VISUAL COMPOSITION IN COMPONENT ORIENTED DEVELOPMENT

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

MURAT MUTLU ÖZTÜRK

IN PARTIAL FULLFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

AUGUST 2005

Approval of the Graduate School of Natural and Applied Sciences.

Prof. Dr. Canan Özgen
Director

I certified that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Ayşe Kiper
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Ali H. Doğru
Supervisor

Examining Committee Members

Assoc. Prof. Dr. İ. Hakkı Toroslu	(METU, CENG)	
Assoc. Prof. Dr. Ali H. Doğru	(METU, CENG)	
Assoc. Prof. Dr. Veysi İşler	(METU, CENG)	
Dr. Ayşenur Birtürk	(METU, CENG)	
Dr. Erhan Gökçay	(METU, II)	

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name:

Signature :

ABSTRACT

VISUAL COMPOSITION IN COMPONENT ORIENTED DEVELOPMENT

Öztürk, Murat Mutlu

MS, Department of Computer Engineering

Supervisor: Assoc. Prof. Dr. Ali Hikmet Doğru

August 2005, 91 pages

This thesis introduces a visual composition approach for JavaBeans components, in compliance with the Component Oriented Software Engineering (COSE) process. The graphical modeling tool, COSECASE, is enhanced with the ability to build a system by integrating domain-specific components. Such integration is implemented by defining connection points and interaction details between components. The event model of the JavaBeans architecture is also added to the capabilities.

Keywords: component orientation, visual composition, case tool, component integration.

ÖZ

BİLEŞEN YÖNELİMLİ GELİŞTİRMEDE GÖRSEL TÜMLEME

Öztürk, Murat Mutlu
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Ali Hikmet Doğru

Ağustos 2005, 91 sayfa

Bu tez çalışması Bileşen Yönelimli Yazılım Mühendisliği (BYYM) sürecine uygun bir görsel tümleme yaklaşımı ortaya koymaktadır. Grafiksel bir modelleme aracı olan Bileşen Yönelimli Yazılım Mühendisliği Modelleme Aracı (BYYMMA), bir sistemi özel bir alana özgü bileşenleri tümleyerek oluşturabilme kabiliyetine sahip olacak şekilde geliştirilmiştir. Bu tümleme, bileşenler arasındaki bağlantı noktalarının ve etkileşim detaylarının tanımlanması ile gerçekleştirilir. JavaBeans mimarisinin olay modeli eklenen kabiliyetler arasındadır.

Anahtar Kelimeler: bileşen yönelimi, görsel tümleşim, BYYM aracı, bileşen tümleşimi.

To my wife, Hülya

ACKNOWLEDGEMENTS

I am obliged to my supervisor, Assoc. Prof. Dr. Ali Hikmet Doğru, not only for his graceful and forbearing support and guidance, but also his very helpful personal advisory. I would like to thank to my sweat heart for her great effort making me finish this work on time and for her endeavor on the duties for our marriage preparation. Finally, I would like to thank to my family for their extravagant belief in me for everything that I have included.

TABLE OF CONTENTS

PLAGIARISM iii

ABSTRACT iv

ÖZ v

ACKNOWLEDGEMENTS vii

TABLE OF CONTENTS viii

LIST OF TABLES x

LIST OF FIGURES xi

LIST OF ABBREVIATIONS xiv

I. INTRODUCTION 1

II. BACKGROUND 5

II.1. Component vs. Object 9

II.2. Component Based Development 14

II.3. Component Orientation 18

II.4. Component Frameworks 20

II.4.1. Java Component Framework: JavaBeans 21

II.4.1.1. Event Model in JavaBeans 24

II.4.1.2. Introspection 25

II.4.1.3. JavaBeans Packaging 26

III. COSE APPROACH 28

III.1. Previous Research in the Area 28

III.2. Defining the COSE Modelling Language 30
III.3. COSE Process Model....................................34
IV. VISUAL COMPOSITION....................................37
IV.1. Previously Proposed Component Integration Approaches 38
IV.2. Visual Composition in Component Oriented Development 49
IV.2.1. Component Framework Selection 49
IV.2.2. Visual Composition in COSEML....................................51
IV.2.3. JavaBeans Integration in COSEML....................................57
IV.2.4. Sample Applications 69
IV.2.4.1. Sample System 1....................................69
IV.2.4.2. Sample System 2....................................71
IV.2.4.3. Sample System 3....................................73
IV.2.4.4. Sample System 4....................................75
IV.2.4.5. Sample System 5....................................77
V. CONCLUSION 79
V.1. Evaluation 79
V.2. Future Research....................................81
REFERENCES 83

LIST OF TABLES

Table 1 Classification for Component Definition (adapted from [43]) 7

Table 2 Classification for Component Orientation (adapted from [43]) 8

Table 3 Details of the Graphical Symbols used by COSEML (adapted from [42]). 32

LIST OF FIGURES

Figure 1 Reference Model for Component Architecture (adapted from [51]) 6

Figure 2 Objects and Components (adapted from [16]) 11

Figure 3 Comparison of Object and Components (adapted from [71]) 12

Figure 4 Inheritance in Object Orientation (adapted from [16]) 12

Figure 5 Component Based Software Development (adapted from [53]) 15

Figure 6 A General Process Model for Component-Based Software Development 18

Figure 7 Component Oriented Software Development (adapted from [16]) 19

Figure 8 Differences between Object Orientation and Component Orientation 20

Figure 9 Overview of the Event Model in JavaBeans (adapted from [73]) 25

Figure 10 Modeling Emphasis for Different Approaches (adapted from [72]) 29

Figure 11 Graphical Symbols in COSEML (adapted from [72]) 31

Figure 12 Modeling of a Small Business Automation System with COSEML (adapted from [72]) 34

Figure 13 COSE Process Model 36

Figure 14 CoOWA Diagram [adapted from [59]] 48

Figure 15 Event Model in JavaBeans Component Framework 52

Figure 16 Event Conduction with Data Transferring 53

Figure 17 Event Conduction with Method Calls without Argument 53

Figure 18 Abstract Level Connectors 54

Figure 19 Method and Event Links 54

Figure 20 JavaBeans Event Model and COSEML Mapping 56

Figure 21 Mode Change Button 57

Figure 22 Main Window of COSECASE 58

Figure 23 Bean Instantiation Window 58

Figure 24 Bean Instantiation Window in Design Mode 59

Figure 25 A System Design with Two Components 60

Figure 26 System Design before Assigning JavaBean Components.......................61

Figure 27 Triggering Component Loading 61

Figure 28 JAR Selection Dialog...62

Figure 29 System Design after Assigning JavaBean Components..........................62

Figure 30 Components Associated with "Connector" Link 63

Figure 31 Creating Association 64

Figure 32 Event Assignment Dialog...65

Figure 33 Target Method Assignment Dialog...66

Figure 34 Source Method Assignment Dialog 67

Figure 35 Bean Instantiation Window 67

Figure 36 Main View 68

Figure 37 Execution Scenario of the Sample System 1 69

Figure 38 Main Window of COSECASE for Sample System 1 70

Figure 39 Bean Instantiation Window for Sample System 1 70

Figure 40 Execution Scenario of the Sample System 2 71

Figure 41 Main Window of COSECASE for Sample System 2 72

Figure 42 Bean Instantiation Window for Sample System 2 72

Figure 43 Execution Scenario of the Sample System 3 73

Figure 44 Main Window of COSECASE for Sample System 3 74

Figure 45 Bean Instantiation Window for Sample System 3 74

Figure 46 Execution Scenario of the Sample System 4 75

Figure 47 Main Window of COSECASE for Sample System 4 76

Figure 48 Bean Instantiation Window for Sample System 4 76

Figure 49 Execution Scenario of the Sample System 5 77

Figure 50 Main Window of COSECASE for Sample System 5 78

Figure 51 Bean Instantiation Window for Sample System 5 78

LIST OF ABBREVIATIONS

ACOEL	A Component-Oriented Extension Language
ADP	Abstract Design Paradigm
CASE	Computer Aided Software Engineering
CB	Component Based
CO	Component Oriented
COM	Component Object Model
CoOWA	Component Oriented Web Application Model
CORBA	Common Object Request Broker Architecture
COSE	Component Oriented Software Engineering
COSECASE	Component Oriented Software Engineering Computer Aided Software Engineering
COSEML	Component Oriented Software Engineering Modeling Language
COTS	Commercial Off-The Shelf
DCA	Distributed Component Architecture
DCOM	Distributed Component Object Model
EJB	Enterprise JavaBeans
IDE	Integrated Development Environment
IDL	Interface Definition Language
I/O	Input Output
OO	Object Oriented
TIL	Tool Interconnection Language
UML	Unified Modeling Language

CHAPTER I

INTRODUCTION

In the software engineering literature, the term “component” has been described in a wide variety of definitions according to its usage and understanding. Some of the definitions stated by the authorities in component technologies are as follows:

- Szyperski defines the components as “units of implemented software building blocks” [1]. In the context of deployment perspective, Szyperski rephrases the definition as, “a unit of composition with contractually specific interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties” [1].

- According to Paul, Sattler and Endig [2], a component is defined as follows:

“A component is a software entity with well–defined interfaces. An interface is an abstraction of the component behavior and encapsulates the internal representation of state and implementation. The interfaces and the implementation of a component are defined by its component class. In principle there are two kinds of component interfaces:

  - Operational interfaces define a set of operations which can be performed by a component or which a component can invoke at another one.

  - Event interfaces define a set of events which can be announced by a component."

- A component is also defined as "a unit which provides its clients with services specified by its interface and encapsulates local structures that implement these services" [3, 4, and 5].

- As interpreted from the UML point of view, "a component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces." [6].

- In terms of Booch, "a reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction." [9].

- As a last definition, "a component is a self-contained piece of software, with well defined interaction points and modes, and is intended to be combined with other components into larger software entities" [8].

As it can be concluded from the variation in the definitions of the component term, it is a new concept within the software engineering discipline. In accordance with the maturement process of the engineering approaches, "software also first enjoys new technologies; engineering methodologies come after to exploit them. Programming Languages were followed by traditional methodologies. Object-Oriented (OO) languages were followed by OO methodologies. Finally, Component-Based (CB) technologies have recently been introduced and a fully Component-Oriented (CO) approach is not yet widely known. COSE attempts to pioneer such an approach. Its modeling language, COSEML, is also a new representation based on related work that emphasizes a hierarchical decomposition of systems into components" [11, and 12].

A graphical design tool is developed for supporting the design terminologies elaborated by COSEML. This graphical tool enables the user to hierarchically decompose a system's requirements, and to view structural relations among

components graphically [13], which fits into the design phase of the COSE Process Model [14].

"Despite difficulties, components are meant for run-time connecting. In a purely component-oriented approach, assuming that most functions have been implemented in components, the system development problem reduces to locating and connection of defined components" [10]. If system definition manner in COSE Process Model is considered, all the connections in between resolved components are defined while hierarchically decomposing the system. The only issue that needs to be addressed is the composition of the physical building blocks -components- of the system.

Indisputably, one of the most effectual factors of the rapid development through component oriented software engineering is "reusability". The search for efficient reusability techniques motivated the developers to seek new approaches for the application development process. One of these is component orientation, which offers great beneficial future for reusability. Effective reuse of software presupposes the existence of tools to support the organization and retrieval of components according to application requirements and the interactive construction of running applications from components [18]. COSEML is encouraged for filling up a gap in software engineering.

In this thesis, a visual component integration (composition) mechanism is introduced into the COSE approach by combining the design and integration phases defined in the process. This work improves COSECASE with framework integration capabilities. In the integration module of COSECASE, an approach for component integration based on the description and runtime mappings of component interconnections is proposed.

JavaBeans framework is selected as the component framework that is introduced to the COSEML Modeling Media. The main reason for selecting JavaBeans is the appropriateness of the Event Model in the JavaBeans framework to

the COSEML. Other reasons that makes JavaBeans framework suitable are described in detail in the following sections.

Beyond this first chapter, the thesis is organized as follows: In chapter II, required background on objects, components, component based development, component orientation and component architectures are described. Additionally, JavaBeans component framework is illustrated in detail. Chapter III describes the COSE approach and defines the modeling language. In addition to this, a COSE based process model is summarized. In chapter IV, visual composition in COSE is explained and the adaptation approach of JavaBeans Event Model to COSEML is described. As a last chapter, chapter V presents a brief conclusion for the thesis study, and presents further work that can be performed over this study.

CHAPTER II

BACKGROUND

Existing object oriented techniques usually consider the developer to make changes to the part being reused. It takes extra time and effort for the developer to adopt software. Also, when an original part gets modified, updating the adopted version can be difficult and may require additional human effort and interaction. Component oriented programming provides an easier way for reuse [26].

Commonly, because of being very close to each other, the borders between objects and components seem to be flue in developers' minds. Hence, it would be helpful to clearly expose the definitions of component and component orientation. In the literature, it is hard to find a generally accepted definition of a component.

For the general classification of components, [51, and 52] consider three dimensional discrepancy directions for the scope of viewpoint division, as it can be seen in Figure 1:

- granularity of components (Axis 1)
- level of specialization in actuality (Axis 2)
- level of specialization in implementation (Axis 3)

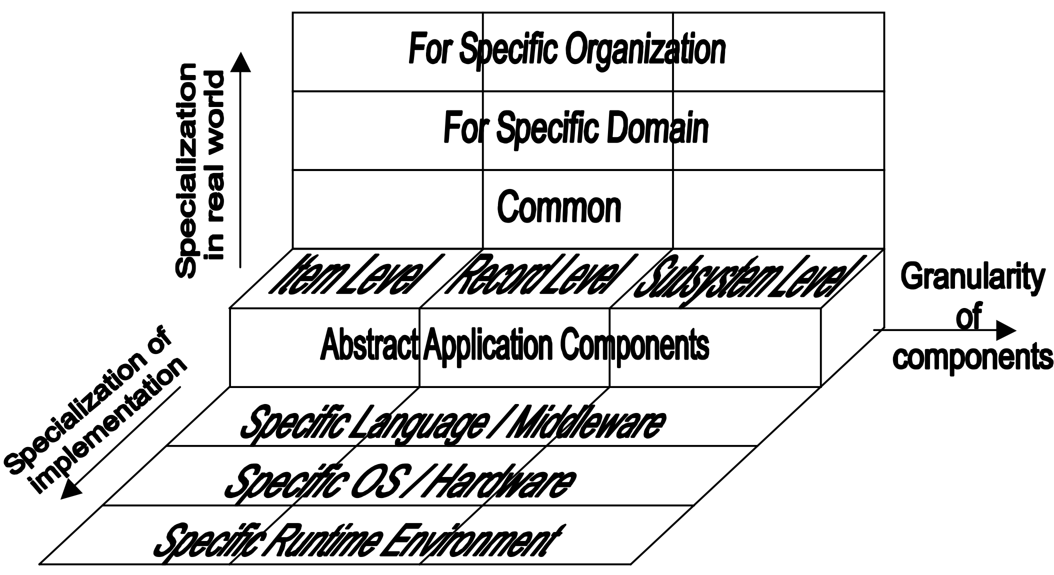

Figure 1 Reference Model for Component Architecture (adapted from [51])

To achieve a clear understanding of what the core features of a software component are, [43] provided a classification framework to clearly categorize each of the proposed models, frameworks, or standards. The goal of this classification is to obtain a consolidated and clear definition of what a component constitutes. Morphological classification attributes of components are summarized in Table 1. In this table, shaded fields constitute the commonly accepted component attributes.

In addition to the classification framework of components, classification framework for component-oriented approaches is also provided by [43] as it can be seen in Table 2. Shaded cells in Table 2 state an assignment to the corresponding concept, in component orientation.

Table 1 Classification for Component Definition (adapted from [43])

CHARACTERISTIC		VALUE OF CHARACTERISTICS						
Unit of Composition		Block of Code	Module	Class	Object	Macro/ Template	Abstract Data Type	Component
Reuse	Platform Dependency	Proprietary	Restricted	Independent				
Reuse	Inter-Component Dependency	Coupled	Loosely Coupled	Self-Contained				
Reuse	Paradigm	Implementation Inheritance	Overwriting	Generative	Compositional			
Interface	Definition	Subject to Change	Contract					
Interface	Standard	Proprietary	Industry Standard	Open Standard				
Interoperability		Proprietary	Restricted	Unrestricted				
Granularity		Coarse	Fine	Elemental				
Hierarchy		Flat	Eligible	Tree-like				
Visibility		White-Box	Glass-Box	Gray-Box	Black-Box			
Composition	Time	System Implementation	System Integration	Run-Time				
Composition	Role	Component Developer	Application Designer	Application User				
State		Non Persistent	Persistent	Stateless				
Extent of Deployment		Part of Component	Full Component					
Marketability		Not Planned	Restricted	Full				
Supported Object Oriented Concepts		Encapsulation	Inheritance: Implementation	Inheritance: Interface	Polymorphism			

Table 2 Classification for Component Orientation (adapted from [43])

CHARACTERISTIC		VALUE OF CHARACTERISTICS						
Unit of Composition		Block of Code	Module	Class	Object	Macro/ Template	Abstract Data Type	Component
Reuse	Platform Dependency	Proprietary	Restricted	Independent				
Reuse	Inter-Component Dependency	Coupled	Loosely Coupled	Self-Contained				
Reuse	Paradigm	Implementation Inheritance	Overwriting	Generative	Compositional			
Interface	Definition	Subject to Change	Contract					
Interface	Standard	Proprietary	Industry Standard	Open Standard				
Interoperability		Proprietary	Restricted	Unrestricted				
Granularity		Coarse	Fine	Elemental				
Hierarchy		Flat	Eligible	Tree-like				
Visibility		White-Box	Glass-Box	Gray-Box	Black-Box			
Composition	Time	System Implementation	System Integration	Run-Time				
Composition	Role	Component Developer	Application Designer	Application User				
State		Non Persistent	Persistent	Stateless				
Extent of Deployment		Part of Component	Full Component					
Marketability		Not Planned	Restricted	Full				
Supported Object Oriented Concepts		Encapsulation	Inheritance: Implementation / Interface	Polymorphism				
Model Class		Component-System-Framework	Component-Application-Framework	Business Component				
Distribution		Homogeneous	Heterogeneous					
Domain Standard		Proprietary	Industry Standard	Open Standard				
Adaptation		Implementation Changes	Customizing	Exchange of Modules, Parts or Components				
Adaptation Time		Implementation	Integration	Run-Time				
Conflict-Solving		Technical Level	(Business) Domain Level	(Business) Domain Level and Standardized				

In general, a component can be defined as an autonomous unit of code, which has clearly defined interfaces, and is a black box to the user. Based on this definition, a developer can ignore the components' internal implementation, and directly focus on only the interface and connection of the interfaces.

In order to define the beneficial usage grade of component technologies, Szyperski proposed a component maturity model according to the component concept explored which includes the following maturity levels: [48]

1. Maintainability: modular solutions
2. Internal reuse: product lines
3.a.1 Closed composition: make and buy from closed pool of organizations
3.a.2 Open composition: make and buy from open market
3.b Dynamic Upgrade
4. Open and Dynamic

These approaches solidify the scope and the meaning of the component and component oriented framework concepts.

II.1. Component vs. Object

During the evaluation of the software methodologies, object oriented methodologies have appeared earlier than the component-oriented methodologies. The main reason for proposing a new concept, namely component oriented software engineering, to the literature is the inefficient addressing of reusable frameworks in object-oriented methodologies. According to Pfister and Szyperski, weakness of

the object oriented programming is that, it too often concentrates on individual objects, instead of whole collections of objects, i.e., components [15].

In a very general perspective, objects are meant to encapsulate services, on the other hand, components are considered as abstractions of group of objects. Objects have identity, state and behavior, and are always run-time entities. Components, on the other hand, are generally static entities that are needed at system build time and consist of interface and implementation. The interface, in addition to the state and behavior, contains events that are the actions to be performed when an event occurs in the component or in the environment. The implementation is a set of objects that are not visible to the outside, and it can be changed without affecting the client of the component [13]. Components do not necessarily exist at run-time and may be of finer or coarser granularity than objects: e.g., classes, templates, mix-ins, and modules. Components should have an explicit composition interface, which is type-checkable [16].

Figure 2 depicts the difference the between objects and components, as far as input/output connectivity is concerned. In the perspective of this research, the resemblance between the two concepts is trivial. Component orientation is interpreted as a revolutionary paradigm to software development. As long as a module complies with its interface specifications, it is regarded as a component. The development of this component does not have to be object oriented. No matter how a component was modeled or developed, once it is available, it is executable code that presents interfaces. Now, component oriented approach, having started since the requirements, in a decomposition approach, can utilize these components. The idea is to integrate available components, rather than developing new code.

Figure 2 Objects and Components (adapted from [16])

As being one of the main differences between objects and components, while designing the systems, generally, objects uses the inheritance mechanism meanwhile component-based integration is suited for composition. Figure 3 presents a comparison of objects and components, within the integration perspective.

In fact, most of the problems arise because inheritance is basically a white box reuse mechanism. Inheritance conflicts with encapsulation since subclasses are dependent upon implementation details of super-classes in a way that is not described by an explicit interface (Figure 4) [16]. Consequently, object oriented programming hasn't created a viable software component industry. From a technical perspective, the reason for this failure lies in an insufficient consideration of the unique requirements of component software [15].

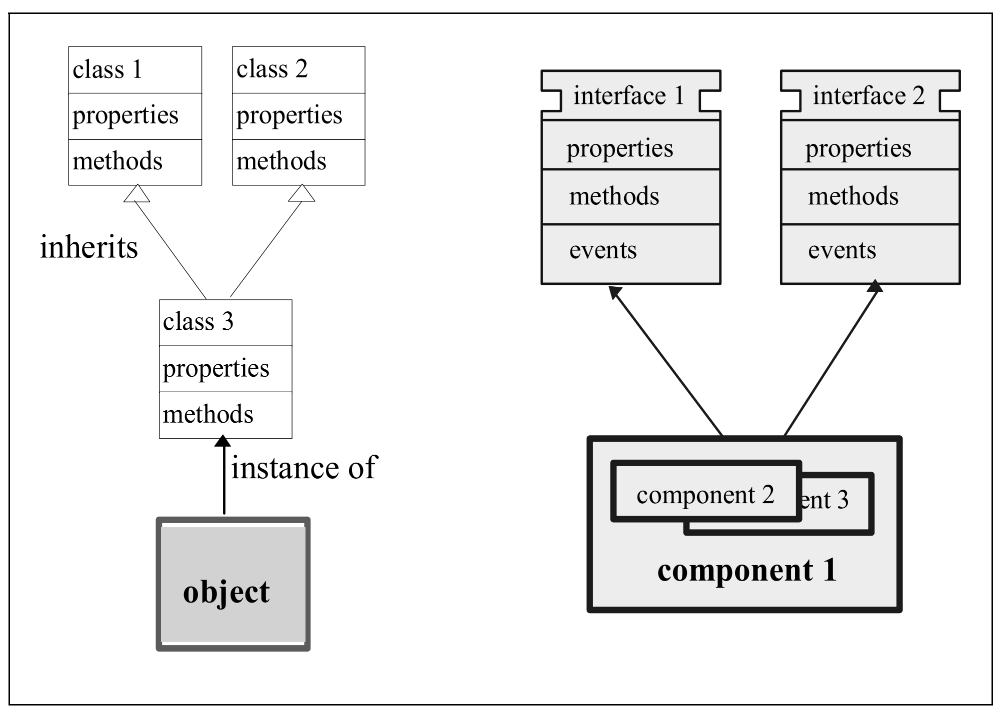

Figure 3 Comparison of Object and Components (adapted from [71])

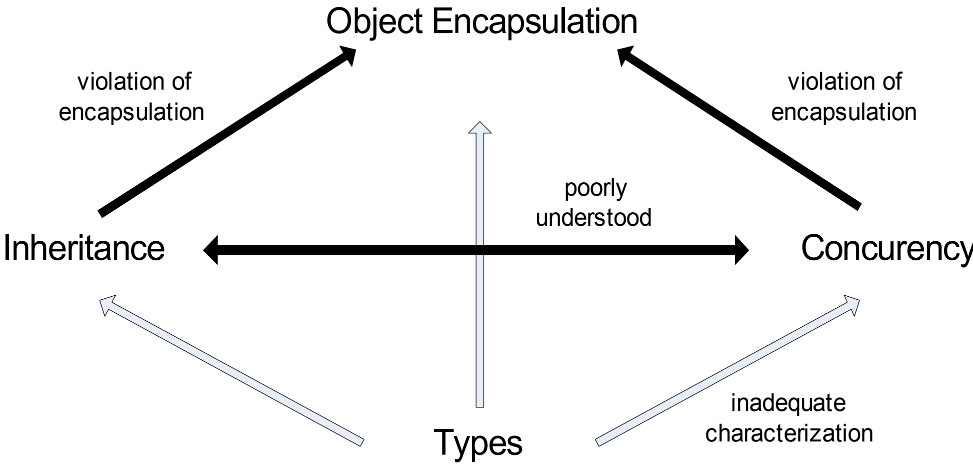

Figure 4 Inheritance in Object Orientation (adapted from [16])

According to [16] issues addressed by components, considering objects are as follows:

- Reuse: “Software reuse” is the key to building the systems: not only are the cooperating resources (software components) used across multiple applications, but also the components that realize non-functional behavior will be used in various configurations to address a variety of requirements.

- Fast time-to-market: Applications built from reusable components can be developed more quickly and less costly than custom-made applications.

- Reliability: Components that are reused across many applications are bound to be more reliable than new, hand-coded components. Applications built according to tested frameworks are bound to be more reliable than newly designed and implemented applications.

- Division of labor: Components with well-defined interfaces are natural units for distribution to software teams. Development of applications from software components and the development of reusable components themselves are tasks requiring different kinds of skills and experience.

- Variability: Families of applications can be developed using a common software base if the software base can accommodate sufficient variability. Software components support variability through parameterization. Parameters represent functionality that must be provided by the client of the component, (as is often the case with object-oriented components). An alternative way to variability is through overriding default functionality.

- Adaptability: A flexible application is one that can be easily adapted to changing requirements. Software components support adaptability if an application can be viewed abstractly as a configuration of components that are linked together. If the components have been well-designed, many changes in requirements can be addressed at this abstract level by

reconfiguring the application's components. In well-understood application domains, many possible changes in requirements can be anticipated and incorporated into the design of the components and the ways in which they may be combined. Note that adaptability may be viewed as a form of reusability, since it entails the reuse of an existing application to create a changed version. However, it does not focus on building (larger) new systems from (smaller) existing software components. Variability is a pre-requisite to adaptability but increasing variability in the components may damage adaptability since adaptation becomes increasingly complex.

- Distribution and concurrency: In order to use hardware resources optimally, systems are becoming more distributed and consequently concurrent. Since distributed systems are notoriously difficult to implement correctly, application developers need software abstractions that can simplify the task. Components offer on the one hand natural units for distribution, and on the other hand may encapsulate protocols and concurrency abstractions, thus hiding the complexity of distributed programming from application developers.

- Heterogeneity: Open systems are inherently heterogeneous. Components of a distributed system will be developed using different platforms and programming languages. Components help by hiding differences in implementation platform behind interfaces that are (in principle) independent of programming languages, as in component models such as Component Object Model (COM) and Common Object Request Broker Architecture (CORBA) [24, and 25].

II.2. Component Based Development

During the evolution of software development methodologies, the main objective is improving the development practices by means of improving the design

techniques. One of the best practices in this improvement is utilizing the reusability. In spite of the volatile successes, because of the absence of the methodological approaches, there exist issues to be resolved in software development with components.

Since the component-based methodology is not as mature as desired different understandings of the methodology exist [14]. In the remaining of this section, component-based development is offered to resolve the misleading understandings.

Component-based software system development has been thought of as a solution for remarkably improving both software productivity and quality ever since standard component infrastructure technologies appeared in the software industry.

Component-based software development approach is based on the idea to develop software systems by selecting appropriate of-the-shelf components and then to assemble them with a well-defined software architecture [53]. Figure 5 summarizes the easiness of the process.

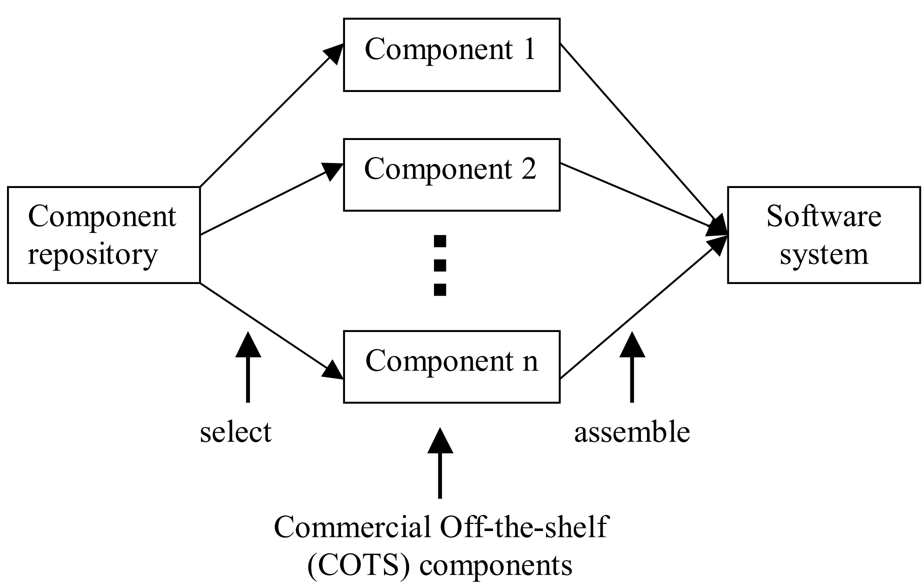

Figure 5 Component Based Software Development (adapted from [53])

Another driver is that several underlying technologies have matured such that they permit building components and assembling applications from sets of those components. Development environments and the supporting tools make it possible to share and distribute application pieces.

Component technologies, such as ActiveX, CORBA, JavaBeans, and Enterprise JavaBeans (EJB), have become widely available and are in common use. Lots of software components using these technologies are already available. Nowadays, an application system may be built simply by collecting appropriate software components and putting those together [51].

Component based software development environments are generated for supporting the interfacing mechanism in component integration. Two of those significant component based application architectures are ComponentAA [51] and, SYNOPSIS [65], which are proposed for developing business application systems. ComponentAA is made up of an effective combination of frameworks, patterns, and methodologies. Besides, there exist various approaches for Component based integration tools and scripting languages that counsel the usage of scripting component glue definitions to solve the integration issue of components [66, 67, and 68].

Component-based development is an important maturement level for the organizations that are exhibiting an improvement strategy for their software development process. By means of this approach, software development is carried from code based programming to integrational system construction approach by using pre-implemented, reliable, mature, and well-defined software components. General life-cycle of a component-based development is presented in Figure 6.

According to this lifecycle, system development starts with defining the system specification (a). After defining the requirements clearly, the system is decomposed into building blocks within the second stage (b). In the third stage, specification of the components is described according to the decomposed system

(c). Each functional block of the decomposed system is treated and specifications are designated. After the specification process, component repository is searched for correspondence to the functional components of the decomposed system. Retrieved components are examined for appropriateness and the modification needs are determined. After identifying the reusable ones, components that need to be implemented from scratch are identified. If there is need for new components, component development is performed according to the decomposed component specifications. After acquiring the necessary components for the system, the integration process takes place (d). In this phase, the components are integrated according to the decomposition specifications described in previous stages. If the system decomposition and component specification is not carried out successfully, then this phase brings all inappropriateness's to the light. After this phase, the generated system is ready for testing (e). Types and specifications of these tests are described in detail in [13].

As a final remark for this section, the differences between component-based and object oriented development is that, in component-based development, software systems are designed and developed by using components, whereas object orientation uses objects for the system development. There are many advantages of component-based development over object oriented development. Briefly, it separates the specification from implementation which provides greater flexibility of control and enables the software developers to design with interfaces which offers more efficient reusability.

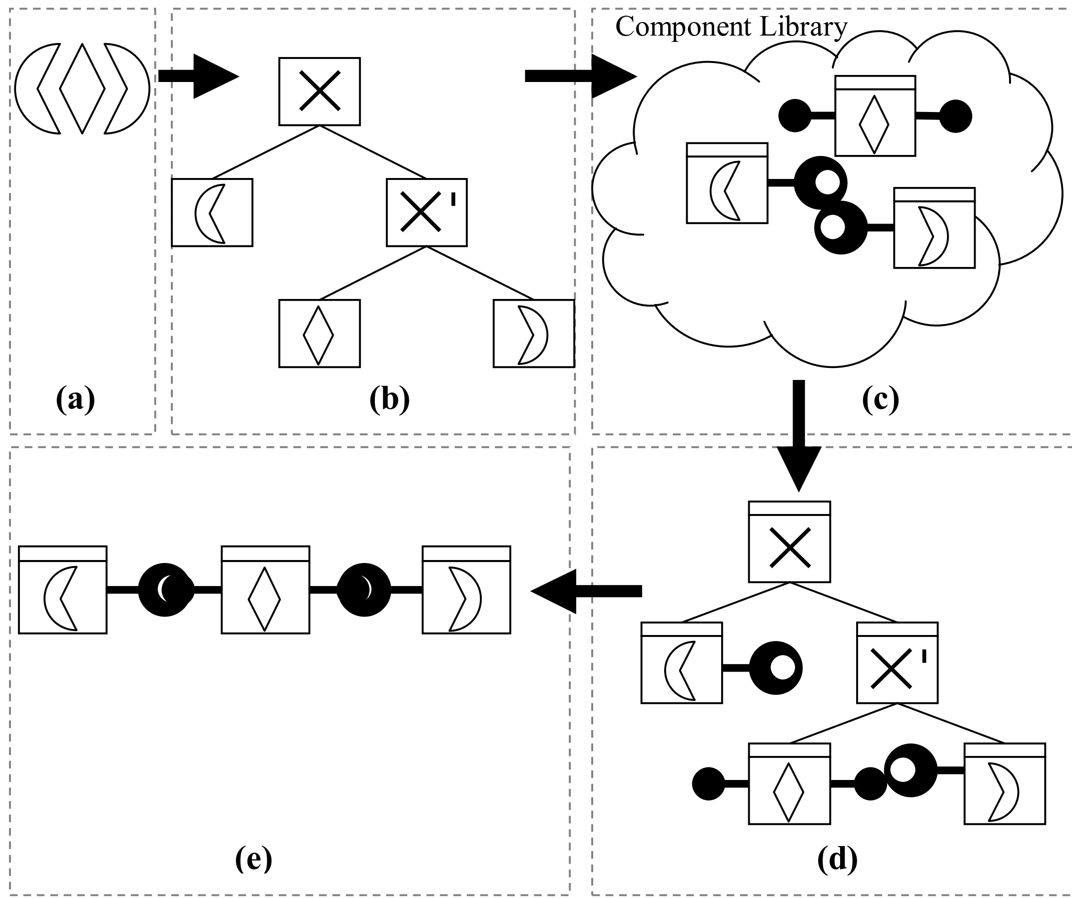

Figure 6 A General Process Model for Component-Based Software Development

II.3. Component Orientation

The evolution from object-oriented systems to component-oriented systems is accepted as the state of the art and components and component-orientation is often assumed as the next step after object orientation [17, and 43]. Component oriented programming replaces monolithic software systems with reusable software components and hierarchical component frameworks [1].

Component oriented software development can be explained as solving a puzzle, whose parts are ready for integration. In that manner, for building up the whole system, the only problem to solve is finding out the appropriate parts and

putting them together. By using the same approach, COSE paradigm is regarded as system development by putting the pluggable components together.

Figure 7 summarizes the activities of the component oriented software development in a graphical way. Accordingly, development of the system is integration of the components by soldering the interfaces of the components.

Component-oriented programming requires more stringent information hiding, a more dynamic approach, and better safety properties than object-oriented programming [15]. To be more specific, object-orientation uses objects as the building blocks of the systems whereas component-orientation uses components. Figure 8 depicts the difference between object oriented and component oriented development from the decomposition point of view.

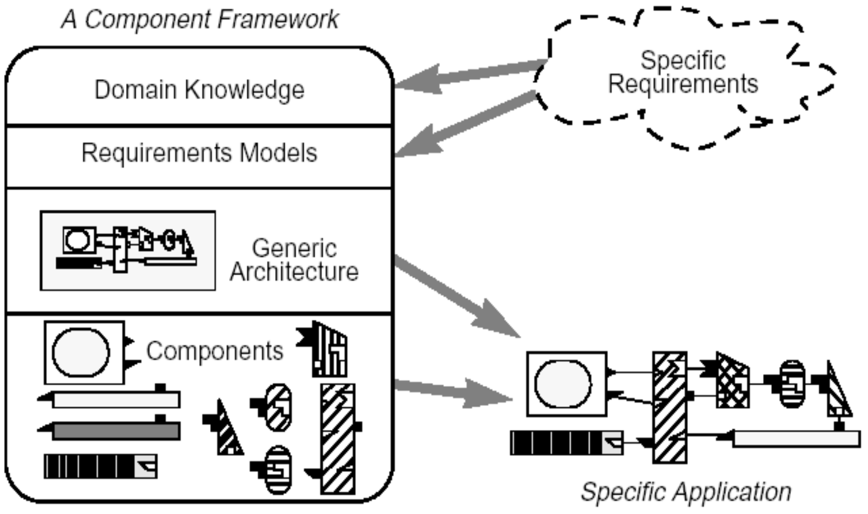

Figure 7 Component Oriented Software Development (adapted from [16])

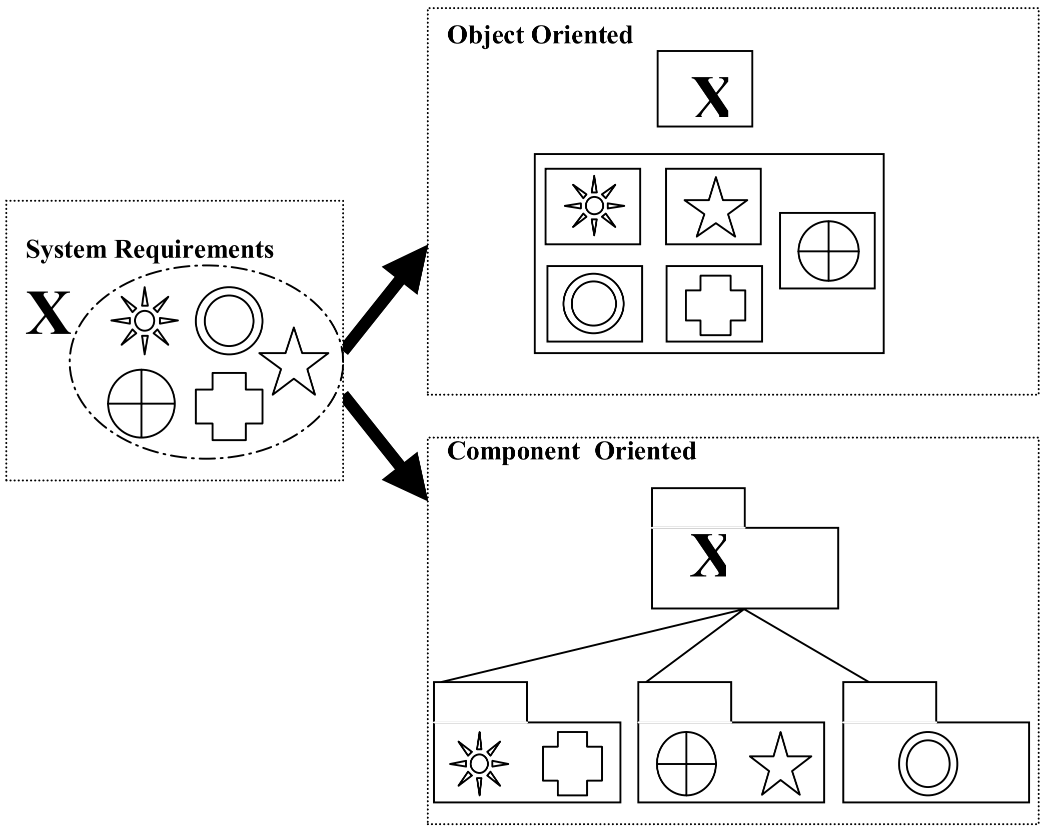

Figure 8 Differences between Object Orientation and Component Orientation

II.4. Component Frameworks

Software component architecture is essential to provide a framework which provides services for the communication of the components without any knowledge of each other. The most popular of these technologies are COM, Distributed COM (DCOM), ActiveX, CORBA, JavaBeans and Enterprise JavaBeans (EJB). Details of JavaBeans Component framework are included because of being related with the scope of this thesis. Detailed information about the rest of those frameworks can be found in [13, 14].

II.4.1. Java Component Framework: JavaBeans

"A Java Bean is a reusable software component that can be manipulated visually in a builder tool."

JavaBeans™, Sun Microsystems, [73]

Sun's Java-based component model consists of two parts: the JavaBeans for client-side component development and the EJB for the server-side component development. With EJB technology the enterprise developer no longer needs to write code that handles transactional behavior, security, connection pooling, or threading, because the architecture delegates this task to the server vendor. EJB component model logically extends the JavaBeans component model to support server components. Server components are similar to client components, but they are generally larger grained and more complete than client components. EJB components cannot be manipulated by a visual Java Integrated Development Environment (IDE) in the same way that JavaBeans components can. Instead, they can be assembled and customized at deployment time using tools provided by an EJB-compliant Java application server [14].The JavaBeans component architecture supports applications of multiple platforms, as well as reusable, client-side, and server-side components [75].

Java platform offers an efficient solution to the portability and security problems through the use of portable Java byte codes and the concept of trusted and un-trusted Java applets [53]. Java provides a universal integration and enabling technology for enterprise application development, including

- interoperating across multi-vendor servers
- propagating transaction and security contexts
- servicing multilingual clients
- supporting ActiveX via DCOWCORBA bridges.

JavaBeans and EJB extend all native strengths of Java including portability and security into the area of component-based development. The portability, security, and reliability of Java are well suited for developing robust server objects independent of operating systems, Web servers and database management servers [53].

One of the main goals of the JavaBeans architecture is to provide platform neutral component architecture. JavaBeans varies in the functionality they support, but they typically include the following features [13, 14, 17, and 73]:

- Introspection: JavaBeans can be analyzed for the properties by using this functionality. This feature is described in detail in section II.4.1.2.
- Customization: While using on an application builder, the appearance and the behavior of a JavaBean can be changed by using this functionality.
- Events: Used for connecting the JavaBeans. A JavaBean can be an event source, event listener or both. Event sources fire events to registered listeners. Event listeners register themselves with event sources by calling registration methods. Events provide a convenient mechanism for allowing components to be plugged together in an application builder, by allowing some components to act as sources for event notifications that can then be caught and processed by either scripting environments or by other components. Event model implemented in the JavaBeans architecture is described in detail in the preceding sections.
- Properties: Mainly used for customization. Properties are attributes of JavaBeans that affect the appearance or behavior of them. They can be private, protected or public. Also, they can be read-only, write-only or read-write depending on what access methods are provided. Properties have four types:

- Simple: Simple property is a single value defined with a pair of set/get methods.
- Indexed: Indexed property is an array of values for which the set/get methods take the index parameter. Set/get methods can set/get the entire array at once.
- Bound: Bound property fires a change event when its value changes.
- Constrained: Constrained property allows other objects to veto a value change.

- Persistence: After customization, a JavaBean holds its last execution state by using this functionality.

The methods, fields, constructors, interfaces, and classes of Java have a meta-data class that supports dynamic interrogation, instantiation and invocation [13, 74]. JavaBeans run within the same address space as their container [13, 73]. If the container is a Java application then the contained JavaBean runs in the same Java Virtual Machine as its container. If the container is a non-Java application then the JavaBean runs in a Java Virtual Machine that is directly associated with the application.

The JavaBeans components can run under multi-threaded access. This can be done by making all the methods synchronized. In addition, Enterprise JavaBeans (EJB) can be used for building scalable, distributed and component-based applications. It extends the JavaBeans component model to handle the needs of transactional business applications [13, 73].

The advantages of using JavaBeans include platform-independence, security and better memory management. If the applications and controls need to be portable, it makes more sense to use JavaBeans, but if the application being developed is specific to Microsoft platforms, ActiveX may be a better choice.

Another advantage is that to develop JavaBeans components, developers should not learn new languages or extensive interface/class libraries. Because of the simplicity and ease of development of JavaBeans, it is an alternative to other component models [76].

II.4.1.1. Event Model in JavaBeans

This section describes the Event Model in JavaBeans component architecture. Figure 9 presents a detailed event generation and consumption sequence.

The model can be outlined as follows:

- Event notifications are propagated from sources to listeners by Java method invocations on the target listener objects [73].
- Each distinct kind of event notification is defined as a distinct Java method. These methods are then grouped in EventListener interfaces that inherit from *java.util.EventListener* [73].
- Event listener classes identify themselves as interested in a particular set of events by implementing some set of EventListener interfaces [73].
- The state associated with an event notification is normally encapsulated in an event state object that inherits from *java.util.EventObject* and is passed as the sole argument to the event method [73].
- Event sources identify themselves as sourcing particular events by defining registration methods that conform to a specific design pattern and accept references to instances of particular EventListener interfaces [73].
- In circumstances where listeners cannot directly implement a particular interface, or when some additional behavior is required, an instance of a

custom adaptor class may be interposed between a source and one or more listeners in order to establish the relationship or to augment behavior [73].

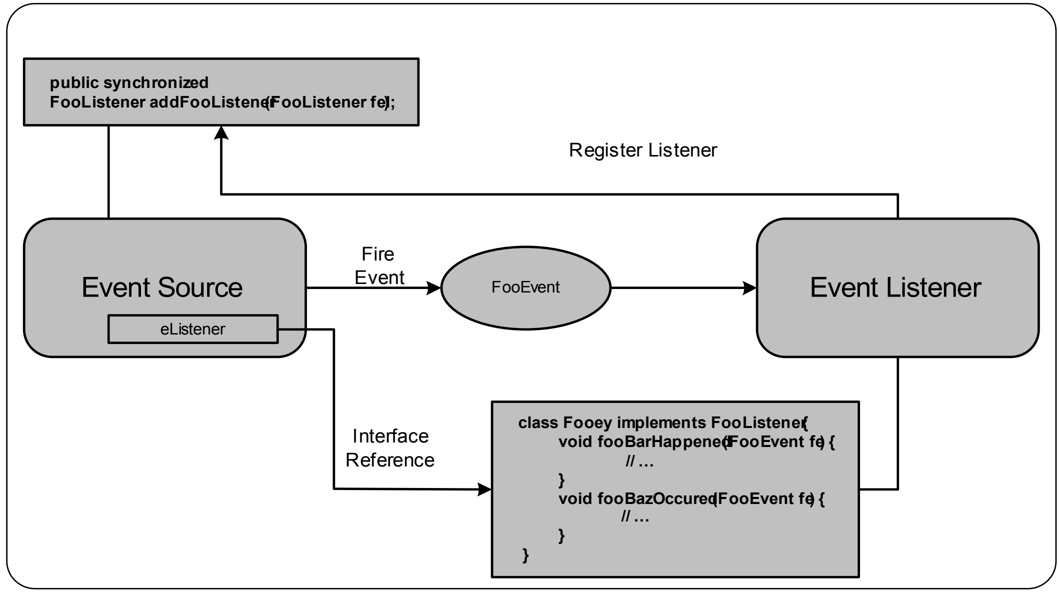

Figure 9 Overview of the Event Model in JavaBeans (adapted from [73])

II.4.1.2. Introspection

Introspection is a mechanism used for examining arbitrary beans to analyze and specify properties, events, and methods of it. This mechanism is proposed to avoid using a separate specification language for defining the behavior of a JavaBean. The key goal of JavaBeans is to make it very easy to write simple components and to provide default implementations for most common tasks. Therefore, any kind of an environment can be able to introspect simple beans without requiring the beans developer to do an extra work. However, for more sophisticated components, it allows the component developers' full and precise control over which properties, events, or methods are exposed. Consequently in

both ways, a composite mechanism is provided. By default, a low level reflection mechanism is used to study the methods supported by a target bean. Then, simple design patterns can be applied to deduce from those methods what properties, events, and public methods are supported. However, if a bean developer chooses to provide a BeanInfo class describing a bean then this BeanInfo class will be used to programmatically discover the beans behavior. To allow application builders and other tools to analyze beans, an introspector class is provided that understands various design patterns and standard interfaces and provides a uniform way of introspecting on different beans. For any given bean instance it is expected that its introspection information to be immutable and not to vary in normal use. However if a bean is updated with a new improved set of class files, then of course its signatures may change [73].

As a summary, JavaBeans uses a two-level approach for introspection mainly automatic analysis using reflection and explicit specification using BeanInfo interface. Reflection provides information about the fields, methods and constructors of a bean. BeanInfo interface provides information about the description of a bean, the icons for representing a bean in design tools, toolbars, properties of a bean, the events fired by a bean and the methods supported by a bean and so on.

II.4.1.3. JavaBeans Packaging

JavaBeans are packaged and delivered in JAR files. JDK1.1 supports JAR files to collect JavaBeans, class files, serialized objects, images, help files and similar resource files. A JAR file is a ZIP format archive file that may optionally have a manifest record file with additional information describing the contents of the JAR file. All JAR files containing beans must have a manifest describing the beans. A single JAR file may contain more than one JavaBean; this simplifies packaging and allows for sharing of classes and resource files at packaging time

[73]. Classes in a JAR file may be signed by using third party verification methods. This may reward the JavaBeans with additional privileges.

CHAPTER III

COSE APPROACH

"Systems are best viewed as flexible compositions of software components designed to work together as part of a component framework that formalizes a class of applications with a common software architecture. "

T.D. Meijler and O. Nierstrasz, [16]

According to [44], we are in the midst of a paradigm shift toward component-oriented software development, and significant progress has been made in understanding and harnessing this new paradigm. However, as a result of the recent assault of both commercial and academic studies, the maturement of the paradigm is not far ahead of us. This section summarizes Component Oriented Software Engineering (COSE) as component oriented software development approach, and previous studies performed on this topic.

III.1. Previous Research in the Area

"*Methodologies have two parts: modeling formalisms and a detailed process prescribing how and when to use the models. The models represent the three dimensions of software systems - namely data, function, and structure - and sometimes a fourth dimension, control. Traditional approaches (also called structured) have disjoint graphical models for different system cross sections based on data, function, and structure. Object-oriented approaches do not represent these dimensions in isolation. Any approach must address all three either directly or indirectly. Commonly, traditional methodologies arrive at functional system decomposition as the essential model; whereas object oriented methodologies*

decompose the system with respect to data, with a slight hint about structure. A component-based methodology must be oriented toward structure more than any other dimension." [72]

Figure 10 shows the modeling emphasis on different dimensions, for different approaches. Theoretically, Data, Function, and Control are the important dimensions that should be addressed during design. All together those aspects constitute a Turing Machine equivalent capability. However, starting with the traditional approaches, the structure dimension become important. The only purpose is to help the human designers with their understanding of a complex model. Often it manifests itself as a tool to define part/whole relations in an effort to meaningfully decompose a big model.

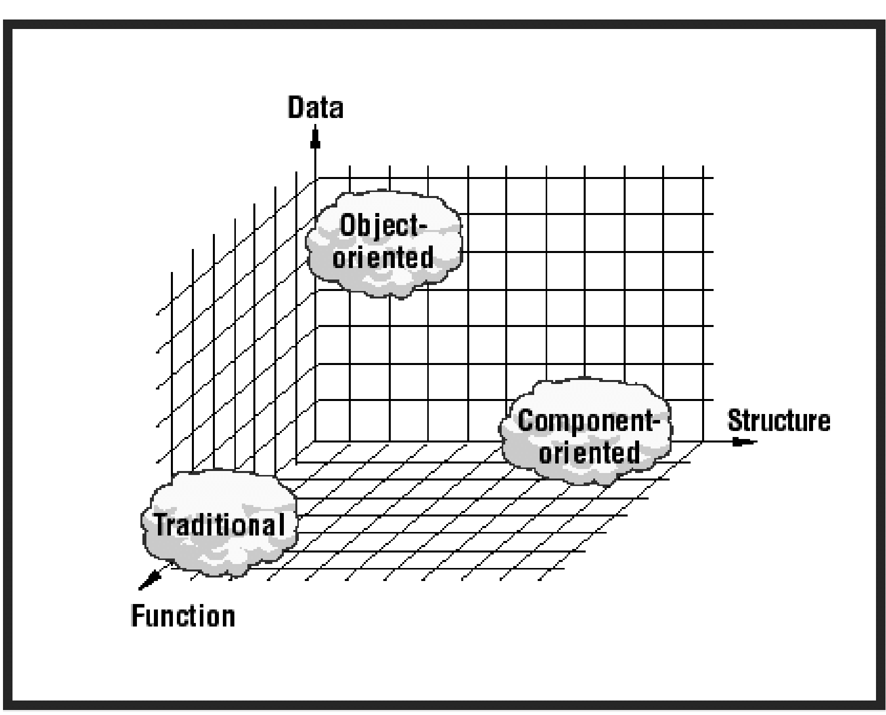

Figure 10 Modeling Emphasis for Different Approaches (adapted from [72])

Information hiding suggests that the fields that are wished to be encapsulated within a component should be recognized first, and then an abstract interface should be provided to define the component's boundaries. However, the decision of what to hide inside a component may depend on the result of some top-down design activity. It has been proven that information hiding is highly effective in supporting design for reusable components. Hence, the bottom-up design activity should proceed in a consistent way [13, 49].

COSE approach suggests the decomposing of the system structure hierarchically (top-down) while estimating the integration of component technologies as a bottom-up process [13, 49]. The reason for such a composite pattern is the difficulty of identifying the components of the system in the early phases of design.

III.2. Defining the COSE Modelling Language

During the decomposition of the system, COSE converts the system specifications into hierarchically grouped entities that are composed of a set of components and a set of connectors. Entities in the connectors group are used as a glue for connecting the entities in the component group in order to build-up the target system. A pair of interfaces at component level is represented by a connector at abstraction level. In COSE approach, there is a recursive component understanding without a container or super-component differentiation. COSEML is designed as the primary modeling language for the COSE approach [13].

As described in the previous section, a hybrid system decomposition method is used in COSE approach. Accordingly, a top-down decomposition method is applied in the initial phases of the modeling process in order to bring out the building blocks of the system into open. While the small grained low level blocks are being identified, interfaces between those items are defined. During the design, a temporary bottom-up approach is applied when a module is expected to match a component. If the granularity of a component covers more than one abstraction

functionalities, it can be decomposed into sub-components. As a final product, the super-component is imported into the system. This ends the temporary bottom-up task.

Abstract components correspond to Abstract Design Paradigm (ADP) elements namely “Packages”, “Data”, “Function”, and “Control” abstractions. “Packages” are represented by Unified Modeling Language’s (UML) package symbol. Internals of the “Packages” are represented by the “Data”, “Function”, and “Control” abstractions. A “Function” abstraction, like UML’s Use Case symbol, is represented by an oval. For the implementation-level components, events and behavioral descriptions need to be appended to the “Properties” and “Methods” [14]. On the other hand, there exist various links for connecting abstractions (components), namely “Composition”, “Inheritance” and “Connector” links. Basic symbols used in COSEML can be seen in Figure 11.

Since the model can have both abstraction and corresponding component levels, a system function could be represented in different levels. Thus, a connector can be represented between two abstractions, as well as between two components [14]. Detailed descriptions of the symbols used by COSEML are given in Table 3.

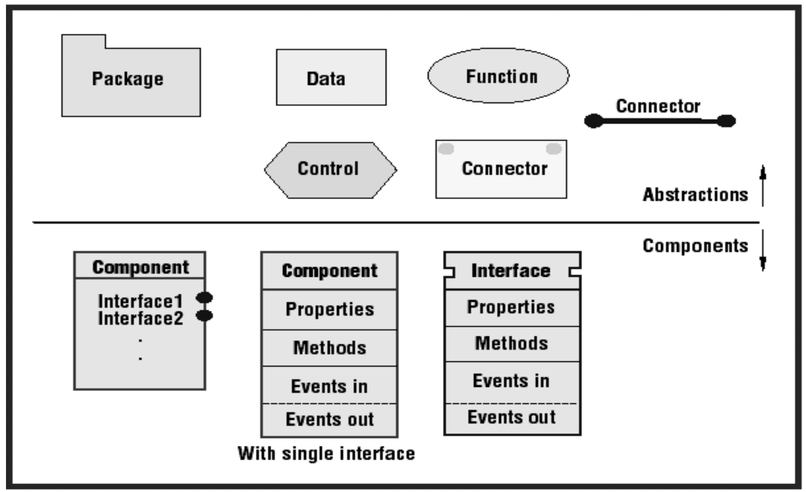

Figure 11 Graphical Symbols in COSEML (adapted from [72])

Table 3 Details of the Graphical Symbols used by COSEML (adapted from [42])

Symbol	Explanation
	Package is for organizing the part-whole relations. A container that wraps system level entities and functions etc. at a decomposition node. Can contain further Package, Data, Function, and Control elements. Also can own one port of one or more Connectors. Can be represented by a Component. The contained elements are within the scope of a package: they do not need connectors for communication.
	Function represents a System-level function. Can contain further Function, Data, and Package elements. Can own Connector ports. Can be represented by a Component.
	Data represents a System-level entity. Can contain further Data, Function, and Package Elements. Can own Connector ports. Have its internal operations. Can be represented by a Component.
	Control corresponds to a state machine within a package. Meant for managing the event traffic at the package boundary, to affect the state transitions, as well as triggering other events. Can be represented by a Component.
	Connector represents data and control flows across the system modules. Cannot be contained in one module because different modules will use its two ports. Ports correspond to interfaces at components level.
	A Component corresponds to existing implemented component codes. Contains one or more interfaces. Can contain other Components. Can represent one abstraction (Package, Data, Function, or Control)
	An Interface is the connection point of a component. Services requested from a component have to be invoked through this interface. A port on a connector plugs into an interface
	A Represents relation indicates that an abstraction will be implemented by a component
	An event link is connected between the output event of one interface and the input event of another. The destination end can have arrows corresponding to the synchronization type.
	A Method link is connected between two interfaces to represent a method call. Arrow indicates message direction.
	UML class diagram relations can be utilized. Diamond: composition, Triangle: inheritance.

A small business automation system design can be given as an example for the systems that are designed by using COSEML. Modeling diagrams can be seen in Figure 12. The system functions include:

- Sales,
- Accounting,
- Personnel,
- Clients, and
- Inventory.

This design diagram includes both the abstract level design and the detailed level design. Accordingly, the high level abstract connectors can be traced in the upper part of the diagram, as seen in between “Inventory” and “Sale” packages. As a consequence of specialization of the design items, components and the specific connectors between them can be observed in the lower parts of the diagram, as seen in between the interfaces of the “inventory” and “sale” components.

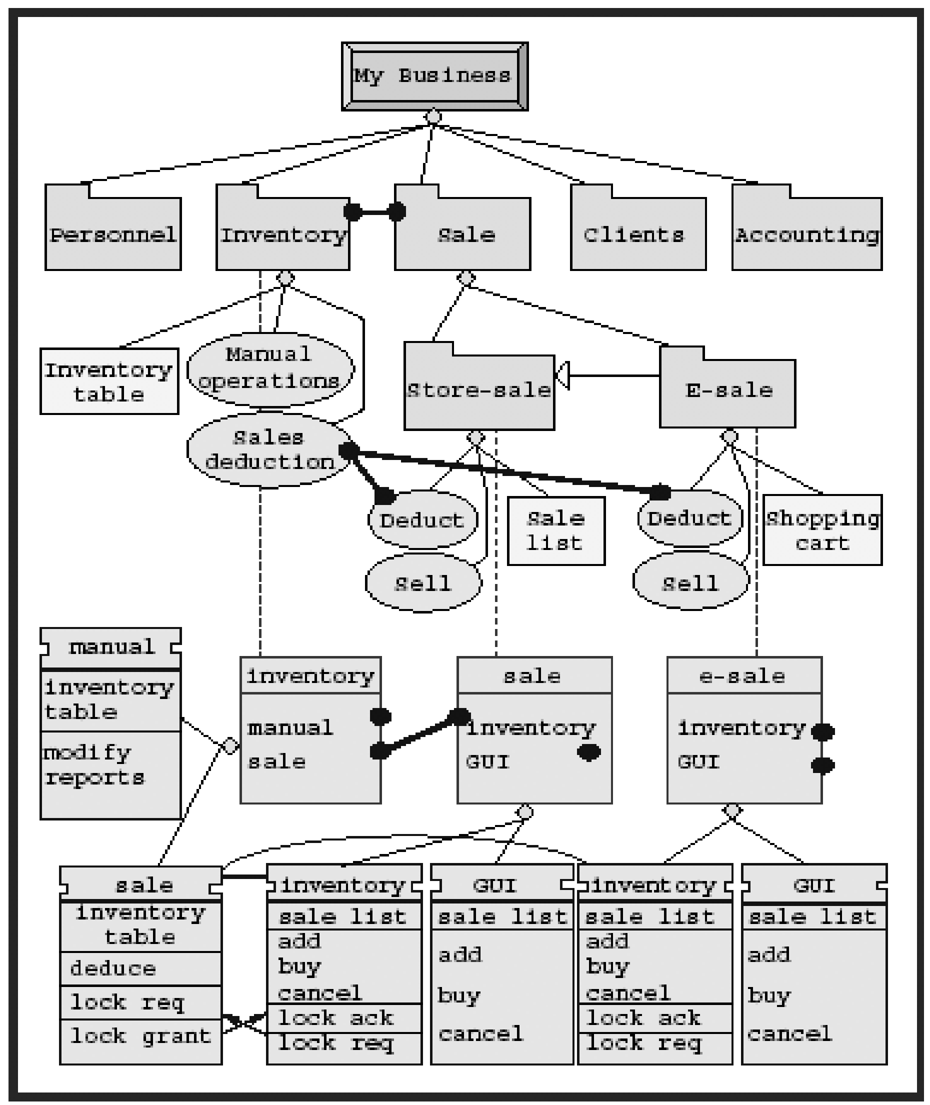

Figure 12 Modeling of a Small Business Automation System with COSEML (adapted from [72])

III.3. COSE Process Model

A COSE-based Process Model is proposed by [14] within the scope of a previous thesis study. In this section, a brief summary of the model is introduced.

According to [14], COSE process model consists of four main developmental phases and a system test phase. Developmental phases are as follows:

- System specification,
- System decomposition,
- Component specification, and
- Integration.

A general representation of the COSE Process Model can be seen in Figure 13. Each rounded rectangle represents a high level process within the specified phase. Arrow directions show process and information flow. Documents that are connected to the processes show information flow depending on the direction of dashed arrows [14].

The starting activity of the COSE Process Model is system specification. The system is quantified and clearly understood, high-level system requirements are documented in this activity. Additionally, a search for existing components is performed in system specification phase. System is analyzed and decomposed in the system decomposition phase. Moreover, functional requirements are detailed and required components and their specifications are stated.

The next step in the flow is the component specification phase. Components that are going to implement the system are specified and developed in this phase, either by using existing components or developing new components [14]. Also, this phase includes the search for existing components and evaluation of located components. If any further decomposition is needed, then system is decomposed more until there is no further meaningful decomposition. After the evaluation of the existing components, the module that does not have any corresponding component needs to be implemented.

After component acquisition activities, integration and testing activities take place. Within the integration phase, components of the system are united for building the expected system.

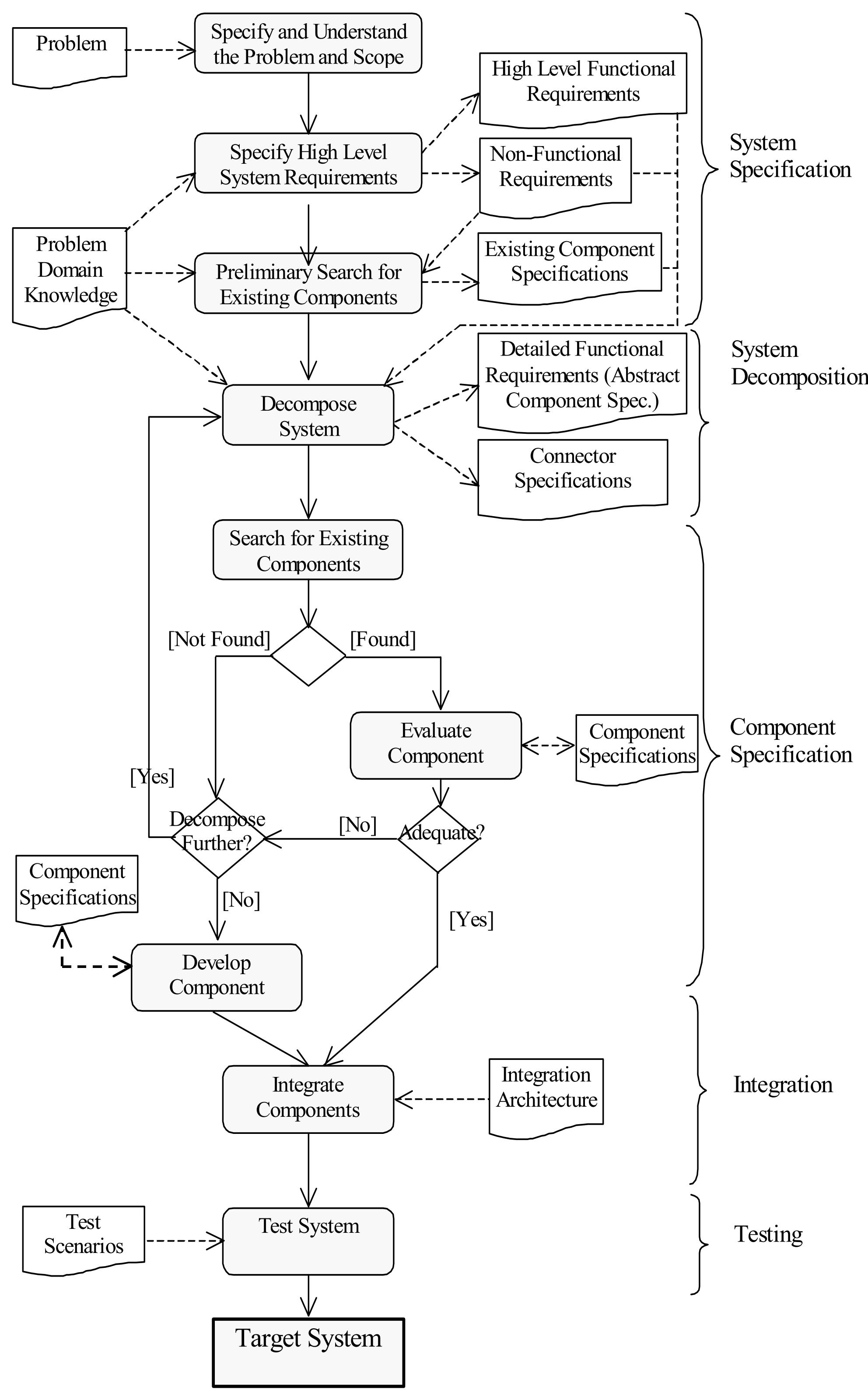

Figure 13 COSE Process Model

CHAPTER IV

VISUAL COMPOSITION

Composition is the act of applying a composition operator that forms part of a composition model and theory in a given context. Components are the subjects of composition [48].

Visual composition is defined as the interactive construction of running applications by the direct manipulation and interconnection of visually presented software components in [31]. The connections between components are governed by a set of plug-compatibility rules specified within a composition model. In another point of view, it is the interactive construction of applications from pre-packaged, plug-compatible software components by direct manipulation and graphical editing. This approach allows both the internal behavior of applications, as well as their user interfaces, to be directly edited and manipulated.

Visual composition is a response to the trends in software development towards more component-oriented lifecycles. With a large number of components supplied by component engineers, application development becomes an activity of composing components into running applications. Visual composition can be used to communicate reusable assets from component engineers to application developers, reusable designs to application developers, and open applications to end-users. A visual composition framework enables the construction of environments and tools to facilitate component-oriented software development [31].

Component integration frameworks are becoming increasingly important for commercial software systems. The purpose of a component integration framework is to prescribe a standard architectural design that permits flexible composition of third-party components. Usually a framework defines three concepts [69]:

- the overall structure of an application in terms of its major types of constituent components;
- a set of interface standards that describe what capabilities are required of those components; and
- reusable infrastructure that supports the integration of those components through shared services and communication channels.

A successful framework greatly simplifies the development of complex systems. By providing rules for component integration, many of the general problems of component mismatch do not arise [70]. By providing a component integration platform for third-party software, application developers can build new applications using a rich supply of existing parts. By providing a reusable infrastructure, the framework substantially reduces the amount of custom code that must be written to support communication between those parts [69].

In the first subsection of this chapter, previously proposed component integration approaches are described in detail. In the second subsection, COSEML approach to the visual composition is expressed by means of various examples.

IV.1. Previously Proposed Component Integration Approaches

This section summarizes the previously proposed component integration approaches by means of integration tools supported by frameworks and programming languages. There exist various proposed tools and languages for the purpose of component integration.

One of the most serious projects performed on creation of a generic development environment that can be easily adapted to various application domains is the ITHACA project [18, 19, and 31]. ITHACA is a Technology Integration Project in the Office & Business section of the European Community's Esprit II Programme. The goal of ITHACA is to produce a complete object-oriented

application development environment that can be easily adapted to various application domains [41]. The environment includes workbenches that constitute the software components and tools specific to a particular application domain. The tool that is related with component integration is the Visual Scripting Tool (Vista). Vista is based on an object oriented framework for visual composition that supports open system development through the notion of domain-specific composition models [31, 32, and 34].

According to the definition in [18], visual scripting is the interactive construction of applications from pre-packaged, plug-compatible software components by direct manipulation and graphical editing. Scriptable software components are characterized by the following properties: [18]

1. Scripting Interface: every component has a set of output ports where it makes available services that it provides, and a set of input ports where it receives connections to services it requires;

2. Visual Presentation: direct manipulation is supported by providing every component and each of its ports with some editable visualization;

3. Run-time Behavior: a component, when all its input ports are connected, provides some set of services to its clients.

A script is a set of software components with compatible input and output ports connected. The types of ports defined for a set of components and the rules that determine plug-compatibility constitute what is called a scripting model. This very general scheme can accommodate a variety of programming and composition models [18, 19].

A scripting tool can guarantee that this connection only be made between plug-compatible fields. In this case, presumably, the view field expects the component to which it is connected to have operations that allow its value to be retrieved and to be updated. Note, however, that the interpretation of the script is

strictly limited to managing the connections. It is the business of the component itself to decide when and how to make use of the connection. A scripting tool, then, functions like a corporate lawyer setting up standard contracts between standard kinds of customers and clients. It oversees and authorizes the signing of the contracts, but it is not concerned with their execution [19].

Nierstrasz claims that by the scripting functionalities of Vista, the ability to encapsulate a script as a component (SAC) makes it possible to develop higher-level abstractions through scripting rather than by programming. To define a SAC, one must provide script with a scripting interface (by specifying which ports of the script are become ports of SAC), and a visual presentation (which can be composed of existing presentation components). But this scripting manner reduces the component orientation properties of components, as covering the pre-component object based units with a scripting language. Another restriction on new components introduced to Vista is that they conform to the rules of a predefined scripting model. Internal representation of a script defined by Vista is a graph in which the nodes of it represent components and the arcs represent links. This basic graph structure has no inherent syntax or semantics. Consequently, there is no framework support for the design activities in Vista.

In the Vista, the definitions of the components are pre-defined and non-editable. Additionally, the composition concept is realized by data flow diagrams that does not cover a system design paradigm. Moreover, there is a wide variety of topics still left open in this first version of the Vista tool which the research group plans to investigate. These topics include: a better way of adding new components, more support for components written in different programming languages, the general treatment of scripting models and generating code from a script [19].

In another point of view, a language called ACOEL (A Component-Oriented Extension Language) for abstracting and composing software components is proposed. Components in ACOEL are black-box components, and each component should consists of

- an internal implementation which contains classes, methods, and fields that is hidden to the external world, and
- an external contract which consists of a set of typed input and output ports.

Components in ACOEL interact with each other only by means of these ports [58]. Components need to be implemented by using ACOEL syntax and input and output ports should be defined by using these implementation details. By this approach, component generation is tried to be standardized, but it does not state any language free integration mechanism.

Another integration approach that also uses the scripting mechanism is the TIFRAME which is an object–oriented framework based on the introduced component model and intended for the control integration layer in tool environments. The framework consists of two parts: the CORBA IDL-like configuration language, Tool Interconnection Language (TIL), for describing components and their interconnections and the runtime environment for instantiating configurations described in TIL. In order to adjust the tool environment to specific tasks it should be possible for the user to describe the configuration on an abstract, implementation–independent level [2].

From the TIL specification the compiler generates an internal representation of a configuration readable by the runtime environment and the adapter code required for interconnecting the components. The compiler checks the specified interaction relations for interface compatibility. Based on the configuration description, the runtime environments are able to instantiate and connect the components directly or via the generated adapters [2].

In the scripting language approaches, by decoupling scripting models from the underlying programming language, a variety of different software composition paradigms can be supported. But this defined prerequisites forces the generation of the script interface for each component like object groups. Consequently, this

approach does not satisfy the interfacing constraint of the component by re-defining auxiliary interfaces for making the components compatible for such tools.

There have been efforts spent on creating tools for the purpose of integration. One of such tools is Fabrik [37] which is a visual programming environment that exploits bidirectional dataflow. The goal of Fabrik was to facilitate the kit approach to programming by taking advantage of emerging technologies such as iconic user interfaces. With kits, fixed rules govern the composition of kit components, thus restricting their possible reusability and the flexibility of application generation. Visual scripting, with scripting models, tries to factor out such restrictions to allow the creation of application templates that can be tailored to particular application domains [19].

Another integration tools is HyperCard [38] which is an authoring tool and information organizer based on the concept of stacks of information. HyperCard provides a fixed set of five components that can be configured interactively or through a high level programming/scripting language called HyperTalk. HyperCard is self-contained, no components can be added and there is no distinction between the tool environment and the application. In contrast, visual scripting supports the composition of components.

Interface Builder [39], one of the integration tools, allows an application designer to define an application's interface graphically and to connect user interface objects to underlying application objects which have been programmed separately. Interface Builder was not intended to be an application generating tool, but it is gradually becoming more versatile in its functionality. Interface Builder, in contrast to visual scripting, does not support hierarchical design, that is, the encapsulation of scripts as components to be reused in future development.

In another lane of software engineering, various programming languages are trying to propose an integration methodology that can be used for the integration of

components, either using existing object oriented languages or creating their special languages from scratch.

One of those languages is Lagoona [44] which aims to solve the component composition in an object oriented programming level. According to the proposal in Lagoona, it is tried to approximate certain component oriented programming ideas using a variety of essentially object oriented programming languages such as C++, Eiffel, and Java. In this approach, to increase the benefit of reusability of object oriented artifacts in component orientation, an integration bridge between component oriented and object oriented approaches is attempted to be constructed, consequently it is not a fully component oriented approach. Other approaches that have the same point of view are described in [45] and [54]. The aim of the approach described in [54] is combining the object oriented development with component based approach. According to their manner, if the OO design and OO implementation is collided with component quality assurance and adaptation techniques, the approach becomes component oriented. Unfortunately, the CO methodologies are not meant to be representable with OO methodologies and development approaches.

One of the programming languages which is proposed to define the specification of software components and integration details is the II-language [20, 42]. In this approach a software system is given by defining the connections between them. In this manner, this language is nothing but the component based integration approach that does not propose anything more than other well known component integration languages. Beside on II-language, there exist other architecture description languages, such as CDL.DCUP [26], Darwin [27, 76], Rapide [28], Wright [29], and C2 [30]. Some of these description languages are also used as a basis for automatically integration of components. But these approaches drag the components under the sovereignty of a programming language.

There exist other approaches that try to discover the component phenomenon through object oriented perspectives: For instance, in the study described in [21], it

is aimed to use object oriented languages for taping one component to another. As another example, object oriented modeling languages are attempted to concoct with component integration methodologies in [22]. Another approach to component integration is described in [45]. In this approach, a generic meta-level framework for modeling both object and component-oriented programming abstractions is presented. In this framework, various features, which are typically merged in traditional object-oriented programming languages, are all replaced by a single concept: the composition of forms. Forms are first-class, immutable, extensible records that allow for the specification of compositional abstractions in a language-neutral and robust way. Thus, using the meta-level framework, we can define a compositional model that provides the means both to bridge between different object models and to incorporate existing software artifacts into a unified composition system [45].

In a broader manner, there exist different studies on connecting distributed components. The approach to component integration for distributed application software described in [56] includes an integration mechanism that performs compatibility check of a black box component with the target architecture based on both functional and non-functional requirements. Followings are the non-functional attributes for compatibility checks: [56]

- Types of protocols used in the architecture
- Types of security protocols used
- Types of fault tolerance
- Types of event handling
- Types of exception handling
- Amount and types of resources used

In this approach the Distributed Component Architecture (DCA) [41] is used as the common underlying environment to integrate and use distributed software components. Additionally, an object-oriented distributed component framework is used to facilitate compatibility checks of components with the target architecture during integration time. Concerning the communication aspect, an object-oriented model of distributed connectors is used for connecting components. DCOM is used as the underlying DCA environment to implement the distributed component framework and the distributed connector. The framework is implemented as a COM component. Architectural reflection is implemented through the COM's dynamic interface discovery and invocation mechanism (IDispatch) and maintaining a data structure inside the framework that holds references to the subcomponents, sub-connectors, and their interconnection information. The sub-components are implemented as out-process objects, although both in-process and remote components can be used seamlessly with the framework. The connector is also implemented as a COM component [56].

The entire integration process used in [56] can be summarized as follows:

1. Specify the target architecture of the distributed application software as a model adopted from I/O Automata as described in [36].

2. Select the candidate components from repository based on desired functionality.

3. Perform non-functional compatibility checks on the components from Step 2, which are based on the component framework by supplying a part of the automaton from Step-1.

4. Identify the compatible components based on Step-2 and Step-3.

5. Customize the components from Step-4 that are not completely compatible based on the results in Step-3.

6. Visually integrate the components from Step-4 and Step-5 using distributed connectors to generate the distributed application software.

As it is described in [23], the most important role in these process activities belongs to the integration tool. The tool developed to support this activity visualizes distributed components, generates adapters for resolving parameter mismatches and provides fault tolerance. The following activities describe the integration mechanism used by the component integration tool to integrate two components.

1. The integration tool retrieves a reference to the role object from the corresponding distributed component using its IRole interface which is described in [56] in detail. To accomplish this, a role object should implement a DCA-compatible interface.

2. The reference from Step-1 is passed to the location of the component through the underlying DCA.

3. The component uses its IManage interface which is also described in [56] in detail to store the reference to the role from Step-2.

4. A reference to the IService interface -described in [56] - of the component is passed to the distributed component using the same mechanism described in Step -2.

5. During runtime, the distributed component uses the reference to the component from Step-4 to forward any method invocation to the component. Similarly, the component uses the reference to the role from Step-2 to dispatch its outgoing events.

After the procedure is applied, each component in the application software uses its assigned role object to communicate with other components. The integrated components produce an effect as if the interaction protocol were already built into each component [56].

Another approach for integrating distributed black-box components which is presented within the context of Integration Rules (IRules) project is described in [55]. In this approach, a general-purpose, distributed rule processing architecture for integration of black-box components is proposed.

Another interesting approach for the integration of the components is proposed for the purpose of web-based data analysis in [57]. In this study, the architecture of an intelligent component integration system that supports the component based development of scientific data analysis applications is described. The system relies on formal specifications to describe as the components, wrappers, and architectures and uses automated reasoning techniques to retrieve, adapt, and integrate components intelligently [57].The architecture presented in this approach is a component based approach. Additionally, for the integration of the components, descriptors of the components, namely integrator, should be implemented.

In another proposal, a Component Oriented Web Application Model (CoOWA) is proposed [59]. Within this model, a web application is regarded as a collection of components which has its own functionality and communication with each other through their interfaces. This approach aims to formalize the web applications as a group of components. By the support of that idea, web applications can benefit from the concept of component oriented development. The following figure describes the idea behind this approach.

The usage of component orientation methodologies is getting wider. Day by day, this concept is applied to various areas in software engineering. In [60], component orientation approach is used for embedded software development, and in [61], is applied to integration of simulation components. Additionally, new architectures are under development for real-time component oriented development.

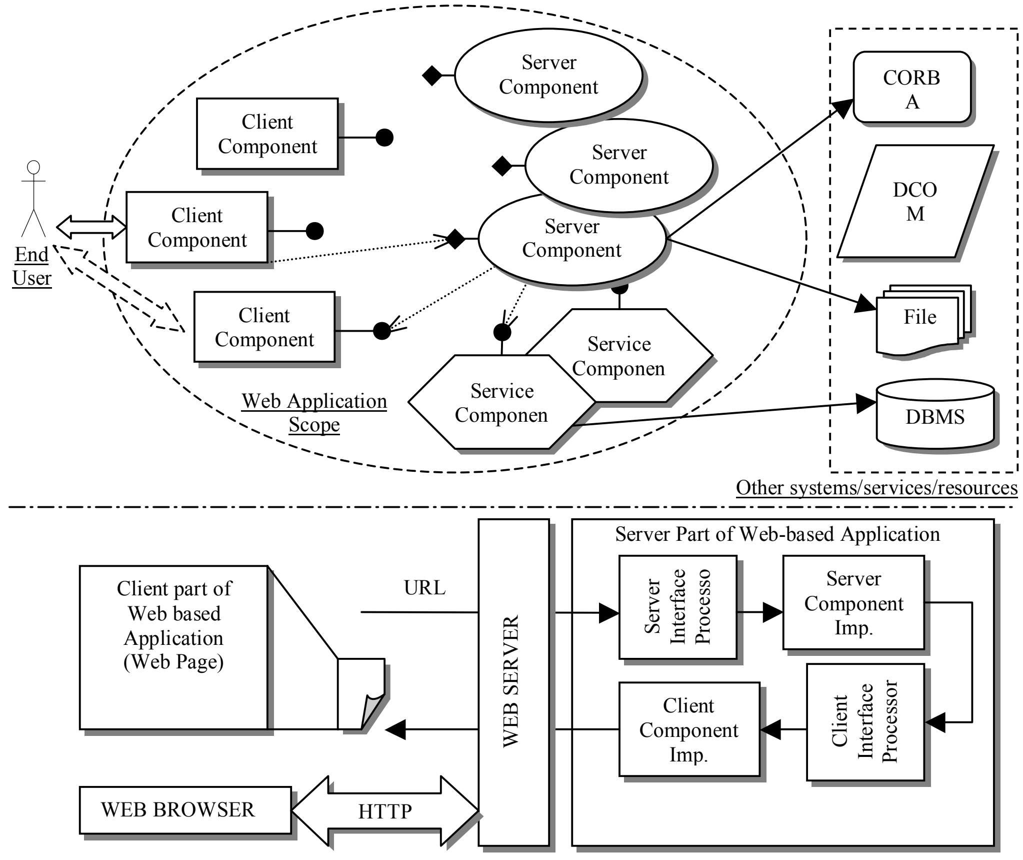

Figure 14 CoOWA Diagram [adapted from [59]]

To support the development activities, component based testing methodologies are also proposed in the literature [64]. In addition to development related studies, there exist contemplations on component oriented requirement analysis approaches [63].

As an additional footnote, various design patterns are proposed for defining common solutions to general problems in component orientations. Some of those patterns are as follows:

- DYNAMIC FACTORY [46], a creational pattern which allows for the creation of late-bound components.
- AGGREGATION [46] and the EMBEDDING [46] pattern. These two structural patterns are archetypal for component-oriented programming.

- PROPAGATOR [47] as an important pattern for message propagation is illustrated. Flexible message propagation, expressed in delegation and forwarding, is a key aspect of component-oriented programming.

Each of these approaches handles significant problems in the application development community but they are all limited to describing particular instances of applications. Whereas COSEML, with the support of COSECASE can enable the development of generic applications by using the JavaBeans integration module in a wide variety of range. The proceeding sections describe this mechanism.

IV.2. Visual Composition in Component Oriented Development

As seen in the previous chapter, the previous studies on component oriented development do not introduce a well-defined modeling media usage. This gap is identified and a new component oriented development with the usage of COSEML as the modeling media is introduced to the literature. In this thesis, visual composition of COSEML is introduced by means of integrating JavaBeans with COSECASE. In the next sections, first the reasons for choosing the JavaBeans as the component framework are listed, after the visual composition concept in COSEML is explained, then the instantiation of visual composition in COSEML is underlined and lastly the JavaBeans integration to COSEML is described in detail.

IV.2.1. Component Framework Selection

In order to support the visual composition in COSEML, JavaBeans component framework is selected. The reasons for this selection are as follows:

- The main reason for selecting JavaBeans is that, the version of COSECASE that is used for this thesis study is implemented in Java programming languages. To make the integration of component framework with the system design environment easier, JavaBeans component framework is selected.

- One of the other reasons is that the JavaBeans architecture provides platform neutral component architecture.

- JavaBeans is also designed to work with component models through software bridges [17]. As an example, on the Microsoft platforms, the JavaBeans APIs can be bridged with COM and ActiveX. Similarly, it will be possible to treat a bean as an OpenDoc Live Object part, or to integrate a bean with LiveConnect inside Netscape Navigator. So a single Bean should be capable of running in a wide range of different environments. Within each target environment it should be able to fire events, invoke service methods just like any other component. This makes the integration of other component frameworks with COSECASE more suitable by considering JavaBeans framework as a model.

- By using JavaBeans framework, it is very easy to develop simple components for testing issues.

- Event Model in JavaBeans framework is considerably appropriate for the COSEML integration concept. This concept was introduced in the proceeding sections.

- By the usage of the customization mechanism in JavaBeans framework, the features of a JavaBeans can be altered at runtime.

- Persistence mechanism allows the changed JavaBeans to be saved and reloaded at runtime.

- The three important features of a JavaBean are the set of properties it exposes, the set of methods it allows other components to call, and the set of events it fires [73]. At runtime, it is possible to figure out which properties, events, and methods a JavaBean supports. This functionality is called introspection as it was described in the previous sections.

Through introspection, it is very easy to deal with the JavaBeans at runtime.

IV.2.2. Visual Composition in COSEML

> *Visual formalisms are important for specifying and representing software composition because they can support multiple views of the same structures, they can provide important visual cues to aid understanding, and because they can directly represent the final application interface, and hence can conveniently support a direct manipulation paradigm during development. Framework developers must be able to specify and represent generic architectures, components, component interfaces and glue using a high-level graphical "syntax." Application developers should be able to instantiate architectures by elaborating, binding and linking framework components. Abstractions must be available as explicitly manipulable (and visually represented) entities in the composition environment. Composition structures must be mappable to language sentences. The visual environment should support the user actively in creating and adapting compositions correctly. The environment should be homogeneous with respect to either object or class level composition.*
>
> T.D. Meijler and O. Nierstrasz, [34]

In previous studies, the graphical editor that supports the system design with COSEML has been implemented. COSECASE enables software designers to use all COSEML symbols during designing the systems. Thus, it enables the user to hierarchically decompose a system's requirements, and to view structural relations among components graphically [13]. User manual of the initial version of COSECASE is supplied in [13].

COSECASE enables users to decompose a system's requirements in two levels according to the details used in decomposition; first, abstraction level decomposition is performed in a single tree, and next, appropriate components and their interfaces are included in a component forest.

In this study, COSECASE is introduced to the visual composition with JavaBeans components. For the success of this aim, the integration should be accomplished in the conceptual basis. Component framework terminologies of JavaBeans are applied to the COSEML specifications.

In JavaBeans, events can be used as a simple communication mechanism to connect beans. JavaBeans Event Model which is used to connect the methods to events is applied as a basis for the compositional association between components. The Event Model in the JavaBeans component framework is re-described briefly in Figure 15.

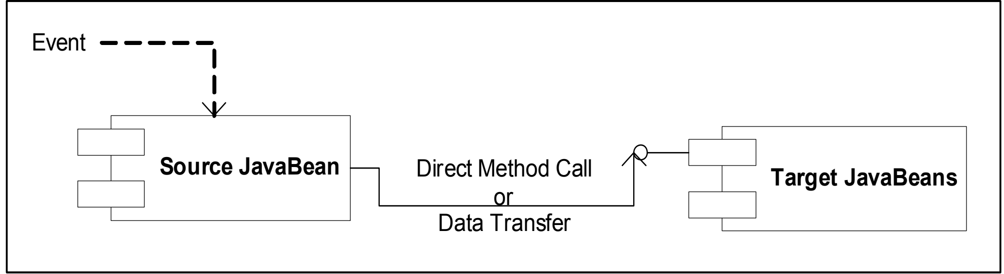

Figure 15 Event Model in JavaBeans Component Framework

According to this model, interaction between two JavaBeans in triggered by an event that has been initiated in the source bean. This event is conducted to the target bean by means of a method call that either can transfer data or solely forward the event. These communication strategies are described in Figure 16 and Figure 17.

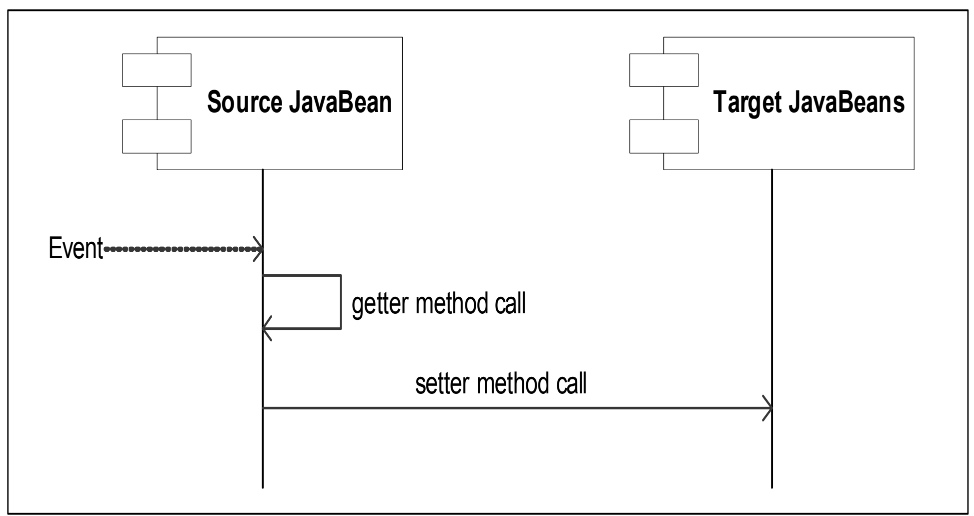

Figure 16 Event Conduction with Data Transferring

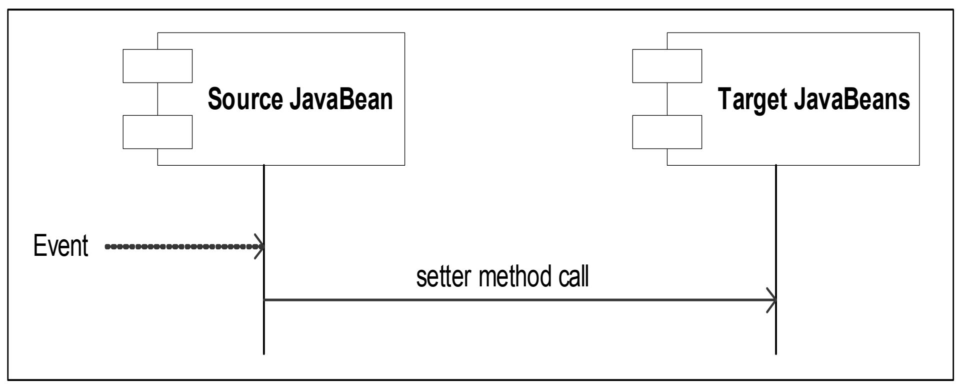

Figure 17 Event Conduction with Method Calls without Argument

COSEML conceptually supports the Event Model of the JavaBeans. As it is described earlier, connections between components are represented by the special connectors of COSEML -"Composition", "Inheritance" and "Connector". As it can be seen from the Figure 18, a connector between the system components describes the high level association between them. While the system decomposition progresses towards lower-levels, interfaces between the components that are

associated in the abstract level are defined. Association between components represented by the connectors in the abstract level are emphasized by more detailed low-level connectors – "Represents", "Event Link" and "Method Link". The sample usage of these connectors is displayed in Figure 19.

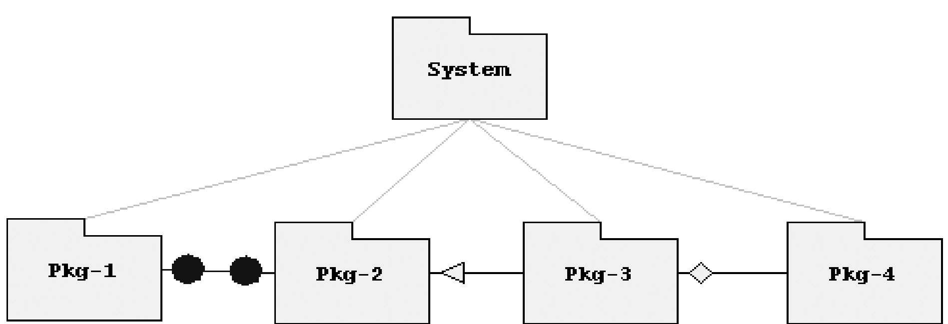

Figure 18 Abstract Level Connectors

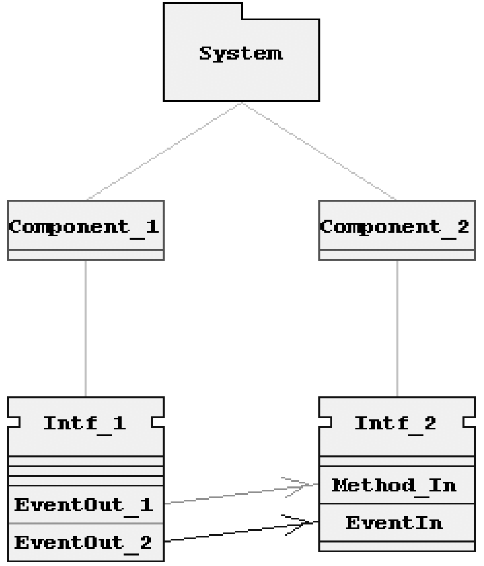

Figure 19 Method and Event Links

In the perspective of COSEML, Event Model of the JavaBeans is implemented as follows:

- Any association between two JavaBeans is represented by the "Connector" link between the two abstractions representing the beans.

- After direct manipulation of the "Connector" association, the abstract connection of these components is specialized by defining the roles of the components. The specialized roles are represented by the "Interfaces". Interfaces also represent the method(s) to be functionalized between two components.

- As a refinement of the abstract "Connector" link specified in the abstract level, "Event Link" connector is used to connect the "Interfaces". The mapping between abstract Event Model in JavaBeans and COSEML is represented in Figure 20.

In order to integrate the JavaBean components, they should be "pluggable". For the plugability concept in the COSEML, there are two "rules of thumb" proposed as follows:

1. JavaBeans can be associated in order to forward data from the source component to the target component in an occurrence of a specific event. This data forwarding is implemented by a getter method in the source component and a setter method in the target component. To be "pluggable", the return type of the getter method in the source component should be compatible with the argument of the setter method in the target component. This rule is called as "getter-setter pairing" within the scope of this thesis. This scenario is described in Figure 16.

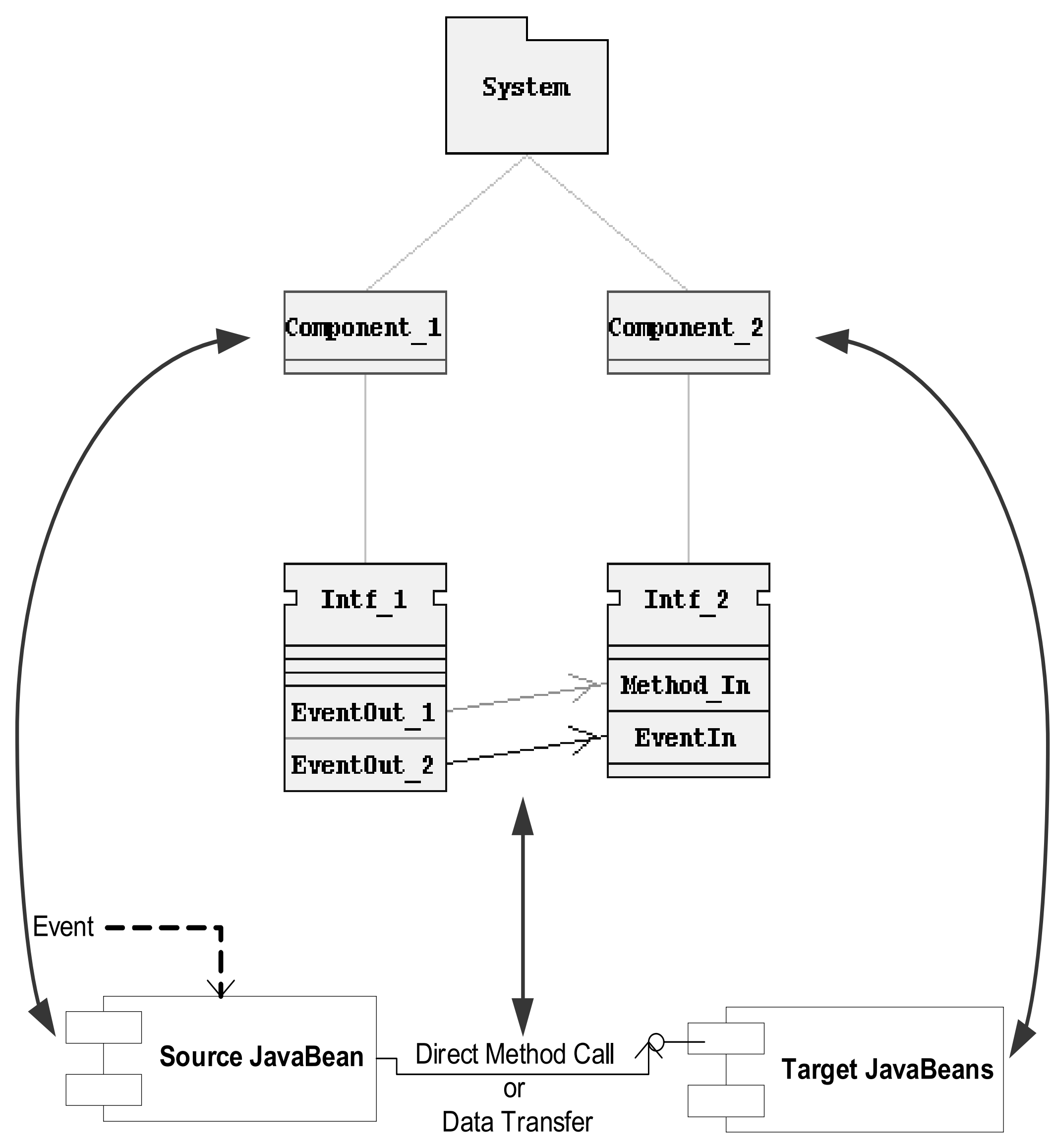

Figure 20 JavaBeans Event Model and COSEML Mapping

2. In case of no data forwarding, a specific event happening in the source component directly calls a method in the target component that does not have any argument. This rule is called as "event-method pairing" within the scope of this thesis. This scenario is described in Figure 17.

These rules are inferred from the design patterns that are applied to the introspection mechanism in the JavaBeans protocol.

IV.2.3. JavaBeans Integration in COSEML

In this thesis, a component in the COSEML can be associated with a JavaBean. After the component assignments, user can realize a connector specified in COSEML. During this realization process, the triggering events and the methods to be called are defined. When all the components are assigned and all the connectors are realized, the full system can be seen in the Bean Instantiation Window. In order to see the full system at work as a software product, Mode Change Button can be pressed as it can be seen in Figure 21.

COSECASE tool was developed before for supporting the COSE process model. JavaBeans activation interface is implemented to support the integration phase of this process model. General specifications of the COSECASE tool are described in [13]. Main window of COSECASE can be seen in Figure 22. For supporting the visual composition, additional features are included to the COSECASE. The main addition is the bean instantiation window that is displayed in Figure 23.

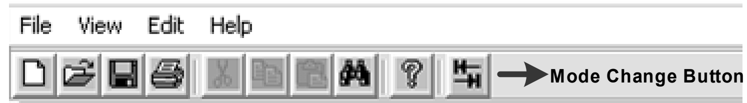

Figure 21 Mode Change Button

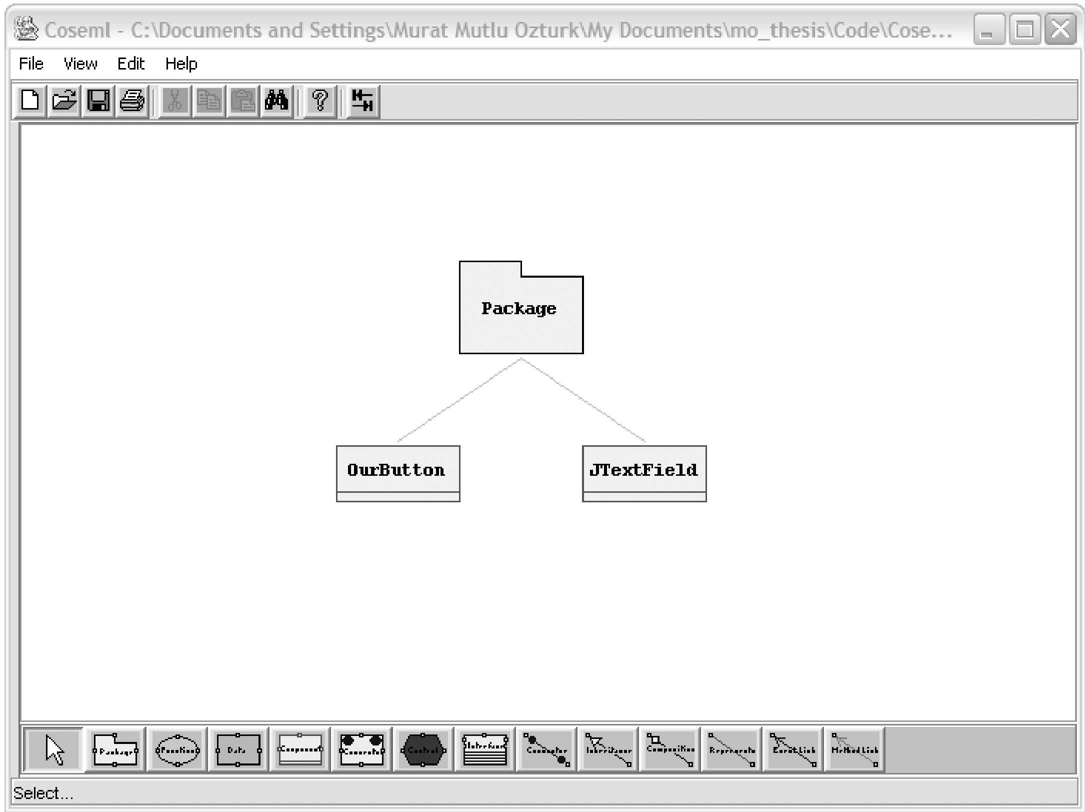

Figure 22 Main Window of COSECASE

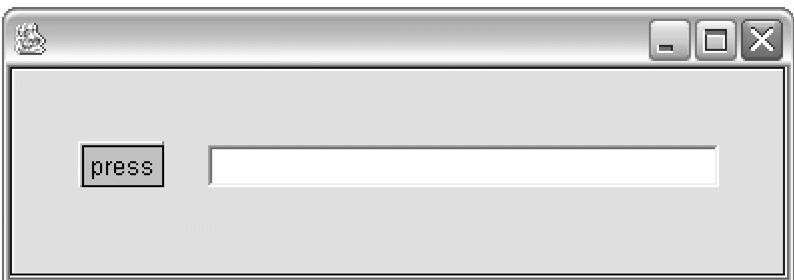

Figure 23 Bean Instantiation Window

Bean instantiation window has two modes: design and execution modes. The modes of the window can be changed by using the mode change button displayed in Figure 21. In the design mode, it is allowed to change the visual properties of the instantiated beans, like the size and position. The bean instantiation window in

design mode is displayed in Figure 24. In the execution mode, beans are executed according to the roles defined. In this mode, it is not allowed to change any property of the beans. The appearance of the bean instantiation window in execution mode can be seen in Figure 23.

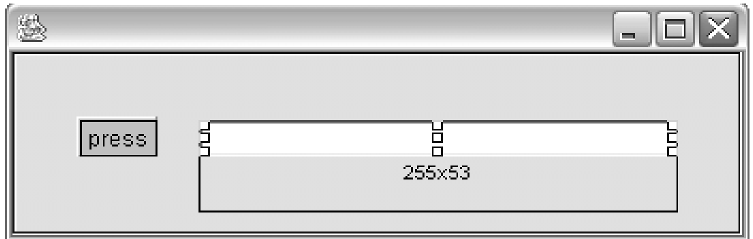

Figure 24 Bean Instantiation Window in Design Mode

The main task in integration by using COSECASE is deciding what role a component plays to collaborate with other components. The choice of roles is restricted to the types of connectors used in the abstractions and specializations.

Once it is decided, the corresponding components and hence the roles are integrated. The steps of the integration process for a system including two components are described as follows:

1. The system is decomposed into its components and the JavaBeans counterparts of the components are acquired as described in [13] and [14]. Figure 25 describes an initial system design with two components.

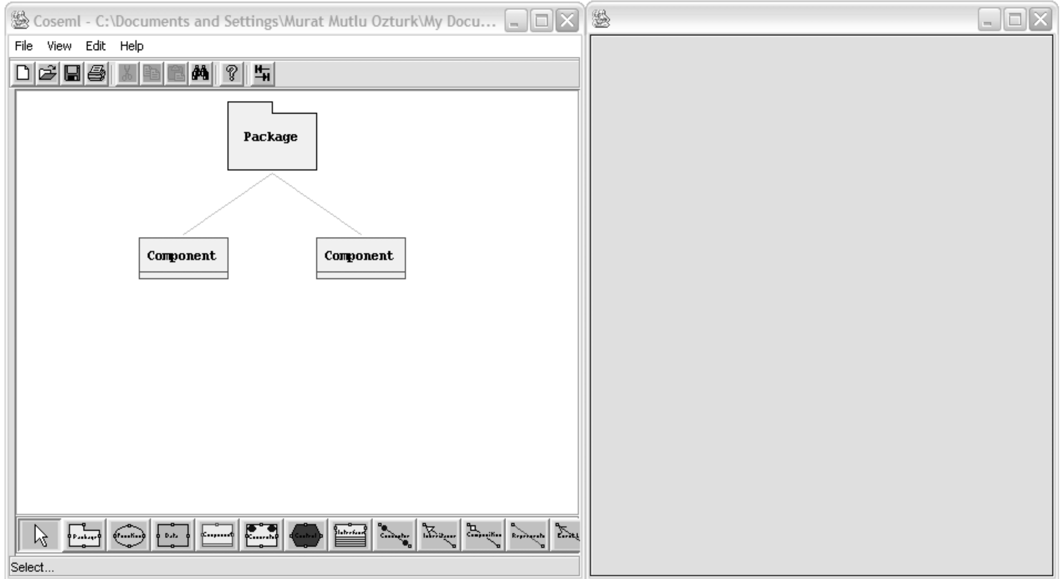

Figure 25 A System Design with Two Components

2. Each component defined in the system design is assigned to a JavaBeans that is in “jar” file format. The assignment process is described in Figure 26, Figure 27, Figure 28, and Figure 29.

3. Each assigned JavaBean is instantiated in bean instantiation window as it is seen in Figure 23.

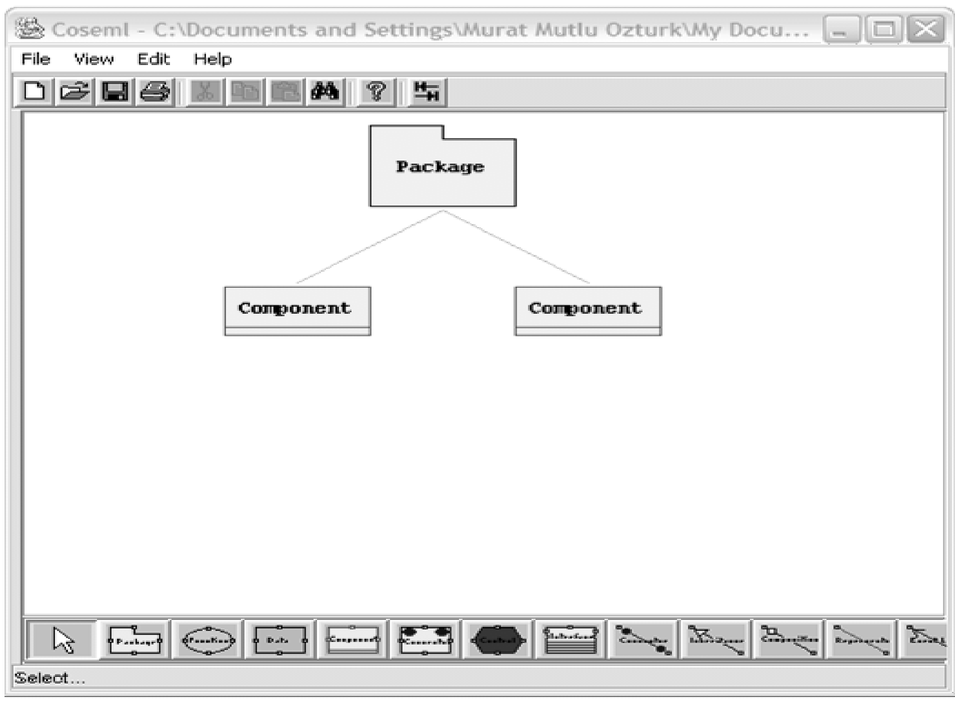

Figure 26 System Design before Assigning JavaBean Components

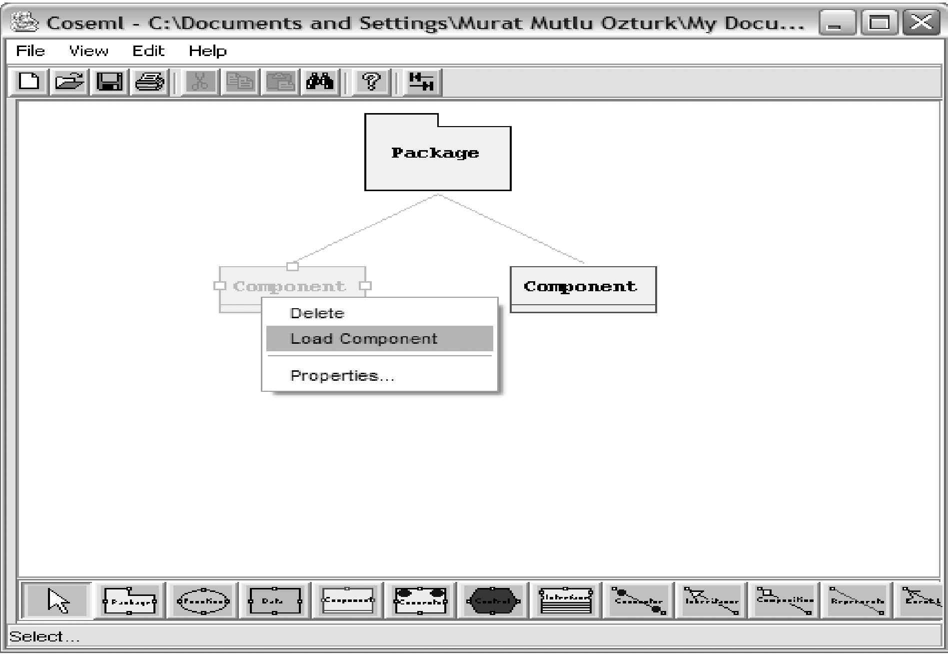

Figure 27 Triggering Component Loading

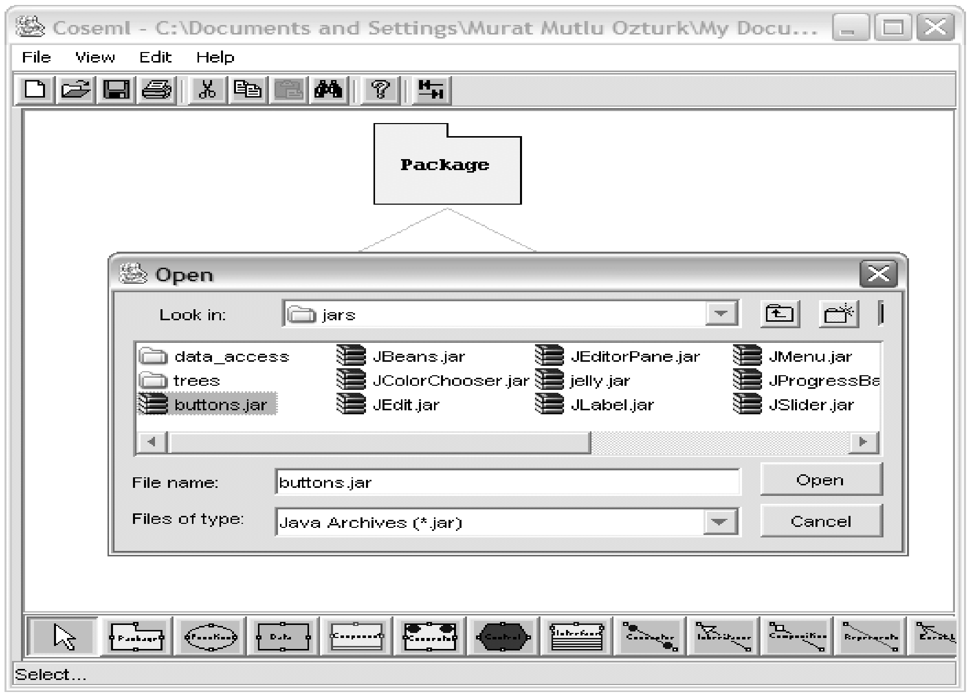

Figure 28 JAR Selection Dialog

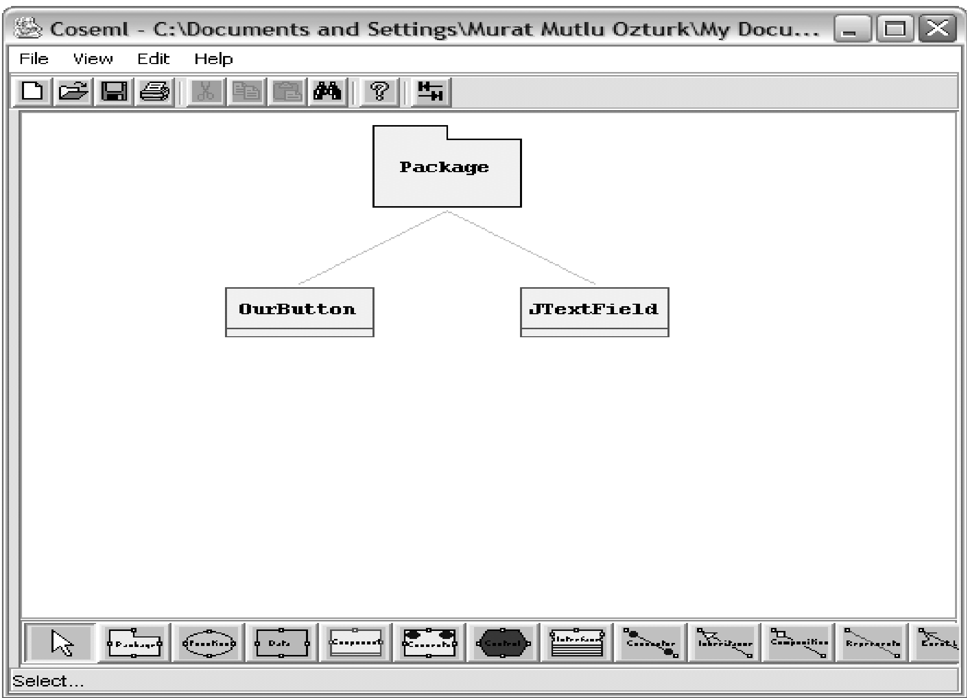

Figure 29 System Design after Assigning JavaBean Components

4. The association between components is displayed by a “Connector” drown between two components as it can be seen in Figure 30.

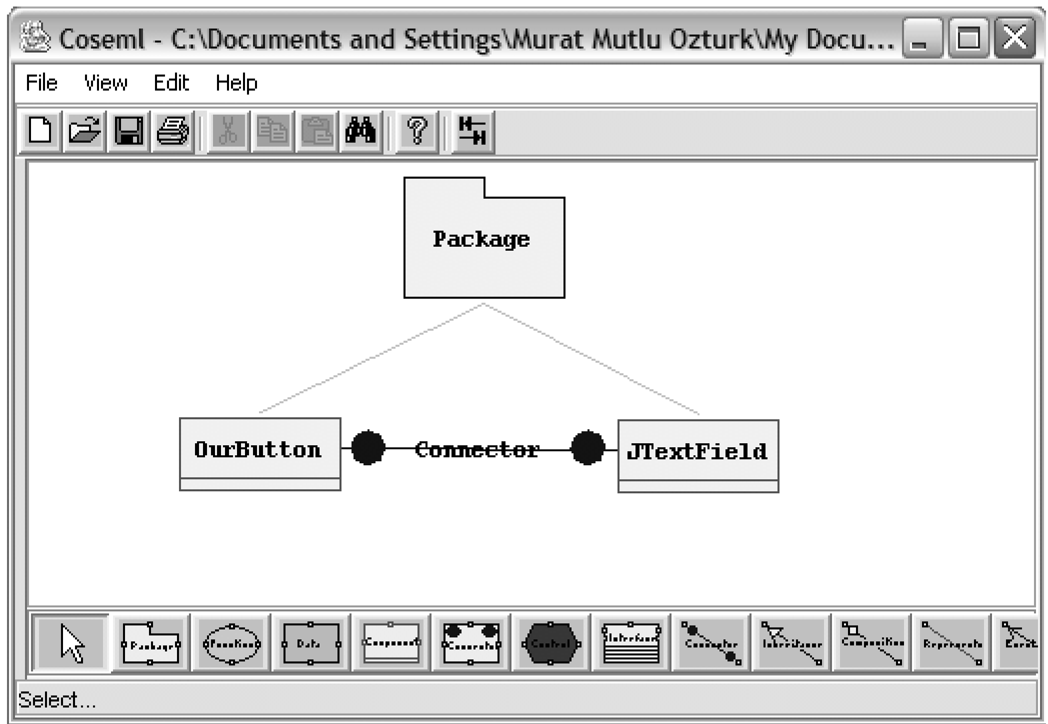

Figure 30 Components Associated with "Connector" Link

5. The roles of the components are assigned by using the help dialogs. There are three sub-steps needed to be completed to define the role assignments that are identified as follows:

   a. Event Assignment: After selecting “Create Association” submenu item (Figure 31) by right clicking the connector stretched between the components, “Role Assignment Interface” dialog is displayed. This first step in this dialog is selection of an event that is going to trigger the other function in the target bean. In this dialog, the event

groups and events of the selected event group are displayed. This dialog box is displayed in Figure 32. After selecting the desired event, process continues with the next sub-step for integration.

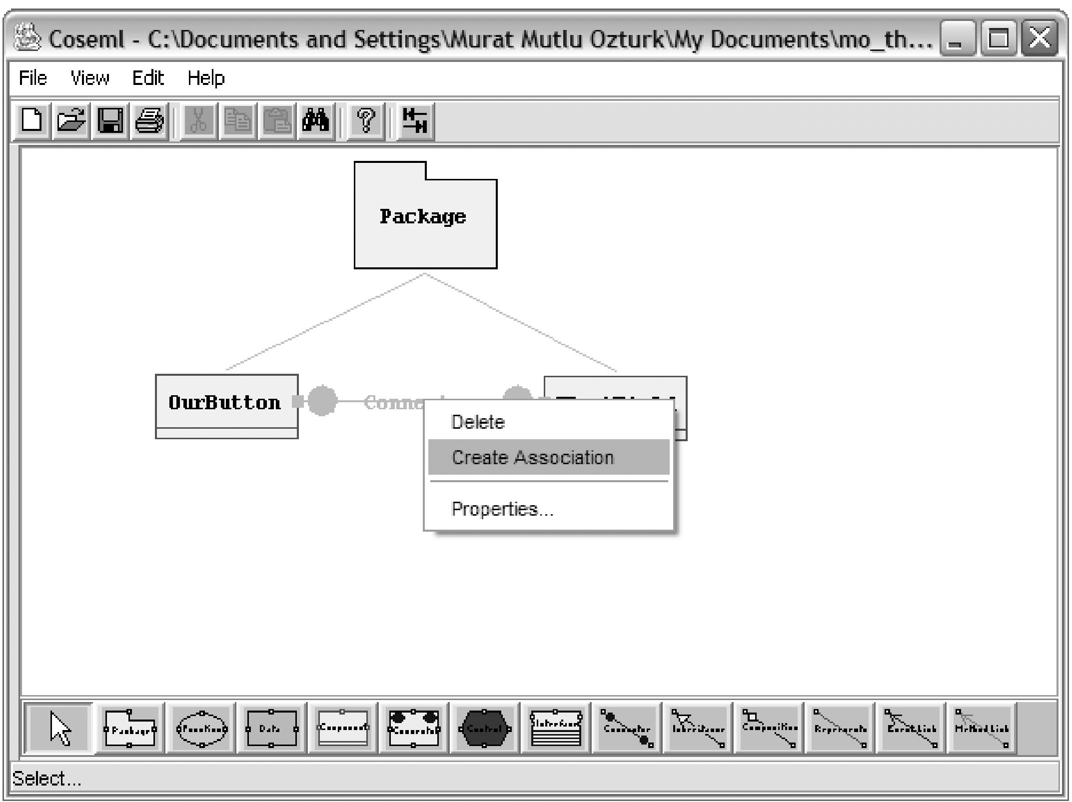

Figure 31 Creating Association

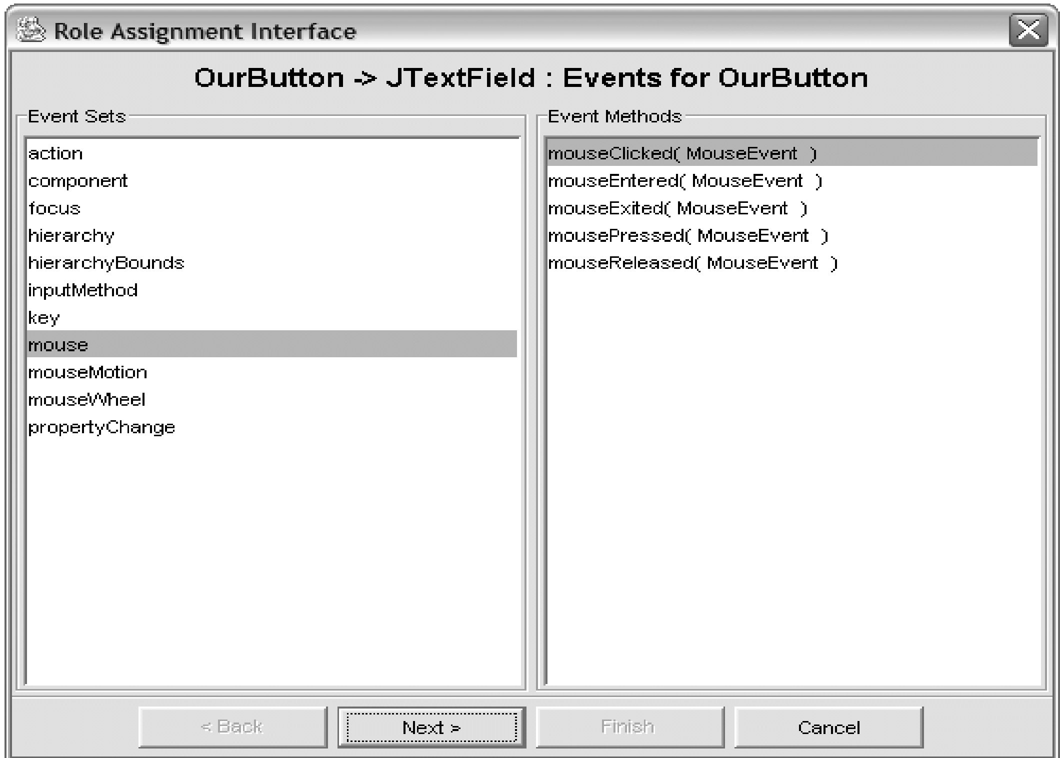

Figure 32 Event Assignment Dialog

b. Target Method Assignment: In this second step, the target method that is going to be called by the source bean in case the selected event happens is selected. The reason for selecting the target method before the source method is that, according to the selected method in this step, source methods displayed in the next step are filtered. The filtering is performed by using the arguments of the selected target method. If the selected target method has any argument, getter-setter pairing typed association is generated. If a target method without any parameter is selected, then there is no need to select any source method for the source component. In that case, an association type that pairs events is generated. The dialog of this step is displayed in Figure 33. Here, the *setText()* method is selected as the target method.

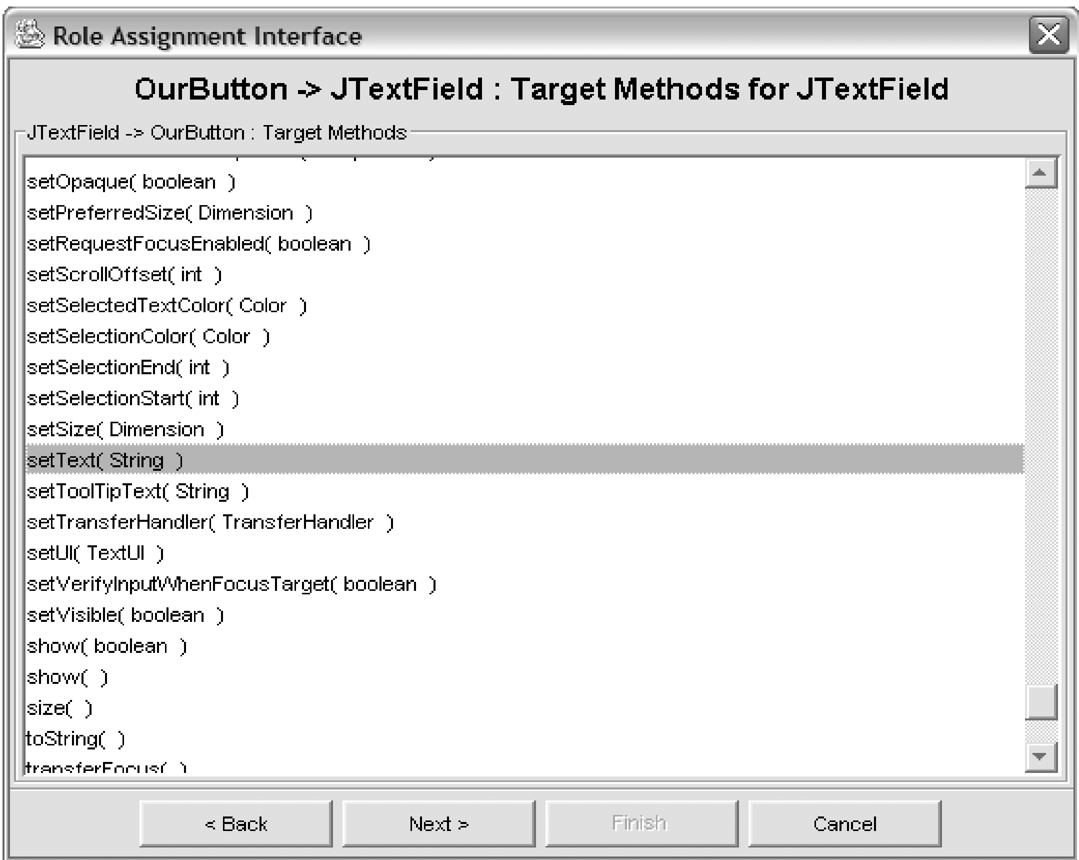

Figure 33 Target Method Assignment Dialog

c. Source Method Assignment: In this last step, the source method that is going to be initiated before calling the method of the target component is selected. The methods displayed in this dialog are filtered as described in the previous step. The dialog box for this last step can be seen in Figure 34. The *getName()* method is selected as the source method.

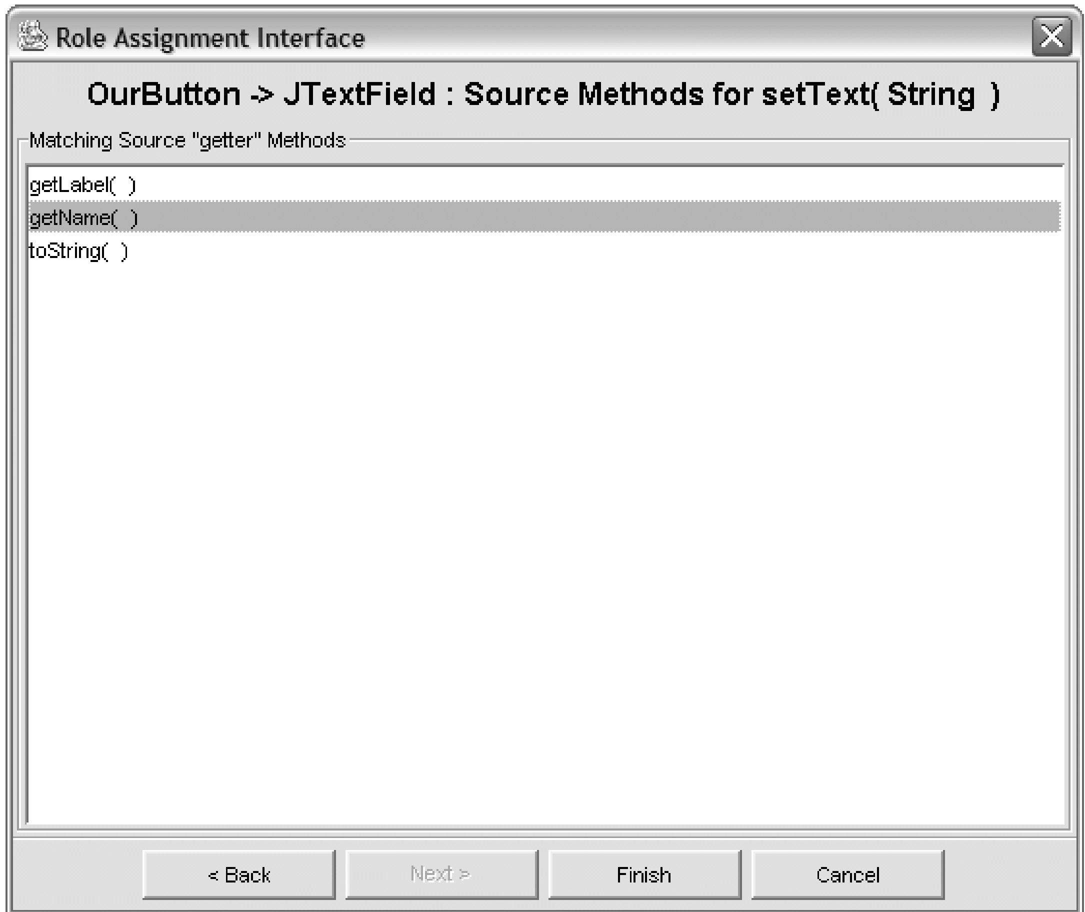

Figure 34 Source Method Assignment Dialog

6. After the assignment of the roles, system design continues with changing the mode of the beans instantiation window as shown in Figure 37.

Figure 35 Bean Instantiation Window

7. Each association between the components can be instantiated by applying steps 1 through 5. Main window is depicted in Figure 36. This screen corresponds to the default design view that is the structural decomposition.

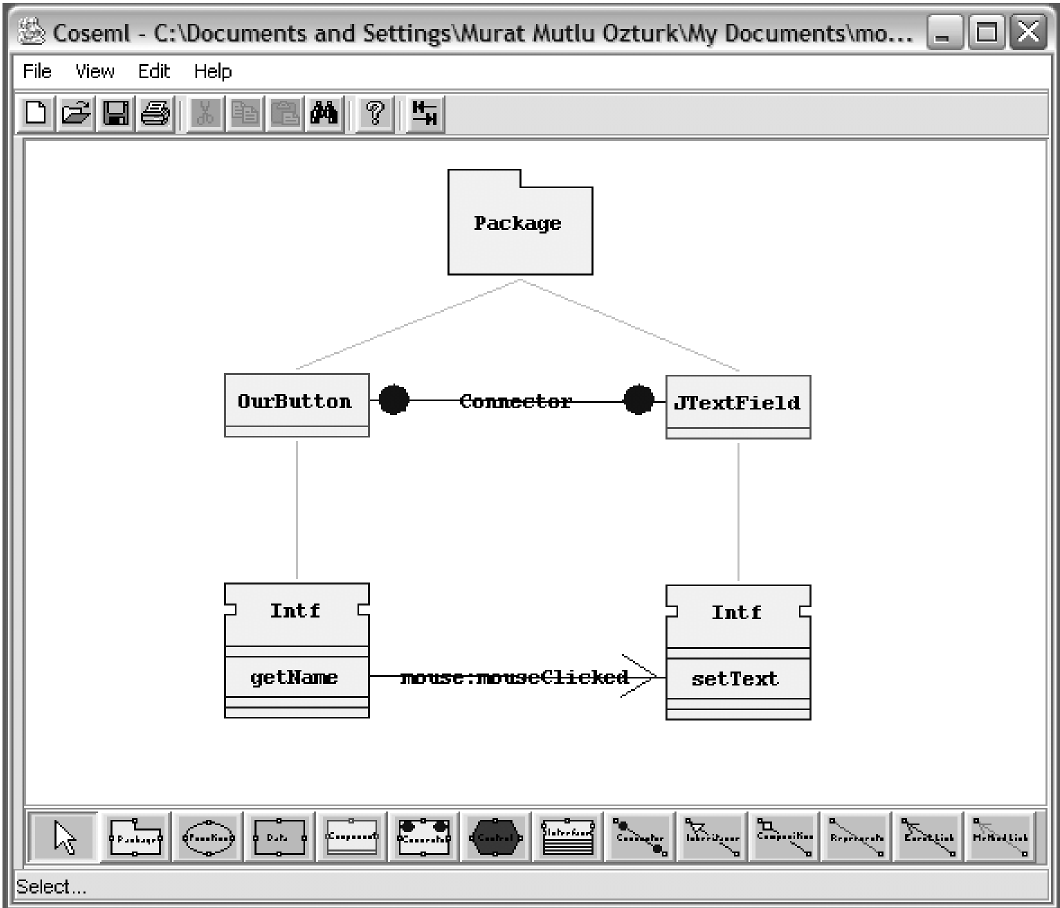

Figure 36 Main View

IV.2.4. Sample Applications

In this section, some simple system designs are presented. In each subsection, brief descriptions about the systems are presented. Additionally, main design and bean instantiation windows of COSECASE are included for each case.

IV.2.4.1. Sample System 1

This sample system design includes two Editor Pane JavaBeans. Interfacing of these two components are based on the “getter-setter pairing” rule. Accordingly, in case the key typing event occurs in source bean, *getText()* method of source bean with the *String* return type is called and the return value of this function is passed to the *setText()* method of the target bean that requires one parameter of type *String*. The execution scenario of this sample application can be seen in Figure 37. Additionally, main window of COSECASE is given in Figure 38 and bean description window takes place in Figure 39.

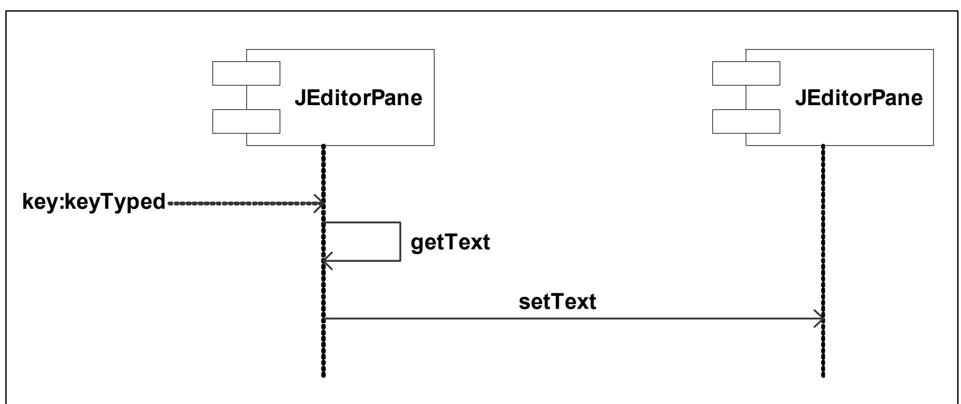

Figure 37 Execution Scenario of the Sample System 1

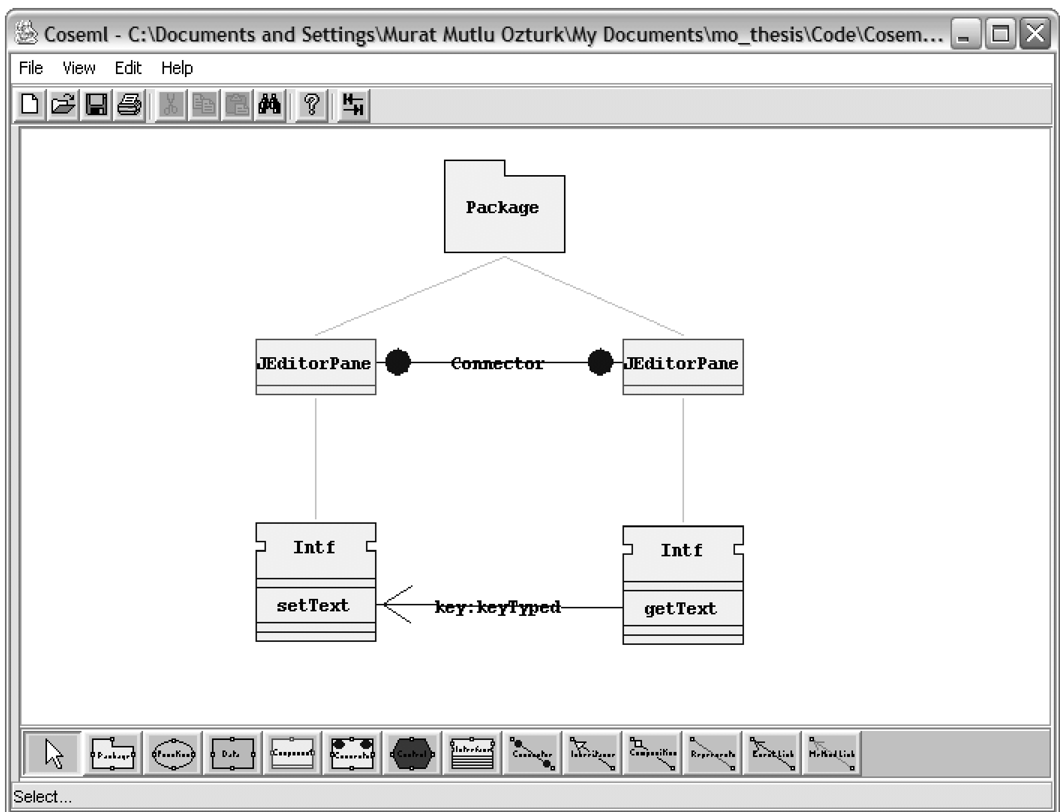

Figure 38 Main Window of COSECASE for Sample System 1

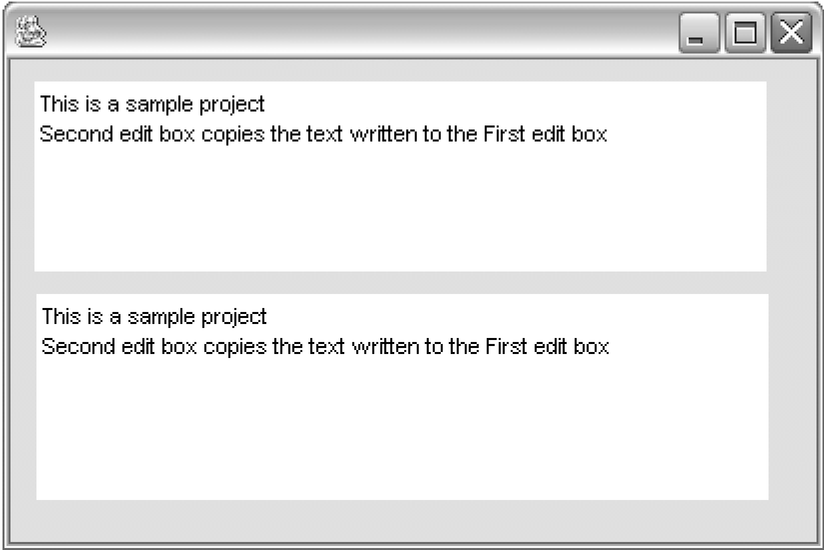

Figure 39 Bean Instantiation Window for Sample System 1

IV.2.4.2. Sample System 2

This sample system design also includes two JavaBeans, an Editor Pane and a Text Field. Interfacing of these two components are based on the "getter-setter pairing" rule as in the previous example. In case the "focus lost" event occurs in the source bean, the *getText()* method with the *String* return type is called and the return value of this function is passed to the *setPage()* method that accepts one parameter of *String* type. *getText()* method retrieves the text written to the bean text area. *setPage()* method opens the given "URL" with the specified protocol supplied in the argument text. If a file in a local disk is requested to be opened through the Windows operating system, the argument should be given as follows:

file://<Partition>:/< File_Path >

The execution scenario of this sample application can be seen in Figure 40. Additionally, main view for the system is given in Figure 41 and bean description window is shown in Figure 42.

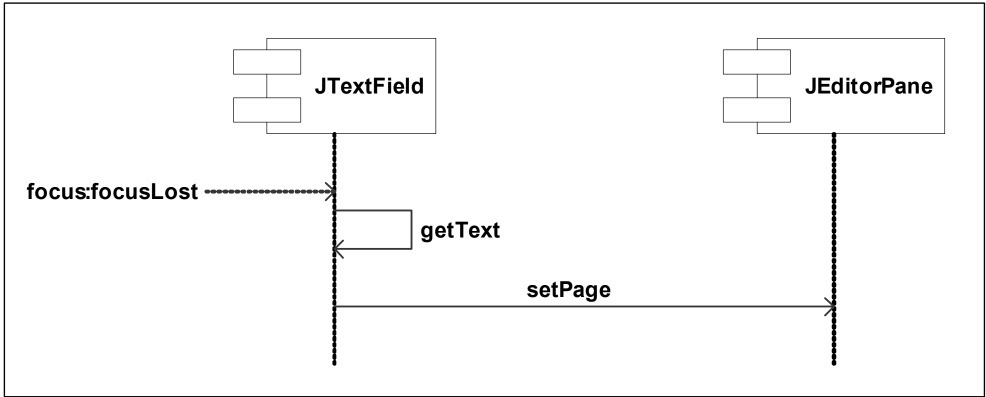

Figure 40 Execution Scenario of the Sample System 2

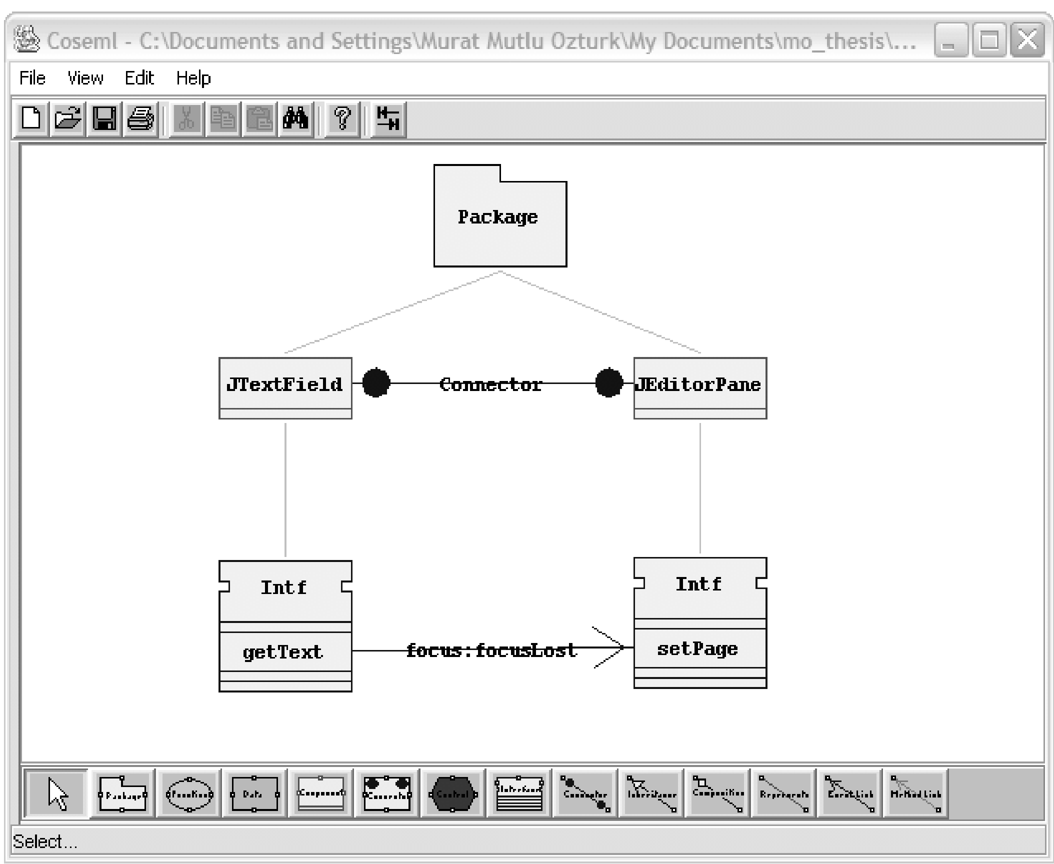

Figure 41 Main Window of COSECASE for Sample System 2

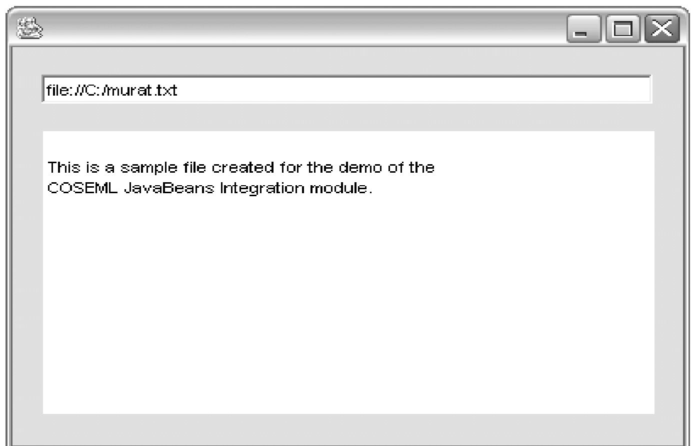

Figure 42 Bean Instantiation Window for Sample System 2

IV.2.4.3. Sample System 3

This sample system design includes two JavaBeans, a Slider and a Progress Bar. Interfacing of these two components are again based on the "getter-setter pairing" rule. Accordingly, in case the mouse dragging event occurs in the source bean, the *getValue()* method with the *int* return type is called and the return value of this function is passed to the *setValue()* method that requires one parameter of *int* type. *getValue()* method retrieves the current position value in integer of the slider bean and *setValue()* method sets the amount of progress displayed by the progress bar. During the execution, the same fluctuations in the progress bar while the slider is moved back and forth are supposed to be observed.

The execution scenario of this sample application is presented in Figure 43. Additionally, corresponding COSEML model is given in Figure 44 and bean description window is given in Figure 45.

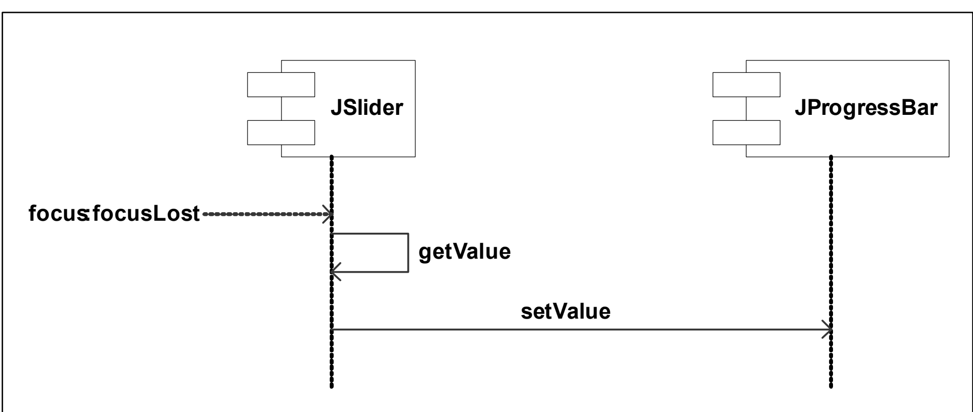

Figure 43 Execution Scenario of the Sample System 3

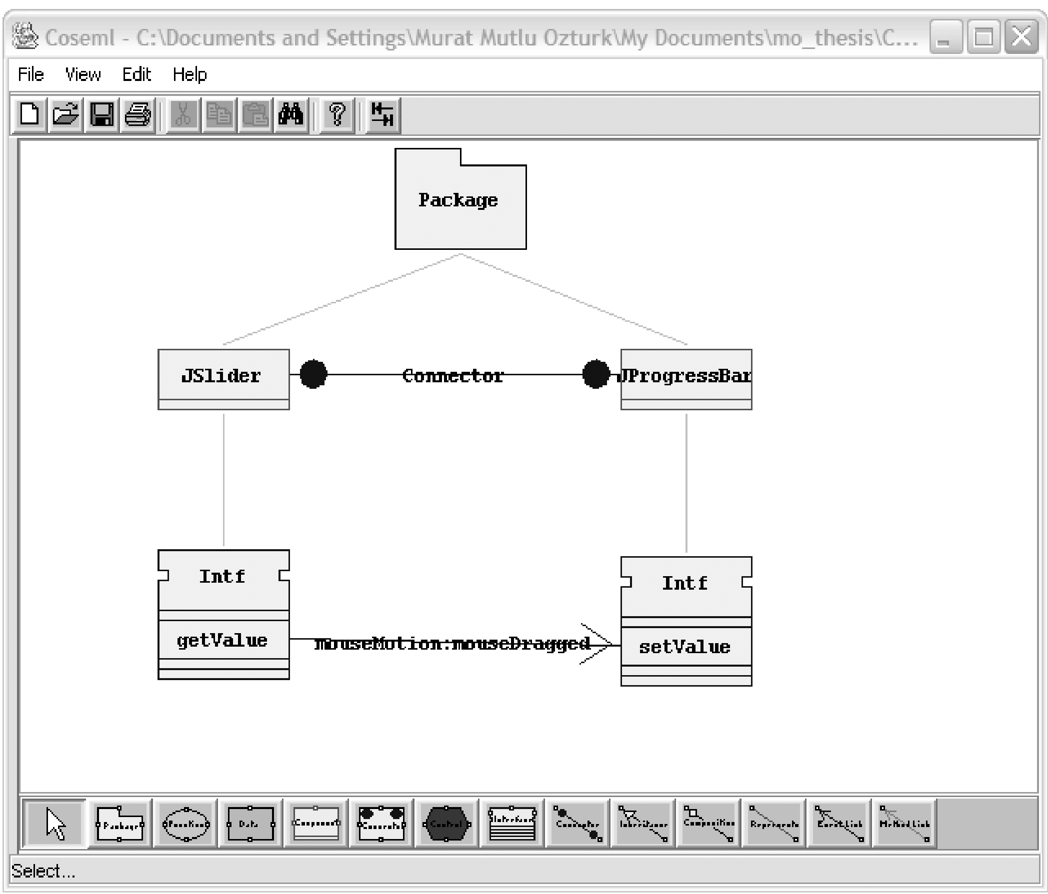

Figure 44 Main Window of COSECASE for Sample System 3

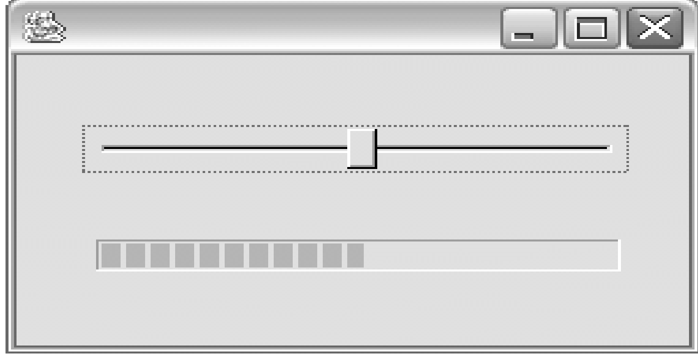

Figure 45 Bean Instantiation Window for Sample System 3

IV.2.4.4. Sample System 4

This sample system design includes three JavaBeans, two Buttons and an Edit Box. Each interfacing of these three components are based on the "event-method pairing" rule. For the interfacing between one of the buttons and the text field is based on the mouse click event. In case this event occurs in the source button bean, the *cut()* method is directly called in the target edit box bean. The *cut()* method cuts the selected text and saves it on the clipboard. On the other hand, the interface between the other button and the text field is also based on the mouse click event. In case this event occurs in the second button bean, the *paste()* method is directly called in the same target edit box bean. The *paste()* method pastes the text saved to the clipboard.

The execution scenario of this sample application can be seen in Figure 46. Additionally, main COSEML view is given in Figure 47 and bean description window is shown in Figure 48.

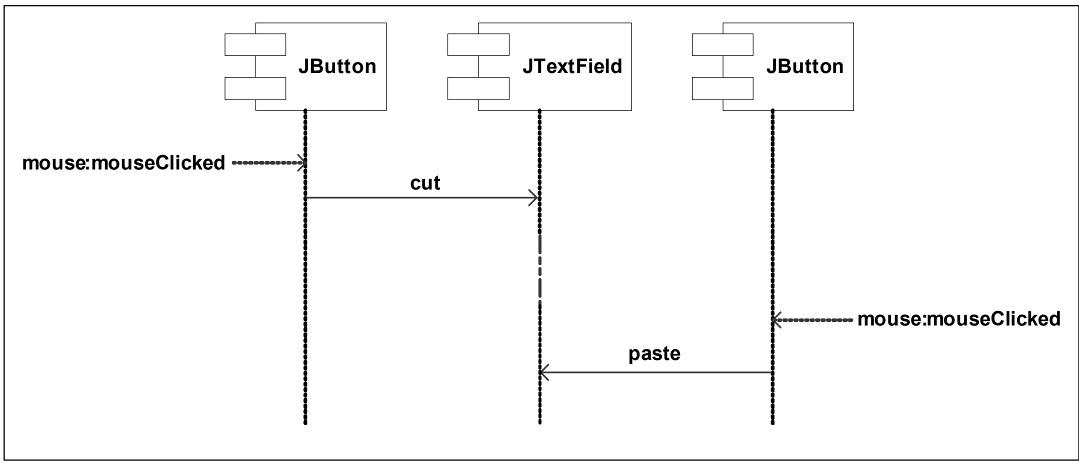

Figure 46 Execution Scenario of the Sample System 4

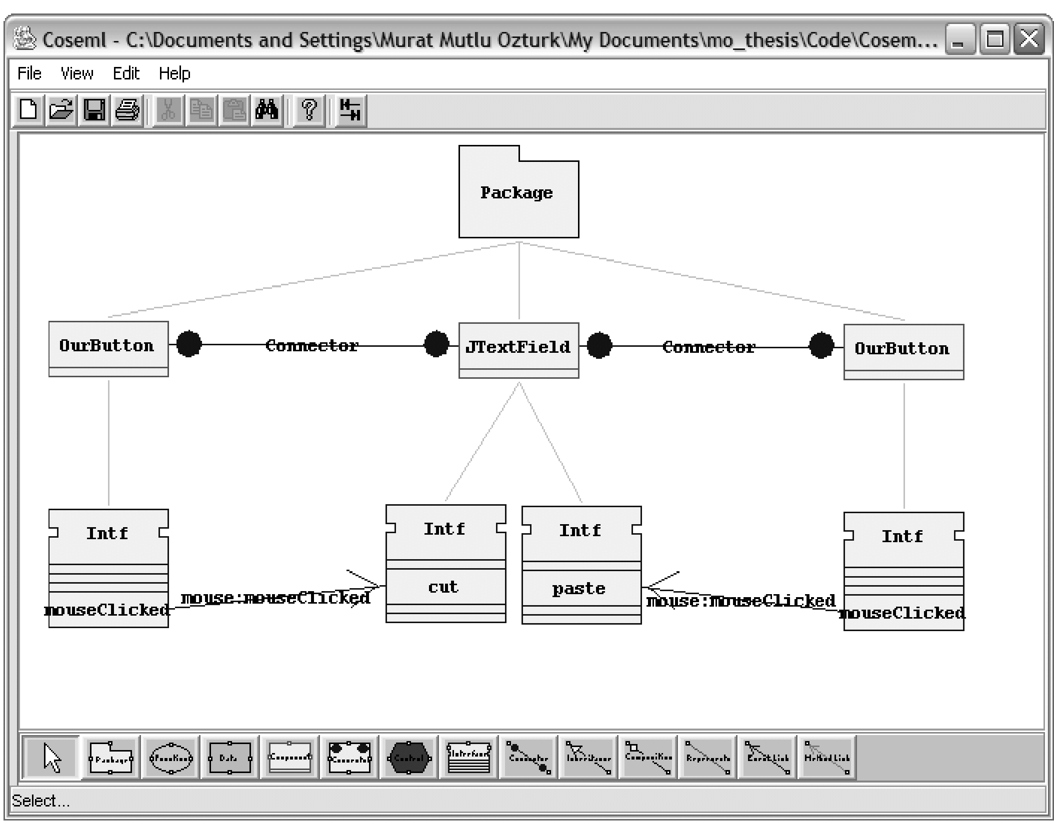

Figure 47 Main Window of COSECASE for Sample System 4

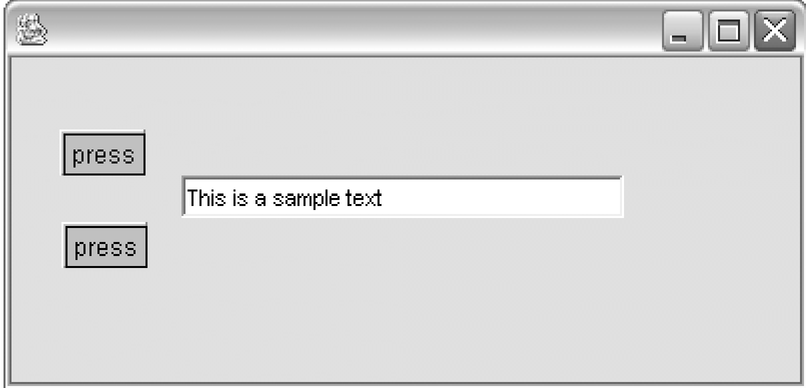

Figure 48 Bean Instantiation Window for Sample System 4

IV.2.4.5. Sample System 5

This sample system design includes two JavaBeans, a Color Chooser and a Text Field. Interfacing of these two components are based on the “getter-setter pairing” rule. In case the mouse click event occurs in the source bean, the *getColor()* method with the *Color* return type is called and the return value of this function is passed to the *setBackground()* method that accepts one parameter in *Color* type. The *getColor()* method retrieves the selected color in Color Chooser bean and the *setBackground()* method sets the background color for the text field.

The execution scenario of this sample application is shown in Figure 49. Also the COSEML view is given in Figure 50 and bean description window is presented in Figure 51.

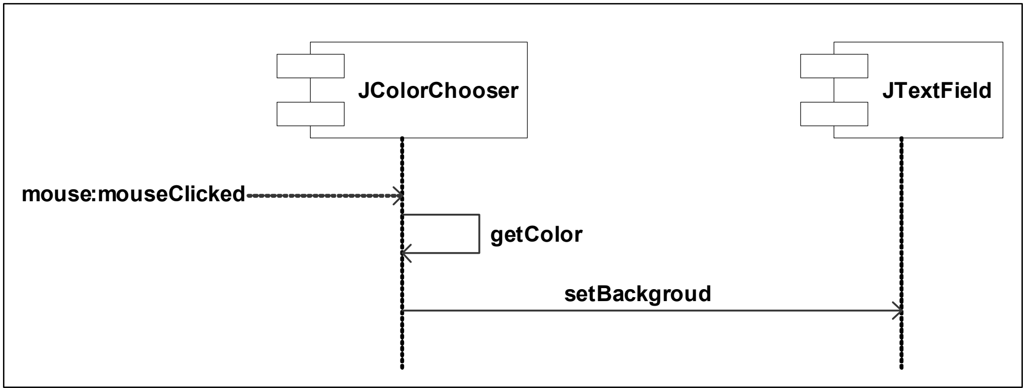

Figure 49 Execution Scenario of the Sample System 5

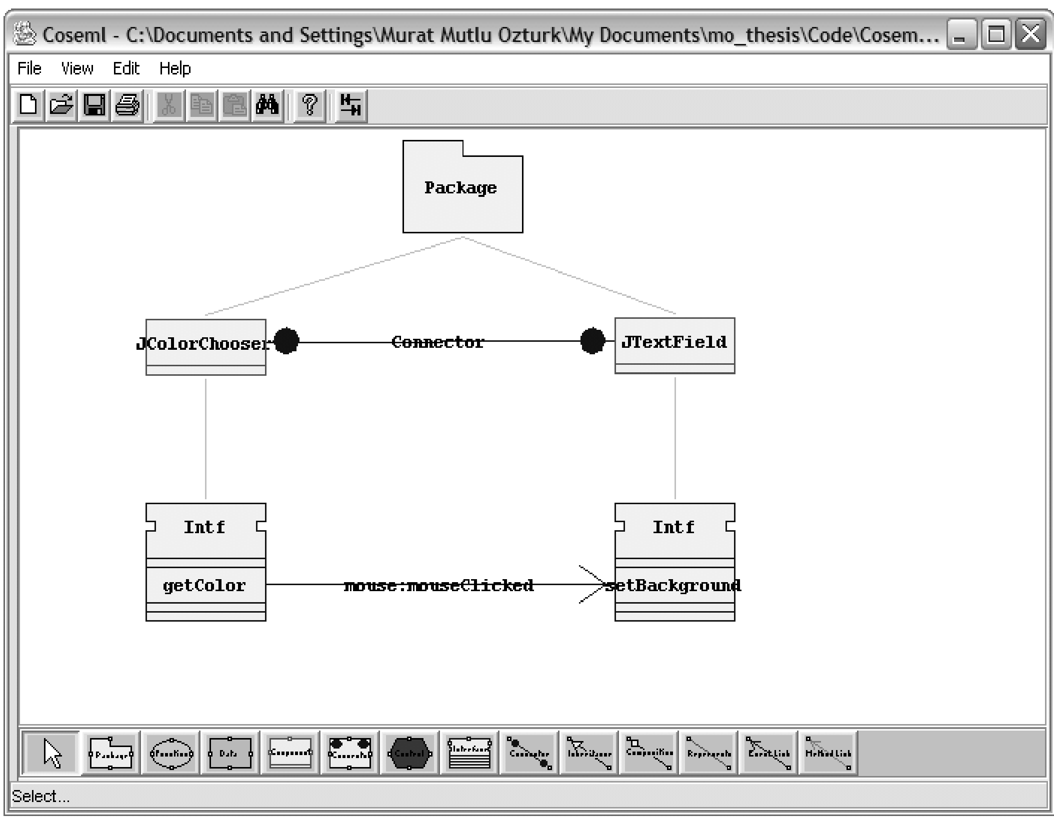

Figure 50 Main Window of COSECASE for Sample System 5

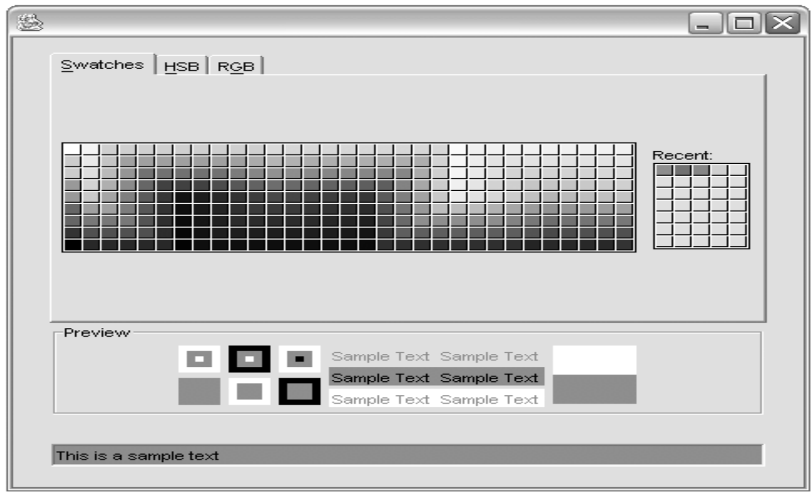

Figure 51 Bean Instantiation Window for Sample System 5

CHAPTER VI

CONCLUSION

This chapter includes the final remarks highlighted in this thesis. The first section includes the evaluation and critique of the work performed and the following subsection discusses future work and items open for improvement.

V.1. Evaluation

The literature on component oriented software development lacks use of a modeling media for visual composition. In this thesis, component oriented development with COSEML for visual composition is introduced. In a more specific definition, an approach to visual composition for JavaBeans framework in COSEML is presented. The details of COSEML modeling language, it's supporting design tool COSECASE and their role and importance in visual composition are discussed in detail.

In the conceptual maturation of this thesis work, the appropriateness of COSEML and component frameworks are investigated for the purpose of visual composition. The JavaBeans framework is selected as the target component framework. The reasons for this selection are discussed in detail.

Event Model in JavaBeans component framework is used as the composition model that is tailored to the COSECASE. By the addition of the new capabilities to the COSECASE, systems decomposed to their components are instantiated by

corresponding JavaBean components. This instantiation process is succeeded with the connectors between the components defined in COSECASE. By using the connectors between the components, composition can be realized by the triggered events of JavaBeans and the methods called. After the association and the realization phase, there is a ready to go software product in hand. Also the testing of the system is always possible in any step of the association and the realization phase.

In the development of the JavaBeans composition module, basic bean supporter classes are used for processing the beans at runtime. Additionally, in the scope of this thesis the beans developed or compiled for the testing purposes supplied a deep understanding of the Java Component technology.

For the easy integration purposes with other versions of the COSECASE that have been implemented by other thesis work, the module is implemented with a very low coupling fashion from the baseline code. A general configuration management repository of the baseline code is under construction with the support of collaborating thesis students.

By means of the visual composition approach applied to the COSEML and realized by COSECASE, the development of a software product is reduced to only design phase in this thesis. The systems that are decomposed into constructing components can build up by assigning the JavaBean conjugates of the components. Additionally, sample applications are developed to support the conceptual proposal of the thesis. The main contribution of this thesis work is the procurement of the component integration by visually composing the building blocks of a system. In this approach the design phase of a component oriented system development is combined with the integration phase by enhancing the COSECASE tool with visual composition of JavaBeans components. This allows the developer to integrate the system components at runtime. By means of this work, both design and integration are conducted using COSECASE. This approach expedites the component oriented development process and helps the emergence of the integration problems earlier.

By means of integrating components on the basis of the real contact points – interfaces – of the hard coded components, the design phase reveals all disagreements in the first place.

As discussed, the previous approaches proposed integration by means of scripting languages that glues the components or by completely proposing another component language. Almost all of the approaches require modification or correction of the existing components. On the other hand, by synthesizing the visual composition with a component oriented modeling language, this study aims the usage of the JavaBean components without any modification.

The visual integration module is not tested extensively. The systems developed include only simple components integrated with simple instantiations. In the sample applications developed, it is observed that the module enables the usage of component libraries efficiently without writing any code. In this manner, by means of combining the design phase with integration, it becomes very easy to develop systems by only defining the contact points and interaction methods between components. Additionally, by combining integration with design, the usual effort in cycling between the design and integration phases is eliminated. However, in this very initial phase of visual composition in COSE, this work should be supported with case studies for revealing the defectiveness and weaknesses of the approach.

V.2. Future Work

Future research includes validation of the constructed software with respect to the target component architecture. After the development of the visual composition module, it is not tested exhaustively with beans that have different methods or events in different ranges. Consequently, it would be helpful to use the tool in some case studies that includes components generated in different domains for different purposes.

As a future study, other component frameworks can be introduced to the COSECASE for extending the usage areas of the COSEML and COSECASE. This broadening would provide the efficient usage of the component libraries. As another future enhancement for COSECASE, an applet or application generation interface can be introduced as a supplemental module to the visual composition module.

Another issue that is open to improvement is that this module is implemented for the purpose of defining the composition between two components. For the execution scenario of the whole system, it would be beneficial to define the roles of the components in execution scenarios. So, introduction of the sequencing to the COSEML with the support of visual composition module would let to develop more complicated systems. In order to support large scale systems, the integration approach in this study should be supported with a sequencing mechanism in-between the components, such as the interaction diagrams in UML.

REFERENCES

[1] C. Szyperski, “Component Software: Beyond Object-Oriented Programming”, *Addison-Wesley*, New York, 1998.

[2] G. Paul, K. Sattler, and M. Endig, “An Integration Framework for Open Tool Environments”, *Chapman & Hall*, 1996.

[3] M. Goedicke, and T. Meyer, “Design and Evaluation of Distributed Component-Oriented Software Systems”, *Turku Centre for Computer Science*, Germany, TUCS Technical Report No. 0, May 1997.

[4] M. Goedicke, J. Cramer, W. Fey, and M. GroBe-Rhode, “Towards a Formally Based Component Description Language – a Foundation for Reuse”, *Structured Programming*, Vol. 12 No:2, Springer, Berlin, Germany, 1991.

[5] M. Goedicke, and H. Schumann, “Component Oriented Software Development with II”,*Fraunhoser Institute for Software-Engineering and Systems Engineering*, ISST report 21/94, 1994.

[6] Software Process Engineering, “The Unified Process Model (UPM)”, *Initial Submission*, OMG Document ad/2000-05-05, May, 12 2000.

[7] J. Ackermann, “Specification Proposals for Customizable Business Components”, S. Turowski (Ed.), *Proceedings 1st International Workshop Component Engineering Methodology (WCEM’03),* Erfurt, pp. 51-62, September 24, 2003.

[8] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. “Component-based Product Line Engineering with UML”, C. Szyperski (Ed.), *Addison-Wesley,* Great Britain, 2002.

[9] Booch, G., “Software Components with Ada: Structures, Tools and Subsystems”, 1987.

[10] A.W. Brown, and K.C. Wallnau, “The Current State of CBSE”, *IEEE Software*, September-October, 1998.

[11] M.M. Tanık, and E.S. Chen, “Fundamentals of Computing for Software Engineers”, Van Nostrand Reinhold, NewYork, 1991.

[12] I. Altintas, “A Feasibility Study For Component Oriented Design Modeling”, *M.S. Thesis*, Computer Engineering Department, Middle East Technical University, Ankara, Turkey, May 2001.

[13] A. Kara, “A Graphical Editor for Component Oriented Modeling”, *M.S. Thesis*, Computer Engineering Department, Middle East Technical University, Ankara, Turkey, April 2001.

[14] V. Bayar, “A Process Model for Component Oriented Software Development”, *M.S. Thesis*, Computer Engineering Department, Middle East Technical University, Ankara, Turkey, November 2001.

[15] C. Szyperski, and C. Pfister, “Why Objects are not enough?”, *Proceedings, First International Component Users Conference (CUC’96),* Munich, Germany, 15-19 July 1996.

[16] T.D. Meijler, O. Nierstrasz, “Beyond Objects: Components”, *Cooperative Information Systems: Current Trends and Directions,* Academic Press, November 1997.

[17] R. Ibrahim, “Component-Based Systems: A Formal Approach”, *Proceedings of Component-Oriented Software Engineering Workshop (COSE'98) in conjunction with Australian Software Engineering Conference (ASWEC'98)*, November 1998.

[18] O. Nierstrasz, S. Gibbs, and D. Tsichritzis, “Component-Oriented Software Development”, *Communications of the ACM*, Vol. 35, No. 9. Special Issue on Analysis and Modeling in Software Development, pp. 160-165, September, 1992.

[19] O. Nierstrasz, D. Tsichritzis, V.D. Mey, and M. Stadelmann, "Objects + Scripts = Applications", *Proceedings, Esprit 1991 Conference*, Kluwer, Dordrecht, pp. 534–552. 1991.

[20] H. Schumann, and M. Goedicke, "Component-Oriented Software Development with II", *Fraunhofer Institute for Software-Engineering and Systems Engineering*, ISST Report 21/94, July 1994.

[21] U. Hölzle, "Integrating Independently-Developed Components in Object-Oriented Languages", *ECOOP'93 Proceedings,* Springer Verlag Lecture Notes on Computer Science*,* 1993

[22] P.J. Hoen, J.N. Kok, G. Busatto, and L.P.J. Groenewegen, "From OO to Components: Components at the Type and Instance Level", The Netherlands, February 29, 2000.

[23] S. Yau, and B. Xia, "Object-Oriented Distributed Component Software Development Based on CORBA", *Proceeding, 22"d Int'l Computer Software and Application Conference (COMPSAC 98)*, pp.246-251, August, 1998.

[24] D. Konstantas, "Interoperation of Object-Oriented Applications", Object-Oriented Software Composition, O. Nierstrasz and D. Tsichritzis (Ed.), *Prentice Hall*, pp. 69-95, 1995.

[25] X. Pintado, "Gluons and the Cooperation between Software Components," Object-Oriented Software Composition, O. Nierstrasz and D. Tsichritzis (Ed.), *Prentice Hall*, pp. 321-349, 1995.

[26] V. Mencl, "Component Definition Language", *M.S. Thesis*, Charles University, Prague, The Czech Republic, April 28, 1998.

[27] K. M. Goudarzi, and J. Kramer, "Maintaining Node Consistency in the Face of Dynamic Change", *Proceedings of 3rd International Conference on Configurable Distributed Systems (CDS'96), IEEE Computer Society Press*, Annapolis, Maryland, USA, pp 62-69, May 1996.

[28] Rapide Design Team, “Rapide 1.0 Language Reference Manual”, *Program Analysis and Verification Group*, Computer Systems Lab, Stanford University, January 1996.

[29] R. Allen, and G. Garlan, “Formalizing Architectural Connection”, In Preceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, pp. 71-80, May 1994.

[30] N. Medvidovic, “A Classification and Comparison Framework for Software Architecture Description Languages”, *Technical Report UCI-ICS-TR-97-02*, 1997.

[31] V.D. Mey, “Visual Composition of Software Applications”, *Object-Oriented Software Composition, Prentice Hall,* O. Nierstrasz and D. Tsichritzis (Ed.), pp. 275-303, 1995.

[32] V.D. Mey, B. Junod, S. Renfer, M. Stadelmann and I. Simitsek, “The Implementation of Vista - A Visual Scripting Tool”, *Object Composition*, D. Tsichritzis (Ed.), Centre Universitaire d’Informatique, University of Geneva, pp. 31-56, June 1991.

[33] O. Nierstrasz, L. Dami, “Component-Oriented Software Technology”, *Object-Oriented Software Composition, Prentice Hall*, O. Nierstrasz and D. Tsichritzis (Ed.), pp. 3-28, 1995.

[34] O. Nierstrasz, and T.D. Meijler, “Requirements for a Composition Language”, *In Preceedings of the ECOOP’94 workshop on Models and Languages for Coordination of Parallelism and Distribution*, Springer Verlag, LNCS 924, pp. 147-161, 1995.

[35] N. Skarmeas, and K. L. Clark, “Component Based Agent Construction”, *International Journal on Artificial Intelligence Tools*, Vol. 11, No. 1, pp139-163, 2002.

[36] N. Lynch, and M. Tuttle, “Hierarchical Correctness Proofs for Distributed Algorithms”, *Proc. 6th Ann. ACM Symp. on Principles of Distributed Computing*, pp. 137-151, August 1987.

[37] D. Ingalls, "Fabrik: A Visual Programming Environment," *Object-Oriented Programming Systems Languages and Applications (OOPSLA), Special Issue of SIGPLAN Notices*, vol. 23, no. 11, pp. 176-190, Nov. 1988

[38] D. Goodman, "The Complete Hypercard Handbook", *Bantam Books*, 1988.

[39] NeXT Inc., *NeXT Preliminary 1.0 System Reference Manual: Concepts*, 1989.

[40] V.D. Mey, and O. Nierstrasz, "The ITHACA Application Development Environment", *In Visual Objects*, D. Tsichritzis (Ed.), Centre Universitaire d'Informatique, University of Geneva, pp. 267-280, July 1993

[41] S. Yau, and B. Xia, "An Approach to Distributed Component-Based Real-time Application Software Development", *Proc. 1st IEEE Int'l Symp. Objectoriented Real-time Distributed Computing*, pp. 275-283, April 1998

[42] A.H. Doğru, and I. Altintas, "Modeling Language for Component Oriented Software Engineering: COSEML", *proceedings of the Sixth World Conference on Integrated Design and Process Technology,* Dallas, Texas, 2000.

[43] K. J. Fellner, and K. Turowski, "Classification Framework for Business Components", *Proceedings pf the 33rd Hawaii International Conference on Syste Sciences*, Hawaii, 2000

[44] P. H. Fröhlick, A. Gal, and M. Franz, "Supporting Software Composition at the Programming Language Level", *Science of Computer Programming 56,* pp. 41-57, 2005

[45] M. Lumpe, and J-G. Schneider, "A Form-Based Meta-Model for Software Composition", *Science of Computer Programming 56,* pp. 59-78, 2005

[46] K. Rege, "Design Patterns for Component-Oriented Software Development", *Proceedings of 25th EUROMICRO Conference*, Milan, Italy, Vol. 2, pp. 220-228, September 8 – 10, 1999.

[47] P. Feiler, and W. Tichy, "Propagator – A Family of Pattern", *TOOLS USA 97*, Santa Barbara, USA, 1997

[48] C. Szyperski, "Component Technology – What, Where, and How?", *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, 2003

[49] A. Kozak, "Component-Oriented Modeling of Land Registery and Cadastre Information System Using COSEML", *M.S. Thesis*, Computer Engineering Department, Middle East Technical University, Ankara, Turkey, November 2002.

[50] S. Baboo, "Java Bean Component Architecture for Java," White Paper http://www.issoln.com/knowledgebase/javabean/javabean.html, International Software Solutions Inc., 1999.

[51] J. Ginbayashi, R. Yamamoto, and K. Hashimoto, "Business Component Framework and Modelling for Component-based Application Architecture", *4th International Enterprise Distributed Object Computing Conference (EDOC'00)*, Makuhari, Japan, pp. 184-193, September 25-28, 2000.

[52] J. Ginbayashi, K. Hashinoto, and K. Yabuta, 'Response to BODTF RFI-I (Common Business Objects)", *Object Management Group document boml97-11-0 1*, Nov. 1997.

[53] X. Cai, M. R. Lyu, K-F. Wong, and R. Ko, "Component-Based Software Engineerign: Technologies, Development Frameworks, and Quality Assurance Schemes", *7th Asia-Pacific Software Engineering Conference (APSEC'00)*, Singapore, pp. 372 – 382, December 5-8, 2000.

[54] L. F. Carpets, M. A. M. Carpetz, and D. Li, "Component-Based Software Development", *IECON'01: The 27th Annual Conference of the IEEE Industrial Electronics Society*, 2001

[55] S.D. Urban, A. Saxena, S. W. Dietrich, and A. Sundermier, "An Evaluation of Distributed Computing Options for a Rule-Based Approach to Black-Box Software Component Integration", *Proceedings of the Third International Workshop on Advanced Issues in E-Commerce and Web-Based Information Systems*, San Jose, CA, pp. 100-109, June 2001.

[56] S.S. Yau, and F. Karim, "Integration of Object-Oriented Software Components for Distributed Application Software Development", *Proceedings of 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, Tunisia, South Africa, pp. 111-116, December 1999.

[57] J. Penix, B. Fischer, G. Pour, J. van Baalen, and J. Whittle, "Automating Component Integration for Web-Based Data Analysis", 2000 IEEE Aerospace Conference. Big Sky, Montana, March 2000.

[58] V. C. Sreedhar, "Mixin'up Components", *Proceedings of the International Conference on Software Maintenance (ICSE'02)*, Orlando, Florida, USA, May 19-25, 2002.

[59] W. Zhao, and J. Chen, "CoOWA: A Component Oriented Web Application Model", *Proceedings of the 31st International Conference on Technology of Object-Oriented Language and Systems*, p.191, September 22-25, 1999.

[60] Y. Vandewoude, and Y. Berbers, "Run-time Evolution for Embedded Component-Oriented Systems", *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, 2002.

[61] A. Teitelbaum, and H. G. Mendelbaum, "Arts'Codes: Generation of Parallel-Automata Real-Time Systems, using a Unifying Diagrammatic Component Oriented Design Methodology", *Proceedings of the IEEE International Conference on Software-Science, Technology & Engineering (SwSTE'03)*, 2003.

[62] V.A. Nagin, I. V. Potapov, and S. V. Selishchev, "A Distributed Component-Oriented Architecture for Real-Time ECG Data Acquisition Systems", *2001 Proceedings of the 23rd Annual EMBS International Conference,* Istanbul, Turkey, October 25-28, 2001.

[63] J. Hutchinson, G. Kotonya, I. Sommerville, and S. Hall. "A Service Model for Component-Based Development", *Proceedings of the 30th IEEE EUROMICRO Conference (EUROMICRO'04)*, Rennes, France , pp.162-169, September 2004.

[64] G. Xie, "Decompositional Verification of Component-based Systems – A Hybrid Approach", *Proceedings of the 19th International Conference on Automated Software Engineering (ASE'04)*, Linz, Austria, pp. 414-417, September 20-24, 2004.

[65] C. Dellarocas, "Toward a Design Handbook for Integrating Software Components", *Proceedings of the 5th International Symposium on Assessment of Software Tools (SAST '97)*, Pittsburgh, pp. 3-13, June 03-05,1997.

[66] F. J. Doucet, S. K. Shukla, and R. K. Gupta, "Typing Abstractions and Management in a Component Framework", *Proceedings of Asia and South Pacific Design Automation Conference*, 2003.

[67] M. G. Christiansen, S. N. Delcambre, E. Demirörs, O. Demirörs, and M. M. Tanık, "Software Development with Transformable Components", *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, Cat. No.91TH0394-7, pp. 558-559, Vol.2, 1991.

[68] C. C. Chiang, "The Use of Adapters to Support Interoperability of Components for Reusability", *Information and Software Technology 45,* pp. 149-156, 2003.

[69] J. P. Sousa, and D. Garlan, "Formal Modeling of the Enterprise JavaBeans™ Component Integration Framework", *Information and Software Technology 43*, pp. 171-188, 2001.

[70] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural Mismatch: Why Reuse is so Hard", *IEEE Software*, November 1995.

[71] A. H. Doğru, "Component Oriented Software Engineering Modeling Language: COSEML," *TR-99-3*, Computer Engineering Department, Middle East Technical University, December 1999.

[72] A. H. Doğru, and M. M. Tanık, "A Process Model for Component Oriented Software Engineering", IEEE Software, Vol.20, No. 2, s.34-41, March-April 2003.

[73] Sun Microsystems, "JavaBeans API Specification," Version 1.01, July 1997.

[74] D. Krieger, and R. M. Adler, "The Emergence of Distributed Component Platforms," *Computer*, vol. 31, no. 3, pp. 43-53, March 1998.

[75] SUN Microsystems, http://java.sun.com/,May 2005

[76] H. Fossa, and M. Sloman, "Implementing Interactive Configuration Management for Distributed Systems", *Proceedings of 3rd International Conference on Configurable Distributed Systems (CDS'96), IEEE Computer Society Press*, Annapolis, Maryland, USA, pp 44-51, May 1996.