USING COLLABORATION DIAGRAMS IN COMPONENT ORIENTED MODELING

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

BY

MEHMET BURHAN TUNCEL

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

JANUARY 2006

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 Hikmet Doğru
Supervisor

Examining Committee Members

Prof. Dr. Volkan Atalay	(METU,CENG)	
Assoc. Prof. Dr. Ali Hikmet Doğru	(METU,CENG)	
Dr. Ayşenur Birtürk	(METU,CENG)	
Dr. Cevat Şener	(METU,CENG)	
Kurtcebe Eroğlu	(HAVELSAN)	

ABSTRACT

USING COLLABORATION DIAGRAMS IN COMPONENT ORIENTED MODELING

Tuncel, Mehmet Burhan

M. S., Department of Computer Engineering

Supervisor: Assoc. Prof. Dr. Ali Hikmet Dogru

January 2006, 80 pages

Component Oriented Software Engineering (COSE) seems to be the future of software engineering. Currently, COSEML is the only modeling language that completely supports the COSE approach. Abstract decomposition of the system and their representing components are shown in a hierarchy diagram to support the COSE process model. In COSEML, only static modeling is supported through this single diagram. However, software is about behavior and static modeling is not sufficient to describe the system. The aim of this thesis is providing the benefits of dynamic modeling to COSEML by adopting collaboration diagrams. For this purpose, first, specification of modified collaboration diagrams is made for COSEML. Then software is developed for supporting collaboration diagrams in COSECASE. Also, an e-store application is modeled with COSEML using the collaboration diagrams. With this work, modeling the dynamic behavior of the system in both abstract and component levels is made possible. Furthermore, use case realization is enabled in the COSE modeling. More important, modeling the sequential interactions among components is made possible. Consequently, a suitable environment is provided for automated testing and application generation from the model.

Keywords: Component Oriented Software Engineering, COSEML, Component Oriented Software Modeling Language, Collaboration Diagrams

ÖZ

İŞBİRLİĞİ DİYAGRAMLARININ BİLEŞEN YÖNELİMLİ MODELLEMEDE KULLANIMI

Tuncel, Mehmet Burhan

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Danışman: Assoc. Prof. Dr. Ali Hikmet Doğru

Ocak 2006, 80 sayfa

Bileşen Yönelimli Yazılım Mühendisliği (BYYM), yazılımın geleceği olarak görülmektedir. Şu an BYYM yaklaşımını destekleyen tek modelleme dili COSEML'dir. Sistemin soyut ayrışımı ve bunu temsil eden bileşenler, BYYM süreç modelini desteklemek amacıyla bir hiyerarşi diyagramı üzerinde gösterilmektedir. COSEML'de modelleme, bu statik diyagram üzerine dayanmaktadır. Ancak, yazılım davranışla ilgilidir ve statik modelleme sistemi anlatmak için yeterli değildir. Bu tezin amacı, işbirliği diyagramlarını kullanarak dinamik modellemenin faydalarını COSEML'e sağlamaktır. Bu amaçla, önce işbirliği diyagramlarının COSEML için belirtimi yapılmıştır. Ardından bu diyagramların COSECASE'de kullanımını destekleyen yazılım geliştirilmesi yapılmıştır. Bunu takiben, bir sanal mağaza uygulaması, işbirliği diyagramları kullanılarak COSEML ile modellenmiştir. Bu çalışmayla birlikte, sistemin dinamik davranışının hem soyut seviyede, hem de bileşen seviyesinde modellenmesi mümkün kılınmıştır. Ayrıca BYYM modellemesinde kullanıcı senaryolarının gerçekleştirimine olanak sağlanmıştır. Daha önemlisi, bileşenler arasındaki sırasal etkileşimin modellenebilmesine imkan verilmiştir. Bunun bir sonucu olarak, model üzerinden otomatik yazılım testi yapılmasına ve uygulama üretilmesine uygun bir ortam sağlanmıştır.

Anahtar Kelimeler: Bileşen Yönelimli Yazılım Mühendisliği, COSEML, Bileşen Yönelimli Yazılım Modelleme Dili, İşbirliği Diyagramları

To My Parents

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Assoc. Prof. Dr. Ali Hikmet Dogru, for his guidance and encouragement throughout the research. To my family, I offer sincere thanks for their emotional support.

TABLE OF CONTENTS

APPENDICES

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Today, in the information age, there is an exponential increase on the demand for software. Scope and complexity of the software have dramatically increased. Current software industry mostly deals with huge government and military applications. Because of the high competition in the industry, such software systems should be built in less time with less cost.

Moreover, technological and business requirements change frequently in these systems. For these reasons, today's software systems are more likely to face the software crisis. Obviously, traditional software approaches that are built on code development are becoming less efficient.

To respond to the demand and overcome the software crisis, new approaches have been developed that benefit from component technologies. Among these approaches, Component Oriented Software Engineering (COSE) [1] proposes building software by integrating existing components.

A modeling language, COSEML [2], and a graphical modeling editor COSECASE [3] were developed for COSE approach. This modeling language is based on a single hierarchy diagram. This static diagram offers "divide and conquer" capabilities to system design on modeling environment. In this static diagram, modeling starts by decomposing the system structure hierarchically. This activity continues until the decomposition model arrives at existing components. Relations among these components are also shown on the model. Then in the integration phase of COSE, these components are integrated to build the desired system.

1.1. Motivation for Using Collaboration Diagrams in COSEML

In the hierarchy diagram of COSEML, messages are not allowed among abstract elements. Only connectors are used to show the relations [2]. It is not possible to show the dynamic behavior. However, software is mostly about dynamic behavior and emphasizes only on static modeling is not appropriate.

In fact, static and dynamic models support each other. Without dynamic models, validity of the static model is left to the intuition of the model designer and its accuracy cannot be proven true. Apparently, a new diagram for modeling dynamic behavior is needed in the component oriented modeling language. Two candidates for this diagram are collaboration and sequence diagrams, which are used to model dynamic interactions in Unified Modeling Language (UML) [4]. Both diagrams are equal concerning their semantic expressiveness but they stress two different views. The sequence diagram emphasizes the temporal perspective of interaction. On the other hand, the collaboration diagram emphasizes the various kinds of relationships among the interacting objects [5].

The collaboration diagrams emphasize on the structural organization of the elements. Since COSE is a structure-oriented approach, using collaboration diagrams in COSEML is more appropriate.

Collaboration diagrams provide a clear picture of collaborating elements and their roles in the model. They are useful to visualize the collaborating parties executing a scenario in terms of a sequence of messages [6]. Therefore, using collaboration diagrams improve the expressiveness of the model. Besides, they allow use case realization, which is important both in analysis and implementation phase of any software engineering process. Allowing use case realization is important for validating and improving the static hierarchy diagram of the COSEML. More importantly, collaboration diagrams contain the information about the sequence of message calls among the components, which creates a potential for automated testing and application generation from COSEML models.

1.2. Organization of the Thesis

In this thesis, first, background information is given in Chapter 2. Then in Chapter 3, specification of collaboration diagrams is defined and their possible benefits to COSE modeling are explained. In Chapter 4, their implementation to COSECASE is described. After that, a case study: E-store Application is modeled using collaboration diagrams, to show the improved COSE modeling activity. Finally, conclusions of the work are expressed.

CHAPTER 2

BACKGROUND

2.1. Software Reuse

Software reuse is the only solution to the software crisis problem. The main idea is to build systems using already developed software pieces. This idea was pointed out by McIlroy in the early ages of software engineering. "Develop systems of components of reasonable size and reuse them. Then extend the idea of *component systems* beyond code alone to requirements, analysis models, design and test. All the stages of the software development process are subject to reuse" [7]. Reuse is still in the center of attention in software engineering. Although the principle is simple, it has been shown that the process is hard and tedious.

There have been different levels of software reuse. Lowest level of them is the *source code copy*. In this usage, copied parts are spread like a virus in the software. If a requirement changes or a bug is found in the copied code, all clients are required to update the changes.

Next level is the *function-libraries*, which provides a better form of reuse. Code is central and any internal change does not affect the clients. However, function-libraries are not extensible. Clients are effected with any change made to the input or output parameters.

Class-libraries offer a higher level of reuse. It has the benefits of OO approach and they are extensible. However, it requires a lot of understanding before classes can be reused. Moreover, it supports only white-box reuse, which means some modifications are required on the software unit to adapt it to the other software. In an OO language, derived classes are coupled to the base class implementation. Any change made to the base classes directly affects derived classes.

To this point, all the mentioned types of reuse are language specific. Reuse is not supported across code in other languages *Components* are the solution to this problem and they offer the highest level of software reuse. Components provide services to the clients through their interfaces and support black-box reuse. Inner implementation of the components is hidden to the outer world. It is the highest level of information hiding. Clients only rely on interfaces. As long as the interfaces remain unchanged, components can be changed internally without affecting clients. For this reason, using components for system development is the most promising approach for utilizing the full power of reuse in software systems.

2.2. Components and Current Component Technologies

A *software component* is an independent, encapsulated software piece with a well-defined functionality. Internal implementation of a component is hidden from the user. They provide their functionality through their interfaces. The goal of component approach is to standardize the interfaces among software components so that they can work together efficiently.

Advances in component technologies in the last decade bring the components in to the real life. Although there is not a unique, comprehensive standard for components, they are made available to usage with the name commercial off-the-shelf (COTS) components. These COTS components can be developed by different developers using different languages and different platforms.

There have been some approaches to share and distribute application pieces, but most of these approaches rely on certain underlying services to provide the communication and coordination necessary for the application. Three of the component infrastructure technologies have become rather standardized: OMG’s CORBA, Microsoft’s Component Object Model (COM) and Distributed COM (DCOM), and Sun's JavaBeans and Enterprise JavaBeans. Brief information about these component technologies, which is provided in [11], is given in the following sub-sections.

2.2.1. Common Object Request Broker Architecture (CORBA)

CORBA [8] is an open standard for application interoperability that is defined and supported by the Object Management Group (OMG), an organization of over 400 software vendor and object technology user companies. CORBA manages details of component interoperability, and allows applications to communicate with one another despite of different locations and designers. The interface is the only way that applications or components communicate with each other. The most important part of a CORBA system is the Object Request Broker (ORB). The ORB is the middleware that establishes the client-server relationships among components. Using an ORB, a client can invoke a method on a server object, whose location is completely transparent. The ORB is responsible for intercepting a call and finding an object that can implement the request, pass its parameters, invoke its method, and return the results. The client does not need to know where the object is located, its programming language, its operating system, or any other system aspects that are not related to the interface. In this way, the ORB provides interoperability among applications on different machines in heterogeneous distributed environments and seamlessly interconnects multiple object systems.

CORBA is widely used in Object-Oriented distributed systems including component-based software systems because it offers a consistent distributed programming and run-time environment over common programming languages, operating systems, and distributed networks

2.2.2. Component Object Model (COM) and Distributed COM (DCOM)

Introduced in 1993, Component Object Model (COM) is a general architecture for component software [9]. It provides a component-based software architecture that is language-independent and platform-dependent (Windows systems). COM defines how components and their clients interact. This interaction is defined such that the client and the component can connect without the need of any intermediate system component. Specially, COM provides a binary standard that components and their clients must follow to ensure dynamic interoperability. This enables on-line software update and cross-language software reuse.

As an extension of the Component Object Model (COM), Distributed COM (DCOM) [9] is a protocol that enables software components to communicate directly over a network in a reliable, secure, and efficient manner. DCOM is designed for use across multiple network transports, including Internet protocols such as HTTP. When a client and its component reside on different machines, DCOM simply replaces the local inter-process communication with a network protocol. Neither the client nor the component is aware the changes of the physical connections.

2.2.3. JavaBeans and Enterprise JavaBeans

Sun's Java-based component model consists of two parts: the JavaBeans for client-side component development and the Enterprise JavaBeans (EJB) for the server-side component development [10]. The JavaBeans component architecture supports applications of multiple platforms, as well as reusable, client-side and server-side components.

Java platform offers an efficient solution to the portability and security problems using portable Java byte codes and the concept of trusted and untrusted Java applets. Java provides a universal integration and enabling technology for enterprise application development. Following list contains most important benefits of Java platform.

1. Interoperability across multi-vendor servers

2. Propagating transaction and security contexts

3. Servicing multilingual clients

4. Supporting ActiveX via DCOM/CORBA 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.

2.2.4. Comparison among Current Component Technologies

None of the current component technologies is superior to others. All have their strong sides on different aspects. Table 1 presents a brief comparison of these component technologies [11].

Table 1. Comparison of current component technologies

	CORBA	**EJB**	**COM/DCOM**
Development Environment	Underdeveloped	Emerging	Supported by a wide range of strong development environments
Binary Interfacing Standard	Not binary standards	Based on COM; Java specific	A binary standard for component interaction is the heart of COM
Compatibility and Portability	Particularly strong in standardizing language bindings; but not so portable	Portable by Java language specification; but not very compatible.	Not having any concept of source-level standard of standard language binding.
Modification and Maintenance	CORBA IDL for defining component interfaces, need extra modification & maintenance	Not involving IDL files, defining interfaces between component and container. Easier modification & maintenance.	Microsoft IDL for defining component interfaces, need extra modification & maintenance

Table 1 (Continued)

	CORBA	**EJB**	**COM/DCOM**
Services Provided	A full set of standardized services; lack of implementations	Neither standardized nor implemented	Recently supplemented by a number of key services
Platform dependency	Platform independent	Platform independent	Platform dependent
Language dependency	Language independent	Language dependent	Language independent
Implementation	Strongest for traditional enterprise computing	Strongest on general Web clients.	Strongest on the traditional desktop applications

2.3. Component Based Software Engineering (CBSE)

Modern software systems are large-scale and very complex. Controlling such systems is not easy in an environment where technologies and requirements change frequently. Results are high development cost, low productivity and unmanageable software quality. Moreover, complexities of the systems are increasing exponential. On the other hand, development time should be shortened because of the high competition in the software sector. It is obvious that, traditional approaches do not have a chance in the future of software industry.

One of the most promising solutions today is the component-based software development approach. This approach intends to accelerate software development and to reduce costs by using prefabricated software components. CBSE practices increases as the COTS components are becoming more available.

This approach can significantly reduce development cost and time-to- market, and improve maintainability, reliability and overall quality of software systems. It has raised a tremendous amount of interests both in the research community and in the software industry.

Component-based software systems are developed by selecting various components and assembling them together rather than programming an overall system from scratch, thus the life cycle of component-based software systems is different from that of the traditional software systems.

The focus is on composing and assembling components that are likely to have been developed separately, and even independently. Component identification, customization and integration are a fundamental activity in the life cycle of component-based systems. It includes two main parts. First, evaluation of each candidate component based on the functional and quality requirements and second, customization of those candidate components that should be modified before being integrated into new component-based software systems. Integration is to make key decisions on how to provide communication and coordination among various components of a target software system.

2.4. Component Oriented Software Engineering (COSE)

Component oriented software engineering (COSE) is a new approach. Although CBSE and COSE seem similar, these approaches have some major differences.

CBSE process is built over the OO approach. This is very reasonable because when the components were arrived the software world; everything was OO, from analysis to testing in the software process. Currently, there is not much change in the situation. Standard modeling language of software UML is built on an OO backbone and current programming languages that are widely used, are all object oriented.

It is obvious that in order to elevate from first floor to the third floor, elevator must pass through the second floor. CBSE is the second floor built at the top of OO approach. When the software industry and research community realize the reuse power of components, all of the OO tools, technologies, languages and processes are adapted to support this magic boxes. Today, OO approaches that support components are defined as CBSE.

The third floor, COSE, on the other hand is not related to OO approach. Building by integration of components without writing code is the paradigm that defines the COSE [1]. Utilization of components is the central concern in the whole software development process. Idea of building systems completely with components makes it easy to solve most of the software engineering problems.

COSE approach is based on structural decomposition of the system. It enables the analysis and design phases of the system to be processed in a higher level of abstraction. Decomposition of the system is very straight, compared to data oriented decomposition of OO approaches. Data has a sensitive and dynamic nature. Changes in the requirements, in the business flow can totally change the way the system handles data. This is why data oriented models are difficult to maintain in the software process. On the other hand, structure oriented approach is not affected much from the changes.

Software process in COSE starts with structural decomposition in a top-down manner. Decomposition divides the system into logical modules. This process continues until reaching the existing components. These components then integrated in a bottom-up fashion in order to build the system.

2.5. Software Modeling

Implementation technologies are not at an enough level of abstraction to facilitate discussions about design, which creates a need for software models. Models describe the desired structure and behavior of a system. They are important for visualizing and controlling the system's architecture. A model is a simplification of reality [12]. It provides a better understanding of the system, which expose opportunities for simplification and reuse.

Defining a model makes it easier to break up a complex application or a huge system into simple, discrete pieces that can be individually studied. It is easy to focus on the smaller parts of a system and understand the "big picture". Hence, the reasons behind modeling are readability and reusability. Readability makes it easy to understand, and understanding a system is the first step in either building or improving a system. This involves knowing what a system is made up of, how it behaves, and so forth. Modeling a system ensures that it becomes readable and, most importantly, easy to document. It involves capturing the structure of a system and the behavior of the system. Reusability is the consequence of making a system readable. After a system has been modeled, similarities in terms of functionality, features, or structure are identified.

Modeling has been used in all the engineering disciplines for a long time. In some disciplines, modeling is highly matured so that, in electrical engineering or civil engineering, a model has a one to one correspondence to the final product.

Main goal of the software modeling should be reaching the level of CAD modeling used in civil engineering, which permits one to one modeling of real systems [13]. Currently there is an approach supported by Object Management Group (OMG), called Model Driven Architecture (MDA) [14]. In this approach, main idea is building systems from models, which is independent form implementation technology. Although there have been considerable work on software modeling, there is still a long way to go.

Graphical modeling languages have been around in the software industry for a long time. In following sub-sections, first, standard modeling language, UML, is briefly described. Then modeling language of the new COSE approach, COSEML and its modeling tool, COSECASE are explained.

2.6. Unified Modeling Language (UML)

Unified Modeling Language (UML) is the official industry standard for object-oriented modeling as defined by the Object Management Group (OMG) [4].

The UML is appropriate for modeling systems ranging from enterprise information systems to distributed Web-based applications. It is a very expressive language, addressing all the views needed to develop and then deploy such systems.

The UML is not limited to modeling software. In fact, it is expressive enough to model non-software systems.

Multiple models are needed to understand different aspects of a system. UML addresses the different views of a system's architecture with different diagrams as it evolves throughout the software development life cycle. Figure 1 shows the formal diagram hierarchy [4] in UML 2. There are two major kinds of diagram types: structure diagrams and behavior diagrams in UML 2 specification as it is shown in figure 1.

Brief description of these UML 2 diagrams can be summarized as follows.

- **Structure Diagrams**: Structure diagrams show the static structure of the objects in a system. Elements are described regardless of time. The elements in a structure diagram represent the meaningful concepts of an application, and may include abstract, real-world and implementation concepts. Structure diagrams do not show the details of dynamic behavior, which are illustrated by behavioral diagrams.
    - **Class Diagram**: A class diagram shows the classes and their relationships in a system. It is one of the most popular types of diagram in OO modeling.
    - **Composite Structure Diagram**: A composite structure diagram shows how objects are composed at runtime
    - **Component Diagram**: A component diagram shows the structural relationships among the components of a system.
    - **Deployment Diagram**: The deployment diagram depicts the configuration of the runtime elements of the application. This diagram is useful when a system is built and ready to be deployed.

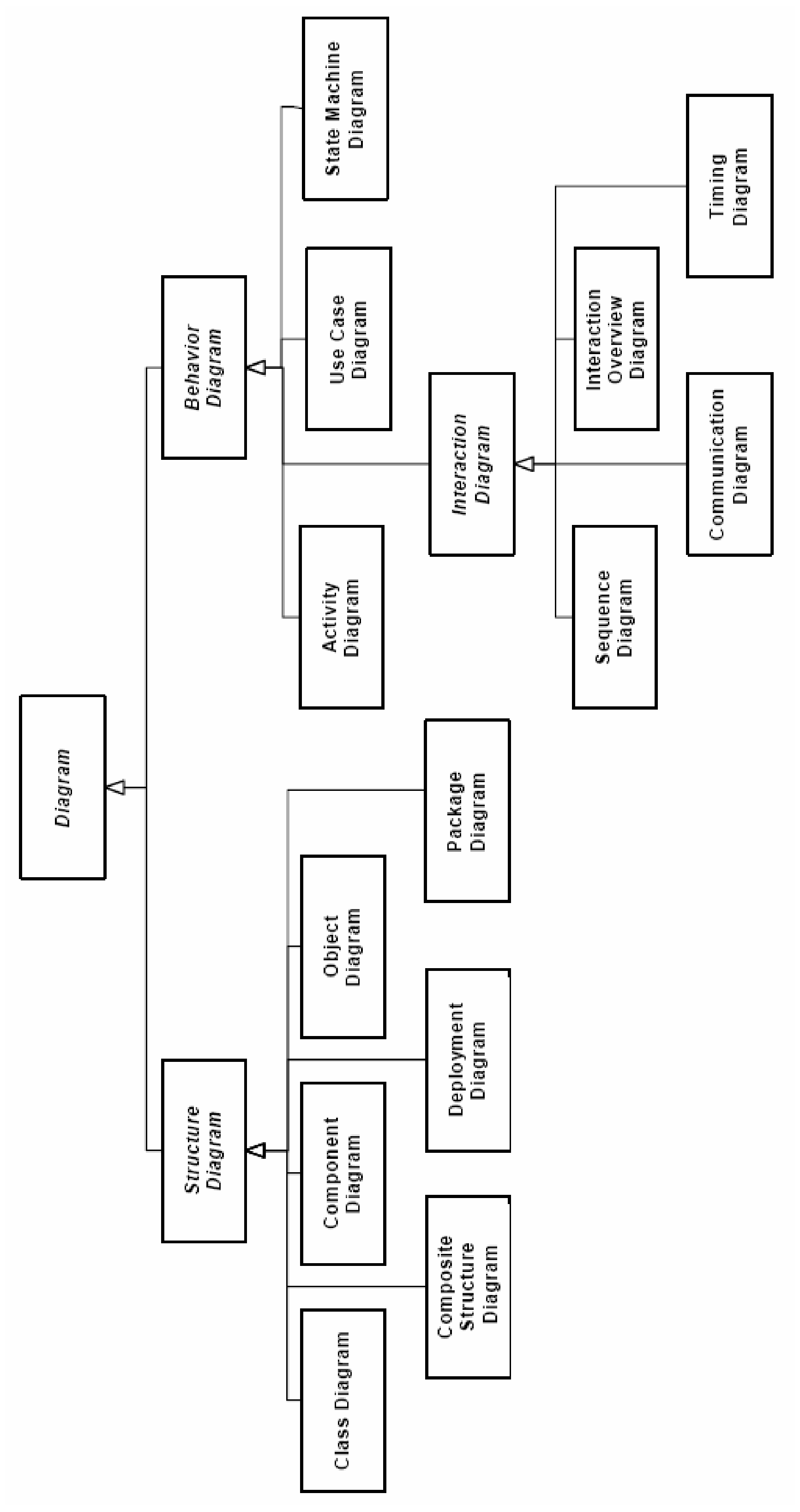


Figure 1. UML 2 diagrams

    - **Object Diagram**: The object diagram shows the state of different classes in the system. Their relationships or associations can be captured at a given point of time.
    - **Package Diagram**: A package diagram shows the organization of packages and their elements. They show compile-time groupings.

- **Behavior Diagrams**: Behavior diagrams show the dynamic behavior of the objects in a system, including their methods, collaborations, activities, and state histories. The dynamic behavior of a system can be described as a series of changes to the system over time. Behavior diagrams are further classified into several other kinds.
    - **Use Case Diagram**: The use case diagram is used to identify the primary elements and behavior that form the system. It shows, what the actors make, to fulfill a system behavior.
    - **Activity Diagram**: An activity diagram captures the process flows in the system. It consists of activities, actions, transitions, initial and final states, and guard conditions
    - **State Machine Diagram**: A state diagram represents the different states of the objects during their life cycle.
    - **Interaction Diagrams**: Interaction diagrams are used to model the dynamic aspect of collaborations and the roles of the elements in the system. An interaction diagram shows an interaction, consisting of a set of objects and messages between them.
        - **Sequence Diagram**: A sequence diagram represents the time-ordered interaction between different objects by showing the messages between them.
        - **Communication (Collaboration) Diagram**: A communication diagrams is used to model the dynamic behavior of the use case. Compared to sequence diagram,

the communication diagram is more focused on showing the collaboration of objects rather than the time sequence.

- **Interaction Overview Diagram**: An interaction overview diagram is a form of activity diagram in which the nodes represent interaction diagrams.

- **Timing Diagram**: A timing diagram shows the behavior of objects in a given period and it is useful for showing timing constraints between state changes on different objects.

2.7. COSE Modeling Language (COSEML)

COSEML is a graphical modeling language and it was developed for use in the COSE approach [2]. The goal of COSEML is to provide the human developer with the natural "divide and conquer" discipline based on structure [6]. Modeling with COSEML emphasizes structural decomposition. It is an adaptation of the earlier structure-based and decomposition oriented specifications [15].

In COSEML, modeling starts with a top-down decomposition of the system. In this phase, abstract building blocks of the system are found. This top-down activity continues while searching the representing components. When they are found, a bottom-up component composition is carried out to reach the desired capability of the system.

The hierarchy that is a key concept in design cognition is not supported effectively in UML and other languages [6]. To address this concept, COSEML utilizes a single hierarchy diagram in which abstract decomposition and component composition are shown together. COSEML addresses both abstract and physical components. The higher-level elements represent the abstractions for package, data, function, and control. Lower-level elements correspond to components and interfaces. Figure 2 depicts the symbols used in COSEML.

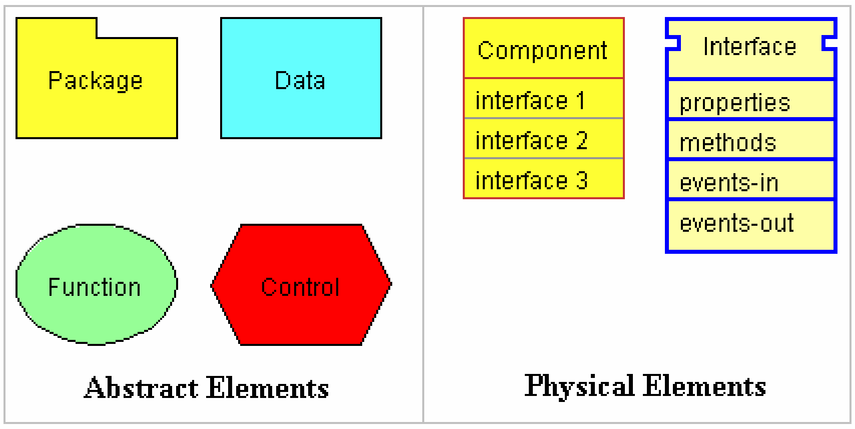


Figure 2. COSEML symbols

Physical component and interface symbols are created while the other appropriate symbols are taken from UML.

These symbols are defined in [6] as follows. Package abstraction groups related elements in an encapsulation. A package can contain further package, function, data, and control abstractions. It is the fundamental structural element used in the definition of part-whole relations. System decomposition is made using packages and decomposition is detailed using the other abstractions. Data abstractions represent data structures. In the requirements model they can model high-level entities. Function abstractions represent high-level system functions. Control abstractions are state machines, accepting messages that cause state changes and outgoing messages. State changes can trigger the execution of function abstractions or of operations inside data abstractions.

Components and their interfaces are represented with symbols as the lowest-level elements. Figure 2 also shows the graphical representation of physical-level elements of COSEML.

There are also connectors to represent communications among components, both in abstract and physical levels. In abstraction, a connector represents many message or event connections between two components. A connector between two components is still an abstract element. It represents at least one, but possibly more than one message (or event) link. A message link represents a function call (local or remote) originating in one component and terminating at the interface of another. Events are similar to messages but semantically they stand for calls initiated by external causes in contrast to calls made under program control. Other than this categorization, messages or events are similar in the way they are represented.

COSEML mainly focuses on the structural decomposition. Components, which represent the decomposed modules, are also showed on the model. In this thesis work, modeling with COSEML is extended to show the interactions among those components

CHAPTER 3

COLLABORATION DIAGRAMS IN COSEML

In this work, two types of collaboration diagrams, abstract and run time collaboration diagrams are proposed. COSEML shows abstract decomposition of the system and corresponding real components together on the hierarchical diagram. Emphasis on these two views is supported by defining these two collaboration diagrams.

In this chapter, a specification is given for their use in COSEML. After that, possible benefits of collaboration diagrams for COSE modeling are explained.

3.1. Specification

On the definition of collaboration diagram specification, applicable rules and methods are adapted from UML. Other rules are defined by considering the COSE approach and the current state of COSEML.

Specification for Abstract Collaboration Diagrams and Run Time Collaboration Diagrams differs only on the element types that can be added to the diagram. Other specifications such as sequence numbering, conditional messages, loop structure or message types are all common for both of the diagram types. These are defined in the following sub sections.

3.1.1. Abstract Collaboration Diagrams

In abstract levels, abstract collaboration diagrams are utilized for supporting the decomposition model of the COSEML. High-level requirements and the behavior of the system can be modeled using this type of collaboration diagram. Utilizing these diagrams can help to find incompleteness and inconsistency in scenarios and requirements in the analysis phase. Thus, they are useful for testing the correctness of the structural decomposition. Adding that, they do not contain any technical or implementation detail. Therefore, they can help to create a common understanding of a system behavior between different users, ranging from domain experts to end-users.

In this diagram, only abstract elements of COSEML are allowed that exist in the main hierarchy diagram. If there is a need to add a new element to the collaboration diagram, then hierarchy diagram should be reconsidered and this new element should be included. Only after that, it can be used in the collaboration diagram. This process, which improves the decomposition model, is shown in figure 3. Any element that is required in the dynamic behavior is forcefully added to the decomposition model.

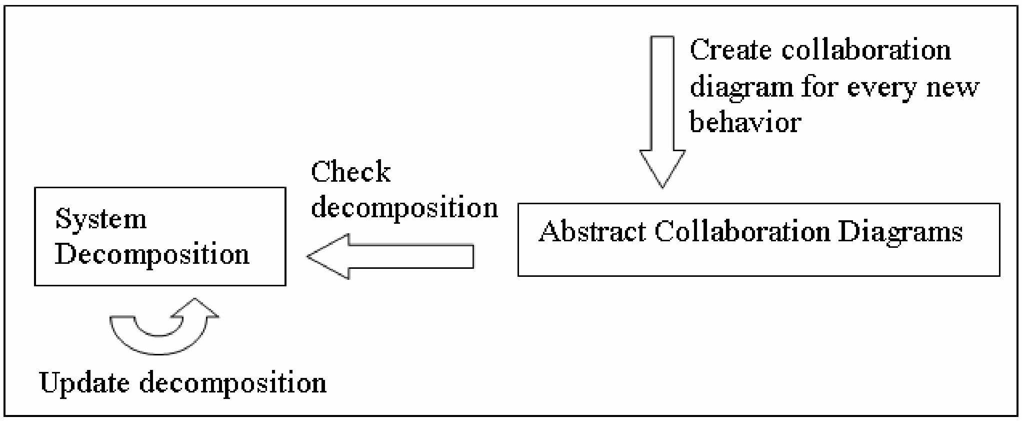

Create collaboration diagram for every new behavior
Check decomposition
System Decomposition
Abstract Collaboration Diagrams
Update decomposition

Figure 3. Decomposition checking using abstract collaboration diagrams

3.1.2. Run Time Collaboration Diagrams

For the physical level, run-time collaboration diagrams are utilized. These diagrams show behavioral aspects of the system on component level. Interactions among interfaces are shown using sequence of method calls and event signals in the diagram. Showing these interactions permit to model the implementation of complex operations. Run time collaboration diagrams are intended for the implementation phase and they mostly contain implementation details. These diagrams are suitable for making wiring-level decisions on the model.

Modeling elements in this kind of collaboration diagrams are only the component interfaces. An interface should exist in the hierarchy diagram in order to be used in runtime collaboration diagram. If a behavior cannot be modeled using existing interfaces, new components and interfaces should be found and added. This rule improves the composition model of the hierarchy diagram by enforcing the addition of new components to fulfill a required behavior. This diagram type permits modeling complex implementation details on the model.

3.1.3. Sequence Numbering

Interactions in a collaboration diagram are shown using sequence of messages among the elements. Order of the messages is indicated by the sequence numbers defined with the messages. UML suggests nested decimal numbering for sequence numbering [4]. However if there are two many nested calls, message numbers can easily get complicated (ex: 1.2.1.1.3). Most of the model designers prefer flat numbering on collaboration diagrams. This notion is straightforward and easy to follow. Figure 4 shows this notion. On the other hand, flat numbering has problems on some situations.

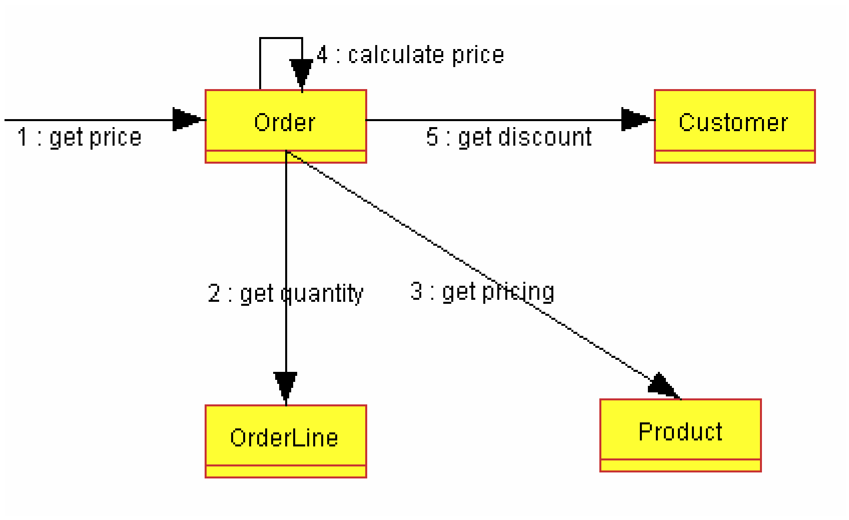


Figure 4. Use of flat numbering on message sequences

In figure 4, it is not clear whether *get discount* is called within *calculate price* or within the overall *get price* method. It is not possible to know if a message is an inner call or not. Nested calls cannot be tracked with flat numbering. Nested decimal numbering scheme solves this problem. Figure 5 shows the same diagram with this numbering scheme.

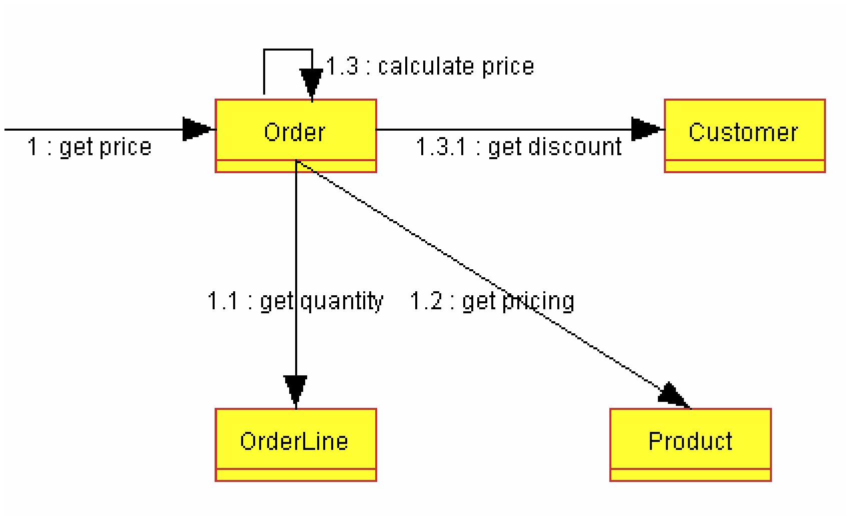


Figure 5. Use of nested decimal numbering on message sequences

In figure 5, it is clearly seen that, *get price* is the main method and others are inner method calls. It is also apparent that *get discount* method does not belong to the main *get price* method. Nested numbering resolves ambiguity and it also covers the flat numbering notion. For these reasons, nested numbering notion is selected for message sequencing in COSEML.

3.1.4. Conditions and Loops

Conditional flows are the essential part of dynamic modeling so they are defined in the specification. A conditional message is denoted by putting the conditional expression between square brackets as shown in figure 6.

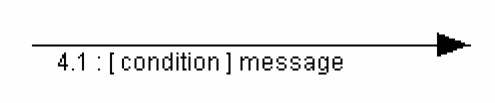


Figure 6. A conditional message

This visualization represents a conditional message, which is activated if and only if the condition between the square brackets evaluates to "true". Like UML 2 specification, no restriction is defined on the use of logical expressions between the brackets.

Any conditional branching can be simulated using conditional messages and concurrency together. Figure 7 shows *if / else if / else* branching on a collaboration diagram.

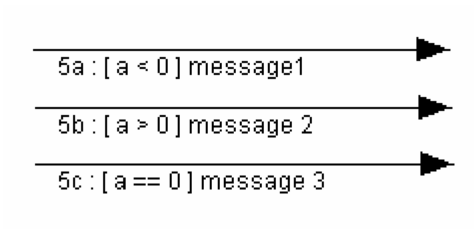


Figure 7. Conditional branching

It is ensured that, only one of the messages in figure 7, which satisfies the condition, is going to run at time t1. By designing proper message conditions, any kind of complex branching is possible.

Another structure, which is also essential for modeling complex flows, is loop structure. When one or more messages are called more than once, these structures are needed. Notation is similar to that of a conditional message; difference is the asterisk in front of the brackets as seen in figure 8.

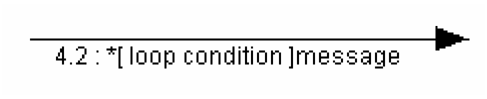


Figure 8. A message with a loop condition

Condition between the square brackets is the loop condition and while it evaluates to true, that message and sub messages are iterated. In order to iterate a group of messages, setting a loop condition to the parent message is sufficient.

To show the usage of conditional and loop structures, an example run time collaboration diagram of an automatic door application is shown in figure 9.

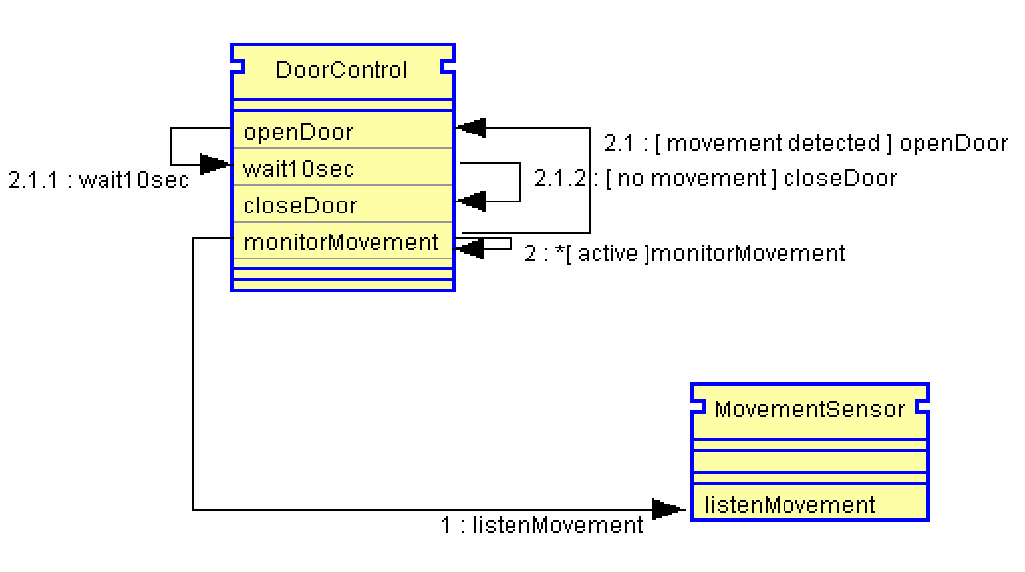


Figure 9. Run time collaboration for automatic door application

In step 1, the *DoorControl* receives movement event from the *MovementSensor*. Step 2 is a message with a loop condition. While the system is active, *monitorMovement* message is called continuously. In a continuous manner, if there is a movement then controller opens the door (2.1). Following that, door keeps open for 10 seconds (2.1.1). Then, if there is no movement at that time, controller closes the door (2.1.2). Until system became passive, step 2 and its sub steps called continuously.

When the *MovementSensor* senses a movement, it informs the listeners and consequently door opens. Table 2 shows the lifetime of the messages.

Table 2. Lifetime of the messages in automatic door application

Sequence	**When called**
1	One time at the start
2	While the door is active
2.1	While the door is active and there is movement
2.1.1	When 2.1 called
2.1.2	When 2.1.1 called and there is no movement

3.1.5. Message Types

The links between the participants in the diagram are annotated with the messages. Messages can be synchronous, asynchronous or flat. Figure 10 shows the message symbols used in COSEML collaboration diagrams.

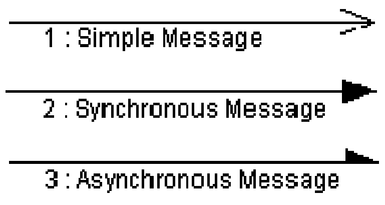


Figure 10. Message types

Different kinds of arrows allow distinguishing between message types. The normal arrowhead stands for simple message, filled arrowhead for synchronous and the half filled arrowhead for asynchronous one.

Concurrent messages between the interfaces are also supported. All message types can also be concurrent. Modeling concurrency is needed especially in distributed, multithreaded and reactive systems. Concurrency is shown by using alphabetic characters next to the sequence numbers. Figure 11 shows two concurrent messages on the first sequence level.

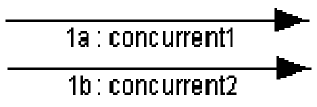


Figure 11. Concurrent message types

In a synchronous message, caller who sends it must wait until the message is done, such as invoking a subroutine. If a caller sends an asynchronous message, it can continue processing and does not have to wait for a response. Concurrent messages are used when a caller needs to send more than one message at the same time. Supporting synchronous, asynchronous and concurrent type of messages is important for modeling complex operations. Asynchronous and concurrent messages are used especially in multithreaded and message oriented applications. It is not possible to model real applications without having such message types on the diagram.

An example model can clarify the necessity of asynchronous messages. Figure 12 shows a collaboration diagram of a car rental scenario [5]. Here all messages are modeled as synchronous which means a message call waits until previous message completes. In the scenario, a customer comes to car rental office and requests a car (1:*walkInRequest*). After that, the desk first, checks the credibility of the customer (1.1:*checkCredibility*), second, it checks the garage for the availability of the requesting car (1.2:*checkAnswer*) and then keeps the availability info for making decision (1.3:*keepAvailAnswer*). If nothing is wrong after those processes, customer gets information about insurance (2:*getInsuranceInfo*). If she finds the insurance plan suitable, then she asks for picking up the car she selected (3:*pickUpCar*). Following that, desk selects the car (3.1:*selectCar*) and takes it from the garage (3.2:*carTaken*).

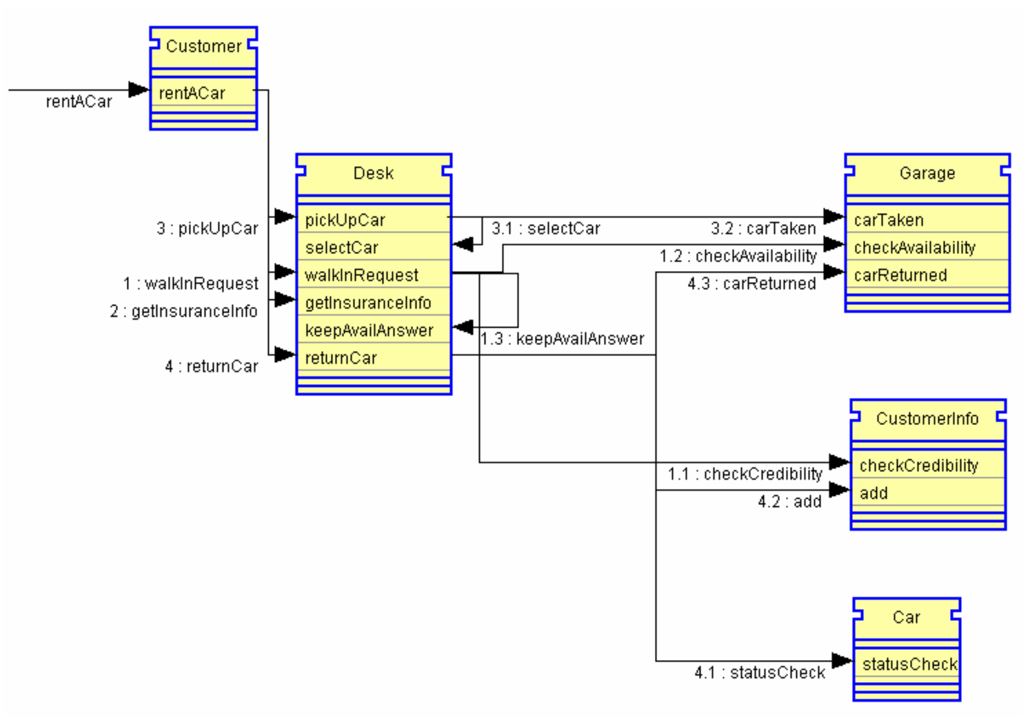


Figure 12. Car rental scenario

At the end of the rental period, customer returns the car to the office (4:*returnCar*). Then desk checks the car (4.1:*statusCheck*), adds information about the rental to customer's file (4.2:*add*) and returns the car to the garage (4.3:*carReturned*).

In the model, it seems that, *getInsuranceInfo* cannot be called until *walkInRequest* operation is completed. However, these are independent operations. Operation *pickUpCar* is dependent only on the answer of the *walkInRequest* operation. This model can be fixed using asynchronous messages. Figure 13 shows the revised diagram.

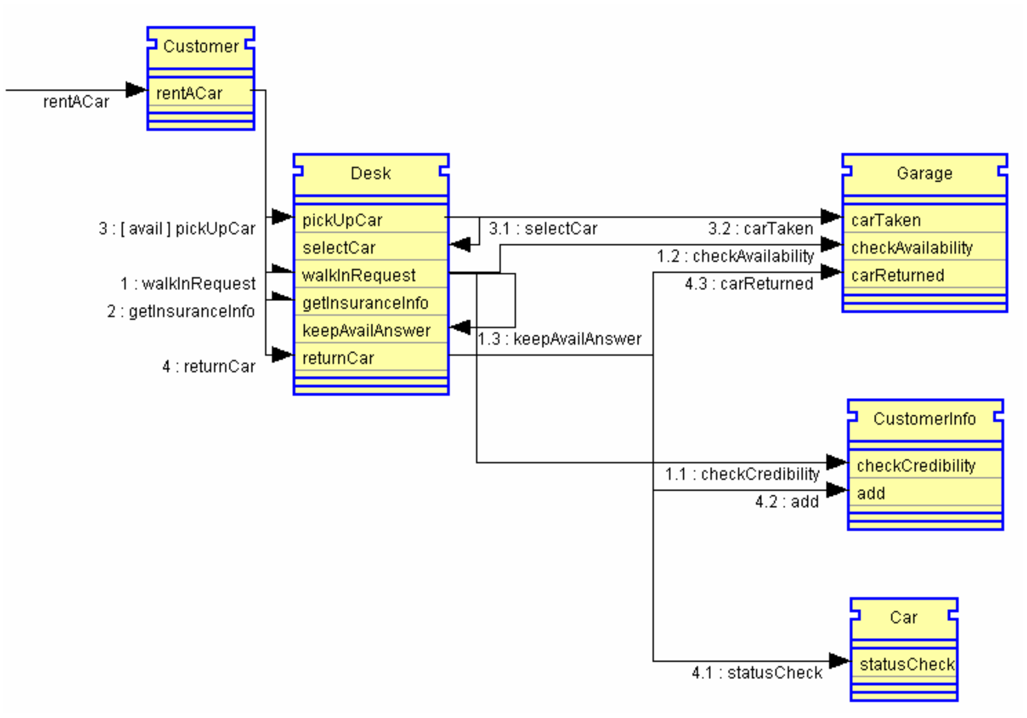


Figure 13. Car rental scenario with asynchronous messages

In figure 13, *walkInRequest* and *getInsurenceInfo* messages are made asynchronous. This means, customer can get information about insurance while desk processes the rental request. In addition, customer can pick up the car as soon as request operations returns positive answer (1.3:*keepAvailAnswer*). Asynchronous method calls improves the performance. Modeling a behavior with a collaboration diagram and examining the interactions make it easy to figure out the possible asynchronous method calls.

Following example scenario is given to show the concurrency modeling. Figure 14 depicts the abstract collaboration diagram of this scenario.

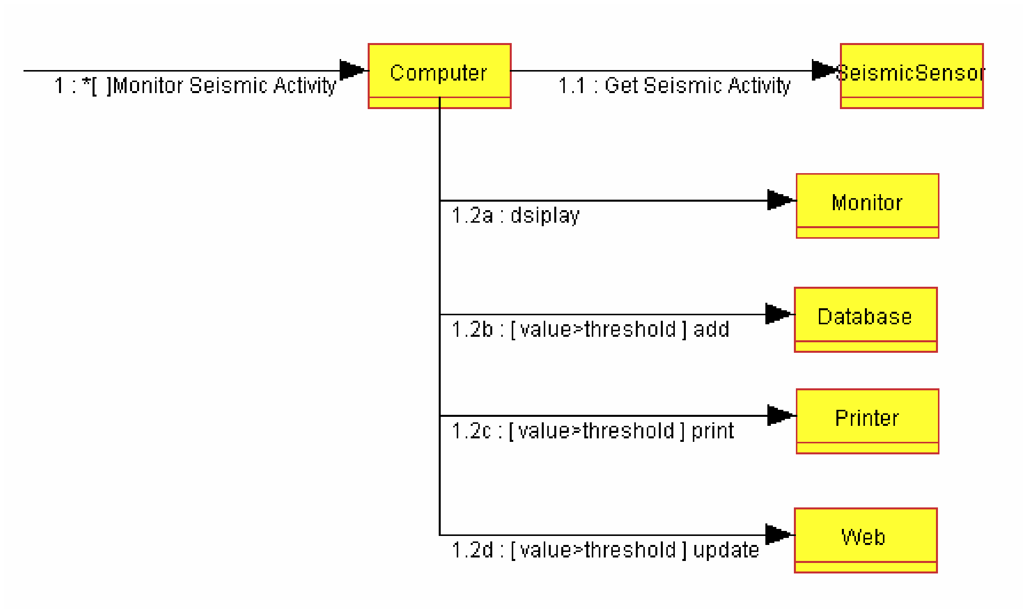

1 : *[ ]Monitor Seismic Activity
Computer
1.1 : Get Seismic Activity
SeismicSensor
1.2a : dsiplay
Monitor
1.2b : [value>threshold] add
Database
1.2c : [value>threshold] print
Printer
1.2d : [value>threshold] update
Web

Figure 14. Seismic activity monitoring scenario

This scenario describes the steps of monitoring seismic activity. A seismic sensor continuously sends seismic activities to the central computer (1.1 at time t1). Computer shows it on the monitor (1.2a at time t2) and if it is larger than a threshold value, this data is added to database (1.2b at time t2), printed (1.2c at time t2) and updated on the web page (1.2d at time t2) concurrently.

Concurrency is important in multi threaded and reactive applications. However, concurrency is error prone especially on shared resources. For this reason it is useful to model such behavior and study all possibilities on the model.

3.2. Benefits of Collaboration Diagrams to COSE Modeling

In this section, possible benefits of collaboration diagrams to COSE modeling are explained.

3.2.1. Use Case Realization

Use cases are a technique for capturing the functional requirements of a system. They describe the typical interactions between the users of a system and the system itself, and provide a description of how a system is used [16].

A use case captures the intended behavior of the system. It specifies this behavior using sequences of actions. They can be considered as the foundation for the rest of development process. As the system evolves, they help to validate the development. They also provide a common understanding of the system to developers, end users and domain experts. For these reasons, use case approach plays a key role in a software development process.

Use cases do not specify implementation details. However, they have to be implemented and in COSE, implementation is the connection of the components through interfaces. For this purpose, collaboration diagrams, which show the sequence of connections, are very suitable for use case realization. They provide a complete path for the realization of use cases [17].

Two collaboration diagram types, introduced in this thesis can be used for use case realizations in different levels.

Abstract collaboration diagrams are closer to the analysis phase of the development. This diagram type describes the externally visible actions and their sequences and it does not get into the implementation details. For this reason, use case realizations with abstract collaboration diagrams are useful for validating and improving abstract decomposition of the system and requirement refinement. In addition, they are useful for showing the behaviors of the system.

On the other hand, run time collaboration diagrams are closer to the integration phase of the COSE. When they are used to describe the realization of a run time level use case, they provide information that is more specific and detailed. Sequence of method calls, event subscriptions and data flow between the interfaces of components, can be modeled using run time collaboration diagrams.

Following section is a textual description of the use case of a mobile translation service example [13]. In this use case scenario, a picture message, containing an English text, is translated to Turkish and sent to the customer.

Translate Picture Message

Main Path:

1. Customer takes a picture of an English text with mobile phone.
2. Customer sends the picture to the translation service.
3. Service processes the image and converts it into text.
4. Service translates the text to Turkish language.
5. Service sends this translated text to the customer as an SMS.
6. Service bills the customer for 10 SMS.

Extensions:

3a. Service could not recognize image

3a.1. Service bills the customer for 1 SMS.

3a.2. Service sends error message to the customer as an SMS

4a. Service could not translate text.

4a.1. Service bills the customer for 1 SMS

4a.2. Service sends error message to the customer as an SMS

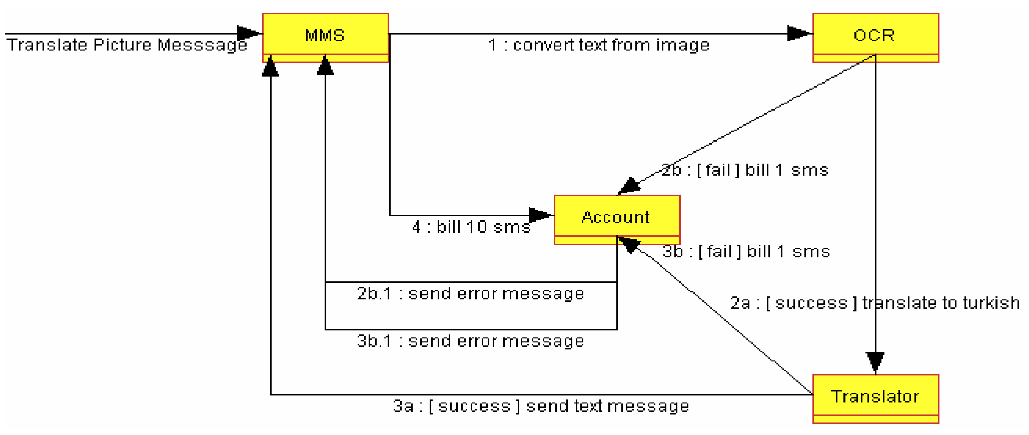

Translate Picture Message
MMS
1 : convert text from image
OCR
2b : [fail] bill 1 sms
Account
4 : bill 10 sms
3b : [fail] bill 1 sms
2b.1 : send error message
2a : [success] translate to turkish
3b.1 : send error message
Translator
3a : [success] send text message

Figure 15. Abstract collaboration diagram of the mobile translation service

Abstract elements in this scenario are *MMS*, *OCR*, *Translator* and *Account* components [13] as show in the collaboration diagram in figure 15. This diagram presents a graphical description of the use case. It shows the high-level system functionality and hides the implementation details. This abstract picture of the use case helps the system designer to review and validate the scenario. End users, domain experts and developers can efficiently negotiate system requirements on this diagram.

To build the system, behaviors should be implemented. Implementation phase in COSE is indeed the integration of the components. Run time collaboration diagrams are very suitable environment for integrating the components through their interfaces. Figure 16 shows the run time collaboration diagram of the "Translate Picture Message" use case.

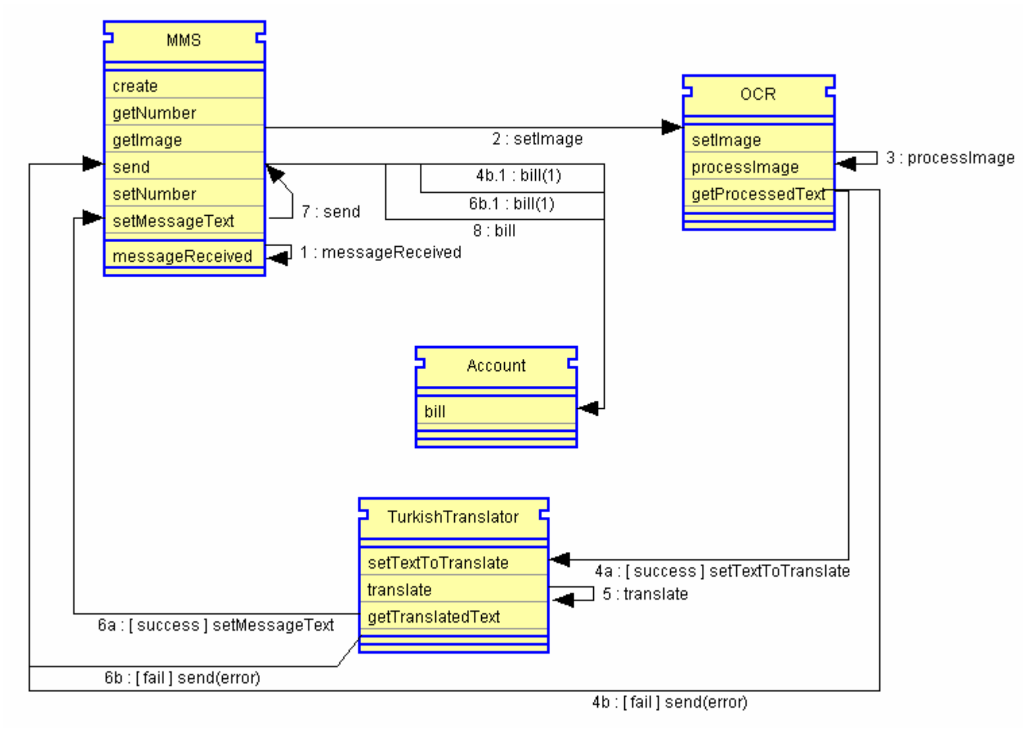


Figure 16. Runtime collaboration diagram of the mobile translation service

As it is clearly seen from the figure, implementation level details can be modeled using run time collaboration diagrams. Method calls, event subscriptions and data flows can be modeled. In addition, implementation level issues such as concurrency, asynchronous calls, conditional calls, looping are supported. Such supports enable the construction of wiring-level decisions. Furthermore, these diagrams can be extended so that they can be used for application generation from the model.

3.2.2. Improved Hierarchy Diagram

COSEML emphasizes modeling the structural view of the system. The hierarchy diagram, which was the only diagram in COSEML, has been used for abstract decomposition of the system. This diagram also shows the components and component compositions that represent the decomposition elements. It shows a static model of the system. However, accuracy and efficiency of the static models cannot be proven true without the help of dynamic models [18]. To prove this claim, a telephone system given in [18] is modeled with COSEML.

A telephone system consists of elements like speaker, microphone, buttons, dialer, display and a network. Therefore, the first decomposition model that comes to mind is, creating a telephone entity and connecting mentioned elements using composition links as shown in figure 17.

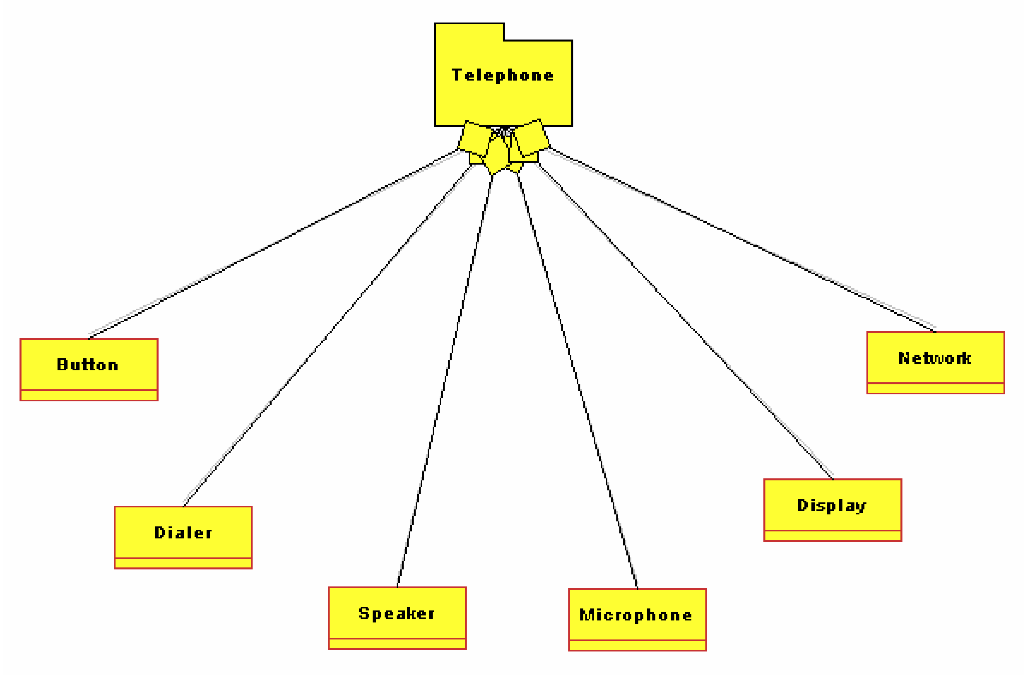


Figure 17. Decomposition model of Telephone System

This model shows the components in a telephone system and it seems a valid static model. However, as explained above, without examining a dynamic model of the system, one cannot ensure that this model is valid. A system is not only a static structure; its functioning is determined by the dynamic aspects. For this reason dynamic behavior should be investigated.

Dynamic behavior is explored by use case analysis. Most obvious function of a telephone system is calling a person. Therefore, studying the “Call a Person” use case is appropriate. Steps of this use case are as follows.

- Caller presses the buttons on the telephone to call the desired number
- Caller presses dial button.
- Dialer dials the number.
- While dialing, dialer creates a tone on the speaker.
- While dialing, dialer display digits on the display.
- Dialer sends the number to the network.
- Network sends ringing status tone to speaker.
- When called person responds, network sends the voice to the speaker.
- Caller speaks to the microphone
- Microphone sends the voice to the network.
- Caller presses the close button
- Dialer sends close message to network and call ends.

Figure 18 shows the representing collaboration diagram. This diagram clearly shows the sequence of actions and communication among components. For example, it is obvious that Button only interacts with Dialer and isolated from the rest of the system. Similarly, Microphone component interacts only with Network. The previous static model in figure 17 was not containing such information. For this reason, previous decomposition model was not complete.

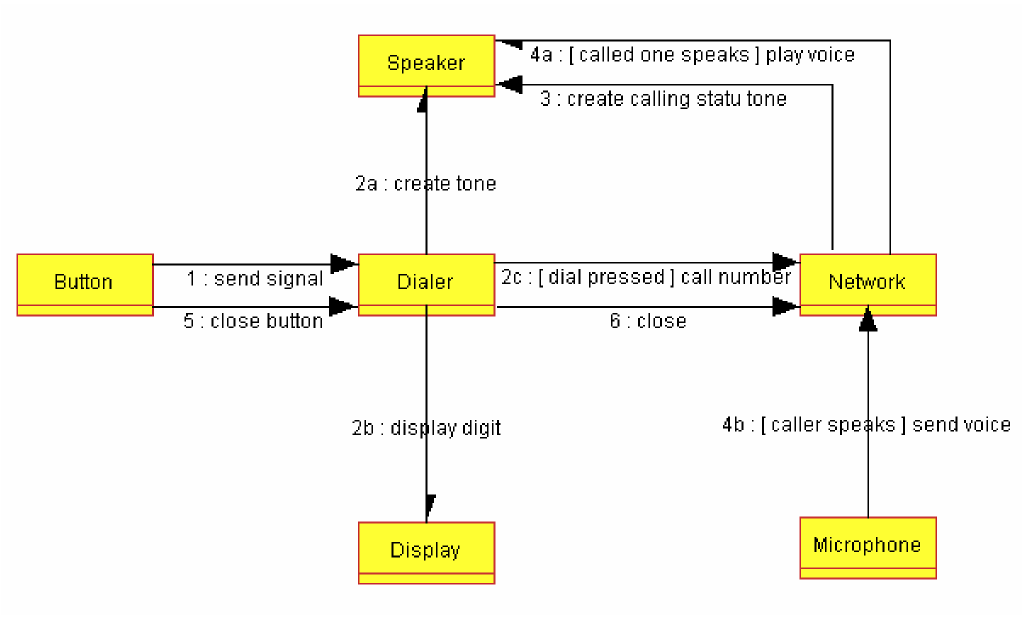

Speaker
4a : [called one speaks] play voice
3 : create calling statu tone
2a : create tone
Button
1 : send signal
5 : close button
Dialer
2c : [dial pressed] call number
6 : close
Network
2b : display digit
4b : [caller speaks] send voice
Display
Microphone

Figure 18. Collaboration diagram of "Call a Person" use case

After observing the dynamic behavior using the collaboration diagram, changing the previous decomposition model is straightforward. Figure 19 shows the improved decomposition model of the telephone system. This decomposition model now contains more information about the system. It is clear that, modeling the dynamic behavior of a system with logic level collaboration diagrams helps to validate and improve the abstract decomposition part of the hierarchy diagram. As the number of abstract collaboration diagrams increase in the model, decomposition part becomes more complete. What this means is improving the static model by exploring behaviors of the system.

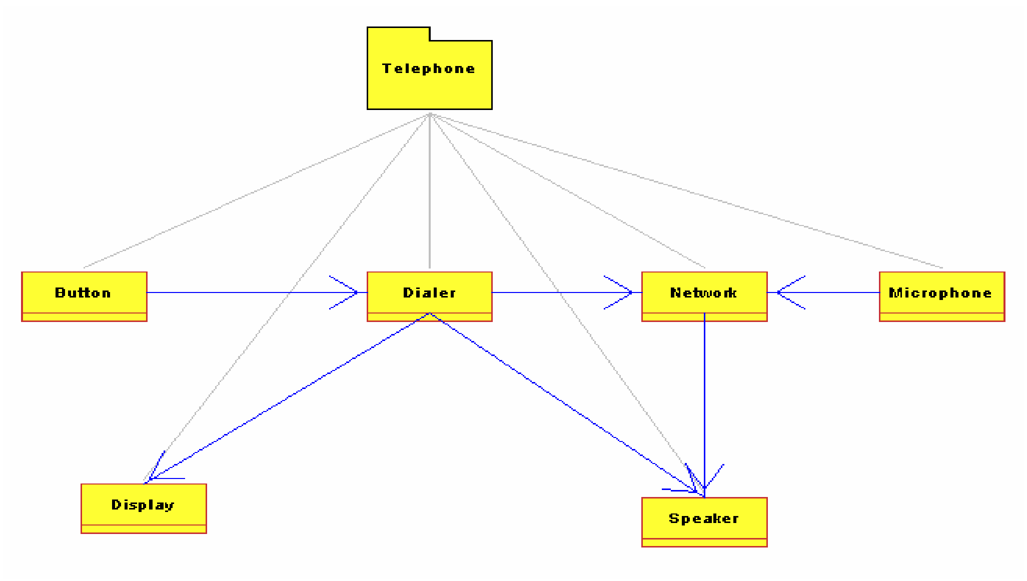


Figure 19. Improved decomposition model of Telephone System

This is also true for the component composition part of the hierarchy diagram of COSEML. In that part, links are used to show the relations among interfaces. These links can be improved by exploring the dynamic behavior at the run time level. Run time collaboration diagrams, which show the sequence of messages among the component interfaces, can be utilized for this purpose.

A static model that is produced without the benefit of dynamic analysis is bound to be incomplete. The appropriate static relationships are a result of the dynamic needs of the application. Collaboration diagrams are a good way to depict dynamic models and compare them to the supporting static models [18].

3.2.3. Automated Software Test

There is an increasing need for effective testing of software. In the domain of military and e-government applications, reliability and robustness of the software is very important. Some software testing approaches focus on test generation from source code. However in the component oriented approaches, source code is hidden in a black box, which is not accessible for testing purposes.

In [17], advantages of generating test data from high-level design notations over code-based generation were proposed. It is claimed that, one of the major costs of testing can be significantly reduced by using design notations as a basis for output checking. Design problems can also be discovered within such a testing process. This eliminates the problems in the early stages, which means saving time and resources. In addition, early testing allows more effective planning and utilization of resources. It is also underlined that testing from design makes the testing process independent from any particular implementation of the design.

Such design oriented testing approach is adaptable to the work cited in this thesis. Collaboration diagrams represent a significant opportunity for testing because they precisely describe how the provided software functions are connected in a form that can be easily manipulated by automated means [17]. Static and dynamic testing of the system can be handled using abstract and run time collaboration diagrams that are provided with this work. Tests can be generated automatically from these diagrams. Such an approach, the utilization of COSEML collaboration diagrams for software testing, creates an open research area in COSE.

3.2.4. Automated Application Generation

There is an increasing interest on the idea of creating applications from the software models both in the research community and in the software industry. Model Driven Architecture (MDA) is currently the most popular approach on this subject. The aim in MDA is to build systems from models, which is independent from implementation technology [14]. From a given model, code is generated by MDA-CASE tools. However, there is a problem in this approach. Quality, robustness and reliability of the generated software are left to the tool, which cannot guarantee the standards required by the current software systems.

In the next generation of software engineering, existing components should be used and there should be no coding. Components have proven quality, robustness and reliability. Therefore, application generation approach could utilize existing components as an alternative to the model transformation. Collaboration diagrams proposed in this thesis provide a good opportunity for application generation from a model using components.

In [19], collaboration diagrams were proposed for java code generation from the model. Their appropriateness for code generation is also explained in that work. COSEML collaboration diagrams are built on the component approach and the problems cited in [19] are not valid for these diagrams. There are no low-level coding issues in a component-oriented approach. This makes use of COSEML collaboration diagrams very feasible for application generation.

Run time collaboration diagrams enable inputting all the necessary information such as event transactions, sequence of method calls, control and loop structures. With this information, a run time collaboration diagram can generate a specific functionality of the system. Later these generated application parts can be combined to generate the final application.

CHAPTER 4

IMPLEMENTING COLLABORATION DIAGRAMS IN COSECASE

Before this work, COSECASE was based on a single diagram concept. For this reason, implementation was not generic enough to extend the editor and to support multiple diagrams. Another problem was, high coupling among the irrelevant types of classes. Any code change made on a single object was affecting the many others. Furthermore, readability of the source codes was poor. Since more than one developer had worked on the development on different times, there was no consistent code style. Variable naming was improper; there were unused variables and methods. In addition, objects in the code were taking irrelevant responsibilities. Such factors were complicating the development efforts. These problems are investigated with the help of Eclipse, which is a highly functional development environment.

After solving these problems, some sorts of precautions are taken for preventing similar problems in future. One of the most important problems is the versioning. COSECASE is a research project and there is an ongoing development on it. Multiple developers are working on this project at different times and a versioning system is mandatory. To solve this problem, source codes of COSECASE are put into an online repository, which supports version controlling. In addition, requirement and feature tracking is made possible on this online repository.

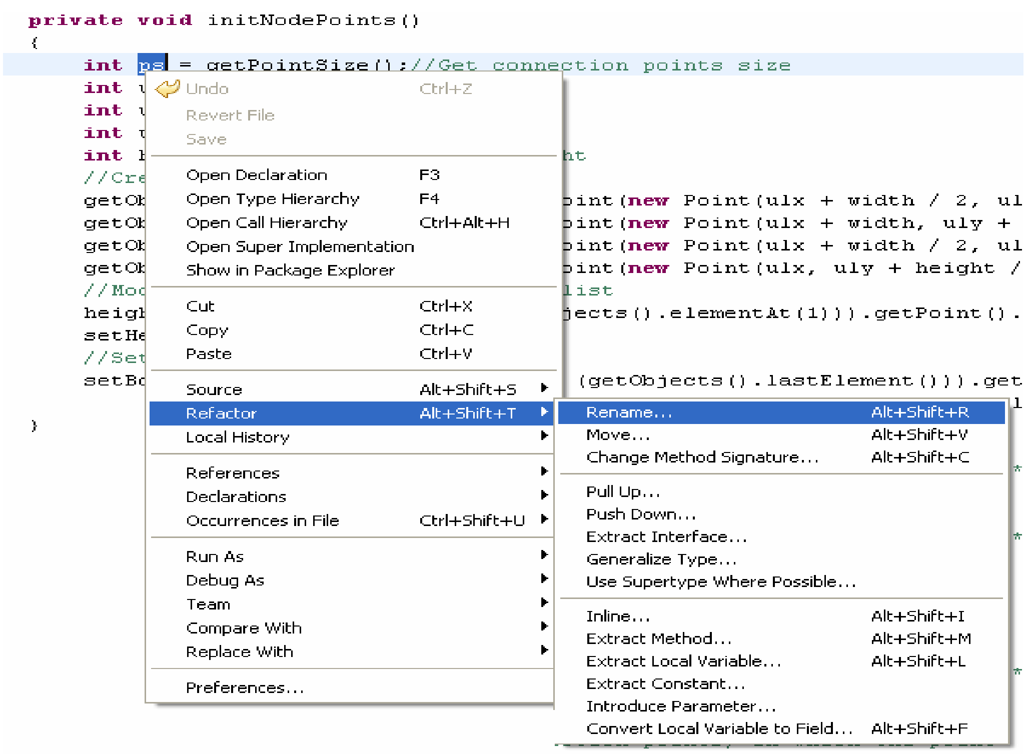
After creating a robust foundation for development, new features that are required by collaboration diagrams are added to the modeling tool. These are explained in this chapter.

4.1. Code Review and Improvements on COSECASE

In the previous version, development was made on the single hierarchy diagram idea. For this reason, most of the objects were designed around this single diagram concept. Even the simplest objects, symbol and link objects were keeping the reference to the hierarchy diagram. These objects were accessing the global resources provided by this diagram. To support new diagram types, these objects should be made independent. An object should not keep the reference of an unrelated object. For these reasons, all the unrelated references are removed. While doing that, some business functionality were also moved to more responsible objects.

4.1.1. Code Improvements with Eclipse

Eclipse is an open source software development project, which provides a high quality, full-featured Integrated Development Environment (IDE) [20]. This IDE has a high level of code re-factoring support.

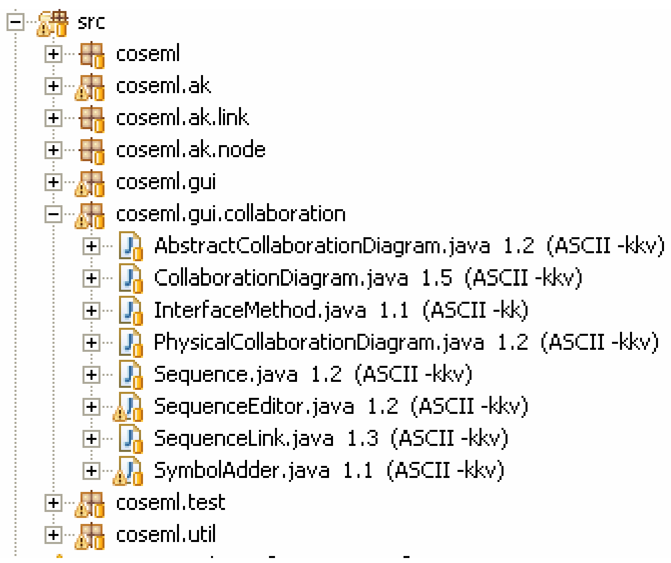


```
private void initNodePoints()
{
        int ps = getPointSize();//Get connection points size
        int u
        int u
        int u
        int h                                       ht
        //Cre
        getOb                                       oint(new Point(ulx + width / 2, ul
        getOb                                       oint(new Point(ulx + width, uly +
        getOb                                       oint(new Point(ulx + width / 2, ul
        getOb                                       oint(new Point(ulx, uly + height /
        //Mod                                       list
        heigh                                       jects().elementAt(1))).getPoint().
        setHe
        //Set
        setBo                                         (getObjects().lastElement())).get
}
```

Figure 20. Eclipse development environment

Figure 20 shows a snapshot of the development environment. Source codes of COSECASE are reviewed using rich set of code improvement functionalities provided by this environment. There were unused variables and methods in the code. They were making the code unreadable and difficult to understand. Using eclipse, they are detected and totally removed from the source code. Another readability problem was the format of the codes. Different parts of the software were having different code style. With this tool, all of the source codes are reformatted using the conventional code styles.

High number of source files was another problem. It was difficult to distinguish which classes were related. This problem also is solved by creating logical groups of packages and putting the related classes to the same packages. Figure 21 shows the use of packages.



```
src
  coseml
  coseml.ak
  coseml.ak.link
  coseml.ak.node
  coseml.gui
  coseml.gui.collaboration
    AbstractCollaborationDiagram.java 1.2 (ASCII -kkv)
    CollaborationDiagram.java 1.5 (ASCII -kkv)
    InterfaceMethod.java 1.1 (ASCII -kk)
    PhysicalCollaborationDiagram.java 1.2 (ASCII -kkv)
    Sequence.java 1.2 (ASCII -kkv)
    SequenceEditor.java 1.2 (ASCII -kkv)
    SequenceLink.java 1.3 (ASCII -kkv)
    SymbolAdder.java 1.1 (ASCII -kkv)
  coseml.test
  coseml.util
```

Figure 21. Re-factoring functionalities in Eclipse

With this package use, classes, which are related to a specific functionality are grouped so that a novice COSECASE developer can easily locate them.

To sum up, readability and understandability of the COSECASE codes are highly improved using the functionalities provided by the Eclipse development environment.

4.1.2. CVS

Concurrent Versions System (CVS) is an open source version control system [21]. It is widely used by software development teams.

Version controlling is an important component of source code management and is necessary even for a single developer. Changes made on source codes may need to be rolled back. In addition, before a risky code review, a tag may be given to a group of source files. This is similar to creating a backup disk and putting it in a safe place. When there are multiple developers, situation is more critical as in the case of COSECASE. Changes made by a developer should be synchronized by others in order to keep the integrity of the software. When multiple developers work on a single source, conflicts can be solved and changes can be merged by using the facilities provided by CVS.

There has been no version controlling on the development of COSECASE. As a result, there have been different versions of COSECASE that have different functionalities, different bugs and different solutions. This was slowing the development of COSECASE project. For these reasons, source files of COSECASE were put on an online CVS server for the sake of current and future development of the modeling editor.

4.2. Implementation of Collaboration Diagrams

Previous version of the COSECASE was built on the single hierarchy diagram concept. There was a single diagram class, *CosemlDrawPanel*, which was handling all of the modeling work. This class is replaced by a hierarchy of classes. New structure supports collaboration diagrams and other types of diagrams to be used in COSECASE. Figure 22 shows the new implementation of the diagrams in COSECASE. *DrawPanel* is the base class and it contains the common properties of diagrams that can be used in the tool. It supports use of any modeling symbols without any constraints. Also all common drawings, mouse and keyboard events are handled in this class.

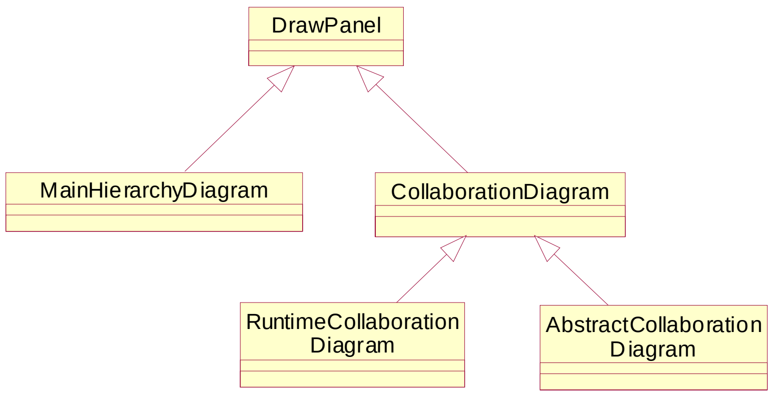


Figure 22. New diagram structure in COSEML

As its name implies, *MainHierarchyDiagram* is the new implementation of the previous *CosemlDrawPanel* class. Its functionality and the user interface remained same. In addition, two new diagram types, *RuntimeCollaborationDiagram* and *AbstractCollaborationDiagram* are added to the diagram structure.

To enable browsing the diagrams in the model, a component, which is a floating window, is designed. This component, *DiagramTree*, is a *JTree* structure that contains a main hierarchy diagram and zero or more collaboration diagrams. Other than browsing, *DiagramTree* also responsible for adding and managing diagrams. New collaboration diagrams can be added; existing ones can be renamed or removed. On the other hand, main hierarchy diagram can only be renamed. It cannot be removed from the diagram. Functionalities such as duplicating a collaboration diagram or saving it to disk are also supported in the *DiagramTree*.

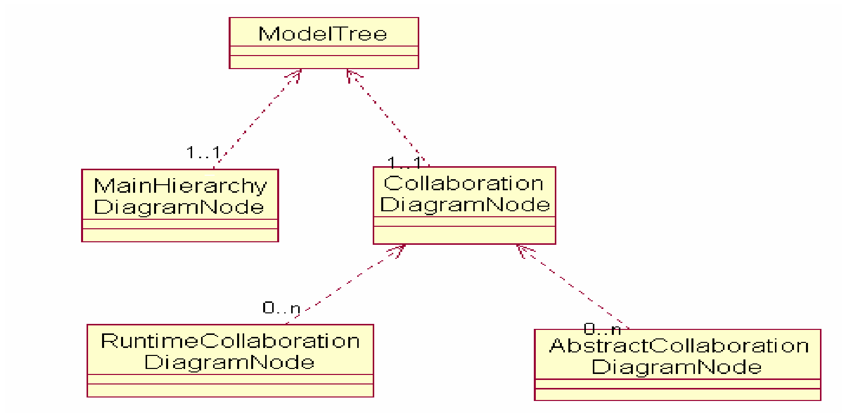

ModelTree
1..1
1..1
MainHierarchy DiagramNode
Collaboration DiagramNode
0..n
0..n
RuntimeCollaboration DiagramNode
AbstractCollaboration DiagramNode

Figure 23. Diagram model contained in *DiagramTree*

Tree model of the *DiagramTree* is shown in figure 23. All these functionalities are accessed via pop-up menus to save space. In addition, *DiagramTree* can be made invisible with a menu item defined in View Menu.

Each model in COSECASE has a default hierarchy diagram. Collaboration diagrams are added by the model designer and they are optional. Model can have as many collaboration diagrams as needed. Figure 24 shows a snapshot of the *DiagramTree* on the COSECASE. In figure 24, diagram with the name "Online Banking" is the main hierarchy diagram and it is always shown at the top of the diagram tree.

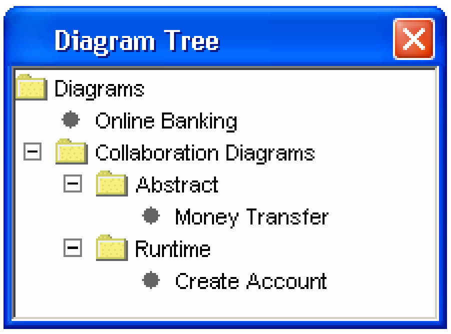


Figure 24. *DiagramTree*

While implementing collaboration diagrams, some infrastructural changes are made to COSECASE. For those changes, general diagramming guidelines [22] and UML collaboration diagrams are observed.

4.2.1. Links

Links are used to connect elements for showing relations between them. A class diagram for the old link classes of the previous COSECASE is shown in figure 25.

In this work, two major additions are made to the link structure. First is the support for segmented-lines. One of the diagramming principles suggests avoiding diagonal and curved lines for connecting elements. They are difficult to follow on a diagram and model can easily become complicated. For this reason segmented-lines are implemented.

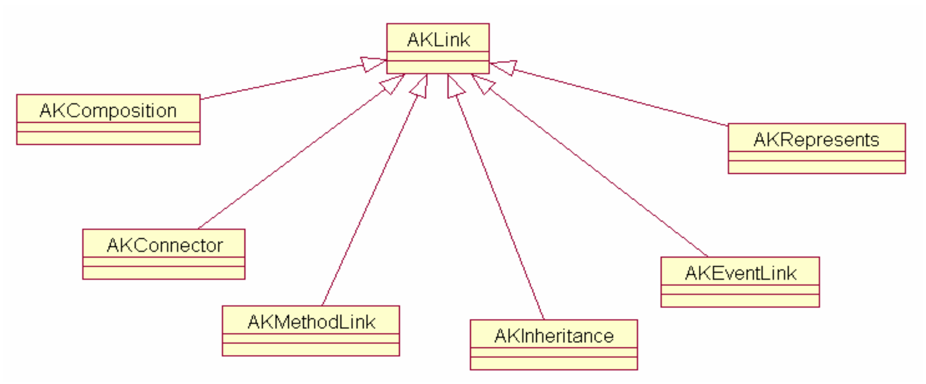


Figure 25. Old structure of link classes in COSEML

Figure 26 shows a segmented link. Segmented-lines allow creating a line with multiple joint points. They allow creating flexible paths on the model. In addition, auto straightening functionally is added to the segmented-lines. This also improved the ease of connecting elements on the diagram. Joint points are implemented by segment drag points. A segment drag point can be added or removed by double-clicking a point on the segmented-line. Then the line can be shaped by dragging these points using mouse.

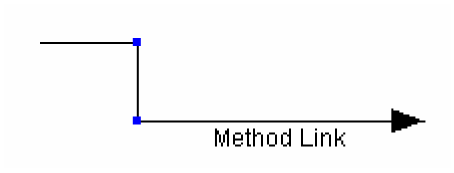


Figure 26. Segmented links

Second addition is the support for sequencing messages. They are required for showing the sequence of messages in a collaboration diagram. Figure 27 shows a sequence message.

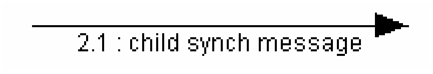


Figure 27. A sequence message

New classes are created for supporting sequence messages and segmented links. Figure 28 shows the updated state of link classes. As seen in the figure, all previous link classes are made segmented by extending from the *SegmentedLink* class. In the implementation of the *SegmentedLink*, a new class that handles the drawing is used. This class, *SegmentedArrow*, contains two *SegmentDragPoints*, on it, one at the tail and the other at the head. Later, any number of *SeqgmentDragPoints* can be added by double clicking on the *SegmentedLink*.

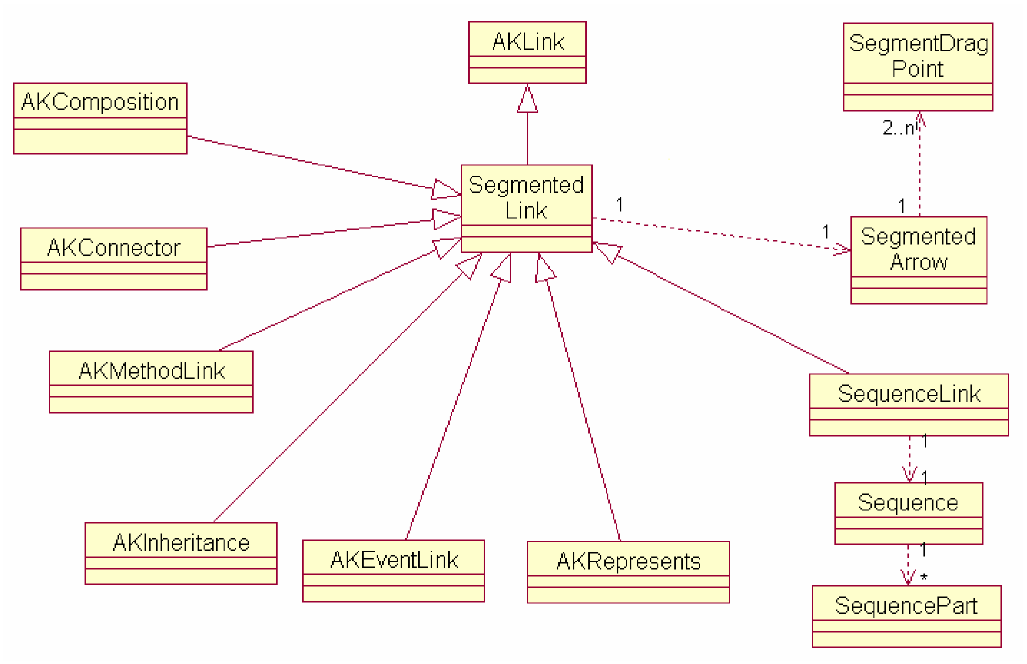


Figure 28. New structure of link classes in COSEML

In the new structure, a new link type, *SequenceLink* is introduced. This link contains a *Sequence* class, which handles the decimal number sequencing. *SequencePart* class in this class is simply the number parts defined in a sequence. As an example, "Sequence 2.1.3" contains three *SequenceParts*.

4.2.2. Dialog Windows

New dialog windows are added to support functionalities of collaboration diagrams. Figure 29 shows the new dialogs added to COSECASE.

AddDiagramDialog and *RenameDiagramDialog* are used for adding new collaboration diagrams and renaming diagrams in the model. *SymbolAdder* is used to add symbols from main hierarchy diagram to a collaboration diagram. *SequenceEditor* manages the sequence messages in the collaboration diagrams. Defining message name, message type, condition and loop structures are all handled in this dialog.

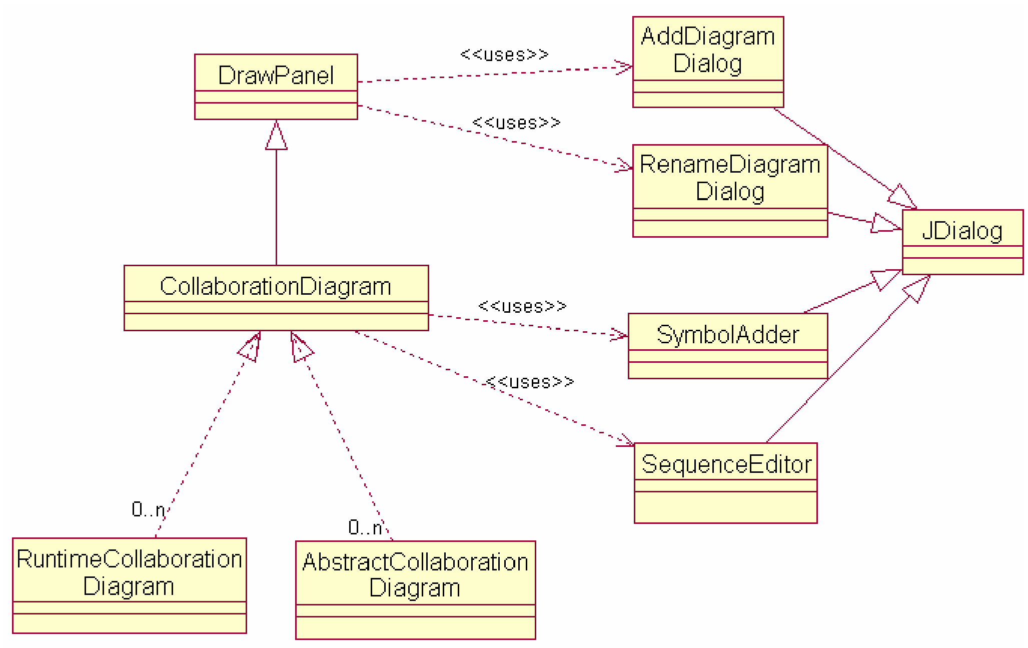


Figure 29. New dialog classes in COSEML

CHAPTER 5

CASE STUDY: E-STORE APPLICATION

In this case study, an e-store application is modeled with COSECASE. Modeling power of collaboration diagrams in the Component Oriented Software Modeling approach is clearly illustrated in this work.

An e-store application provides a virtual store for online shopping. A customer visits the e-store and creates an account in order to benefit from shopping. At this stage, personal information, contact information (e-mail, phone number) and address of the customer are saved. After that, customer can browse the product catalog and examine the features of the products. While browsing, customer makes a selection and adds them to a virtual shopping cart provided by the e-shop application. During this process, customer may edit the contents of this shopping cart. Then customer clicks to “proceed to checkout” button to buy the items in the shopping cart. In the checkout process, customer selects the shipping and payment options and then approves the order. Following that, system charges the customer and sends the products to the shipping address. Meanwhile, customer can track the order status using the system. When the products arrive, order status is changed as completed and details of this order are saved in the “order history” of the customer.

5.1. Logical Decomposition

COSE modeling approach starts with system decomposition. Possible packages at the first level of the decomposition are listed below.

- Account
- Product Catalog
- Shopping Cart

- Order
- Customer
- Web

Figure 30 shows the first level decomposition of the e-store application on COSECASE.

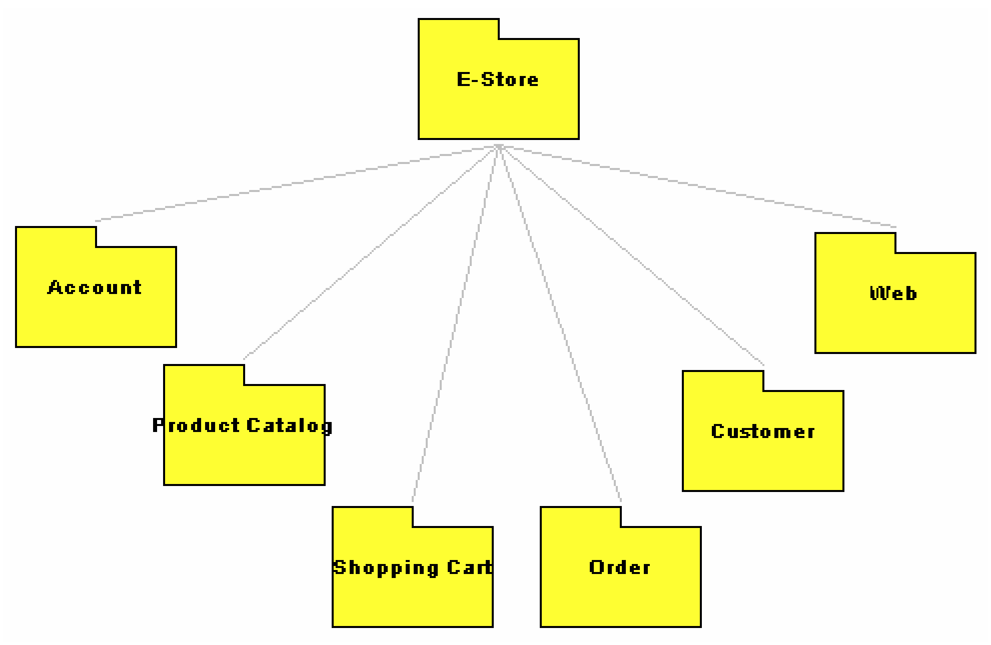


Figure 30. First level decomposition of the e-store application

Creating accounts and login operations are handled in the *Account* package. Information about products, prices and product stocks are managed in *Product Catalog* package. This package also has search functionality for products, prices and stocks in the e-shop. *Shopping Cart* package manages a virtual shopping cart. This package handles buying decisions of the customers. Products can be added or removed until payment approval. Another package in the e-store application is the *Order* package. This package manages the checkout of the items in the shopping cart, shipping of the items to the customers and payment process. Keeping the order status is also handled in this package. *Customer* package represents the real customers and keeps user preferences and other customer related information in the system. Finally, *Web* package represents the web pages on the e-store application. Customers interact with the system using the web pages provided by this package.

COSE development continues with further decomposition and reviewing of the system specification. Decomposition of the packages that are shown in figure 30 is conducted and modeled as follows.

Figure 31 shows the decomposition of the *Account* package, which is responsible for the login and registration operations.

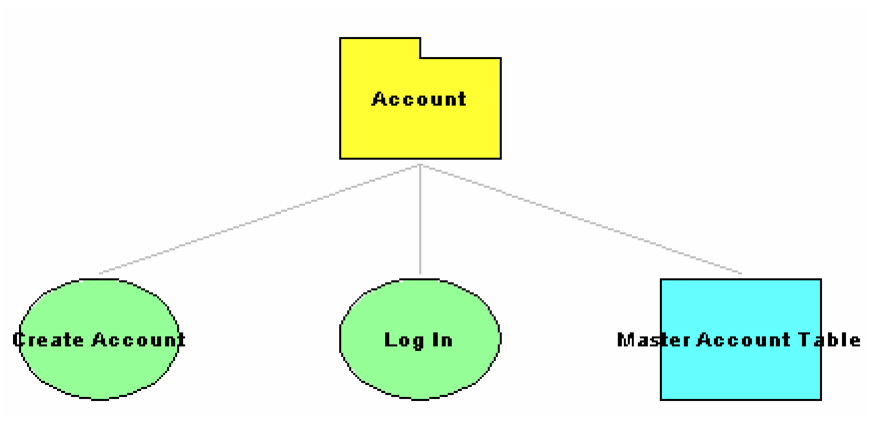


Figure 31. Decomposition of the *Account* package

Figure 31 shows two main functionalities of the *Account* package. A new customer first creates an account using *Create Account* function. Username and password of the customer are added to the *Master Account Table*. Then customers log in the e-store using *Log In* functionality with their username and password.

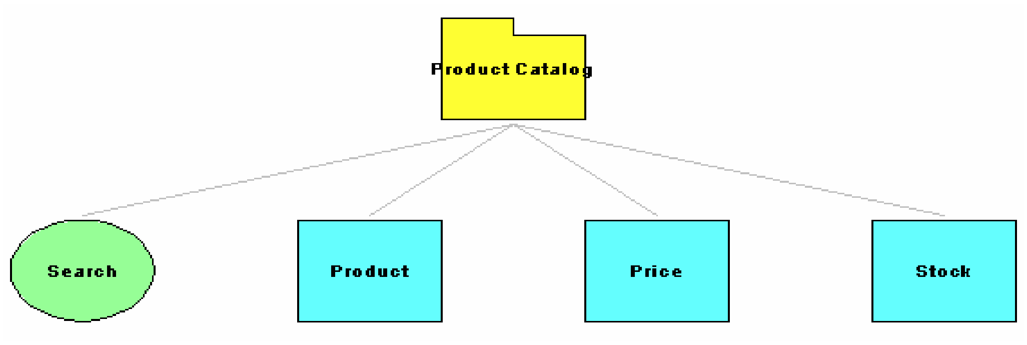


Figure 32. Decomposition of the *Product Catalog* package

In figure 32, decomposition of the *Product Catalog* package is shown. Products and their properties are stored in the *Product* data abstraction. Up-to-date prices of the products are kept in *Price* and current stock values are maintained in *Stock* data abstractions.

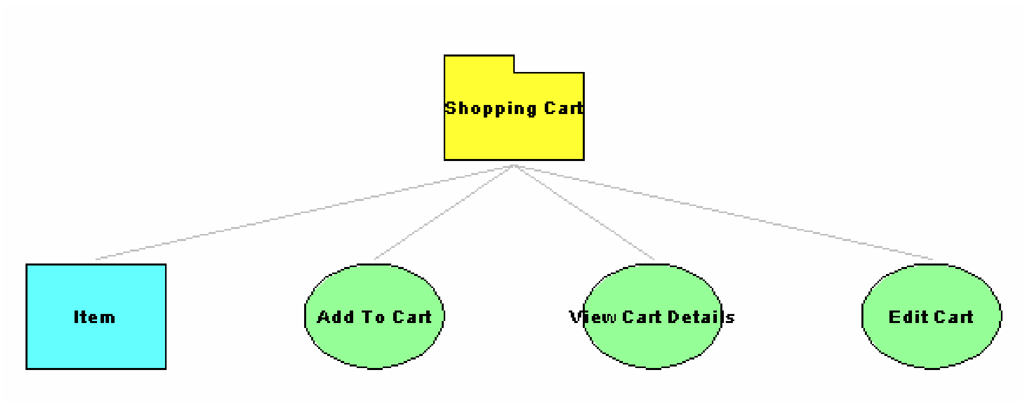


Figure 33. Decomposition of the *Shopping Cart* package

Shopping Cart package contains three main functionalities. Products selected by the user can be added to the shopping cart using *Add to Cart* functionality. Updating or deleting an item is handled by the *Edit Cart* functional abstraction. Another functionality of the package, *View Cart Details*, shows the price, quantity and product information of the items in the shopping cart. Such information about the items in the cart is stored in *Item* data abstraction in *Shopping Cart* package. This decomposition is shown in figure 33.

Order package is more detailed than the other packages in the application. Figure 34 shows the decomposition of this package.

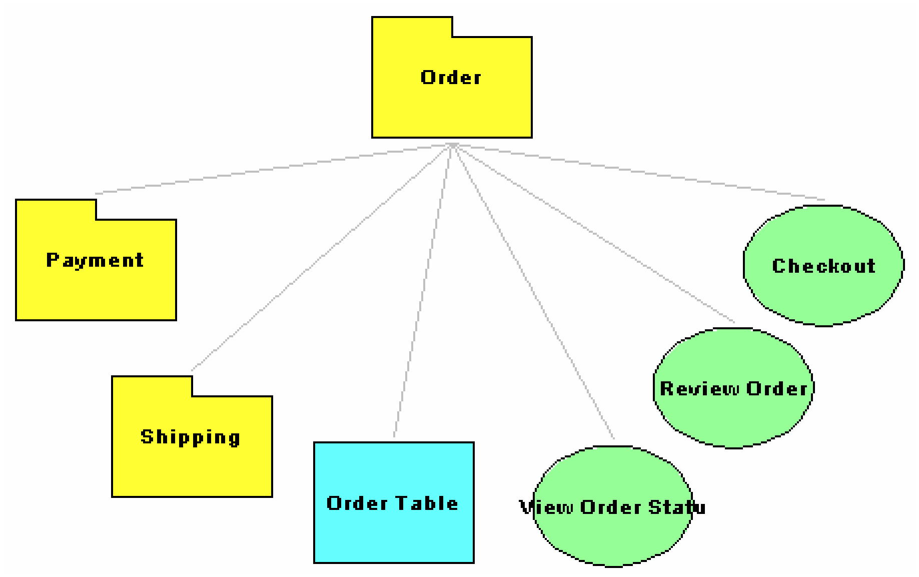


Figure 34. Decomposition of the *Order* package

It encloses two sub packages, three function abstractions and a data abstraction. *Payment* and *Shipping* are the packages defined under the *Order* package. Reviewing of an order before payment, confirming the checkout and viewing the order status are the main functionalities. They are represented by the function abstractions defined in this package.

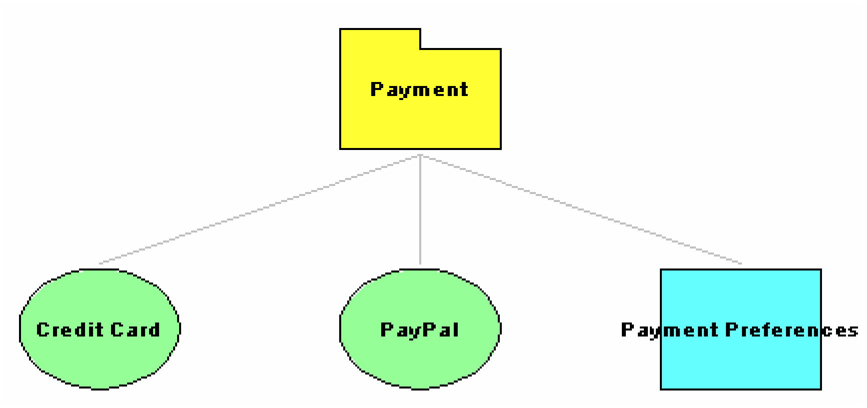


Figure 35. Decomposition of the *Payment* package

Figure 35 shows the decomposition of the *Payment* package. This package manages the payment methods, and it keeps the customer's payment preferences. Security of the payment process, validation of customer's credit card and communication with the bank are handled in this package.

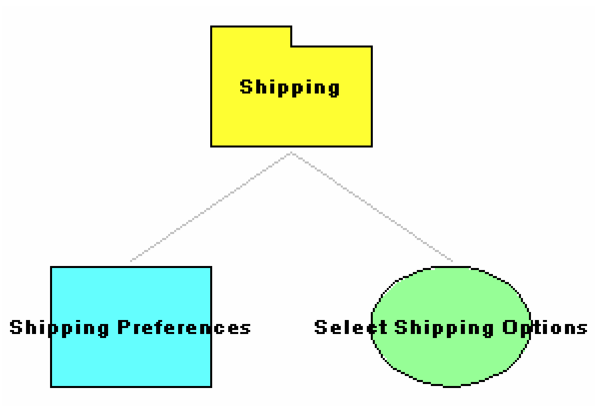


Figure 36. Decomposition of the *Shipping* package

Figure 36 shows the *Shipping* package defined in the *Order* package. This package manages the shipping of the products to customers. Shipping address, invoice name, selection of shipping company, shipping time and other shipping related issues are handled in this package. Figure 37 shows the whole decomposition model.

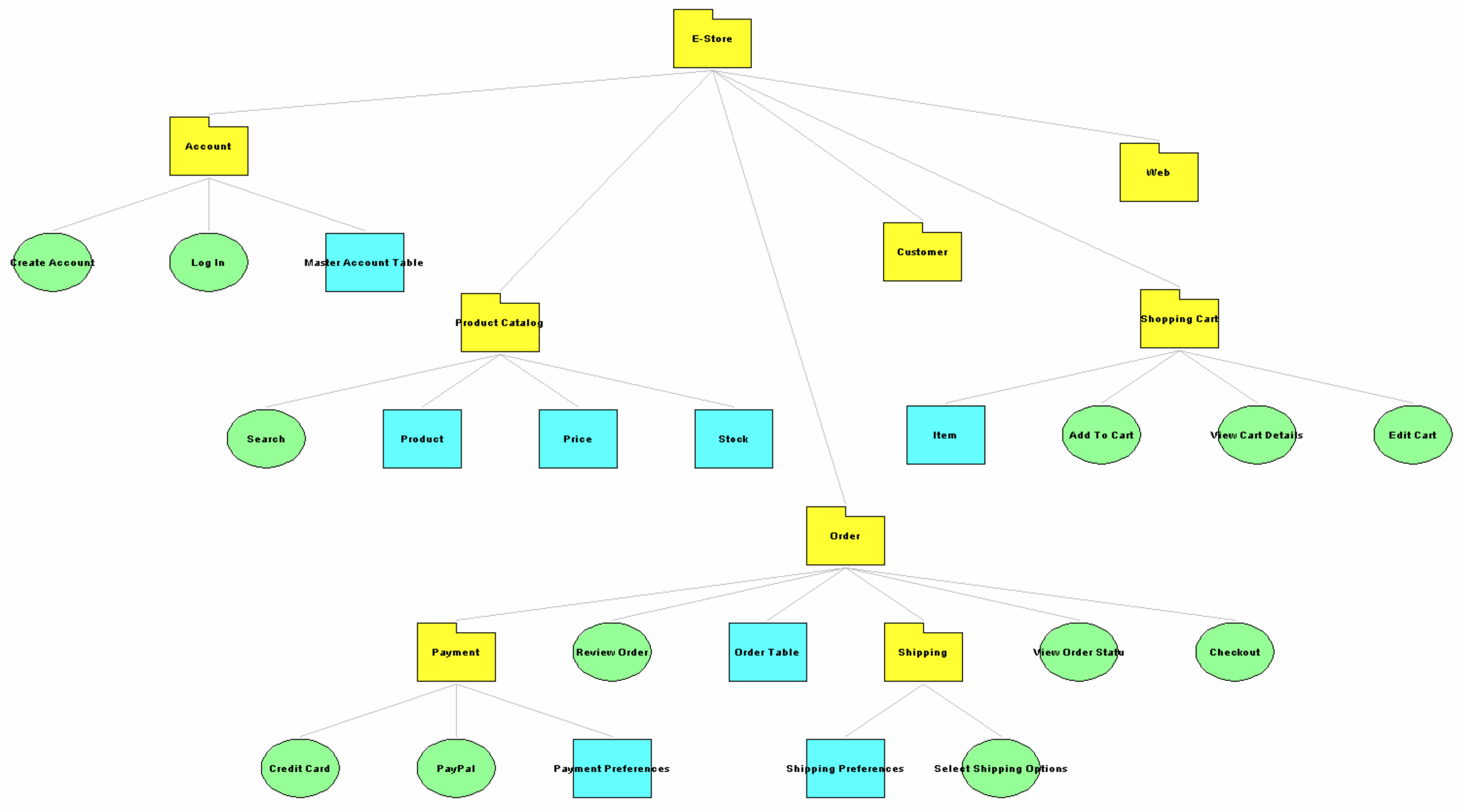


Figure 37. First level decomposition of the e-store

5.2. Use Cases Realizations with Abstract Collaboration Diagrams.

Use case analysis helps to figure out the requirements and help to explore the important behavior. In this section, some of the most important use cases of the system are observed and their realization is modeled using abstract collaboration diagrams.

To model the realization of a use case using an abstract collaboration diagram, possible collaborating elements should be identified. After that, their existence in the main hierarchy diagram should be checked. If these elements do not exist, then they should be added to the main hierarchy diagram before using in a collaboration diagram.

Use Case - Create Account

To use the e-store application, customers first need to create an account. A use case with the name "Create Account" is obvious.

Steps:

- Customer opens the registration page of the e-store.
- Customer enters registration data (username, password, e-mail) to the registration page.
- Registration page checks the validity of the data.
- If check fails, an error message is displayed.
- If check succeeds, registration page creates the account.
- *Account* package inserts a new row to the *Master Account Table* with given username and password.
- Registration page mails the account details to the customer.

Possible collaboration participants in this use case are *Customer*, *Web*, *Account* and *Master Account Table* defined in the main hierarchy diagram. Figure 38 shows the realization of "Create Account" use case with the abstract collaboration diagram.

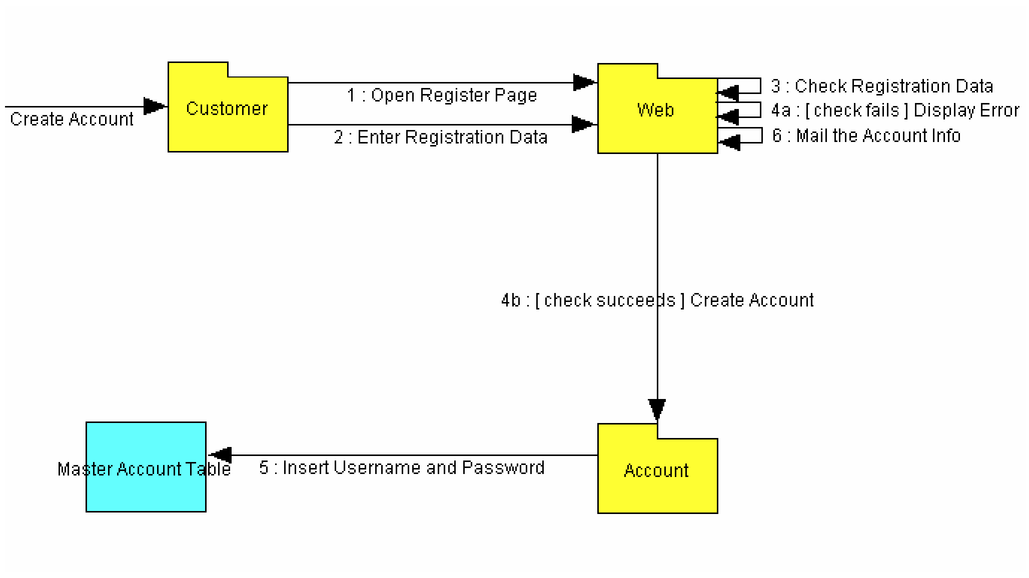

Create Account
Customer
1 : Open Register Page
2 : Enter Registration Data
Web
3 : Check Registration Data
4a : [check fails] Display Error
6 : Mail the Account Info
4b : [check succeeds] Create Account
Master Account Table
5 : Insert Username and Password
Account

Figure 38. Abstract collaboration diagram of "Create Account"

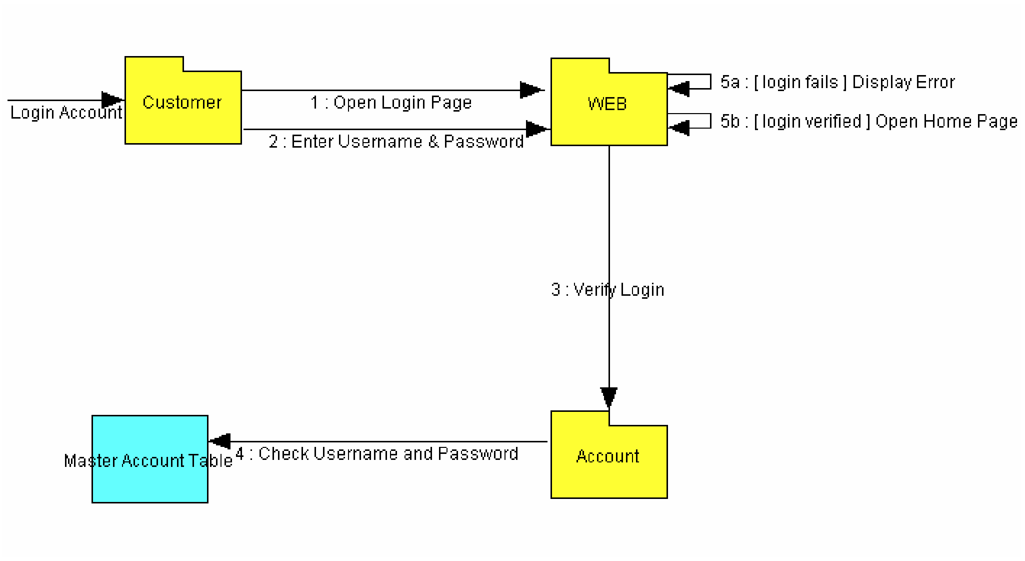
Use Case - Login

Registered users should login to system for improved service and security. This use case assumes that customer has registered to system.

Steps:

- Customer opens login page.
- Customer enters username and password to the login page.
- Login page verifies login data from *Account*.
- *Account* checks username and password using the *Master Account Table*.
- If login fails, login page displays error.
- If login is successful, login page opens the home page.

Figure 39 shows the abstract collaboration diagram for this use case.



Login Account
Customer
1 : Open Login Page
2 : Enter Username & Password
WEB
5a : [login fails] Display Error
5b : [login verified] Open Home Page
3 : Verify Login
Master Account Table
4 : Check Username and Password
Account

Figure 39. Abstract collaboration diagram of "Login"

Use Case – Browse Products

Customers often visit an e-shop for browsing a product category (e.g. video camera). In this use case scenario, a customer browses the product catalogs by selecting a product category.

Steps:

- Customer selects a product category from the e-store home page.
- Page selects a category form the *Product Catalog*.
    - If exists, *Product Catalog* selects subcategories
- *Product Catalog* searches products for selected categories.
- Search function finds products by the selected category from the *Product* table.
- Page displays the product list from the selected category.

Figure 40 shows the representing collaboration diagram.

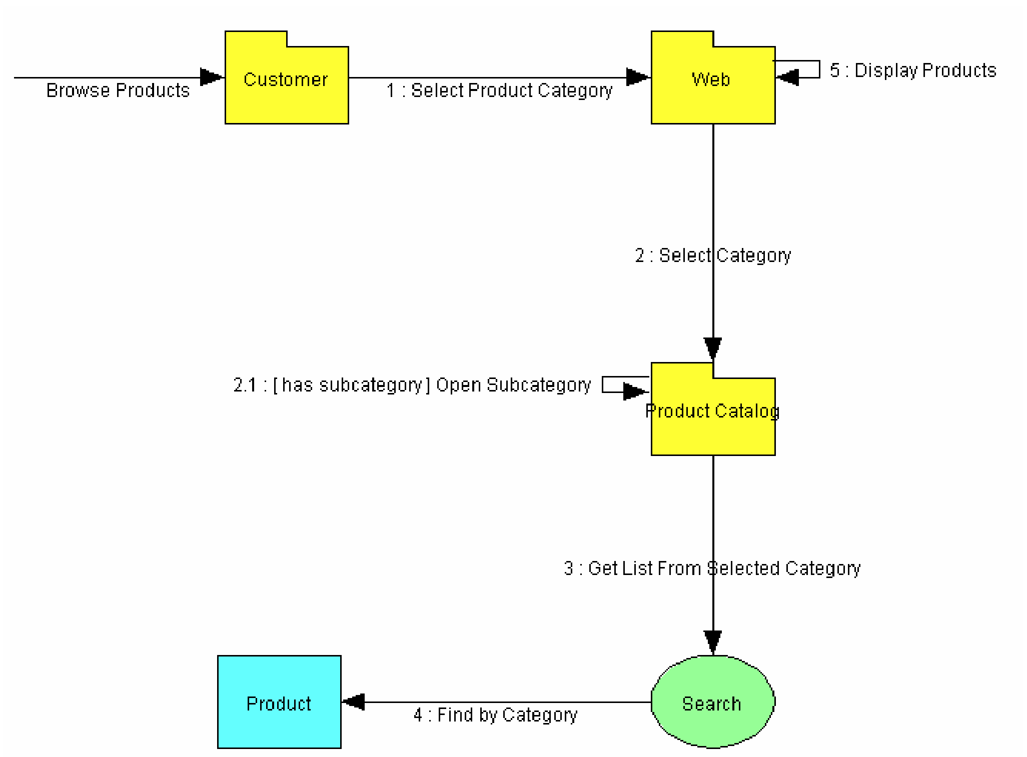


Figure 40. Abstract collaboration diagram of "Browse Products"

Use Case – Search by Product Property

Customers should be able to make a search on a specific product property.

Steps:

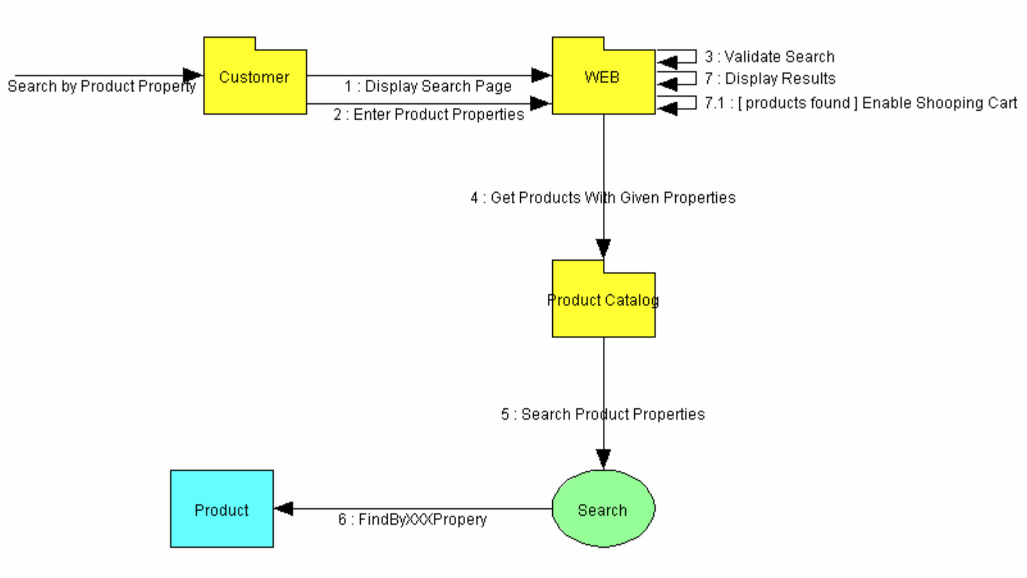
- Customer opens the search page of the e-store application.
- Customer enters the product properties to the input fields provided by the search page and clicks the search button.
- Search page checks the input values for validity.
- Search page asks for the products with given properties from the *Product Catalog*.
- *Product Catalog* uses the search functionality to get the products with given properties.

- Search function makes a query for the given properties on *Product* table.

- Search page displays the products with the given properties.

    - If products found, Search page activates “Add to Shopping Cart” use case.

Collaboration diagram of this use case is shown in figure 41.



Search by Product Property
Customer
1 : Display Search Page
2 : Enter Product Properties
WEB
3 : Validate Search
7 : Display Results
7.1 : [products found] Enable Shooping Cart
4 : Get Products With Given Properties
Product Catalog
5 : Search Product Properties
Search
6 : FindByXXXPropery
Product

Figure 41. Abstract collaboration diagram of “Search by Product Property”

Use Case – Add to Shopping Cart

When customers want to buy a product, they add it to the shopping cart provided by the e-shop and continue shopping. Adding a product to the shopping cart is given with the following use case steps.

Steps:

- Customer selects a product

- Customer inputs the product quantity and clicks the “add to shopping cart” button.
- Page checks the stock from *Product Catalog*.
- *Product Catalog* gets the stock data of the product from *Stock* table.
- If stock is not available, page displays “Stock not available” error and use case ends.
- If the requested quantity of the product is available in the stock, page calls the *Add to Cart* function.
- *Add to Cart* function adds the product with given quantity to the Item table
- *Add to Cart* function updates the data on the *Shopping* Cart.
- Page displays the contents of the *Shopping Cart*.

Figure 42 shows the collaboration diagram for this use case.

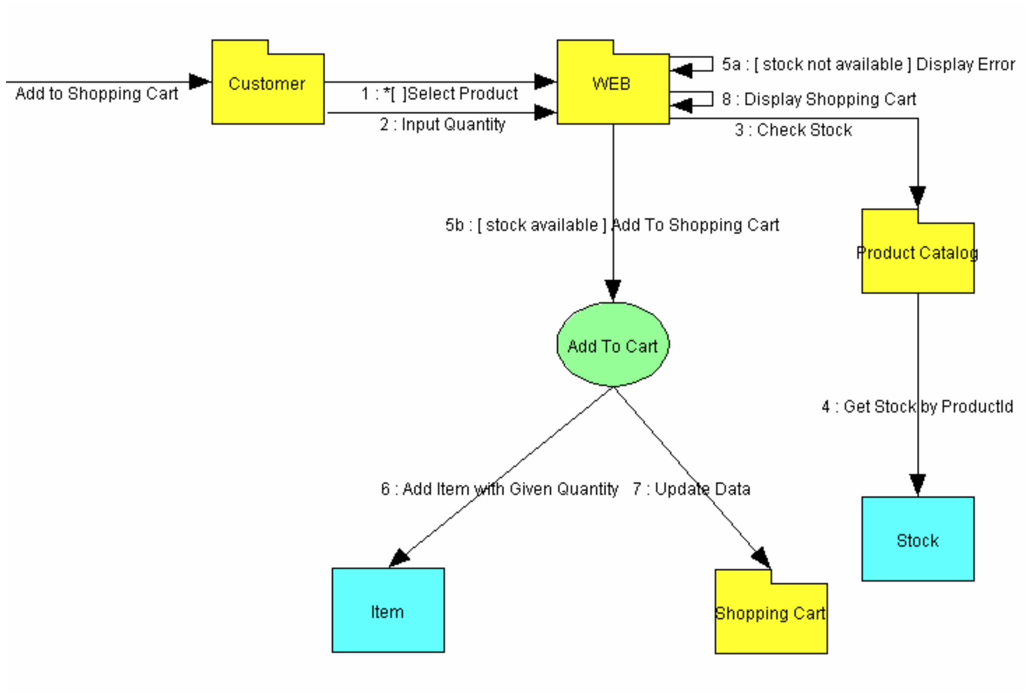

Add to Shopping Cart
Customer
1 : *[ ]Select Product
2 : Input Quantity
WEB
5a : [stock not available] Display Error
8 : Display Shopping Cart
3 : Check Stock
5b : [stock available] Add To Shopping Cart
Product Catalog
Add To Cart
4 : Get Stock by ProductId
6 : Add Item with Given Quantity
7 : Update Data
Stock
Item
Shopping Cart

Figure 42. Abstract collaboration diagram of "Add to Shopping Cart"

Use Case – Edit Shopping Cart

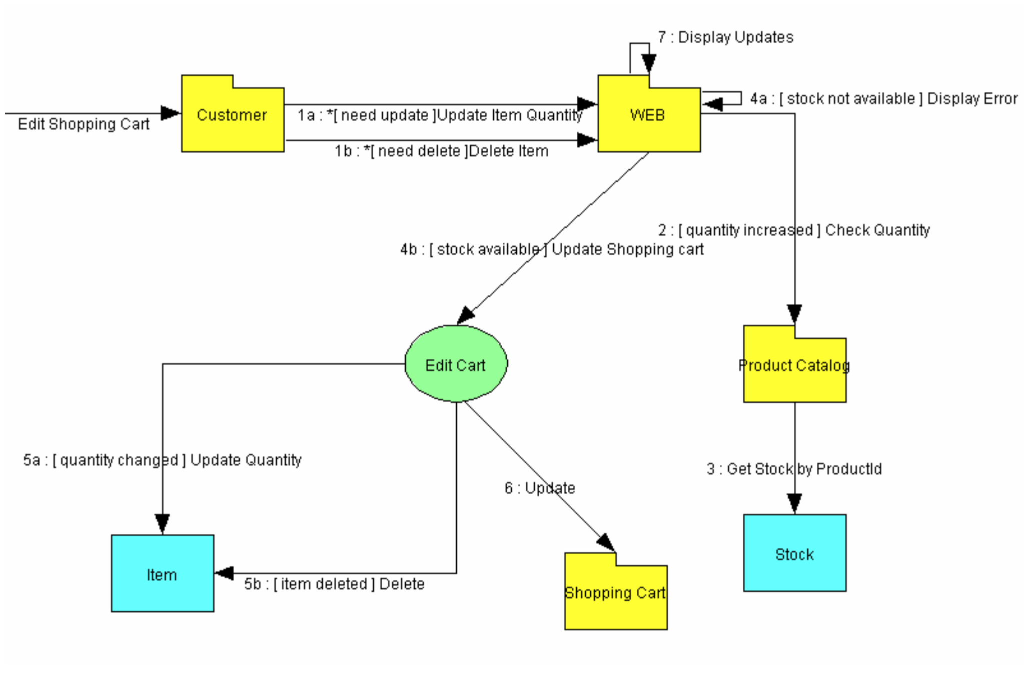
Customers may decide to edit the contents of the shopping cart. They may cancel the shipping process. Alternatively, they may update the quantity of the products in the shopping cart. This use case describes the steps of deleting or updating the items in the shopping cart.

Steps:

- Customer updates the quantity of the selected products on the shopping cart page.
- Customer deletes the selected products on the shopping cart page.
- If any product quantity increased, page checks the stock from *Product Catalog*.
- If any product quantity increased, *Product Catalog* gets the stock data from *Stock* table.

- If any product quantity increased and stock is not available, page displays an error and use case ends.
- Else, page calls *Edit Cart* function for updating the shopping cart.
- If there is a quantity change, *Edit* Cart function updates the quantity on Item table.
- If there is an item deletion, *Edit Cart* function deletes the item from Item table.
- *Edit* Cart function updates the contents of the *Shopping Cart*.
- Page displays the updates.

This use case is realized with the collaboration diagram shown in figure 43.



7 : Display Updates
Edit Shopping Cart
Customer
1a : *[need update]Update Item Quantity
1b : *[need delete]Delete Item
WEB
4a : [stock not available] Display Error
2 : [quantity increased] Check Quantity
4b : [stock available] Update Shopping cart
Edit Cart
Product Catalog
5a : [quantity changed] Update Quantity
6 : Update
3 : Get Stock by ProductId
Item
5b : [item deleted] Delete
Shopping Cart
Stock

Figure 43. Abstract collaboration diagram of “Edit Shopping Cart”

Use Case – Proceed Checkout

Checkout is the most complicated use case of the e-shop application. It involves shipping approval, payment method approval, and finally the checkout. Steps are defined as follows.

Steps:

- Customer clicks "Proceed to Checkout" button.
- Page gets the saved shipping preferences from *Shipping* package.
- *Shipping* package get the saved preferences from *Shipping Preferences* table.
- Page displays shipping details.
- If customer wants to change the shipping preferences, page calls the "Update Shipping Preferences" use case.
- Customer approves the shipping preferences by clicking approve button on the page.
- Page gets payment preferences from *Payment* package.
- *Payment* package retrieves the preferences from *Payment Preferences* table.
- Page displays the previously saved payment preferences.
- If customer wants to change the payment preferences, page calls the "Update Payment Preferences" use case.
- Customer approves the payment preferences by clicking approve button on the page.
- Page gets review data from *Review Order* function.
- *Review Order* function gets preferences from the *Shipping Preferences* table.

- *Review Order* function gets preferences from the *Payment Preferences* table.
- *Review Order* function gets shopping cart contents by calling *View Cart Details* function.
- *View Cart Details* function gets the details from *Shopping Cart* package.
- Page displays the review.
- Customer controls the review and clicks the "Checkout" button on the page.

Updating the shipping and payment preferences process is not included in the realization diagram. Otherwise, diagram can become complicated and difficult to understand. For this reason they are modeled as separate collaboration diagrams and referenced from this diagram as "Call Update xxx Details Collaboration Diagram". Collaboration diagram of this use case is shown in figure 44.

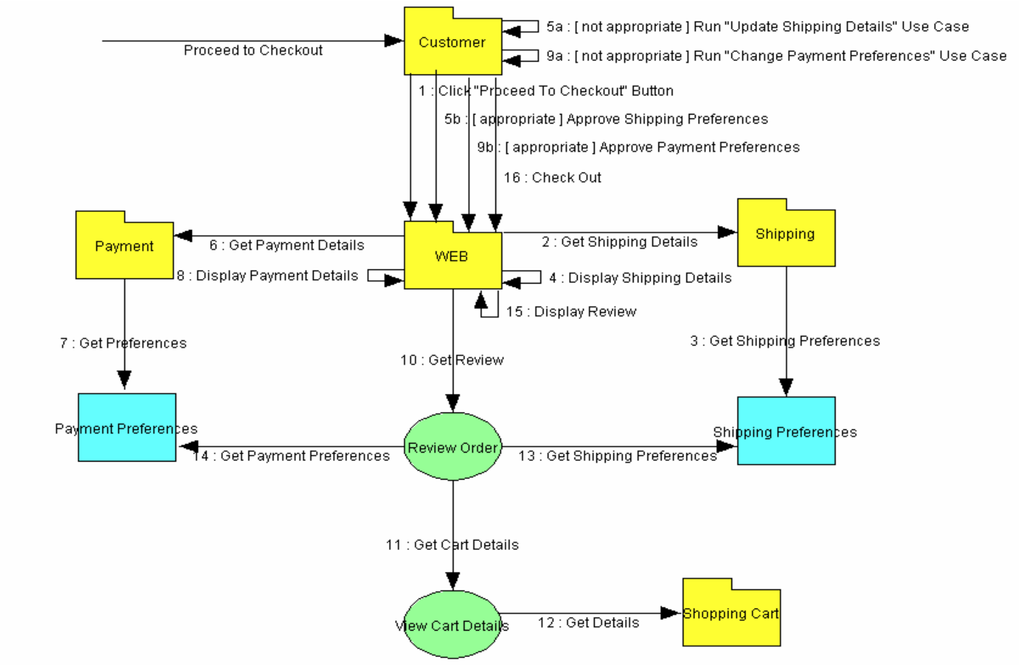

Proceed to Checkout
Customer
5a : [not appropriate] Run "Update Shipping Details" Use Case
9a : [not appropriate] Run "Change Payment Preferences" Use Case
1 : Click "Proceed To Checkout" Button
5b : [appropriate] Approve Shipping Preferences
9b : [appropriate] Approve Payment Preferences
16 : Check Out
Payment
6 : Get Payment Details
8 : Display Payment Details
WEB
2 : Get Shipping Details
4 : Display Shipping Details
15 : Display Review
Shipping
7 : Get Preferences
10 : Get Review
3 : Get Shipping Preferences
Payment Preferences
14 : Get Payment Preferences
Review Order
13 : Get Shipping Preferences
Shipping Preferences
11 : Get Cart Details
View Cart Details
12 : Get Details
Shopping Cart

Figure 44. Abstract collaboration diagram of "Proceed to Checkout"

Use Case – Update Shipping Preferences

A customer may be shopping from a different location. Consequently, she may update the previously entered shipping preferences. This is a sub use case of the checkout use case and referenced there. Steps of this operation are as follows. Figure 45 shows the collaboration diagram for this use case.

Steps:

- Customer enters new shipping options to the page.
- Page calls *Select Shipping Options* function to update the options.
- *Select Shipping Options* function sends the new options to *Shipping* package.
- *Shipping* package updates the preferences on *Shipping Preferences* table.

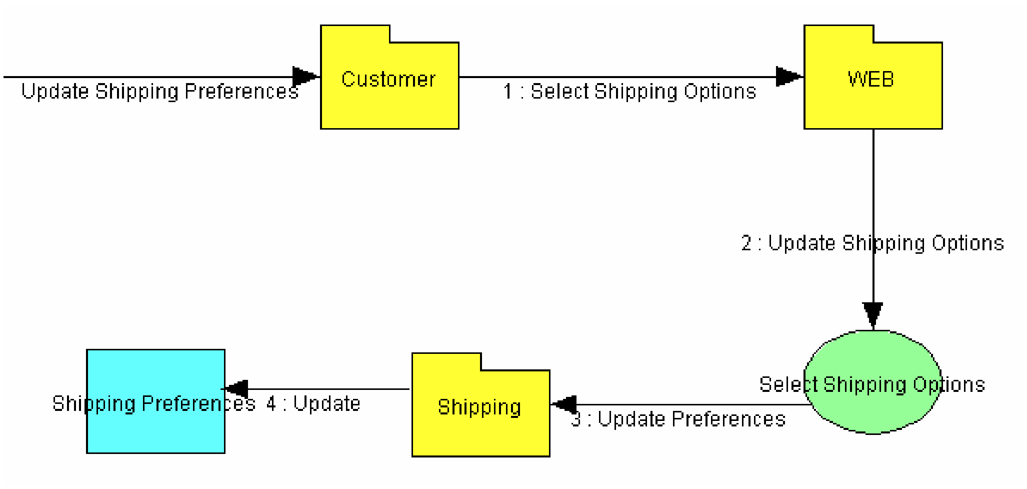


Figure 45. Abstract collaboration diagram of "Update Shipping Preferences"

Use Case – Update Payment Preferences

Customers can change the payment method on the e-shop. This is also referenced from the checkout use case. Following steps describe the process.

Steps:

- Customers enter new payment preferences to the page.
- Page updates preferences on the *Payment* package.
- *Payment* package updates the *Payment Preferences* table.

Figure 46 shows the collaboration diagram.

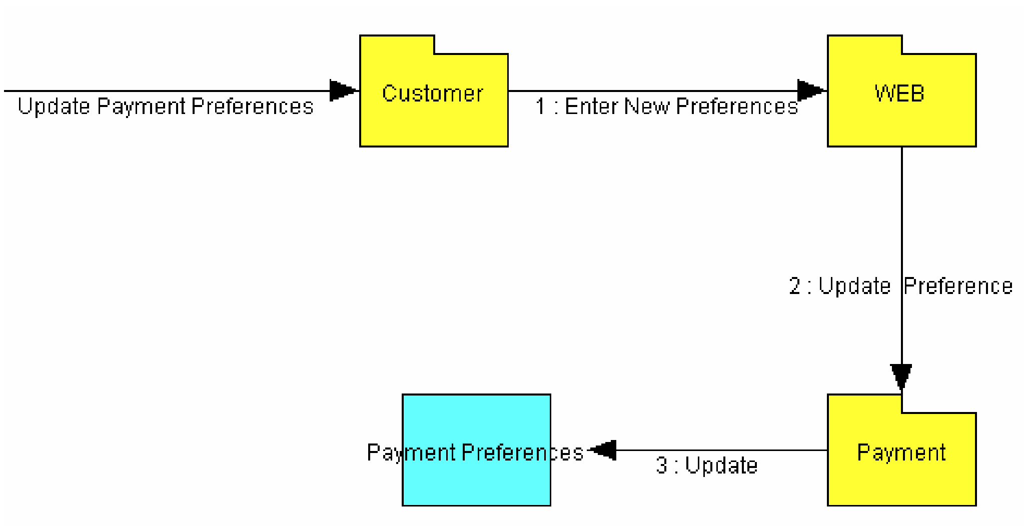


Figure 46. Abstract collaboration diagram of "Update Payment Preferences"

5.3. Physical Composition

After decomposition, matching components should be found to build the system [6]. Then, these physical components should also be added to the hierarchy diagram. By this way, abstractions and their representing components are shown on the same model, which provides an overall system structure.

Up to this point, decomposition of the e-store application was modeled. In addition, a use case analysis of the system is made and use cases that are found, are modeled using abstract collaboration diagrams. Next task is to find the real components that represent the abstractions defined in the decomposition model. While selecting components, abstract collaboration diagrams should also be kept in mind.

First of the packages that is found in the decomposition model is *Account Package*. This package is further decomposed into two function abstractions and a data abstraction, which are *Create Account*, *Log In* and *Master Account Table*. This package can be represented by a component. This component is assumed to have the same name with the package. Figure 47 shows the representing component.

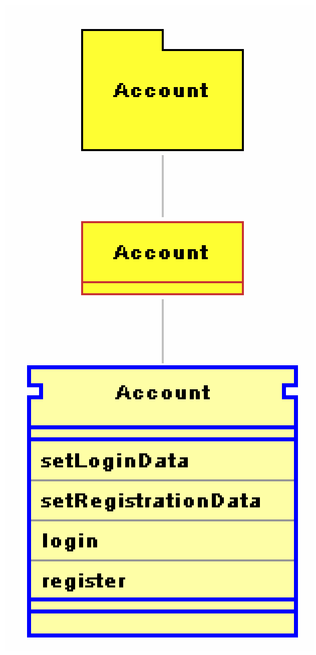


Figure 47. *Account* component and its interface

This component has a single interface and satisfies the required functionality of the *Account Package*. It has methods that provide login and register functionality. There are two input functions. In order to execute login method, some other interface in the system should provide *LoginData* to this interface by calling *setLoginData*.

Likewise, in order to invoke the *register* method, *setRegistrationData* should be called first with *RegistrationData*.

Product Catalog Package can also be represented by a single component. Figure 48 shows the representing component.

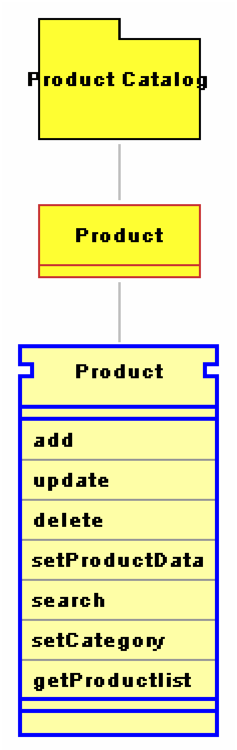


Figure 48. *Product* component and its interface

This component also has a single interface. In this interface, *setProductData* and *setCategory* are the input methods. All other methods require *setProductData* to be invoked first. Exception is that, *getProductList* can be invoked, if the *setCategory* method is called previously.

Another package abstraction is *Shopping Cart Package*. Figure 49 shows the component that represents this package.

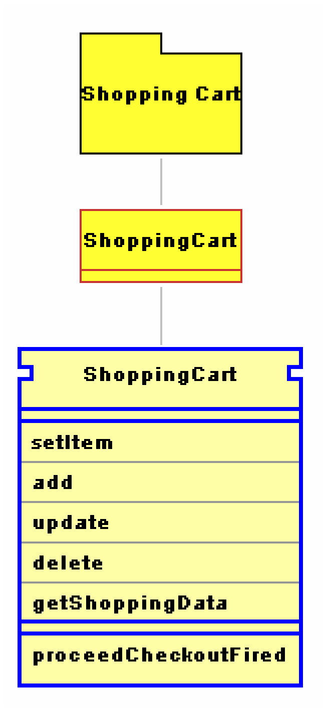


Figure 49. *Shopping* component and its interface

This component also has a single interface. There is a single input method, *setItem* that is required by the other methods, defined in the interface. There is also an output event with the name *proceedCheckoutFired*. When the user decides to buy the items in the shopping cart and clicks the "Proceed Checkout" button, this event is activated.

Most detailed package in the decomposition model is the *Order Package*. It has two sub packages and four other abstractions. *Payment Package* and *Shipping Package* are represented by two components with the same name. A component with the name *Order* is also added to the model. Figure 50 shows the components that represent *Order Package.*

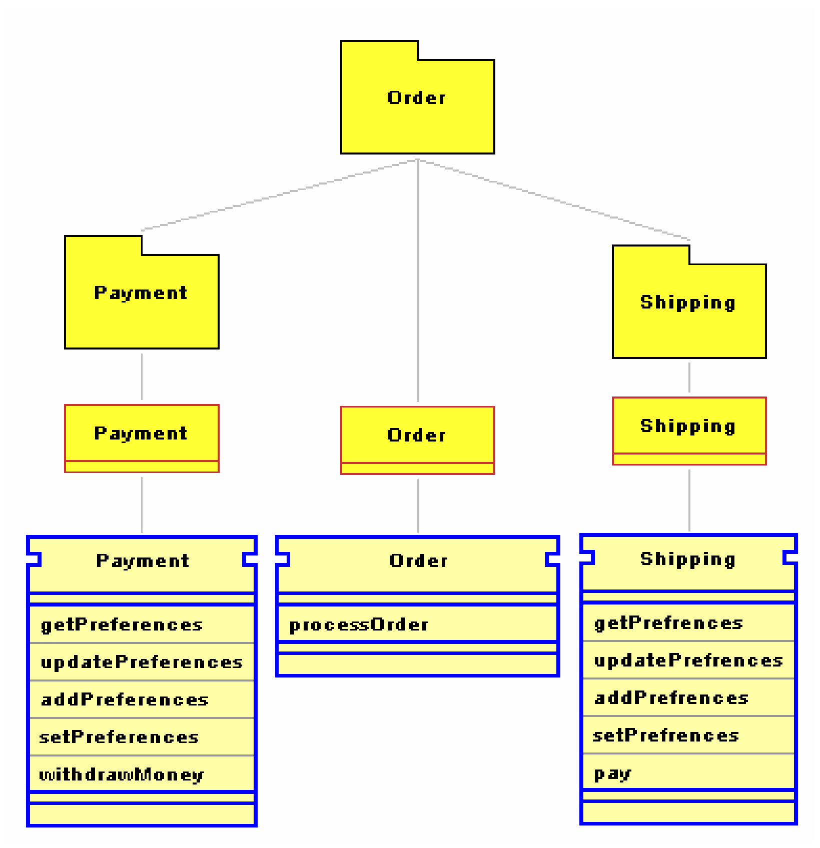


Figure 50. *Payment*, *Order*, *Shipping* components and their interfaces

All components shown in the figure have single interface. *Shipping Interface* and *Payment Interface* have similar methods for adding, deleting, and updating the payment and shipping preferences. The *withDrawMoney* method in *Payment Interface* draws money from customer and *pay* method in *Shipping Interface* send money to the shipping company. The only method in *Order Interface* is the *processOrder*. This method bills the customer, pays the shipping company and ends the shopping.

Interface between the customers and the e-store is the Web *Package*. The Web component should have all the functionalities for creating interactions between the customers and the e-store. Figure 51 shows the corresponding component, and its three interfaces.

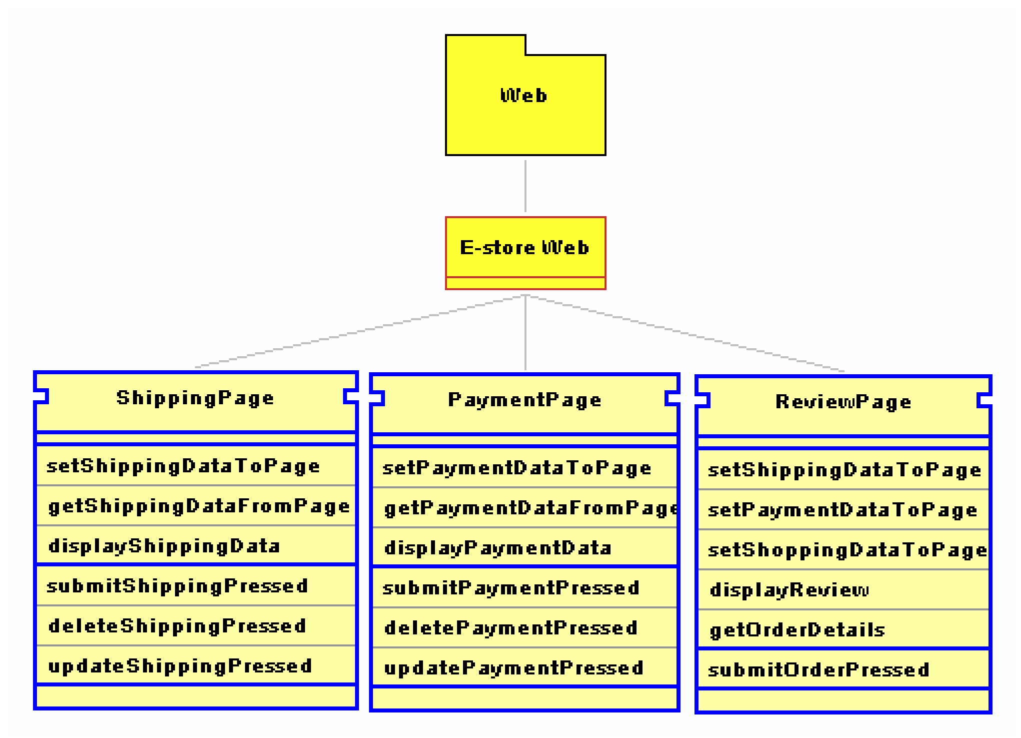


Figure 51. *E-store Web* component and its three interfaces

ShippingPage interface is responsible for displaying the shipping preferences. Customers can update, delete, or approve the preferences using this interface. In this interface, before calling *displayShippingData*, *setShippingDataToPage* method should be called first. There are also three events defined in this interface. These events inform the listeners about user actions on the page. For example, *submitShipingPressed* event informs that customer has approved the shipping preferences and *getShippingDataFromPage* method can be used to get the approved preferences.

Similarly, *PaymentPage* interface is responsible for displaying and managing the payment preferences. Methods and events defined in this interface are similar to that of *ShippingPage* interface.

ReviewPage interface provides the review of shopping cart, shipping and payment details. When a customer decides to buy and clicks the approve button on the page, *submitOrderPressed* event is fired. This event informs the listeners that customer selected the shipping and payment preferences and bought the items in the shopping cart.

Although there are three interfaces in the figure, *E-storeWeb* component has more interfaces like *LoginPage* or *RegistrationPage*. However, to keep the things simple and clear they are omitted. In the following section, only these three interfaces of the *E-storeWeb* component are used.

Figure 52 shows all the abstract packages and their representing components that have been defined in this section.

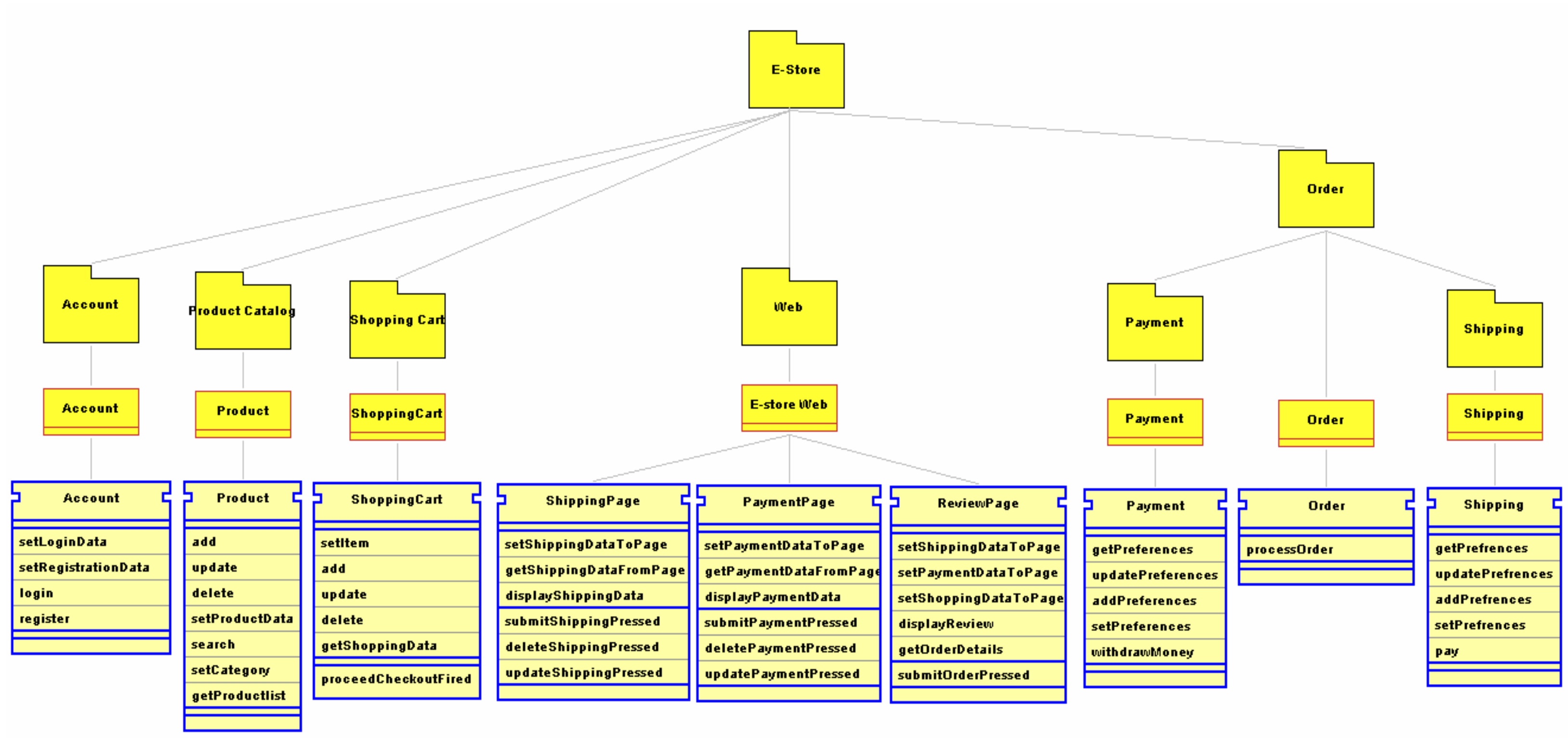


Figure 52. Components of the e-store application

5.4. Runtime Collaboration Diagrams

Runtime collaboration diagrams show implementation details of a system behavior and they provide a very suitable medium for component wiring. In the previous section, components of the e-store system are found and included into the main hierarchy diagram. With run time collaboration diagrams, implementation of the use cases can be modeled by showing the interactions among the real components. In this section, the “Proceed to Checkout” use case, which is the most complex scenario in the e-store application, is modeled using a run time collaboration diagram.

In the “Proceed to Checkout” use case scenario, customer first clicks to “Proceed to Checkout” button. Then, shipping preferences are displayed on the web page. Customer updates the preferences if she wants and approves the shipping preferences. Next, the page displays the payment preferences. Again, customer may change the preferences and she approves the payment preferences. After that, page displays the review of items in the shopping cart, shipping and payment details. If everything is suitable, customer clicks “Submit Order” button and process ends.

Component interfaces, which are required for this scenario, are *ShoppingCart*, *ShippingPage*, *Shipping*, *PaymentPage*, *Payment*, *ReviewPage* and *Order*. Figure 53 shows the first phase of the run time collaboration.

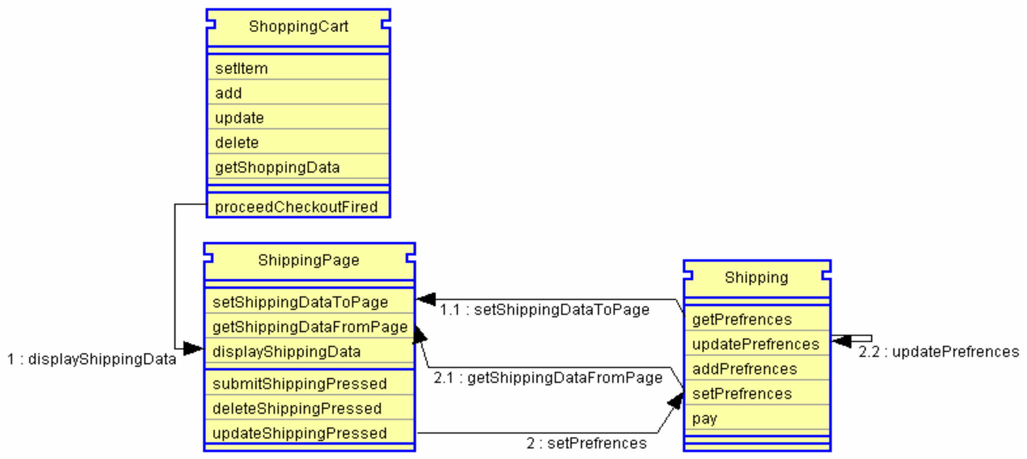


Figure 53. First phase of the run time collaboration

ShoppingCart component fires the *proceedCheckoutFired* event when the customer clicks the "Proceed to Checkout" button. This event is listened by the shipping page. When it is fired, use case starts and *ShippingPage* displays the preferences by getting the data from *Shipping* component. If customer wants to update the preferences, she clicks the update button. Then, the *Shipping* component fetches the updated shipping preferences from the *ShippingPage* and updates it. Figure 54 shows the next phase.

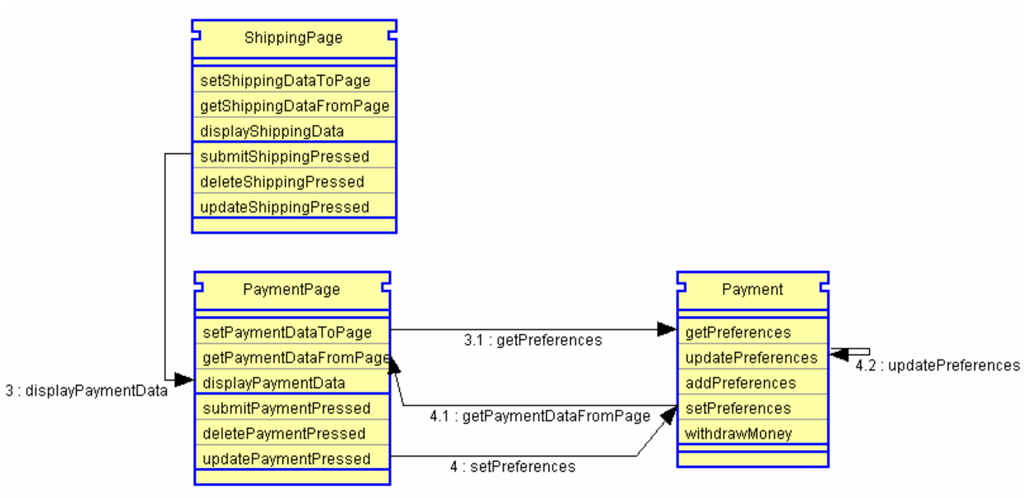


Figure 54. Second phase of the run time collaboration

Customer approves the shipping preferences by clicking submit button on the shipping page. At this point, *ShippingPage* fires the *submitShippingPressed* event. *PaymentPage* listens to this and when it is fired, the *Payment* component fetches the updated payment preferences from the *PaymentPage* and updates it.

Figure 55 shows the last phase. When the customer approves the payment preferences by pressing the submit button, *displayReview* method in the *ReviewPage* is invoked. To display the review, first, *ReviewPage* fetches the shipping preferences from the *Shipping* component. Next, payment preferences are fetched from the *Payment* component. Finally, shopping data is fetched from the shopping cart. After these flows, *ReviewPage* displays the review. When the customer presses order button, *submitOrderPressed* event is fired on the *ReviewPage*. Then, this event calls the *processOrder* method in the *Order* component, which concludes the shopping.

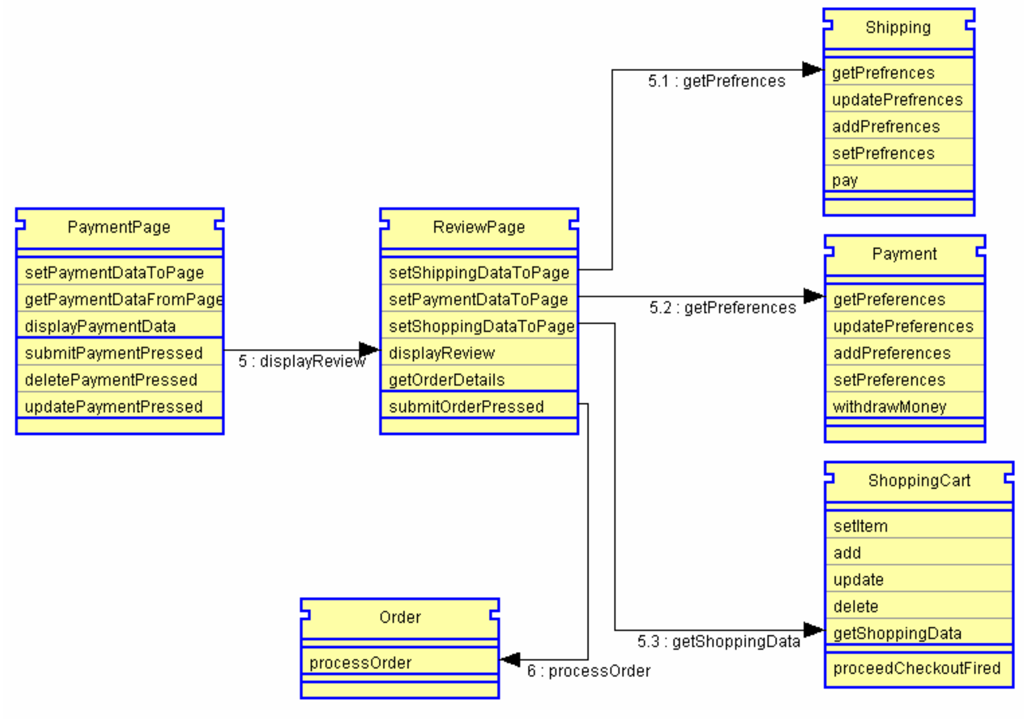


Figure 55. Third phase of the run time collaboration

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1. Conclusions

COSEML was proposed before as the primary modeling language for Component Oriented Software Engineering approach [2]. Since then, some researches have been continuing for improving this modeling language. However there are still more areas for enhancement in COSEML. Having a single static hierarchy diagram in the modeling language is not sufficient. It is not possible to extensively describe a software system without modeling its dynamic behavior. Before this thesis, practiced benefits of dynamic modeling in UML were not available to COSEML.

With this thesis, a collaboration modeling is added to the COSE approach. Supported with the implementation of two levels of collaboration diagrams, the previous COSECASE tool is now capable of representing dynamic models. The case studies conducted through modeling example systems showed that it is beneficiary to include collaboration modeling to a COSE approach. These diagrams give a better view of system functionalities. They enable visualizing the expected system behavior on the model. Use case realization, which is a very important requirement capturing process, is also made possible. It is observed that the sequencing information provided with the collaboration modeling is very helpful for the construction phase of COSE based development. This sequencing information also creates a possibility for generating applications and automated tests from COSEML models. A future commercial version of this tool can be very instrumental for the industry to adopt Component Oriented development methodologies.

6.2. Future Work

An important motivation for starting this research was the missing sequence information for the messages, to be used during the composition phase. There have been different applications of COSE based tools that offered framework kind of environments where components can be composed to yield executable applications [23][24]. Such composition, however could not be guided effectively, by the COSEML model: the order of the messages to be fired were totally left to the designer's intuition during the composition. Future frameworks can import the presented collaboration modeling abilities for a further automated and guided composition. Then, the order of message invocations can be retrieved from the COSEML model.

Another future work is needed for providing some abstractions on collaboration diagrams. In complex systems, modeling the dynamic behavior can produce too many collaboration diagrams. This may create difficulties to see the overall dynamic behavior of the system. A more abstract view is needed that shows the cooperation among the collaboration diagrams. In UML 2, interaction overview diagrams [4] are used for similar approach. Likewise, such a modeling view can be incorporated to COSEML. Moreover, if executable application generation from collaboration diagrams is made possible, then this view can also be used for composing those application parts to build the final application.

REFERENCES

[1] A.H. Dogru and M.M. Tanik, “A Process Model for Component Oriented Software Engineering”, IEEE Software, vol. 20, no 2, pp. 34-41, January 2003.

[2] A.H. Dogru, “Component Oriented Software Engineering Modeling Language: COSEML”, TR-99-3, Computer Engineering Department, Middle East Technical University, December 1999.

[3] A. Kara, “A Graphical Editor for Component Oriented Modeling”, M.S. Thesis, Middle East Technical University, December 2001.

[4] OMG, “Unified Modeling Language: Superstructure”, Version 2.0, Formal/05-07-04, http://www.omg.org/cgi-bin/doc?formal/05-07-04, August 2005.

[5] G. Engels, L. Groenewegen and G. Kappel, “Coordinated Collaboration of Objects”, Advances in Object-Oriented Data Modeling, pp. 307-331, The MIT Press, 2000.

[6] A.H. Dogru, “Component-Oriented Software Engineering”, The Academy of Learning and Advanced Studies (The ATLAS), to be published in 2006.

[7] M.D. McIlroy, “Mass Produced Software Components”, NATO Conference on Software Engineering, Garmisch, Germany, October 1968.

[8] OMG, "CORBA Basics", http://www.omg.org/gettingstarted/corbafaq.htm, 2005.

[9] Microsoft, "Component Development", http://msdn.microsoft.com/library/en-us/dnanchor/html/componentdevelopmentank.asp, 2005.

[10] Sun, "Reference Documentation", http://java.sun.com/products/ejb/index.jsp, 2005.

[11] X. Cai, M.R. Lyu, K. Wong, R. Ko, "Component-Based Software Engineering: Technologies, Development Frameworks, and Quality Assurance Schemes", Proceedings of the Seventh Asia-Pacific Software Engineering Conference, p372, 2000.

[12] G. Booch, J. Rumbaugh, I. Jacobson, "The Unified Modeling Language User Guide", Addison Wesley, April 2000.

[13] M.B. Tuncel, "COSEML'de Isbirligi Diyagramlarinin Kullanimi", Ulusal Yazilim Muhendisligi Sempozyumu, September 2005.

[14] OMG, "OMG Model Driven Architecture", http://www.omg.org/mda, 2005.

[15] M.M. Tanik and E.S. Chan, "Fundamentals of Computing for Software Engineers", Van Nostrand Reinhold, New York, 1991.

[16] M. Fowler, "UML Distilled", 3rd Edition, Addison Wesley, 2004.

[17] A. Abdurazik and J. Offutt, "Using UML Collaboration Diagrams for Static Checking and Test Generation", Third International Conference on the Unified Modeling Language, pp. 383-395, York, UK, October 2000.

[18] R.C. Martin, "UML Tutorial: Collaboration Diagrams", Engineering Notebook Column, November 1997.

[19] G. Engels, R. Hucking, S. Sauer and A. Wagner, "UML Collaboration Diagrams and Their Transformation to Java", Proceedings of UML99, Lecture Notes in Computer Science, vol.1723 pp. 473-488, 1999.

[20] The Eclipse Foundation, "What is Eclipse", http://www.eclipse.org/org, 2005.

[21] Free Software Foundation, "Concurrent Versions System - CVS", http://www.nongnu.org/cvs, 2005.

[22] S.W. Ambler, "The Elements of UML Style", Cambridge University Press, 2002

[23] E. Ozdogru, "A GIS Domain Framework Utilizing Jar Libraries As Components", M.S. Thesis, Middle East Technical University, May 2005.

[24] M. Ozturk, "Visual Composition in Component Oriented Development", M.S. Thesis, Middle East Technical University, August 2005.

APPENDIX A

A BRIEF USER MANUAL FOR USING COLLABORATION DIAGRAMS IN COSECASE

COSECASE was first introduced in thesis “A Graphical Editor for Component Oriented Modeling” by Aydin Kara [6]. A manual for using COSECASE was also given in that work. Modeling in the main hierarchy diagram was described there. Here, new implementations are explained.

A.1. Diagram Tree

To support collaboration diagrams and any other diagrams that can be added in the future, a floating window that contains a tree structure is created. This diagram tree is responsible for managing the diagrams in the model. It can be visible or hidden by selecting a menu item in the “view” menu of the main application. In figure 56, diagram tree is shown at the left side of the main window.

Currently diagram three supports three types of diagram. The first one is the main hierarchy diagram, located at the top of the tree. This is the default diagram in the model and it always exists in the diagram tree. Other two types of diagrams are run time collaboration diagram and abstract collaboration diagram. These diagrams are located under the “Collaboration Diagrams” node in the diagram tree. Model does not contain collaboration diagrams at the beginning; they are added, as they are needed.

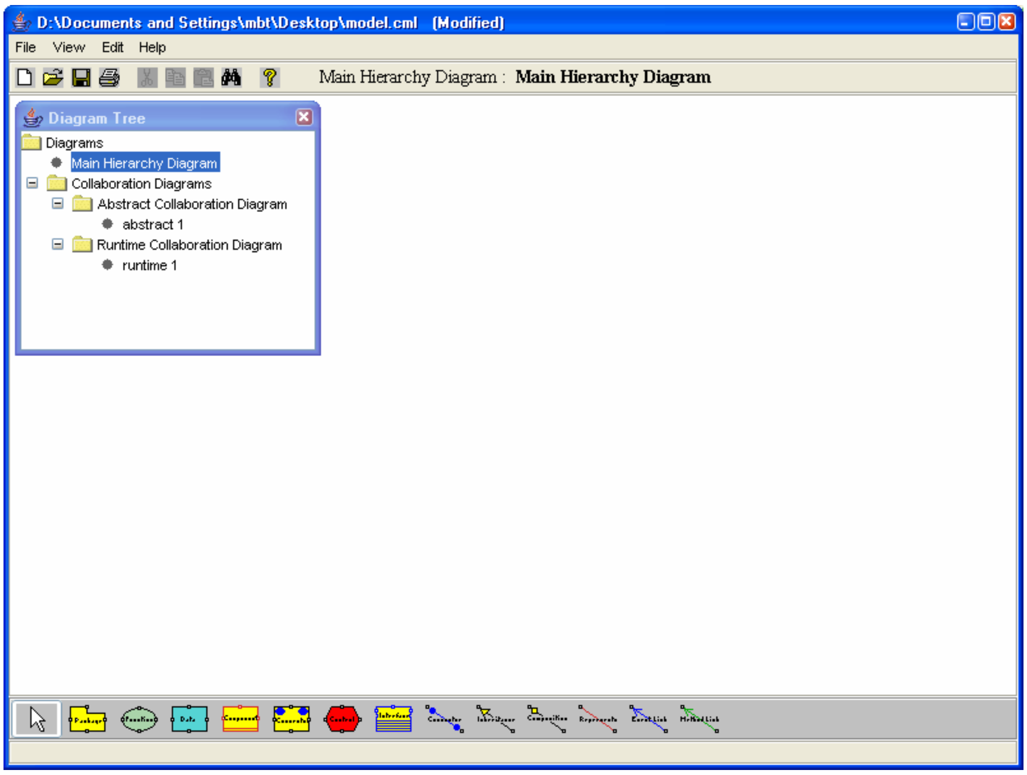

D:\Documents and Settings\mbt\Desktop\model.cml (Modified)
File View Edit Help
Main Hierarchy Diagram : Main Hierarchy Diagram
Diagram Tree
Diagrams
Main Hierarchy Diagram
Collaboration Diagrams
Abstract Collaboration Diagram
abstract 1
Runtime Collaboration Diagram
runtime 1

Figure 56. Main window in COSEML

Basic function of the diagram tree is browsing the diagrams in the model. Double clicking the diagram nodes on the diagram tree shows the selected diagram on the main window. Other functionalities of the diagram tree are mostly available for collaboration diagrams. Since the main hierarchy diagram is the default diagram and there cannot be multiple hierarchy diagrams, only renaming is allowed for this diagram. On the other hand, new collaboration diagrams can be added to the model. Selecting the appropriate collaboration diagram type and right clicking on it, opens the “Add New Diagram” pop-up menu as shown in figure 57.

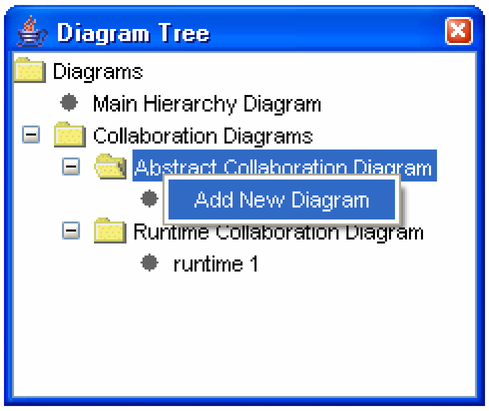
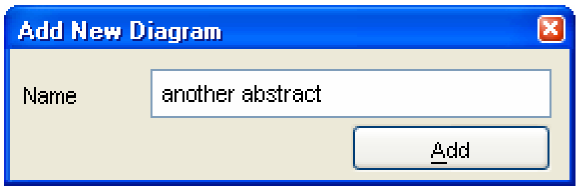


Figure 57. Add new diagram pop-up in *DiagramTree*

Selecting this menu item opens the "Add New Diagram" dialog window as shown in figure 58.



Name: another abstract

Figure 58. Add new diagram dialog

Entering the name of the diagram adds the diagram under the selected collaboration diagram type in the diagram tree. Right clicking on a collaboration diagram reveals other functionalities with a pop-up menu. Figure 59 shows this pop-up menu on the diagram tree.

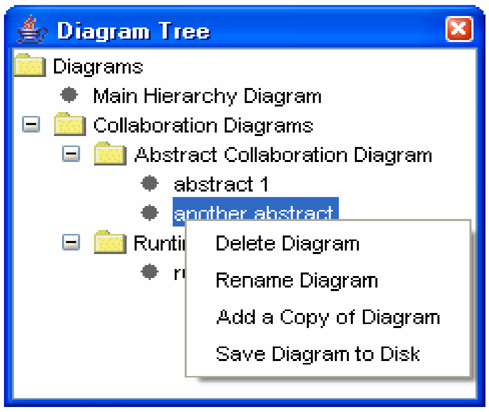


Figure 59. Pop-up menu for managing diagram in *DiagramTree*

First of these menu items is "Delete Diagram". When selected, a confirmation dialog opens and if user confirms the deletion, it deletes the selected diagram collaboration diagram from the model. Second is the "Rename Diagram" menu item. When selected, it opens the "Rename Diagram Dialog". Figure 60 shows this dialog. As it is obvious, this diagram renames the selected collaboration diagram in the diagram tree.

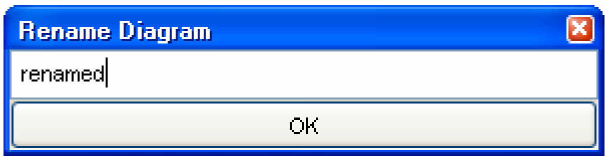


Figure 60. Rename diagram dialog

Third menu item is "Add a Copy of Diagram". This menu item creates the duplicate of the selected collaboration diagram in the model. This functionality is very useful when modeling similar collaboration diagrams. With minor changes on the copied diagram, similar use cases can be modeled efficiently.

The last one is the "Save Diagram to Disk" menu item. When this is selected, it opens a save dialog as shown in the figure 61.

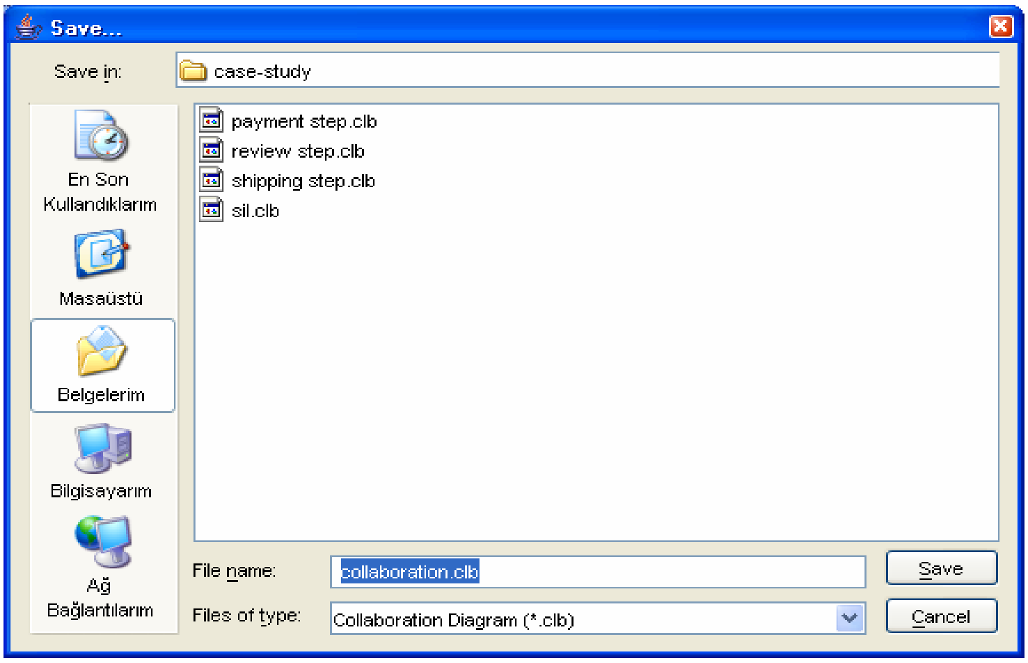


Figure 61. Save diagram to disk dialog

This menu item allows saving a collaboration diagram independently from the model. It allows saving different versions of a diagram on the disk. It is also useful for sending a single collaboration diagram to other users/developers.

A.2. Modeling Tool Bar

Dynamic creation of the tool bar for selected symbols, are made possible in this work. This will be also useful when new diagram types are needed in future. Figure 62 shows the tool bar for collaboration diagrams.

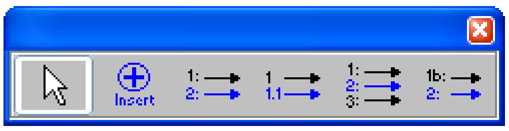


Figure 62. Collaboration diagram tool bar

Buttons on the toolbar and their brief descriptions are given in the table 3.

Table 3. Description of buttons of the modeling tool bar

Button	**Meaning**
↖	"Select Button" selects elements on the diagram.
⊕ Insert	"Insert to Collaboration Diagram" button opens a list of elements that exist in main hierarchy diagram. If the diagram is an abstract collaboration diagram, list contains only abstract elements. If it is a run time collaboration diagram, list contains only component interfaces.
1: → 2: →	"Creates Next Sequence" button creates the next sequence message for the selected message.
1 → 1.1 →	"Creates Child Sequence" button creates child sequence message for the selected message.
1: → 2: → 3: →	"Insert Between Sequence" button creates the same sequence message for the selected message and shifts the selected and the following sequences one step up.
1b: → 2: →	"Insert Next Non Concurrent" button creates the next non concurrent message for the selected message.

This tool bar is used by both abstract and run time collaboration diagrams. Only the "Insert to Collaboration Diagram" button behaves differently on these diagrams.

A.3. Inserting Elements to Collaboration Diagrams

All elements should exist in the main hierarchy diagram before using them in the collaboration diagrams. Then, clicking to "Insert to Collaboration Diagram" button on the tool bar brings a list of elements from the main hierarchy diagram. Figure 63 shows this process on the main window.

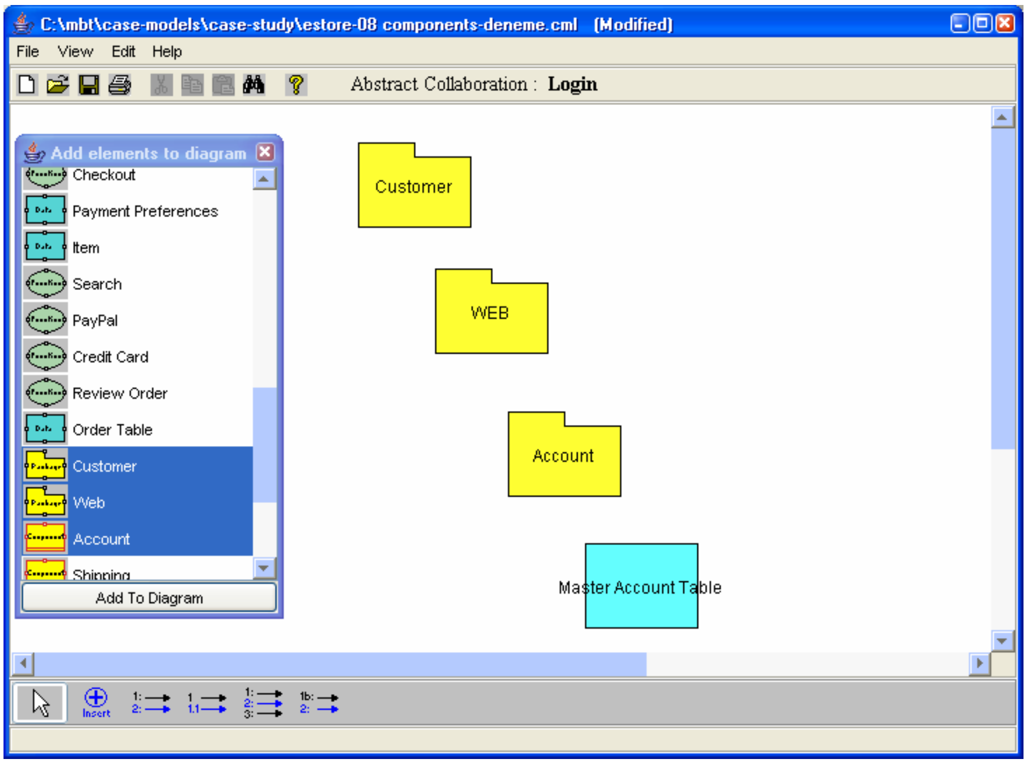


Figure 63. Add elements to diagram dialog.

If the active diagram is an abstract collaboration diagram, then the list contains only abstract elements defined in the main hierarchy diagram, as it is the case in figure 63. If it is a run time collaboration diagram, then the list contains only component interfaces defined in the main hierarchy diagram.

Selecting the elements on the list and clicking the “Add to Diagram” button, adds the selected elements to the active collaboration diagram.

A.4. Sequence Messages

To show the interactions among the elements in a collaboration diagram, sequence messages are used. Figure 64 shows all the possible message types and sequencing in a collaboration diagram. First message is added to diagram by clicking the “Creates Next Sequence” button on the tool bar and clicking on the diagram. To add the next messages, an existing message should be selected and the appropriate button on the tool bar should be clicked.

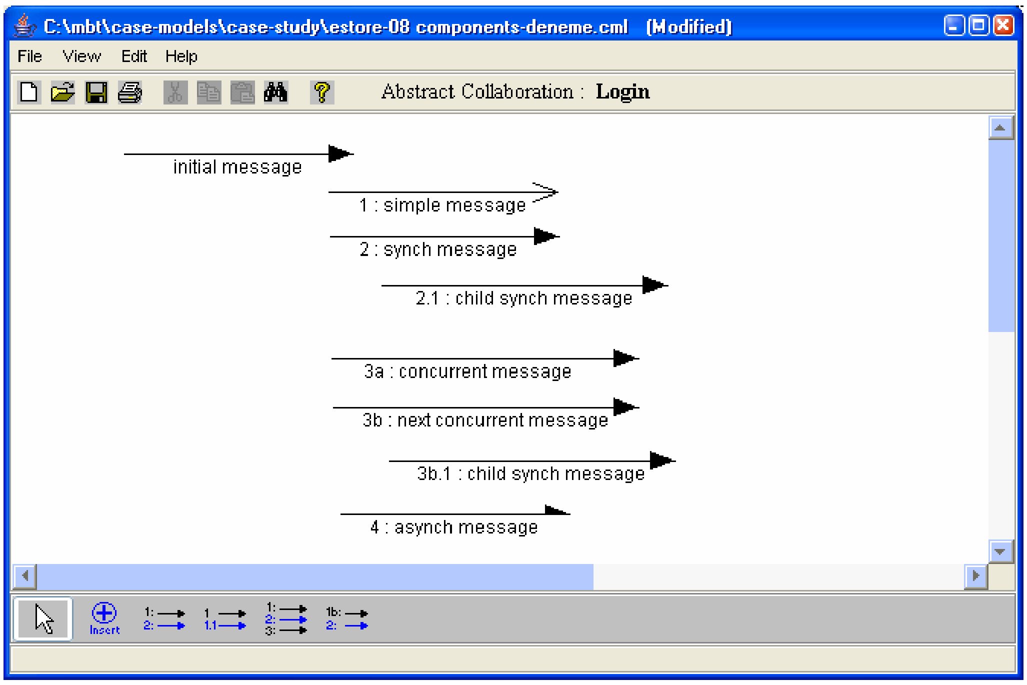

C:\mbt\case-models\case-study\estore-08 components-deneme.cml (Modified)
File
View
Edit
Help
Abstract Collaboration : Login
initial message
1 : simple message
2 : synch message
2.1 : child synch message
3a : concurrent message
3b : next concurrent message
3b.1 : child synch message
4 : asynch message
Insert

Figure 64. Different message types on Collaboration Diagram

Table 4. Managing sequence messages with the tool bar buttons.

Button	Selected Sequence	Previous Set	Next Set
1: → 2: →	**2**	1, 2	1, 2, **3**
1 → 1.1 →	**3**	1, 2, 3	1, 2, 3, **3.1**
1: → 2: → 3: →	**2**	1, 2, 3, 3.1	1, **2**, 3, 4, 4.1
1b: → 2: →	**3a or 3b**	1, 2, 3a, 3b	1, 2, 3a, 3b, **4**

Table 4 explains how to create desired sequence messages by using the buttons on the tool bar. By default, all newly added messages are synchronous. However, their types can be changed by an editor dialog.

A.5. Editing Sequence Messages

Selecting a sequence message and right clicking it opens a pop-up menu. This menu is showed in figure 65.

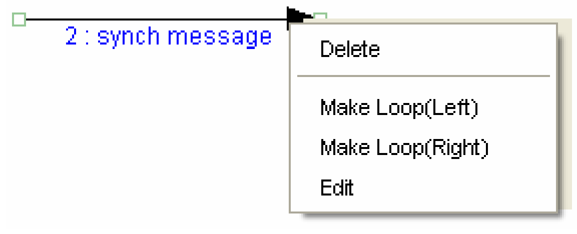


Figure 65. Pop-up menu for sequence managing

Clicking the "Delete" menu item on the pop-up menu, deletes the selected message from the diagram. This action shifts the messages that follow this message one level up. If the sequence message set is "1, 2, 3, 3.1, 3.2, and 4", then deleting 2 from the diagram makes the set as “1, 2, 2.1, 2.2, 3”. Deleting a message also deletes its children. Again deleting 2 from “1, 2, 2.1, 2.2, 3” makes the resulting set as “1, 2”.

Selecting “Make Loop (Left)” or “Make Loop (Right)” on the menu changes the shape of the selected message as shown in figure 66. This option makes it easy to create self-calling messages. Selecting the “Edit” menu item opens the “Edit Message” dialog window. This dialog is shown in figure 67.

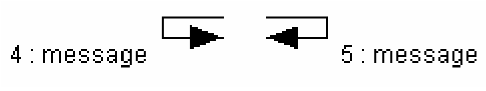
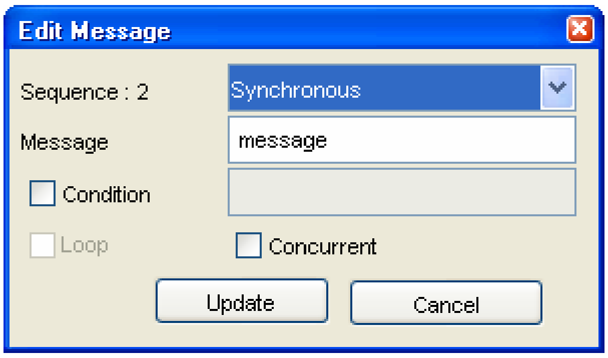


Figure 66. Self calling sequence messages


Edit Message

Sequence : 2 — Synchronous

Message — message

☐ Condition

☐ Loop — ☐ Concurrent

Update — Cancel

Figure 67. Edit message dialog

Message name and type can be updated with this dialog. Possible message types are “Simple, Synchronous, Asynchronous” that are shown in the combobox. Here, a message can also be made concurrent by selecting the checkbox. In addition, a message condition or a loop structure can be constructed with this dialog.

A.6. Properties of Sequence Messages

A.6.1. Segmented Structure

Sequence messages are made segmented so that any complex collaboration can be showed without massing up the diagram. To make segmented messages, segment drag points are implemented. A segment drag point allows stretching the message with the mouse. An algorithm is implemented so that segment lines are automatically made rectangular. Any number of segmented drag points can be added to a sequence message. Figure 68 shows the drag points on a segmented sequence message link.

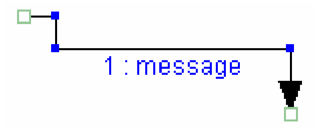


Figure 68. Segment drag points on a sequence message link

Drag points are shown as blue rectangles on the message link. Double clicking on a message link creates a drag point on it. To remove a drag point, double clicking on that point is sufficient.

A.6.2. Custom Message Text Positioning

Segment drag points are also used for managing the location of message text. Right clicking on a segment drag point opens a pop-up as shown in figure 69.

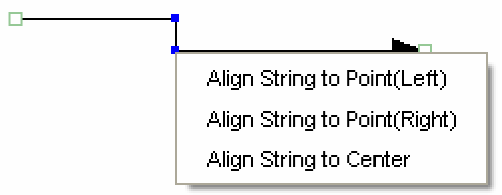


Figure 69. Pop-up menu to align text with segment drag points

Selecting first two menu items align the message text to the left or right of the selected drag point. Selecting “Align String to Center” menu item, places the text between the selected and the next drag points. This creates many possibilities ((number of drag points –1) * 3) for placing text on the link. So, in complex models, interference of text messages is minimized.

A.7. Modeling the Collaboration

How to add elements and how to insert sequence messages to a collaboration diagram are explained in previous sections. However, to model the collaboration, added elements should be connected using sequence messages. In abstract collaboration diagrams, connection is made to the element itself. Figure 70 shows the connection of abstract elements with sequence messages.

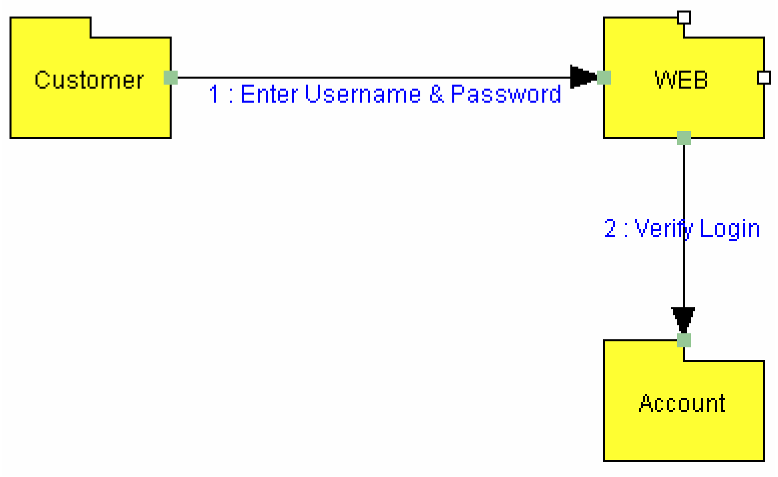


Figure 70. Connections in abstract collaboration diagrams

In run time collaboration diagrams, connection is made between the methods and events of the interfaces. Figure 71 shows the connection of interfaces through their methods and events.

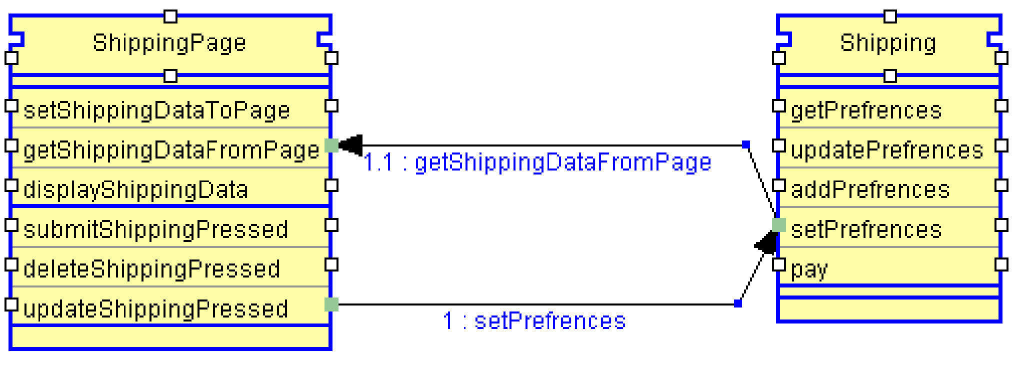


Figure 71. Connections in run time collaboration diagrams

For every method and event in a component interface, two connection points are defined on the both side of the interface. Therefore, method level messages can be shown by connecting over these connection points.