermore, software maintenance, modification and reuse can be significantly eased and simplified.

It can be argued that this model incurs a certain computational overhead due to initialization tasks and per instance composition descriptor file loading. This observation can be made about most of component models, however.

Moreover, the computational overhead incurred when using CSCM components might be reduced by means of native language support and optimization.

We think that the power of CSCM components can be efficiently tackled in the development of software application families. Software application families are most likely to reuse coarse to large-grained software components across families of applications with different functional configuration and capabilities.

10. References

[1] P. Maurer, *Component-Level Programming*: Pearson education Inc., 2003.

[2] M. D. McIlroy, "Mass Produced Software Components," NATO Software Engineering conference, 1968.

[3] D. E. C. Microsoft Corporation, "The Component Object Model Specification." http://www.microsoft.com/com/resources/comdocs.asp, 1995.

[4] C. Szyperski, "Emerging component software technologies--a strategic comparison," *Software--Concepts & tools*, pp. 2-10, 1998.

[5] I. Jacobson, "Component-based Development with UML," 1998.

[6] T. Heineman, G. and T. Council, W., "Component-Based Software Engineering," Addison-Wesely, 2001.

[7] J. Li, "A Survey on Microsoft Component-based Programming Technologies," Concordia University, Montreal 1999.

[8] L. Vanhelsuwe, *Mastering JavaBeans*: Sybec Inc., 1997.

[9] J. Ongg, "An Architectural Comparison of Distributed Object Technologies," MIT june 1997.

[10] S. Ghosh, "Improving Current Component Development Techniques for Successful Component-Based Software Development," ICSR7 2002 Workshop on Component-based Software Development Processes, 2002.

[11] A. Rashid, "Aspect-Oriented and Component-Based Software Engineering," IEEE Proceedings-Software, 2001.

[12] SEI, "CBS Overview." [Online]: http://www.sei.cmu.edu/cbs/index.html: Software Engineering Institute, 2003.

[13] C. P. Clements, "From Subroutines to Subsystems: Component-Based Software Development," in *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute*. Los Alamitos, CA: IEEE Computer Society Press, 1996.

[14] A. Abran, J. Moore, P. Bourque, R. Dupuis, and L. Tripp, *Guide to the Software Engineering Body of Knowledge - SWEBOK*. Los Alamitos (USA): IEEE-Computer Society Press, 2001.

[15] J. Brown, T., Spence, T., Kilpatrick, P., "Mixin programming in Java with reflection and dynamic invocation," Proceedings of the Inaugural conference on the Principles and Practice of programming, and Second workshop on Intermediate representation engineering for virtual machines, Dublin, Ireland, 2002.

[16] H. Mili, J. Dargham, A. Mili, O. Cherkaoui, and R. Godin, "View programming for decentralized development of OO programs," Proceedings on Technology of Object-Oriented Languages and Systems, 1999., 1999.

[17] R. J. Walker, E. L. A. Baniassad, and G. C. Murphy, "An initial assessment of aspect-oriented programming," Proceedings of the 1999 International Conference on Software Engineering, 1999.

[18] C. Krzysztof, Eisenecker, W., U., Steyaert, P., "Beyond Objects: Generative Programming," presented at Proceedings of the Aspect-Oriented Programming Workshop At ECOOP97, Finland, 1997.

[19] S. K. Miller, "Aspect-oriented programming takes aim at software complexity," *Computer*, vol. 34, pp. 18-21, 2001.

[20] M. S. Al-Hatali and H. G. Walton, "Smart Features for Compositional Wrappers," ICSR7 2002 Workshop on Component-based Software Development Processes, Austin, Texas, 2002.