# USING DESIGN PATTERNS AS SUPER COMPONENTS IN C.O.S.E. AND APPLYING THEM IN COSEML

Okan Avkaroğulları

Department of Computer Engineering
Middle East Technical University
December 2003
okana@sbd.havelsan.com.tr

**Abstract.** It is widely believed and empirically proven that component reuse improves both the quality and productivity of software development. This brings the necessity of a graphical editor to model the projects by using components. A graphical editor was implemented for the development of Component Oriented software. The editor facilitates modeling efforts through application of the graphical modeling language COSEML. Both design patterns and software components have come to play important roles in software development. The correlation between software components and design patterns is apparent. In the design phase of the projects design patterns are used widely both in component and object oriented projects. Design patterns can be used as super components in component-based development . Software reuse, software components, design patterns, use of design patterns in component-based development, and component architectures are studied in details to address the need for the approach in this study.

## Contents

## 1. Introduction

A brief look at the history of the computer science suggests that many of the most important developments had happened by advancing the abstraction in development and introduction of a tool associated with this new paradigm.
In the early days of computer science, programmers had to write op-codes of the entire program. Later on mnemonics were introduced which abstracted the machine code into instruction names. A tool associated with the abstraction was a piece of software that converted the instructions to op-codes.
This phase was followed by introduction of another abstraction mechanism and its associated tool: the assembler. The assembler is a software tool that converts not only instructions but also labels into op-codes that a machine can understand. It usually works together with a preprocessor that handles things like defines and such.
As the complexity of software grew assemblers become inadequate. The problem is resolved by introduction of another abstraction in thinking and its associated tools: programming languages and compilers. Over time the raise in software complexity and of failure rate on software engineering projects required different approaches and as a result object oriented programming came to rescue. The tools for object-oriented techniques were new languages that supported this new paradigm.
Object-oriented programming techniques promote a new approach to software engineering in which reliable, open applications can be largely constructed, rather than programmed, by reusing software components.
Object oriented programming supported software reuse and better labor division, there has been still to much of different individuals implementing same things over and over for different projects and in some cases in the same project.
It became apparent that for better software reuse a new method was required. Software components and component based development was the apparent answer to this problem.
Although the dream of a components-based software industry is very old, only now does it appear that we are close to realizing the dream.
Software reuse improves productivity and quality. A suitable example of software reuse is reusing the software components. A software component is any standard, reusable, and previously implemented unit that has a function in a well-defined architecture. When the quality software components are used in a target system, the overall product quality and reliability of that system improves.
In component based software development software components are used to develop a bigger system, namely, component-based development, transforms the development from code writing to integration of components. To achieve this first the specification of the target system is defined. Then, according to this specification, the target system is decomposed according to the components available. Since the components are known next the adaptation and creation of resulting components are performed. After getting the needed components, these components are integrated to build the target system. There are lots of design tools available in the market. But none of them is specialized for component based software development.
Towards the end of eighties and early nineties, the concept of design patterns emerged in parallel with the work on components. Design patterns literature provided a common

language, and the works on patterns provided well thought solutions to problems that are common to most of the software projects.
As it is pointed above, so far the most powerful abstractions, paradigms are all have tools that help the developers implement them with little effort and high efficiency. As we pointed out before, tools for component based software development are not available in the market. There is no powerful tool that helps the developer use design patterns with little effort and high efficiency except for just a few design patterns (such as model-view-controller implemented in Java Swing).
The aim of our work is close this gap by implementing a visual tool that implements a Component Oriented methodology which makes it possible to hierarchically decompose a system's requirements. Modeled in Component Oriented Software Engineering Modeling Language (COSEML) developed before, such a hierarchy corresponds to the structural relations among components. The graphical editor, implemented in this research, gives only the structural views of target systems. So, it enables the user to hierarchically decompose a system's requirements, and to view structural relations among components graphically. It also introduces the design patterns to the picture by treating them as super components.

## 2. Background

New generation of software reuse is the using software components. Software components lead to component-based development style. The system developed with component-based development style may contain the software components written in different languages, compiled by different compilers, and run on different platforms. This heterogeneity problem of the systems can be solved by component architecture.

### 2.1. Software Reuse

A broad definition of software reuse is using existing software artifacts to construct a new software system. The most commonly reused software product is source code. In addition to code, any intermediate life cycle products can be reused. Reusable software development knowledge and experience exists at different abstraction levels: the architecture level, the modular design level, and the program (or code) level. Research on software architecture is currently aiming to define different software architecture styles for different families of software. A software architecture style describes the formal arrangement of architectural elements, and can be reused by software developers to construct their new software systems once the style is well defined.
Reusable knowledge on modular design can be codified in design patterns and frameworks. A design pattern is the description of a solution to recurring problems. It specifies a problem to be solved, a solution that has stood the test of time, and the context in which the solution works. Design patterns provide a common vocabulary for software developers to discuss their designs and can be passed from one developer to another developer for reuse. The concept of framework comes from object-oriented programming languages. A framework describes the interaction pattern among a set of collaborative classes or objects, and can be represented as a set of abstract classes that interact with each other in a particular way. Programmers can reuse frameworks directly in their development after providing implementations for those abstract classes. Framework reuse is a mixture of knowledge reuse and code reuse. Programming knowledge at the level of

code is represented as program plans that can also be reused by programmers if a suitable representation form is defined.

## 2.2. Software Components

There is no shortage of definitions of component in the literature. Advocates of software reuse equate components to anything that can be reused; practitioners using commercial off-the-shelf (COTS) software equate components to COTS products; software methodologists equate components with units of project and configuration management; and software architects equate components with design abstractions. When these definitions combined, the following definition can be found: A software component is the independent and replaceable part of a system that has a function in a well-defined architecture. It is any standard, reusable, and previously implemented unit that is used to enhance the programming language constructs, and to develop applications.

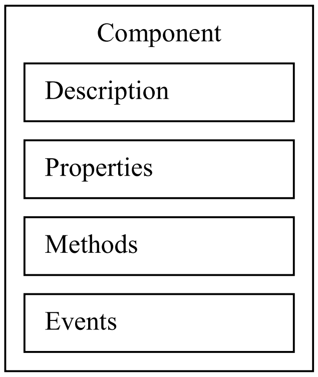

**Figure 1.** The fields of a component [1]

Any component contains four fields as in Figure 1 *Description* field gives the information about the component to semantically search for the component. *Properties* field is used to change the characteristics of the component. *Methods* field gives the services provided by the component. *Events* field is used to set the actions to be performed when an event occurs in the component or from the environment [1].

## 2.3. Component Based Software Engineering

Component-based Software Engineering (CBSE) is concerned with the development of systems from software components, the development of components, and system maintenance and improvement by means of component replacement or customization. Building systems from components and building components for different systems requires established methodologies and processes not only in relation to development / maintenance phases, but also to the entire component and system lifecycle. In addition to objectives such as component specification, composition, and component technology development that are specific to CBSE, there are a number of software engineering disciplines and processes that require methodologies be specialized for application in component-based development. Many of these methodologies are not yet established in practice, some have not yet been developed. Experiences from other areas, such as system engineering can be successfully applied on component-based development, as there are many similarities in the basic concepts. Also, with its focus on components and their specifications, CBSE can give better understanding of building systems in general, and in particular of computer-based systems whose significant part is software.

## 3. Design Patterns

A reusable design should be specific to the problem at hand but also general enough to address future problems and requirements [2]. When a good enough design is found it is reused again and again. At each use the solution comes more flexible. This gives the ability to use it next time. By time these design solutions (patterns) are used to solve specific design problems and make designs more flexible, elegant, and ultimately reusable. Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem. Patterns are about communicating problems and solutions. Patterns enable us to document a known recurring problem and its solution in a particular context, and to communicate this knowledge to others. The main goal of a design pattern is to foster conceptual reuse over time.

### 3.1. Contribution of Design Patterns in Reusability

During the last ten years design patterns, frameworks and software components have become very promising as different tools available to increase reuse in software development projects. The two most common techniques for reusing functionality in systems are class inheritance and **object composition**. Class inheritance lets to define the implementation of one class in terms of another's.
Object composition is an alternative to class inheritance. Here, new functionality is obtained by assembling or composing objects to get more complex functionality. A good pattern shows ways to solve problems and is structured in a style that lends itself well to explaining the aspects of the problem and solution at work. From this view it is clear that a design pattern is a reusable component itself. And also design patterns can be reused by using object composition. Each design pattern has responsibility for one specific problem solution. When a complex solution is needed for a problem, firstable the problem is divided into smaller and acceptable subproblems and then design patterns are used to solve these subproblems. Since design patterns have the ability to connect each other, the subproblem solutions are connected and then a solution to the main problem is found. Patterns tend to reference and built upon each other. This connection between patterns serves to create what is called pattern language: a series of interconnected patterns that, as a whole, often suggest an overall architecture for an application. Certain sets of patterns in combination will form a suitable reusable architecture that can be applied to projects again and again.

### 3.2. Correlation of Design Patterns and Component Based Development

Both design patterns and software components have come to play important roles in software development. The correlation between software components and design patterns is apparent. Components can benefit from design patterns. Design patterns are well suited to describe the power of different strategies in component-based software development. Programmers developing components can take advantage of already written design patterns, when developing new components and the design pattern community might extract new or improved design patterns from existing successful component-based

applications that can later bring benefits to other component developers. There are design patterns giving guidance on how to make loose connections between different subsystems, but also, when it comes to the inner workings of a module or a component, there are patterns helping the programmer to identify a suitable implementation.
Another interesting reuse technique is frameworks. A framework is a highly reusable design for an application, or part of an application, in a certain domain. It often defines the basic architecture of the applications that can be built by using it. Another way to view a framework is as an abstraction of a set of possible solutions to a problem. A framework is different from a design pattern, though. Whereas a framework is a concrete powerful solution that can be described in source code, a design pattern is more abstract and general. Only example usage or applications of a design pattern can be described in source code. Furthermore, a framework is often built by the application of a number of design patterns, and thus, patterns describe microarchitectures often used in frameworks.
Developers use existing frameworks by adapting them to form their particular application. So called whitebox frameworks let developers reuse and extend functionality by inheriting from base classes in the framework and provide application specific implementations. In blackbox frameworks, developers use object composition to plug in new functionality. Blackbox frameworks are generally easier to use and extend than whitebox frameworks, since developers need to have much more detailed knowledge about the internal parts in whitebox frameworks.
There is a strong relation between design patterns and components. Actually, one or more design patterns can be applied to build a component, but also, as a realization of a design pattern; one or more components can be used. Furthermore, components can be used as parts in for example a framework and a framework can even be viewed as the glue code that makes components work together. In fact, technologies like Java Beans, COM/DCOM or Corba, are different specialized frameworks making it possible to connect components. Figure 2 illustrates the relationships (using UML syntax) between patterns, frameworks and components.

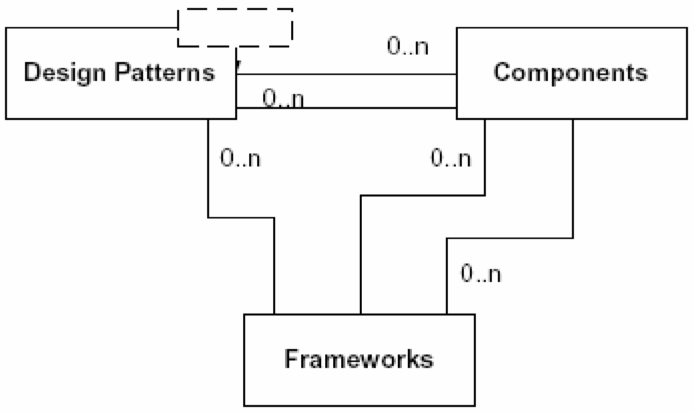

**Figure 2.** The relationships between patterns, frameworks and components [7]

The relation and hierarchy between these reusable techniques and applications which use these is shown in figure 3

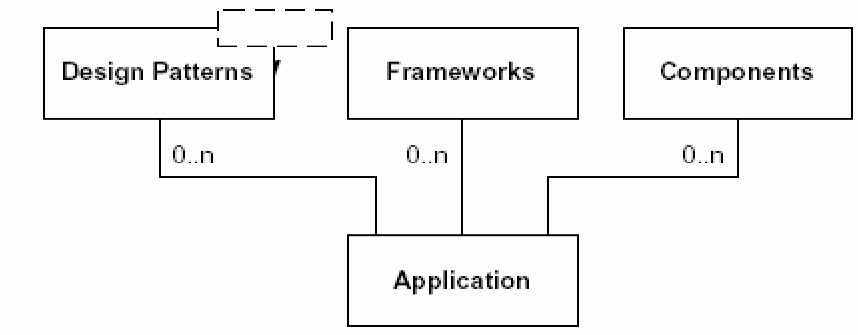

**Figure 3.** The relation and hierarchy between reusable techniques [7]

### 3.3. Design Patterns as Super-Components

As time goes on, probably more and more specialized component patterns will be presented. Such domain specific patterns could be collected to form a useful component pattern catalog. This has already happened in other fields. For example, patterns aimed at business modeling. Another kind of domain specific patterns is patterns aiming at distributed computing, which is also very interesting and essential in CBD. For example, the major component models of today support distribution and the usage of distributed applications increases.

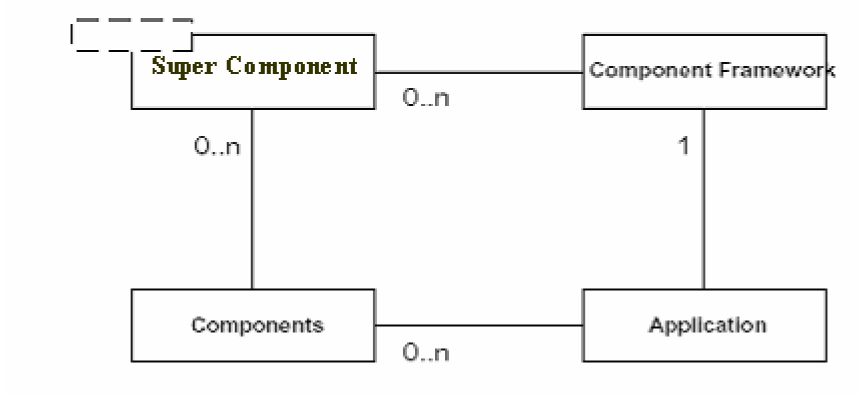

**Figure 4.** The role of super-components in CBD [7]

Domain specific patterns for CBD would work as some sort of super components, that is, components that can help to create other components that conform to a certain set of constraints. This set of constraints could be the implementation of a set of design patterns. As an example, a component framework (system that promotes the use of components i.e. COM/DCOM, CORBA or EJB) could be described as a set of collaborating design patterns. The intersection of all components in such a system would be the component framework itself and possibly solutions to some common problems, Looking at design patterns as super components might inspire the pattern community to develop pattern languages for CBD. In figure 4 an idealized view of super components role in CBD is presented. The application developer uses some component framework and integrates suitable components into the application. Super components were used to form the component framework and also to create the individual components themselves. CBD might also benefit from a formal method of describing design patterns. Research is done in the field to formally specify design patterns and mathematically describe and transform design patterns. This is not mainstream research and has many adversaries due to the fact that design patterns are described in a natural language and the ambiguity of such enhances the design patterns themselves. This is not to say that it hasn't its place. This kind of research in the design pattern field could contribute to CBD. It could stimulate the discovery and invention of new component patterns, the classification and description of them and also the automatic application of patterns.

## 4. Using Design Patterns in CBD as Super Components

To show that design patterns can be used as super components in component based development, the design pattern feature is added to the only CBD case tool COSECASE v1.0. To use Design Patterns in COSECASE a new symbol is added to the system. This symbol represents the design pattern in the design phase shown in figure 5. Some extra new added capabilities are defining and removing design patterns to the system and using them in abstract and structure level. Then a framework designed that uses some design patterns is discussed. This framework is designed in COSECASE and after the framework designed it's realized by using IBM Web sphere Application Developer 5.0.

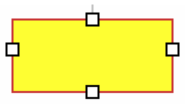

**Figure 5.** The COSEML Figure That Represents Abstract Phase of a Design Pattern

Since the framework decided to code in java, framework is designed containing Java and EJB design patterns. There is main package that holds all the client operations. Framework decided to connect LDAP, Content Manager (for data management system) , IBM MQSeries (for workflow operations) and a database. All of the requirements for these except database requirements are provided by the vendors. So these behave like external systems for our framework (They can also be defined as components). To make a servlet for each request the command design pattern is used as super component. This design pattern decides the external system that must be interacted and sends the request to this [3] . The only job done for programmer is to fill the xml file that holds the commands. The source code of the pattern found from the Internet. If the command is interested in a database operation the wrapper design pattern is activated. This design pattern is an EJB design pattern that decides which session base class or classes will be activated and in which order these will be activated [4]. It formed appropriate to our framework with some modification in code. Session Base design patterns use session beans or in other words session beans inherit session base design patterns. Session base design patterns handle all the responsibilities for transaction and rollback operation. It also decides how to connect to database, by using entity beans or Data Access Object Design patterns. The data access object design pattern handles all the JDBC (Java Database Connection) requirements such as providing a connection and resources from the pool, removing connections and handling exceptions [4]. The queries that will run are decided by the DAO using the

command. Therefore the programmer provides another xml file that holds commands and queries.
This framework is designed by using COSECASE and implemented in IBM WSAD. The design patterns used are found from and they adapted with no or little modifications. Its tested for different requirements and seen that it works properly in nearly all cases. The time required to implement, adapt and use this framework is nearly nothing when compared with all the job is done by the programmer. This framework can also be used in other projects and maybe after some use its own can be a component design pattern. This example showed us that design patterns can be used as super components in component based development. This is not also true for component based framework design and also for component based system design. The generic and simple design done for framework using COSECASE is given in figure 6.

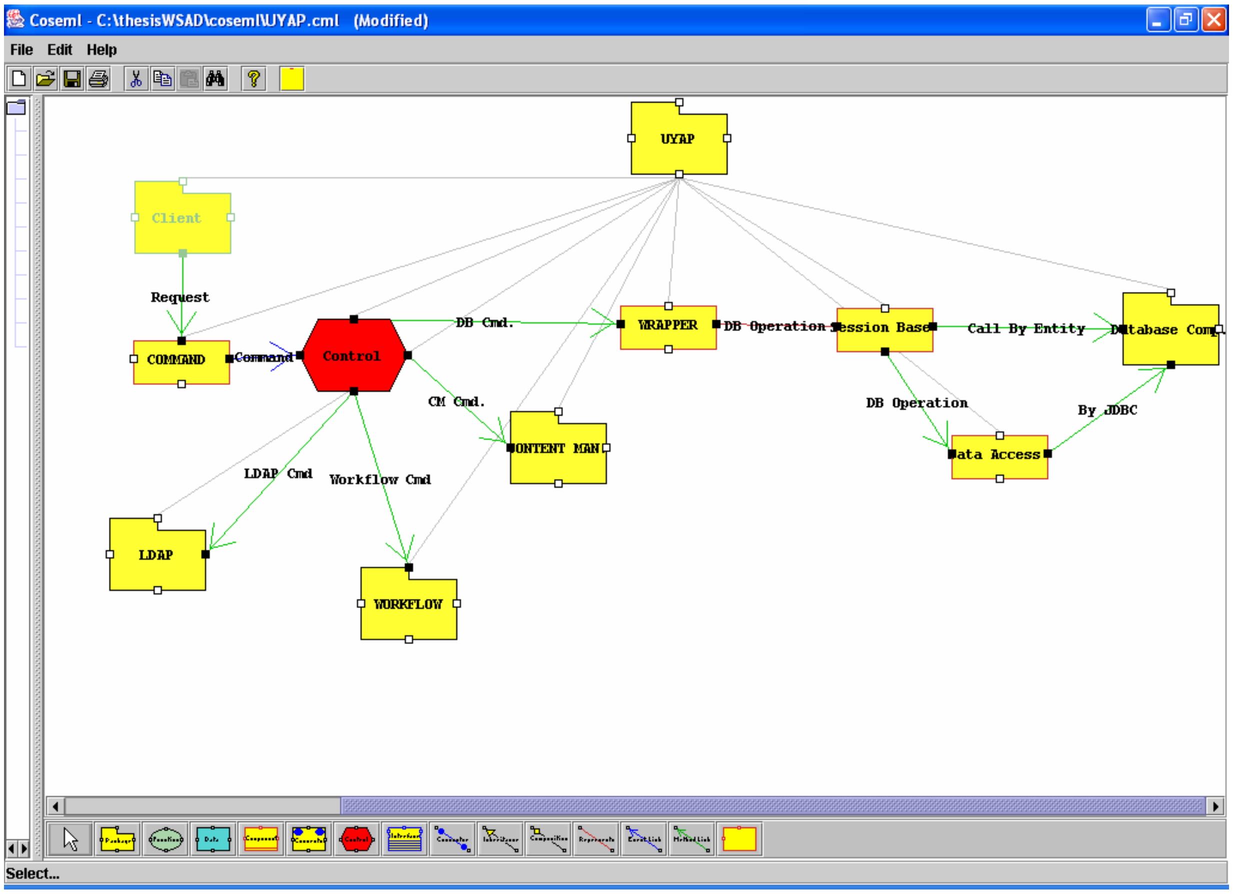

**Figure 6.** The simple COSEML design for the UYAP framework

## 5. Discussion

In object-oriented approach design patterns are used widely. So there are lots of special design patterns for object-oriented approach. For example C++ design patterns. Component based approach is a new approach for software design. Since it is not widely used as object oriented approach, it has not have specialized design patterns such as

component design patterns. However the question “can all design patterns be used in design phase?” is same for the object-oriented approach.
Design patterns can be classified into three groups according to their purpose. These are creational design patterns, structural design patterns and behavioural design patterns. Each group can be divided into two subgroups according to their scope (specifies whether the patterns applies primarily to classes or objects). Creational patterns concern with creation of objects. Structural patterns concern with the composition of classes and objects. Behavioural patterns characterize interaction of classes and objects.
While designing object-oriented systems, designer usually concerns with classes. If one of the creational patterns (for example factory or singleton design patterns) is used in creating classes in your design there is no need to give this information. Therefore UML did not deal with extra class architecture for classes created from creational design patterns or not. But this information is given in the operations defined in the class. Like UML, COSECASE behaves in the same way to this problem. In COSECASE most creational pattern take role only in methods defined in interfaces. Because there is no need and no way to show it in another way since it is a component based approach and in this approach the detail of component is not given, only the usage of component concerned.
But structural design patterns differ from creational patterns. For example adapter design pattern (also known as wrapper design pattern) can be used as a super component in component based system design. Adapter design pattern is used for mainly converting the interface of a class into another interface clients expect. The use of this design pattern in component-based systems is adding a component between the component that interacts each other, and this new component converts the interaction of these components into an appropriate form for each other. This new component is a super component since it is derived from adaptor design pattern. In COSECASE it is enabled to use this as a super-component and reuse this whenever needed.
Like structural design patterns, behavioural design patterns can be used as super components in component based development. For example command design pattern can be used as a super component in component based system design. Command design pattern is used to encapsulate a request as an object, thereby lets to parameterize clients with different requests, queue or log request, and support undoable operation In design phase command design patterns can be used as an super components between sale component and other components, thus the operations can be controlled from this super component.

## 6. Conclusion

As mentioned above design patterns can be used in component-based systems (and also in COSECASE) both in abstract phase and structural phase. The reason for the requirement to use in structural phase is the need to make modifications on design patterns. It is seen that design patterns correspond to super components when used in abstract phase.

However this does not mean that all design patterns can be used as super component directly. To use a design pattern as a super component in the COSECASE, this design pattern must be usable as a component by itself. This means it must be giving information about how to use, not how to code. Because the main issue in component based systems is not coding, using with no or less adaptation in coding. Structural and behavioural design patterns can be used in component-based system designs but this does not mean all structural and behavioural design patterns can be used.

## References

[1] A. H. Doğru, “Component Oriented Software Engineering Modeling Language: COSEML,” *TR-99-3*, Computer Engineering Department, Middle East Technical University, Dec. 1999.

[2] Erich Gamma, Richard Helm, Ralp Johnson, John Vlissides, “Design Patterns”, Addison&Wesley, 1994

[3] Floyd Marinescu , “EJB Design Patterns: Advanced Patterns, Processes, and Idioms”, Jon wiley & Sons, 2002

[4] Deepak Alur, John Crupi, Dan Malks, “Core J2EE Patterns”, Prentice Hall, 2001

[5] Ed Roman, “Mastering Enterprise Java Beans”, John Wiley & Suns, 2002

[6] Kara Aydin, “A Graphical Editor For Component Oriented Modeling”, Engineering Department, Middle East Technical University, Dec. 2001.

[7] Thomas Larsson1, Mikael Sandberg, “Building Flexible Components Based on Design Patterns”, Mälardalen University, Department of Computer Engineering, Sweden,

[8] Felix Bachmann, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, Kurt Wallnau, “Volume II: Technical Concepts of Component-Based Software Engineering, 2nd Edition”, CMU/SEI-2000-TR-008 ESC-TR-2000-007, May 2000

[9] R. Schmidt, U. Assmann, “Concepts For Developing Component-based Systems”, Forschungszentrum Informatik and Universität Karlsruhe,