ENFORCING CONNECTION-RELATED CONSTRAINTS AND ENHANCEMENTS ON A COMPONENT ORIENTED SOFTWARE ENGINEERING CASE TOOL

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

BY

BARIŞ ÖZYURT

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

NOVEMBER 2003

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

Assoc. Prof. Dr. Ali Hikmet Doğru _____

Assoc. Prof. Dr. Nihan Kesim Çiçekli _____

Instructor Dr. Meltem Turhan Yöndem _____

Assistant Prof. Dr. Kayhan İmre _____

Msc. Gürkan Nişancı _____

ABSTRACT

ENFORCING CONNECTION-RELATED CONSTRAINTS AND ENHANCEMENTS ON A COMPONENT ORIENTED SOFTWARE ENGINEERING CASE TOOL

Özyurt, Barış

M. S., Department of Computer Engineering

Supervisor: Assoc. Prof. Dr. Ali Hikmet Doğru

November 2003, 58 Pages

This thesis introduces enhancements over an existing Component Oriented CASE Tool (CoseCase). Constraint checking facility is implemented for the connections provided in the tool: A user programmable set of rules governing the allowed connections among different modeling elements is added as a capability. The previous implementation of the tool did not consider the semantics behind the elements and their interconnection. Also related connection types are tested against cycle formations. Other aspects of the tool have been enhanced such as the dynamic graphical presentation of connection handles and connectors. Deleting a sub-tree from the design diagram is made operational besides the correction of faulty operating routines especially related to attaching new elements to the diagram.

Keywords: Component Oriented Software Engineering, Component Based Development, Component Oriented Modeling.

ÖZ

BİLEŞEN YÖNELİMLİ YAZILIM GELİŞTİRME MODELLEME ARACI İÇİN BAĞLANTILAR İLE İLGİLİ KISIT KONTROLÜ VE İYİLEŞTİRMELER

Özyurt, Barış

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Danışman: Doç. Dr. Ali Hikmet Doğru

Kasım 2003, 58 sayfa

Bu tezde Bileşen Yönelimli Modelleme Aracı (CoseCase) üzerinde bir takım iyileştirmeler yapıldı. Aracın sağladığı bağlantılar üzerinde kısıt kontrolü yapıldı. Değişik modelleme elemanları arasındaki bağlantılar ile ilgili kullanıcı tarafından tanımlanabilen kurallar koyuldu. Aracın bir önceki versiyonunda modelleme elemanlarının arkasındaki mantık göz önüne alınmıyordu. Ayrıca bu bağlantıların gerekli olanlarında döngü oluşup oluşmadığı da kontrol ediliyor. Bunların dışında araçta bağlantı kulplarının dinamik olarak gösterilmesi/gizlenmesi, modelden elemanın, altındaki tüm elemanlarla beraber silinmesi gibi bazı iyileştirmeler yapıldı.

Anahtar Kelimeler: Bileşen Yönelimli Yazılım Mühendisliği, Bileşen Yönelimli Yazılım Modelleme Dili, Bileşen Tabanlı Uygulama Geliştirme

To My Family

ACKNOWLEDGEMENTS

I express sincere appreciation to Assoc. Prof. Dr. Ali Hikmet Doğru, for his guidance and encouragement throughout the research. I offer sincere thanks to my family for their emotional support.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

CHAPTER 1

INTRODUCTION

Software development methodologies have constantly been in a struggle to improve the ways to reduce time, effort and cost of software development. Design techniques and software development process models have advanced and lots of effort has been spent on finding ways to reuse previously developed software units in other projects. Software reuse is using previously written software units in other software projects. Reuse does not only consider source codes, but also data, architecture, technology, utilization and optimization knowledge can also be reused.

Component technology has enabled software developers to easily benefit from software reuse in software projects. A software component is an independent and replaceable part of a system that has a function in a well-defined architecture. Component technology gave rise to a new notion to software development methodology: Building by integration of components rather than writing code from scratch.

Software development by the integration of software components is known to be Component Oriented Software Engineering or Component Based Development. In component based development, firstly, the system's specification is defined, then the system is decomposed into smaller subsystems and these subsystems are decomposed finally into software components. Next, software components satisfying the needs required by the decomposed system, are searched, developed or adapted for integration. Finally system is built by the integration of components.

Software components are integrated according to a component architecture in the integration stage. Component architectures provide standards to create components, to invoke methods, to pass arguments, to retrieve results of methods, etc.

Component Oriented Software Engineering Modeling Language [COSEML] is developed for component oriented development. Component oriented development stages explained above are all supported by COSEML. Also a graphical editor for COSEML was developed by Aydın Kara in his thesis [11]. With the graphical editor, the decomposition of the system into subsystems and software components can be modeled. The user decomposes the system into Package, Function, Data, Control and Connector abstractions, defines the relations among these abstractions and draws the overall architecture of the system, decomposed into abstractions with the relations among them.

1.1. Motivation for Implementing Constraint Checking

Constraints exist in most areas of human endeavor. They formalize the dependencies in physical worlds and their mathematical abstractions. A constraint is simply a logical relation among several unknowns (or variables), each taking a value in a given domain [12]. The possible values that variables can take are restricted by constraints.

Constraints arise in most areas of software engineering. The best example is Database Management Systems (DBMS) A typical DBMS include many constraint checking facilities : *Unique Key Integrity Constraints* or *Referential Integrity Constraints* are only two of the examples of constraint checking facilities in database systems.

Constraints are also used in modeling tools. A model is a representation of a system (from a given point of view) expressed in a formalism or language. A formalism is based on modeling principles, usually named a paradigm, that define the way any system should be considered (e.g., object-oriented, functional). Formalisms include syntactic and semantic constraints that define the meaning of the language primitives as well as the right way to use them. Any legal model should at least fulfill such constraints.

Component Oriented Software Engineering Modeling Language (COSEML) does not provide any constraint checking facility but there is need for constraint checking in COSEML. The links to set the relations among COSEML symbols can be used between any two symbols with no limitation. Allowing the user to use these links freely, brings some inconveniencies and some contradictions. So, COSEML should be extended to provide connection-related constraint checking and this constraint checking facility should be implemented in the graphical editor.

1.2. Organization of the Thesis

The organization of the remaining part of the thesis is as follows: In chapter 2, background information about software reuse, software components, component based software engineering, component architectures and COSE approach are provided. In chapter 3, first the design of the existing case tool and then the design and implementation of connection-related constraint checking are explained. In Chapter 4, other enhancements on the case tool are explained. Chapter 5 concludes the thesis and presents future work discussions.

CHAPTER 2

BACKGROUND

2.1. Software Reuse

Software developers rarely develop all of an application from scratch. Typically, an application is constructed from newly developed pieces, together with a collection of existing fragments produced during previous projects, acquired from third parties, or recovered from legacy systems. Software reuse can simply be defined as using previously written software units in other software projects. Using software units in other software projects sometimes requires some modifications on the software unit to adapt it to the new project. Software reuse which requires such modifications is called *white-box reuse*. But sometimes such modifications may not be necessary, software fragment can directly be used in the other project. This kind of reuse is called *black-box reuse*.

All companies want to benefit from software reuse in development process, because it provides many advantages over development from scratch. The most important of these advantages are:

- Software reuse reduces effort, time and cost of software development.
- Software reuse increases productivity and quality.
- Software fragments which will be reused are already tested and are known to be working correctly, so there will be less bugs in the software.
- Maintenance costs decreases.

Benefiting from these advantages has some cost because building reusable software objects requires some extra work. Firstly it requires extensive analysis and design. Extra time must be invested in testing, quality assurance, optimization and documentation. Different types of application code require varying levels of investment to achieve successful reuse. These are mainly :

- **Reusable GUI objects :** They reduce development time and improve quality and consistency and provide payback in terms of overall application development costs.
- **Server-side components:** Server-side components, which constitute reusable business logic can provide significant payback but require extensive analysis and design. They also require an architectural foundation but may have a short shelf life.
- **Infrastructure components** and **services frameworks:** These components are built for generic services such as transactions, messaging, security and database connectivity. The need to repeatedly build infrastructure that all applications use is eliminated, but require extensive analysis and design, and complex programming. These components can often be purchased off-the-shelf.
- **High-level patterns:** They provide means for organizations to achieve design reuse and identify components with high reuse potential, but developers must build or acquire the components.
- **Packaged applications:** Packaged applications provide the guaranteed form of reuse. Companies can acquire functionality for significantly less than the cost of building it themselves. However, these applications may not offer the exact functionality an organization needs.

2.2. Software Components

Components provide the potential to assemble applications much more rapidly than ever before. A key to assembling applications quickly is the reuse existing pre-built components to meet the application requirements.

There are two elements to designing for reuse: Designing components that can be reused and reusing components that already exist. When designing components within an application, it is possible to take steps to ensure that components are reusable in the future. When designing and developing an application, existing components can be reused to fulfill some, or all, of the application requirements.

2.2.1. Definition of a Component

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

To present a more concrete idea, a component should be considered to be:

- Modular,
- Extensible, and
- Open.

Modularity means that a software component contains everything needed to complete one or more tasks. For example, you could say that a spell-checking application is modular because it does not need anything else to complete its mission. Extensibility means that a software component can be told to fulfill more tasks than it was originally designed to fulfill. Moreover, it can be told to do its original tasks in new ways. For instance, an English-based spell-checking component could be told to check the spelling of German, Welsh, or Spanish documents. Openness means two things to software components. First it points to the capability to function on a number of platforms such as Unix, Windows, Mac OS, and so on. Second, it denotes the capability for one component to talk to another component using a single programming interface. With this type of openness, component developers can instruct one component to become aware of another component. These components can then automatically exchange information and tasks. For example, a spell-checking program could automatically know how to interact with a thesaurus, giving the user the feeling that he is using a single application, when in reality, there are really two applications that simply know how to speak the same language.

An easy way to understand how these features work together in defining software components is to consider the interaction between the various tasks within an imaginary word processing program, say “WordShop”. There is a speller, thesaurus, and many other tasks within WordShop that appear to uphold the ideals of component technology. They interoperate; they run on different platforms; they can be instructed to do more than their original programming through a powerful macro language; and they seem to work independently (as they work one at a time).

These tasks could be broken into separate software components that could be pulled out of the WordShop program and can run independently. This is what component technology is all about, the liberation of individual tasks into a state where they can exist and work together in the most appropriate manner. For example, a spell-checker component could be executed within a Web browser or within an operating system directly.

A component provides some functionalities and properties to be used by applications. Components provide this stuff through their “interfaces”. An interface is how a consumer of a component views that component. For the purpose of consuming a component, the consumer is concerned with the interface. Since the component is the implementation of an interface, the consumer is really concerned with the way the interface behaves. A component provides “Properties”, “Functions” and “Events” through its interface. Figure 1 graphically describes the overall picture of a component.

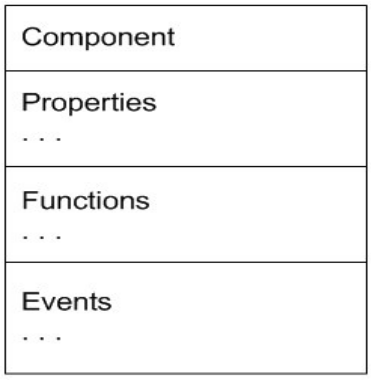


Figure 1. The interface of a component

To have a better understanding, consider a GUI component: a TextBox and its properties such as *font*, *width* and *height*. The function "SetFont(Font f)" to set the font of the TextBox or "GetText()" to retrieve the text written in the TextBox are functions provided by the Textbox component. The TextBox component sets the actions to be performed when an event occurs from the external environment or in the component. Such functions and events should be defined in the "interface" of the TextBox "component". For example, to set the actions when the user clicks on the TextBox, it may provide an event, "OnClick(MouseEvent e)" through its interface.

2.2.2. Understanding Component Concepts

It is easy to provide a general, extent understanding of what is meant by a component: it is a useful fragment of a software system that can be assembled with other fragments to form larger pieces or complete applications. However, to be able to make a comparison among specific component technologies and approaches, a much more precise analysis of component characteristics is required. For this discussion, a detailed conceptual model of component concepts including many important aspects of a component will be explained. Three particular perspectives that reveal many of the most interesting characteristics of components will be considered:

- **Packaging perspective**. A component as the unit of packaging, distribution, or delivery.
- **Service perspective**. A component as the provider of services.
- **Integrity perspective**. A component as a data integrity or encapsulation boundary.

2.2.2.1. Packaging Perspective

The packaging perspective considers a component to be an organizational concept, focusing on identifying a set of elements that can be reused as a unit. The emphasis here is on reuse. This is a very broad definition and covers any reusable software fragment including documents, source code files, object modules, link libraries, databases, and so on.

This is the perspective assumed by UML 1.0 and 1.1, defining a component as follows:

"A component is a reusable part that provides the physical packaging of model elements." [4]

It is useful to distinguish a number of different kinds of artifacts that could be considered a component. UML, for example, identifies a number of specializations (or stereotypes) of component such as executable, document, file, library, and table. From this perspective, each of these can be considered a particular kind of component.

It is also found useful to consider a special kind of packaging component focused on the physical packaging of an executable component. This is valuable, for example, when a component is an executable file or a Dynamic Link Library (DLL)[4]. In these cases, there is often specific information needed about the physical characteristics of the component required for determining where and how that component executes. This is a specialization of a packaging component which will be referred to as a component server, as illustrated in Figure 2. This shows *Component Server* as a specialization of *Packaging Componen*t.

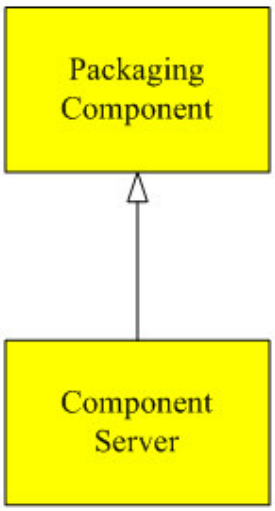


Figure 2. Packaging Component and Component Server (Adopted from [4])

The packaging of a component (executable) into servers generally will be based on the deployment or distribution requirements of that component. For example, a large enterprise-level component that handles one or more disparate databases may be partitioned into a number of DLLs, which can be allocated to specific nodes in an enterprise network. On the other side, simple desktop-based components can be bundled into a single library or executable to simplify delivery.

2.2.2.2. Service Perspective

The service perspective considers a component to be a software entity that offers services (operations or functions) to its consumers. The emphasis here is on components as *service provider*s. Designing and implementing an application involves understanding how a collection of components collaborate by making calls to each others' services.

In this perspective, the importance of a contract between the provider and the consumer of a set of services is highlighted. Services are grouped into coherent, contractual units, *interface*s. The interface can be considered as the contract because it describes everything that a potential consumer of that component's services can rely upon, and is the only way for a potential customer of those services to gain access to them.

This is the perspective taken in the *Component Description Model* (CDE), part of the *Open Information Model* (OIM) supplied with the Microsoft Repository. The CDE offers a services-oriented component definition:

> *"A component is a software package which offers services through interface*s." [4]

The service perspective of a component is a "logical" notion of a component because how a developer decides to partition the required functionality into meaningful service components is essentially a design decision. With respect to the earlier description of the "physical" packaging perspective of a component, a many-to-many relationship may exist between a service component and a component server. A

set of services for managing the maintenance of a list of customers forms the "logical" service component. In a particular implementation, this may be realized as many "physical" component servers via a set of related DLLs. This is illustrated in Figure 3. Both *Service Component* and *Component Server* are specializations of *Packaging Componen*t.

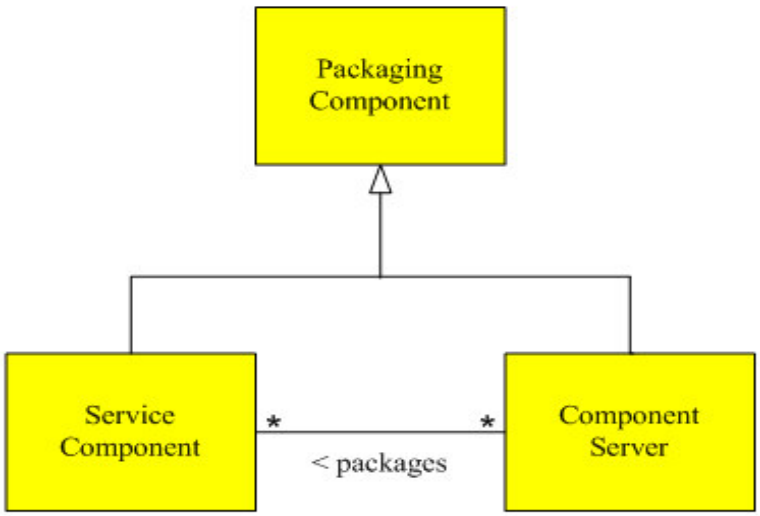


Figure 3. The packaging and service perspectives (Adopted from [4])

Concentrating the notion of a contract, the service perspective introduces an important distinction between the specification of a component *(*what it does) and its implementation (how it does it). This distinction is fundamental to the management of dependencies between components and begins to address the important requirement to be able to *replace* a component with minimal impact on the consumer, often referred to as "plug-and-play".

This distinction between specification and implementation is important. Consumers of a component should only be dependent on the specification of that component. Any dependency on its implementation (whether through direct knowledge or due to unspecified assumptions which happen to be supported) will mean that the application is likely to fail when the component is upgraded or replaced.

2.2.2.3. Integrity Perspective

Although the service perspective makes dependencies between components to be managed, it does not identify the component replacement boundary. This is a further perspective on components emphasizing that a component can provide an independent, replaceable unit of behavior. This can be referred as an *independent* component.

The integrity perspective defines a component as an implementation encapsulation boundary that set of software that collectively maintains the integrity of the data it manages, and, therefore, is independent from the implementation of other components. This criterion is a necessary condition for component replacement.

An integrity perspective is the approach supported by a number of different reuse-oriented technologies. Sterling Software's CBSE96 standard, for example, supports the reuse of business functionality. It defines a component as follows:

"A component is an independently deliverable package of software operations that can be used to build applications or larger components." [4]

This emphasis on independence is important because service components do not necessarily have implementation independence. Typically, they share data or have some other dependency on another component. As a result, collections of service components may be part of one independent, replaceable component. An independent component and all its sub-components form a single implementation encapsulation boundary and therefore can be replaced as a single unit. Sub-components are still components in that they offer services through interfaces, but they do not designate an encapsulation boundary. This is illustrated in Figure 4.

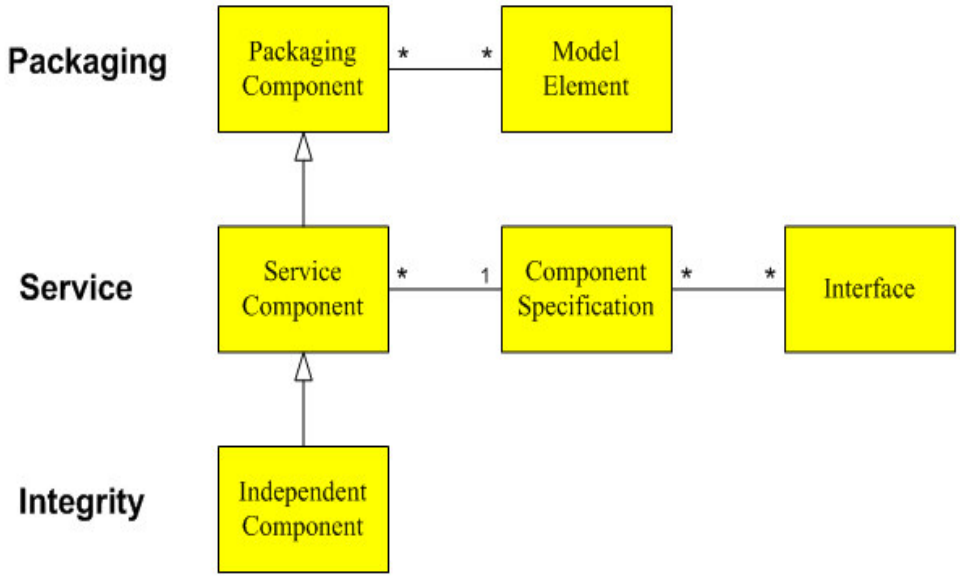


Figure 4. Three Component Services (Adopted from [4])

In Figure 4, the relationships between the three perspectives on components and enrich the conceptual model by introducing interfaces, component specifications, and model elements are illustrated. Each of these perspectives builds on the other. The service perspective, for example, is a specialization of the packaging perspective.

2.2.2.4. An Illustrative Example

To illustrate these different perspectives, consider a familiar component-based application: Microsoft Excel. The packaged item is *excel.ex*e. This is a single "physical" component server and contains a number of "logical" service components such as *Applicatio*n, *Char*t, and *Shee*t. Each of these is an independent component providing an encapsulation boundary. As a result, each of them could potentially be individually replaced. An alternative implementation of the *Sheet* component, for example, could be implemented which could interoperate correctly with the *Application* component, without having any implementation knowledge of the *Application* component.

Within each component, there are a number of sub-components. The *Sheet* component contains the *Range* and *Cell* components, for example. But *Shee*t, *Range*

and *Cell* share implementation and data knowledge and so are not independently replaceable; they can only be replaced as a unit.

Table 1 provides an illustrative categorization of a variety of existing software development artifacts into one of the preceding perspectives. The categorization given for a particular item relies on assumptions about that particular item. Class libraries, for example, have been placed in the packaging column because they do not separate specification from implementation, and reuse (through implementation inheritance) usually makes use of implementation knowledge. Frameworks have been placed in the integrity column on the assumption that they are binary modules that make calls to application-specific extension code, and their implementation logic is not exposed.

Table 1. Example Component Categorization (Adopted from [4])

Packaging	Service	Integrity
Files, documents, directories	Database services	Databases
Source Code Files	Operating System Services	Operating systems
Class Libraries	Function Libraries	Frameworks
Templates, tables	System utilities	ActiveX controls
Executables, dlls	Individual API functions	Some COM classes
	COM classes	Java Applets
		Applications
		Complete APIs

2.3. Component Based Software Engineering

There are efforts for institutionalizing software reuse for more than a decade. The effort of software engineering managers has been the move toward "software factories," where standard pieces of an application are selected from a catalog, assembled with some value-added local pieces or customizations, and rapidly deployed to the field to address a key business need.

In the 1980s, this vision was pursued by considering relatively broad application domains, gathering application pieces and waiting for the flood of customers to make

use of them. It didn't work. The majority of reuse initiatives did not succeed, and in hindsight, it is for all of the obvious reasons: the technical infrastructure supporting reuse was immature, cataloging assets was hard, the assets were diverse and of varying quality, the interfaces and behavior of assets were poorly defined.

In the late 1990s, there has been a number of interesting changes in the needs, tactics, and expectations of application developers. This, in turn, has resulted in important new requirements for the methods, tools, and technologies required to support them. There is a big trend towards developing software by integrating the readily developed software components. This software methodology which encourages the developers to consider the whole software system as a interoperating smaller systems and try to implement these smaller pieces with software components is *Component Based Software Engineering* (CBSE).

2.3.1. Designing Component Based Solutions

To take advantage of CBSE requires developers to begin to think differently about how to design and assemble applications. The goal of application development shifts from building single applications, to building a portfolio of reusable components from which a family of applications can be assembled. Taking this approach raises some important questions, including the following:

- What is the appropriate size and scope for a component?
- How do I describe dependencies that may exist among components in an assembled application?
- How do I document components to allow others to find them, and to assess their value within a given context?
- How does the maintenance and evolution of my application change when I make use of components? How do I manage this evolution?

Traditional software engineering methods mostly cannot answer these questions. To answer them requires fundamental changes in the way in which an organization carries out its software development and maintenance. These CBSE-oriented methods must allow applications to be developed by focusing on interfaces and interface-based

design, and by supporting the selection, evaluation, and assembly of components to create new applications.

2.3.1.1. Designing Reusable Components

During the interface design phase of a project, the architect can already be thinking about reuse of some particular function offered by the application. There are some techniques that will facilitate reuse:

- Look for common behavior that exists in more than one place in the system.
- Try to generalize behavior so that it is reusable.
- Plan on building facade controller interfaces that will make use of generalized components.

While attempting design for reuse, one must be careful. It is very easy to design solutions that are anything but reusable. Designing a solution that is reusable requires keeping the solution as simple as possible. If the component that is being designed for reuse is too complicated then one of two things will happen: it will be unusable because it is too complex; or only a small portion of the component will ever be reused because it was designed to handle much more than a consumer really needs, so a lot of effort was wasted designing and implementing it.

2.3.1.2. Designing Interfaces for Common Behavior

Extracting pieces of common behavior from an application is not a new thing; it has been around since people have been programming computers. The same principle exists when building components, although there are some differences in the way this is done. When building procedural code, common behavior is extracted into some kind of function or subroutine that can be reused in some way, either by having all users call the same function implementation, or by copying the function into code at compile time. When designing a component-based solution, it is possible to extract common behavior so that multiple consumers can use it.

For example, many applications need to make use of state/province codes, often for the purpose of finding these codes. It is possible to design an interface that offers the services necessary for many consumers to use the same state/province component.

Figure 5 shows an example of what this interface might look like. The interface provides the basic services necessary for consumers to find state and province codes. Any consumer that wants to find state or province codes can use the interface of this component.

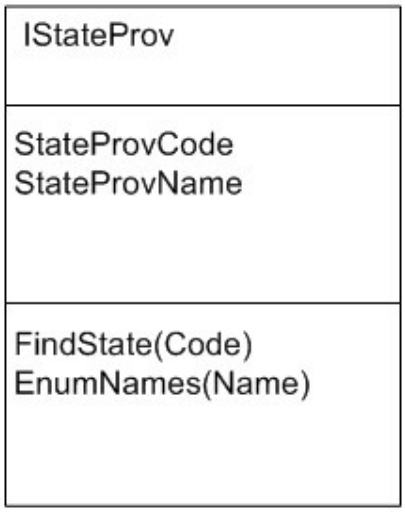


Figure 5. Common Behavior in an Interface (Adopted from [2])

2.3.1.3. Generalizing Function for Reusability

There are cases where the architect can generalize the function offered by an interface in order reuse it. By creating a generic, infrastructure interface it may be possible to reuse the interface, where it may not be possible to reuse the more specific interface.

Figure 6 shows an example of a generalized interface. It is a generic interface for a party, where the party can represent any type of party about which an application may be concerned. These are very simple interfaces that do little more than allow consumers to perform basic operations. If these interfaces are too complex when they are created, they will be difficult to understand and, therefore, difficult to reuse.

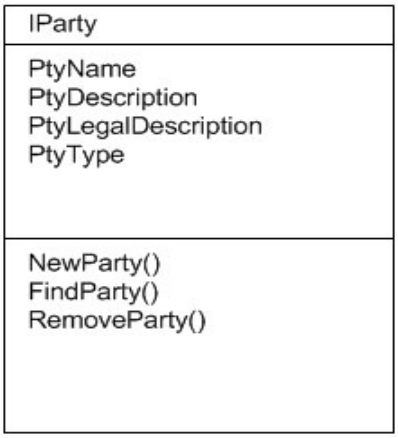


Figure 6. A Generic Interface (Adopted from [2])

2.3.1.4. Building Facade Interfaces

When designing generic interfaces it will be necessary to build facade interfaces since application-level consumers will not be consuming these infrastructure, generic components. A facade is a type of interface that passes control to other interfaces; it provides many of its services by using the services of other interfaces. The specification of a facade will not indicate to the consumer that there is another interface being consumed because it is using implementation-time reuse.

Figure 7 shows an example of a facade interface. The IStateProv interface exists and is used by consumers who are dealing with State and Province information. The interface makes use of the generic ICodeTable interface to provide its services. The IStateProv interface does not indicate anywhere that it is using the services of the ICodeTable, because it is doing so only during the implementation of IStateProv.

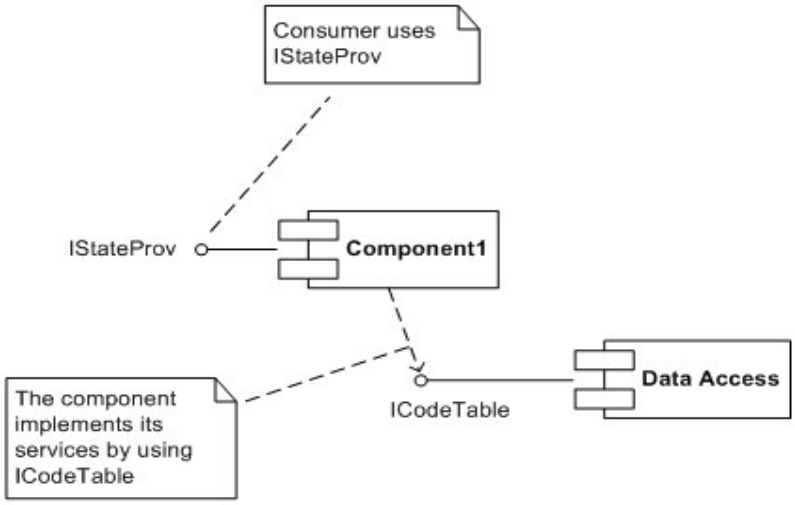


Figure 7. A Facade Interface (Adopted from [2])

This is a very simplistic example; it is a good example of a facade interface because the StateProv interface does not offer any services that are not offered by the CodeTable interface. There are other examples of facade interfaces in the Implementation-Time Reuse section below, where those interfaces offer services other than that of the generic interface that they are consuming.

2.3.1.5. Reusing Existing Components

Reuse of existing components can occur at two different levels: interfaces can be designed to use other interfaces at design time or at implementation time. Where the reuse occurs will affect the design of the application interfaces. If interfaces are associated at design time it means that all implementations of components in the specified system must support the interfaces as they are designed. If one interface consumes the services of another when the component supporting that interface is being implemented, the consumers of the interface have no idea that the second interface is being used to fulfill the implementation.

2.3.1.6. Design-Time Reuse

When designing interfaces for an application, the architect can create an application that has multiple interfaces with associations between them. The architect can create all of these interfaces during the design of the solution, or the architect may

already know of existing interfaces that can be incorporated into the application design. The interfaces that are being reused can be incorporated into the design of the application.

Figure 8 illustrates an example of design-time reuse. A customer interface already exists. As an order interface is being designed, it incorporates the customer interface into its design. In this situation, any implementation of the IOrderEntry interface will always require a component that supports the ICustomer interface. If a consumer is using the IOrderEntry interface and they want information about a customer who is taking part in an order, then they will use the ICustomer interface; IOrderEntry does not offer any of the services of ICustomer.

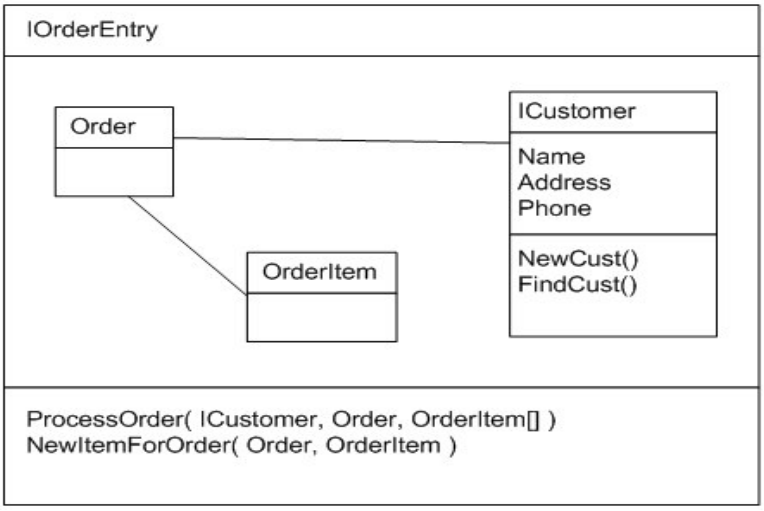


Figure 8. Design-Time Reuse (Adopted from [2])

2.3.1.7. Implementation-Time Reuse

Another method of reuse is to design interfaces that encompass the application's requirements and then look for components that will help to fulfill the obligations of the interfaces. Other components may offer services that will partly fulfill the services of the interface being created. These components can be used in the implementation of the interface.

The generic, infrastructure components that were discussed in the previous section are a good example of implementation-time reuse. A component that offers the ICodeTable interface (Figure 8) can be used at implementation time by another component that offers a different interface.

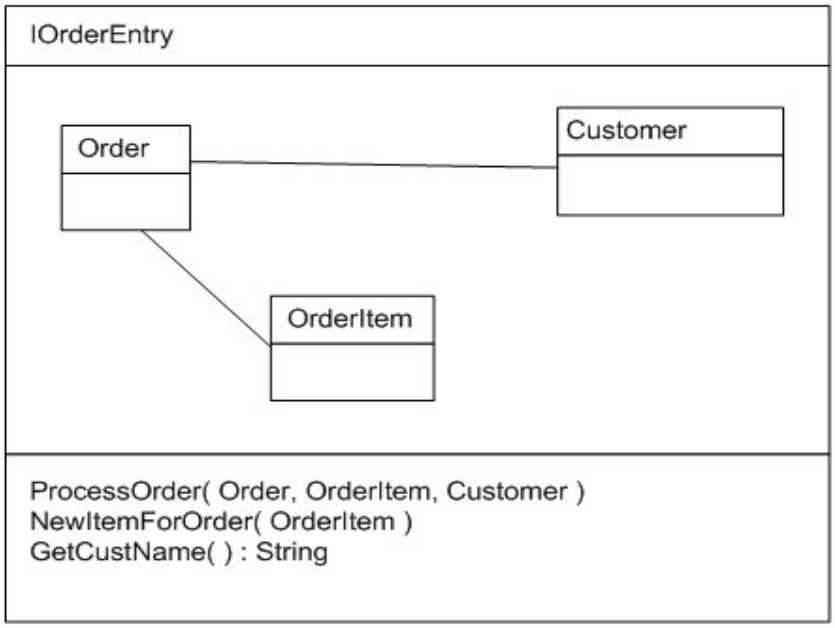


Figure 9. Implementation-Time Reuse (Adopted from [2])

Figure 9 shows another example of implementation-time reuse. The order entry interface is designed without any associations to other interfaces. A specification type has replaced the ICustomer interface that appeared in the specification, so customer information is part of the state information of IOrderEntry. Consumers of the IOrderEntry interface do not know that it is consuming services of other interfaces because this occurs within the implementation code of IOrderEntry. At implementation time, the order entry interface uses the services of the IParty and ICodeTable interfaces to handle customer information, as shown in Figure 10. The IParty interface handles basic customer information and the ICodeTable interface handles the implementation of the customer type code and description.

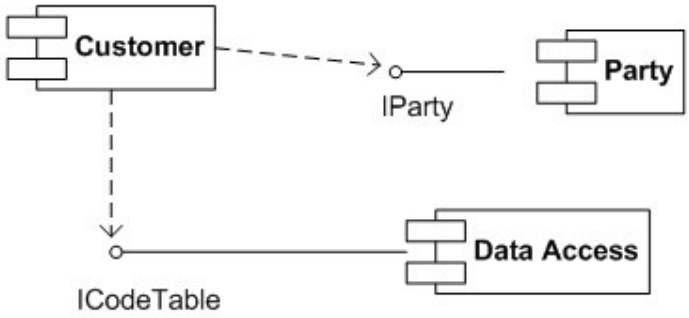


Figure 10. Using Services at Implementation Time (Adopted from [2])

2.4. Component Architectures

In the integration phase of component oriented development methodology, components obtained from different resources (developed, repository, web, ...) are integrated according to the component architecture to establish communication among these components. A component architecture is necessary to enable the components and client applications to communicate with each other through the components' interfaces without the knowledge of the internals of the components. Such software component architectures define a standard and provide a framework to integrate software components from different vendors, across address spaces, networks, and platforms.

In the integration of software modules in the source code level, interfaces among the modules is closed to outside world, it's open only to the vendor. And after the compilation and deployment of the software, these modules are hardly connected to each other and no more distinguishable. This situation prevents software component developers from providing more enhanced solutions and introducing new features. This brings a necessity for a standard to be defined for the integration of software components.

Microsoft provides Component Object Model (COM), its enhanced version COM+ and Distributed Component Object Model (DCOM). Object Management Group (OMG) provides the Common Object Request Broker Architecture (CORBA).

In the Java world, Sun Microsystems provide Java Beans and Enterprise Java Beans (EJB) architectures.

2.4.1. COM

The Component Object Model (COM) is a component software architecture that allows applications and systems to be built from components supplied by different software vendors. COM is the underlying architecture that forms the foundation for higher-level software services, like those provided by OLE. OLE services span various aspects of component software, including compound documents, custom controls, inter-application scripting, data transfer, and other software interactions.

These services provide distinctly different functionality to the user; however, they share a fundamental requirement for a mechanism that allows binary software components, supplied by different software vendors, to connect to and communicate with each other in a well-defined manner. This mechanism is supplied by COM, a component software architecture that:

- Defines a binary standard for component interoperability.
- Is programming language-independent.
- Is provided on multiple platforms.
- Provides for robust evolution of component-based applications and systems.
- Is extensible.
- Provides mechanisms for communications between components, even across process and network boundaries.
- Handles shared memory management between components.
- Makes error and status reporting.
- Enables dynamically loading of components.

The Component Object Model defines several fundamental concepts that provide the model's structural underpinnings. These include:

- A binary standard for function calling between components.
- A provision for strongly-typed groupings of functions into interfaces.
- A base interface providing:
    - A way for components to dynamically discover the interfaces implemented by other components.
    - Reference counting to allow components to track their own lifetime and delete themselves when appropriate.
- A mechanism to uniquely identify components and their interfaces.
- A "component loader" to set up component interactions and additionally in the cross-process and cross-network cases to help manage component interactions.

For any given platform (hardware and operating system combination), COM defines a standard way to lay out virtual function tables (vtables) in memory, and a standard way to call functions through the vtables. Thus, any language that can call functions through pointers (C, C++, Small Talk, Ada, and even Basic) all can be used to write components that can interoperate with other components written to the same binary standard. The double indirection (the client holds a pointer to a pointer to a vtable) allows for vtable sharing among multiple instances of the same object class. The vtable is depicted in Figure 11. On a system with hundreds of object instances, vtable sharing can reduce memory requirements considerably.

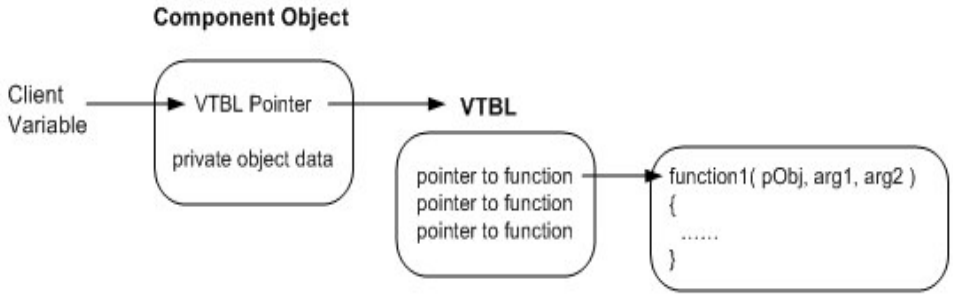


Figure 11. Diagram of a vtable (Adopted from [7])

In COM, applications interact with each other and with the system through interfaces. A COM interface is a strongly-typed *contract* between software components to provide a small but useful set of semantically related operations (methods). An interface is the definition of an expected behavior and expected responsibilities. OLE's drag-and-drop support is a good example. All of the functionality that a component must implement to be a drop target is collected into the **IDropTarget** interface; all the drag source functionality is in the **IDragSource** interface.

COM is language independent. Components can be implemented in a number of different programming languages and used from clients that are written using completely different programming languages. Again, this is because COM, unlike an object-oriented programming language, represents a binary object standard, not a source code standard.

2.4.2. DCOM

Distributed Component Object Model (DCOM) extends the COM to support communication among objects on different computers - on a LAN, a WAN, or even the Internet. With DCOM, applications can be distributed at different locations over the network. DCOM handles low-level details of network protocols so developers can focus on real business rather than complex network operations.

COM defines how components and their clients interact. COM set the standards for components to interact on the same machine. However, distributed applications require components on different machines to interact. When client and component reside on different machines, DCOM simply replaces the local interprocess communication with a network protocol. Neither the component nor its client is aware of the network protocols lying between them. Figure 12 depicts the DCOM architecture.

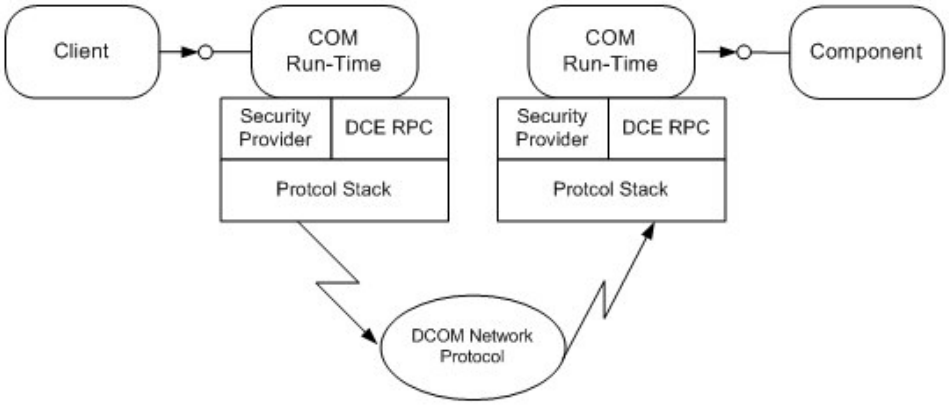


Figure 12. DCOM: COM components on different machines.
(Adopted from [8])

DCOM provides location independence, completely hides the location of a component, whether it is in the same process as the client or on a machine halfway around the world. In all cases, the way the client connects to a component and calls the component's methods is identical. DCOM's location independence greatly simplifies the task of distributing application components for optimum overall performance. Suppose, for example, that certain components must be placed on a specific machine or at a specific location. If the application has numerous small components, network loading can be reduced by deploying them on the same LAN segment, on the same machine, or in the same process.

Network connections are inherently more fragile than connections inside a machine. Components in a distributed application need to be notified if a client is not active anymore, even or especially in the case of a network or hardware failure.

DCOM manages connections to components that are dedicated to a single client, as well as components that are shared by multiple clients, by maintaining a reference count on each component. When a client establishes a connection to a component, DCOM increments the component's reference count. When the client releases its connection, DCOM decrements the component's reference count. If the count reaches zero, the component can free itself.

2.4.3. CORBA

CORBA is the acronym for Common Object Request Broker Architecture, is Object Management Group's (OMG) open, vendor-independent architecture and infrastructure that computer applications use to work together over networks. Using the standard protocol IIOP, a CORBA-based program from any vendor, on almost any computer, operating system, programming language, and network, can interoperate with a CORBA-based program from the same or another vendor, on almost any other computer, operating system, programming language, and network. CORBA automates many common network programming tasks such as object registration, location and activation, request demultiplexing, framing and error-handling, parameter marshalling and demarshalling and operation dispatching.

CORBA applications are composed of *objects*, individual units of running software that combine functionality and data. Typically, there are many instances of an object of a single type - for example, an e-commerce website would have many shopping cart object instances, all identical in functionality but differing in that each is assigned to a different customer, and contains data representing the merchandise that its particular customer has selected. For other types, there may be only one instance.

For each object type, such as the shopping cart above, an interface in OMG IDL(Interface Definition Language) is defined. The interface is the syntax part of the contract that the server object offers to the clients that invoke it. Any client that wants to invoke an operation on the object must use this IDL interface to specify the operation it wants to perform, and to marshal the arguments that it sends. When the invocation reaches the target object, the same interface definition is used there to unmarshal the arguments so that the object can perform the requested operation with them. The interface definition is then used to marshal the results for their trip back, and to unmarshal them when they reach their destination.

The IDL interface definition is independent of programming language, but maps to all of the popular programming languages via OMG standards.

This separation of interface from implementation, enabled by OMG IDL, is the essence of CORBA - how it enables interoperability, with all of the transparencies. The interface to each object is defined very strictly. In contrast, the implementation of an object is hidden from the rest of the system behind a boundary that the client may not pass. Clients access objects only through their advertised interface, invoking only those operations that that the object exposes through its IDL interface, with only those parameters (input and output) that are included in the invocation.

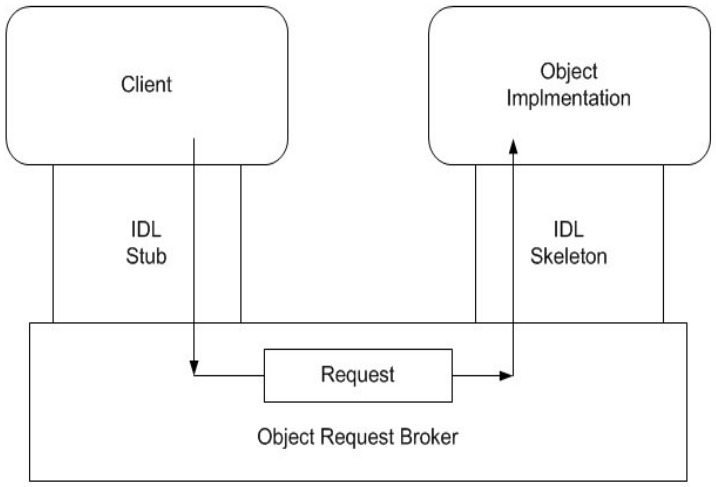


Figure 13. A request passing from client to an object implementation (Local invocation) (Adopted from [6])

Figure 13 shows how everything fits together, at least within a single process: IDL is compiled into client stubs and object skeletons. object (shown on the right) and a client for it (on the left) are written. Stubs and skeletons serve as proxies for clients and servers, respectively. Because IDL defines interfaces so strictly, the stub on the client side has no trouble meshing perfectly with the skeleton on the server side, even if the two are compiled into different programming languages, or even running on different ORBs from different vendors.

In CORBA, every object instance has its own unique *object reference*, an identifying electronic token. Clients use the object references to direct their invocations, identifying to the ORB the exact instance they want to invoke (Ensuring, for example, that the books you select go into your own shopping cart, and not into

your neighbor's.) The client acts as if it's invoking an operation on the object instance, but it's actually invoking on the IDL stub that acts as a proxy. Passing through the stub on the client side, the invocation continues through the ORB (Object Request Broker), and the skeleton on the implementation side, to get to the object where it is executed.

Figure 14 diagrams a remote invocation. In order to invoke the remote object instance, the client first obtains its object reference. To make the remote invocation, the client uses the same code that it used in the local invocation, substituting the object reference for the remote instance. When the ORB examines the object reference and discovers that the target object is remote, it routes the invocation out over the network to the remote object's ORB.

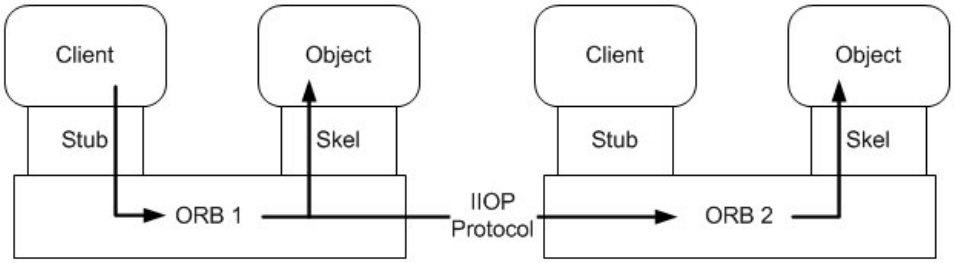


Figure 14. A remote invocation (Adopted from [6])

OMG has standardized this process at two key levels: First, the client knows the type of object it's invoking (that it's a shopping cart object, for instance), and the client stub and object skeleton are generated from the same IDL. This means that the client knows exactly which operations it may invoke, what the input parameters are, and where they have to go in the invocation; when the invocation reaches the target, everything is there and in the right place. Second, the client's ORB and object's ORB must agree on a common protocol - it's the standard protocol IIOP. (ORBs may use other protocols besides IIOP, and many do for various reasons. But virtually all speak the standard protocol IIOP for reasons of interoperability.)

Although the ORB can tell from the object reference that the target object is remote, the client can not. (The user may know that this also, because of other knowledge - for instance, that all accounting objects run on the mainframe at the main office in Tulsa.) There is nothing in the object reference token that the client holds and

uses at invocation time that identifies the location of the target object. This ensures *location transparency* - the CORBA principle that simplifies the design of distributed object computing applications.

2.4.4. Java Beans

The goal of the JavaBeans is to define a software component model for Java. Sun Microsystems gives Java Bean definition as follows:

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

This covers a wide range of different possibilities.

The builder tools may include web page builders, visual application builders, GUI layout builders, or even server application builders. Sometimes the "builder tool" may simply be a document editor that is including some beans as part of a compound document.

One of the main goals of the JavaBeans architecture is to provide a platform neutral component architecture. When a Bean is nested inside another Bean then a full functionality implementation on all platforms will be provided. However, at the top level when the root Bean is embedded in some platform specific container (such as Word or Visual Basic or Netscape Navigator) then the JavaBeans APIs should be integrated into the platform's local component architecture.

The need to bridge to other component models (notably OpenDoc, OLE/COM/ActiveX, and LiveConnect) is one of the constraints in the design of JavaBeans. This means that on the Microsoft platforms the JavaBeans APIs will be bridged through into COM and ActiveX. For example, the *ActiveX – Java Beans* bridge enables Java Beans to be used on Microsoft platform just like ActiveX objects.

Some Java Beans may be simple GUI elements such as buttons and sliders. Other Java Beans may be sophisticated visual software components such as database

viewers, or data feeds. Some Java Beans may have no GUI appearance of their own, but may still be composed together visually using an application builder.

Individual Java Beans will vary in the functionalities they support, but the typical unifying features that distinguish a Java Bean are:

- Support for "introspection" so that a builder tool can analyze how a bean works
- Support for "customization" so that when using an application builder a user can customize the appearance and behavior of a bean.
- Support for "events" as a simple communication metaphor than can be used to connect up beans.
- Support for "properties", both for customization and for programmatic use
- Support for persistence, so that a bean can be customized in an application builder and then have its customized state saved away and reloaded later.

A bean is not required to inherit from any particular base class or interface. Visible beans must inherit from java.awt.Component so that they can be added to visual containers, but invisible beans aren't required to do this.

Each Java Bean component has to be capable of running in a range of different environments. There is really a continuum of different possibilities, but two points are particularly worth nothing. First a bean must be capable of running inside a builder tool. This is often referred to as the design environment. Within this design environment it is very important that the bean should provide design information to the application builder and allow the end-user to customize the appearance and behavior of the bean. Second, each bean must be usable at run-time within the generated application. In this environment there is much less need for design information or customization.

The basic run-time model for Java Bean components is that they run within the same address space as their container. So for example, if the container is a Java application, then the contained bean is run in the same Java virtual machine as its container. If the container is a non-Java application, then the Java Bean will run in a

Java virtual machine that is directly associated with the application. (Normally this virtual machine will be running in the same address space as the application.)

The Java Beans specification does not prescribe a format to be used by an application to store the Java Beans it uses. In particular, for the special but very important case of an application builder, the Java Beans specification does not prescribe a format for projects, nor does it prescribe a format for the delivery of built applications. Although Java Beans does not specify a storage format, JAR is the most widespread used format for storing Java Beans. A JAR file is a ZIP format archive file that may optionally have a *manifest file* with additional information describing the contents of the JAR file. All JAR files containing beans must have a manifest describing the beans.

2.5. COSE Approach

Imperative languages (C, Pascal, ...) require functional decomposition of the system. In Object-Oriented (OO) methodology, system decomposition is data oriented. On the other hand, Component Oriented Software Engineering (COSE) is a methodology based on structural decomposition of the system.

COSE approach starts with top-down decomposition of the system by considering the existing components and the abstract design. Decomposition process continues until decomposed subsystems are expected to correspond to existing components. At this level a bottom-up approach is carried out and existing components are integrated to meet the requirements of the system.

Before discussing the COSE Modeling Language (COSEML), I want to explain the difference between Component-Oriented approach and Component-Based development: Most of the component-based approaches are Object-Oriented: The development starts with an OO requirements analysis and specification. Later, the OO model is supplemented with components that can be represented in the OO model. Although more specific steps are defined for the definition, utilization or creation of components, the graphical modeling of the system is heavily object oriented. Being able to accommodate components within the OO approaches define the component-based environments of today. Whereas, in component orientation, besides containing

components in the diagrams, the whole development process is aligned towards the utilization of components. That is why decomposition of the system specification is very important: the system is separated into logical modules that are expected to correspond to physical components. After the logical decomposition, implementation corresponds to gathering and connecting components. So, component orientation targets the composition of existing components rather than writing code or even rather than developing the components themselves. There can be many benefits or different levels of utilizing components. The component orientation mentioned in this thesis regards the paradigm shift towards composition, rather than code writing. Any approach that targets the development of a system through code writing does not satisfy the required orientation.

2.5.1. COSE Modeling Language

At the end of system decomposition process, COSE transforms the system into sets of connectors and components. The connectors form the skeleton of the system and represent the flows among the components. A pair of interfaces at the component level corresponds to a connector at the abstraction level.

Modeling the system starts top-down. Building blocks of the system are introduced. Recursively system is decomposed into subsystems. As the decomposition continues towards lower level blocks, interfaces among the blocks begin to appear and these interfaces are defined.

2.5.2. Graphical Modeling Elements

Abstract components correspond to “Package”, “Function”, ”Data” and “Control” abstractions. Package abstraction is represented by UML’s package symbol. Function, Data and Control abstractions are represented by oval, rectangle and hexagonal symbols respectively. Implementation level components require additional information to be represented other than object abstractions. Implementation level components can be represented with their *Properties, Methods* and *Events.* Figure 15 shows basic COSEML symbols.

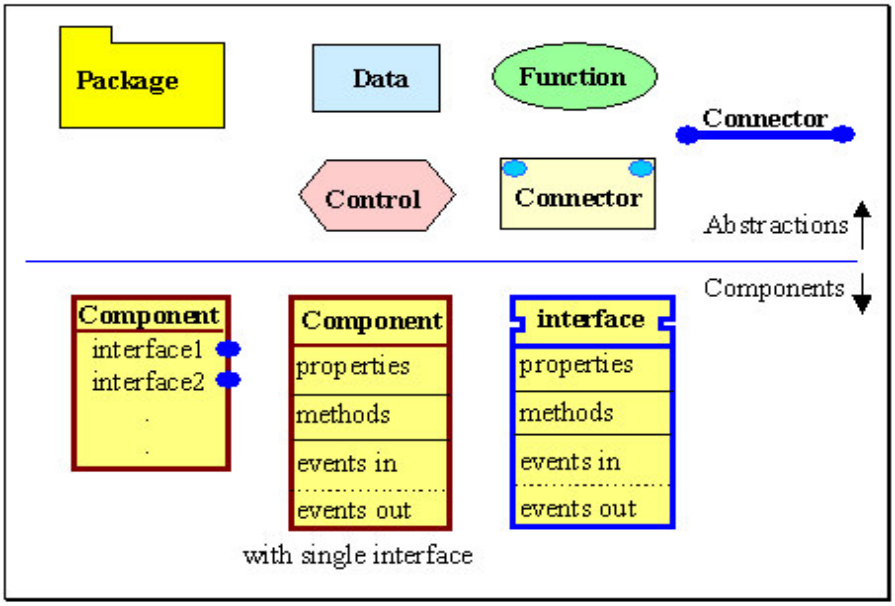


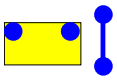
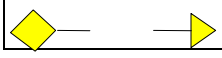
Figure 15. Graphical Symbols in COSEML (Adopted from [1])

Besides the object abstractions, also association links are required to connect the symbols in Figure 15. The default link between two symbols is the association link. The most special link is the "connector" represented by a "line" symbol or "Box" symbol. Line symbol is suitable for information hiding in the interface in the decomposition, whereas the box symbol is used for interface specification.

Since the system is represented by the object abstractions and there is correspondence between these abstractions and the implementation level components, system functions need to be represented at two different levels. The connector can be used between two abstractions as well as between two components. A connector between two abstractions may correspond to more than one message-link at the components level. In this case the synchronization semantics for different arrow-head shapes as defined in UML can be utilized.

Compositional links are used among components to construct super-components. The correspondence between a component and an abstraction is represented by the "Represents" relation. Table 2 presents detailed information.

Table 2. COSEML symbols and their meanings (Adopted from [1])

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

2.5.3. An Example Model

In this section, an example model is represented which is adopted from Associate Professor Ali Doğru's paper, which is released in IEEE with title "A Process Model for Component-Oriented Software Engineering". The example is about a small business automation software. The system consists of five main subsystems: Personnel, Inventory, Sale, Clients, and Accounting. The example model is depicted in Figure 16.

The main system (MyBusiness) is decomposed into subsystems and these subsystems are represented by COSEML abstractions at the upper part of the model. The existing implemented software components are represented at the bottom part of the model. The correspondences among the abstractions and existing implemented components are represented through the "Represents" links.

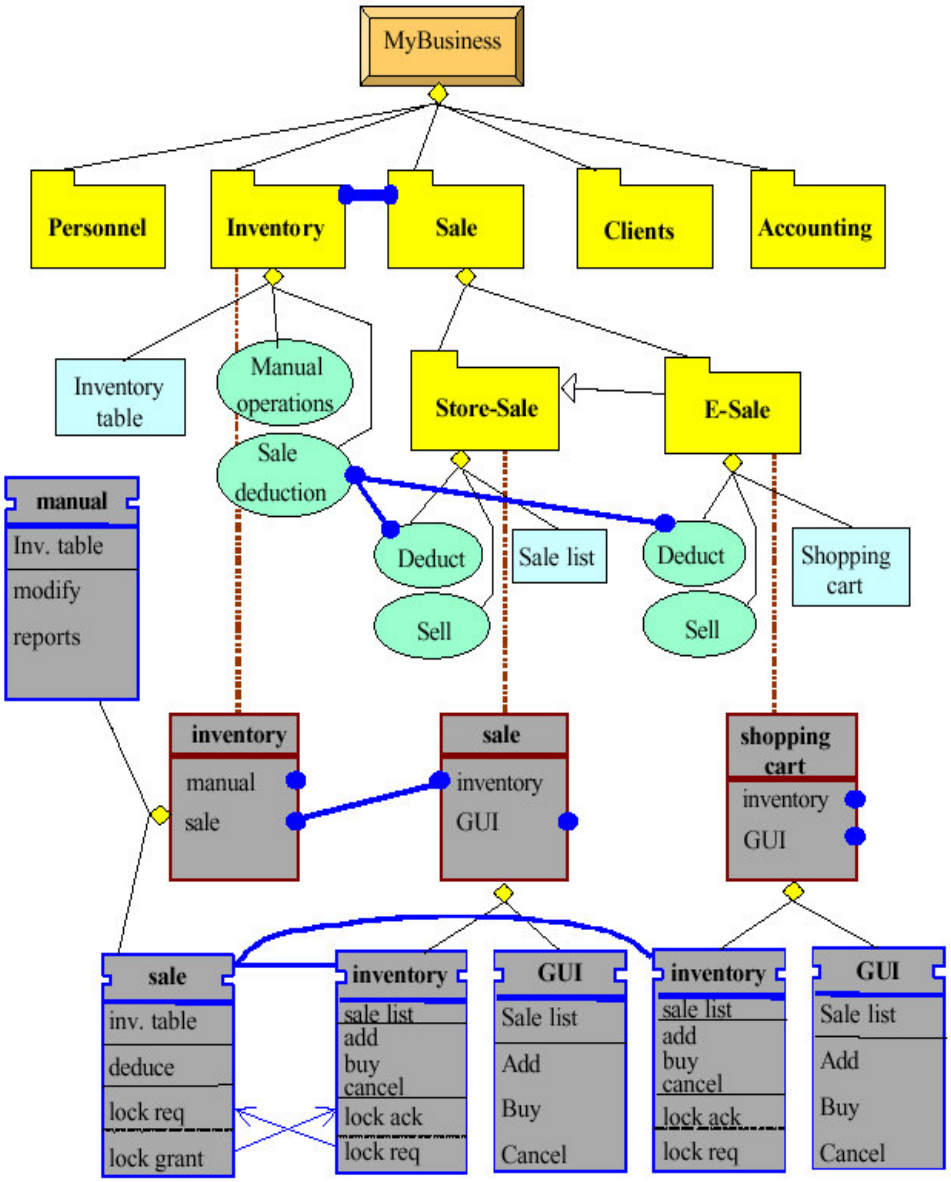


Figure 16 : An Example Model of a Small Business Automation Software (Adopted from [1])

CHAPTER 3

IMPLEMENTING CONNECTION-RELATED CONSTRAINTS IN COSEML

3.1. Constraints and Constraint Checking

Constraints arise in most areas of human endeavor. They formalize the dependencies in physical worlds and their mathematical abstractions naturally and transparently. A constraint is simply a logical relation among several unknowns (or variables), each taking a value in a given domain [12]. The constraint thus restricts the possible values that variables can take, it represents partial information about the variables of interest. Constraints can also be heterogeneous, so they can bind unknowns from different domains, for example the length (number) with the word (string). The important feature of constraints is their declarative manner, i.e., they specify what relationship must hold without specifying a computational procedure to enforce that relationship.

We all use constraints to guide reasoning as a key part of everyday common sense. “I can be there from five to six o’clock”, this is a typical constraint we use to plan our time[12]. Constraint problems are important in computer science. There is a branch of computer science: *Constraint Programming* dealing with the solution of constraint problems. It is the study of computational systems based on constraints. The idea of constraint programming is to solve problems by stating constraints (requirements) about the problem area and, consequently, finding a solution satisfying all the constraints.

Constraints and constraint checking is heavily used in many areas of software engineering. For example Database Management Systems (DBMS) support many constraint checking facilities to satisfy the data integrity. A typical DBMS uses integrity constraints to prevent invalid data entry into the base tables of the database. The user can define integrity constraints to enforce the business rules he wants to associate with the information in a database. If any of the results of a database query language (mostly SQL) statement execution violate an integrity constraint, then the DBMS rolls back the statement and returns an error. Constraint checking is an inevitable feature which must be provided a DBMS.

3.2. Constraints in UML

A model is a representation of a system (from a given point of view) expressed in a formalism or language [13]. A formalism is based on modeling principles, usually named a paradigm, that define the way any system should be considered (e.g., object-oriented, functional). Formalisms include syntactic and semantic constraints that define the meaning of the language primitives as well as the right way to use them. First, any legal model should fulfill these constraints. Then, to eliminate unwanted interpretations, users enforce additional constraints. These constraints stem from the modeled domain (e.g., to give a response before 15 ms), the implementation domain (e.g., no multiple inheritance), and even from the modeling process domain (e.g., a sequence diagram is required for each external event). To define constraints to get only meaningful interpretations, to describe and enforce these constraints, UML includes the notion of constraint and supplies the OCL (Object Constraint Language) that apply to both metamodel and model elements. The UML metamodel, i.e., the model of the formalism, cannot be changed. New notions are added using an extension mechanism, and again constraints are used to define the semantics.

Figure 17 represents a simplified model of the relationships between the main modeling notions. A *system* is anything to model. A *model* is an abstraction that represents a view of this system. The intent of a model is to represent a system into a *formalism* using a set of *abstract primitive*s. These primitives are relevant notions of modeling, (e.g., a *clas*s) that are mapped to forms, i.e., *concrete primitives* (e.g., a *bo*x). The *semantics* of the notions (e.g., what a *class* is) expresses their meaning using any language, including natural language. This semantics includes *constraints* that

induce restrictions on the use of notions in order to ensure that a model has at least one licit interpretation in the modeling domain.

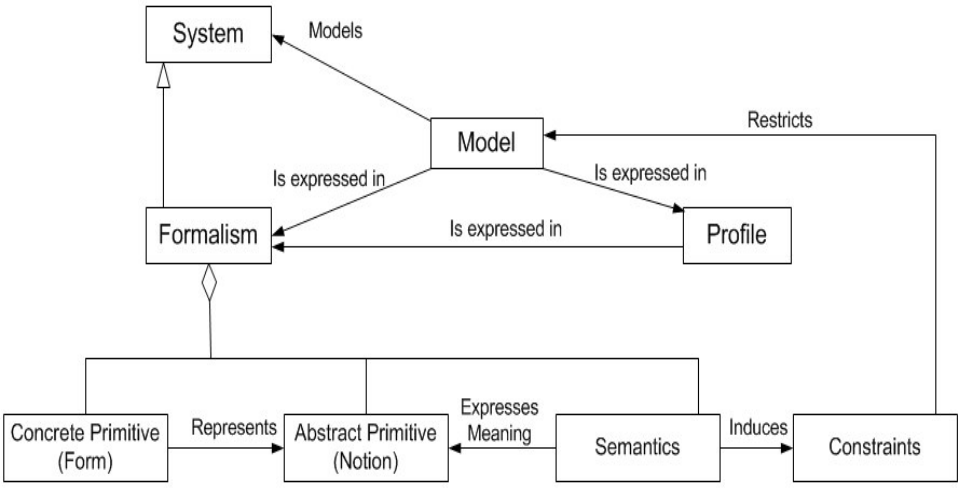


Figure 17. Relationships between system, model and formalism expressed in UML (Adopted from [13])

The entire UML constraints, including the metamodel well-formedness rules, are classified into five levels [13]:

- The *paradigmatic* level in which the semantics of the primitives of the formalism is defined,
- The *paradigmatic extension* level in which the semantics of the extensions is defined,
- The *modeling* level that customizes a formalism for a specific application domain or process and enforces the style guide,
- The *target domain* level that defines the meaning of the notions defined within the user application,
- The *implementation* level that ensures that the translation into a given target language is possible.

3.3. Constraint Checking in COSEML

Previous version of COSEML does not suggest any constraint checking. But in order to obtain a more consistent Component Oriented model, there is a need for constraint checking in COSEML. There are six kinds of links which can be used among abstractions and components in COSEML. They are *Composition, Connector, Inheritance, EventLink, MethodLink* and *Represents*. Detailed information about where and how to use these links is presented in Table 2.

In COSE approach, these links can be used freely. In other words, the user can use any link between any two abstractions or components. If we examine carefully the usage of these links by considering the semantic of the link, we see that it's not suitable to use them freely. For example consider the *Inheritance.* The most important rule of inheritance is that the subclass carries all the properties of the base class. We can freely use a *Inheritance* link from a *Function* abstraction to a *Data* abstraction (i.e. data abstraction inherits from function abstraction). The *Data* abstraction should carry all the properties of the *Function* abstraction. The *Function* abstraction contains some functionality and *Data* abstraction contains only data in itself. "*Data* inherits from *Function*" means *Data* contains all the functionality in *Function*. We said above that the *Data* abstraction only carries data in it. So, it's quite obvious that there cannot be an *Inheritance* from a *Function* abstraction to a *Data* abstraction.

Consider also the *Represents* link. *Represents* link is used to show the correspondence between implementation level components and abstractions. When the user discovers that requirements of an abstraction are implemented by a component, he puts the *Represents* link between the component and the abstraction. For example we cannot put a *Represents* link from a *Package* abstraction to a *Data* or *Function* abstraction.

Another constraint to consider on the usage of the association links is the cycles. Let's assume that we have two Packages: A and B. If A inherits from B, than B cannot inherit from A. Such cyclic inheritance relations among the COSEML symbols are semantically wrong. Inheritance relation should not be allowed to form cycles. This cycle constraint is not valid for all types of links, because cycles on some links are not

semantically wrong. Which links should be allowed to form cycles and which should not, will be discussed in section 3.6.

These are only three examples demonstrating the unsuitability of freely using the association links between two abstractions. Some constraints must be defined to limit the usage of links among the abstractions and components. So, COSE approach should be extended to include such connection-related constraints and these constraints should be implemented in the case tool COSECASE v1.0.

In my thesis, I've defined connection-related constraint checking for the COSE approach and implemented constraint checking on the case tool COSECASE. The design and implementation of constraint checking in COSECASE are described in section 3.5 and the implementation of cycle checking is discussed in section 3.6. But before in section 3.4**,** the existing design of COSECASE is explained.

3.4. Existing Case Tool

The graphical editor COSECASE v1.0 for the modeling COSE approach was developed by Aydın Kara[11]. Firstly COSECASE v1.0 is briefly described.

3.4.1. COSECASE v1.0

The tool is developed by using the Java programming language. Only core Java and swing classes are used during development.

By using COSECASE v1.0, software developers can use all COSEML symbols given in Table 2 to represent the system's component model according to COSEML. The hierarchical decomposition of a system into component abstractions and the relations among these abstractions can be modeled with the tool. Hierarchical decomposition is performed in two levels: Abstraction level and component level. During decomposing a system into abstractions, components corresponding to these abstractions and the relations among them are also drawn with the tool.

3.4.2. Design Overview

The class hierarchy of COSEML symbols including links among these symbols in the tree, squares around the shapes, selected shapes, etc. are depicted in Figure 18. All classes are derived from the Serializable interface to be able to write them to files, and retrieve from these files.

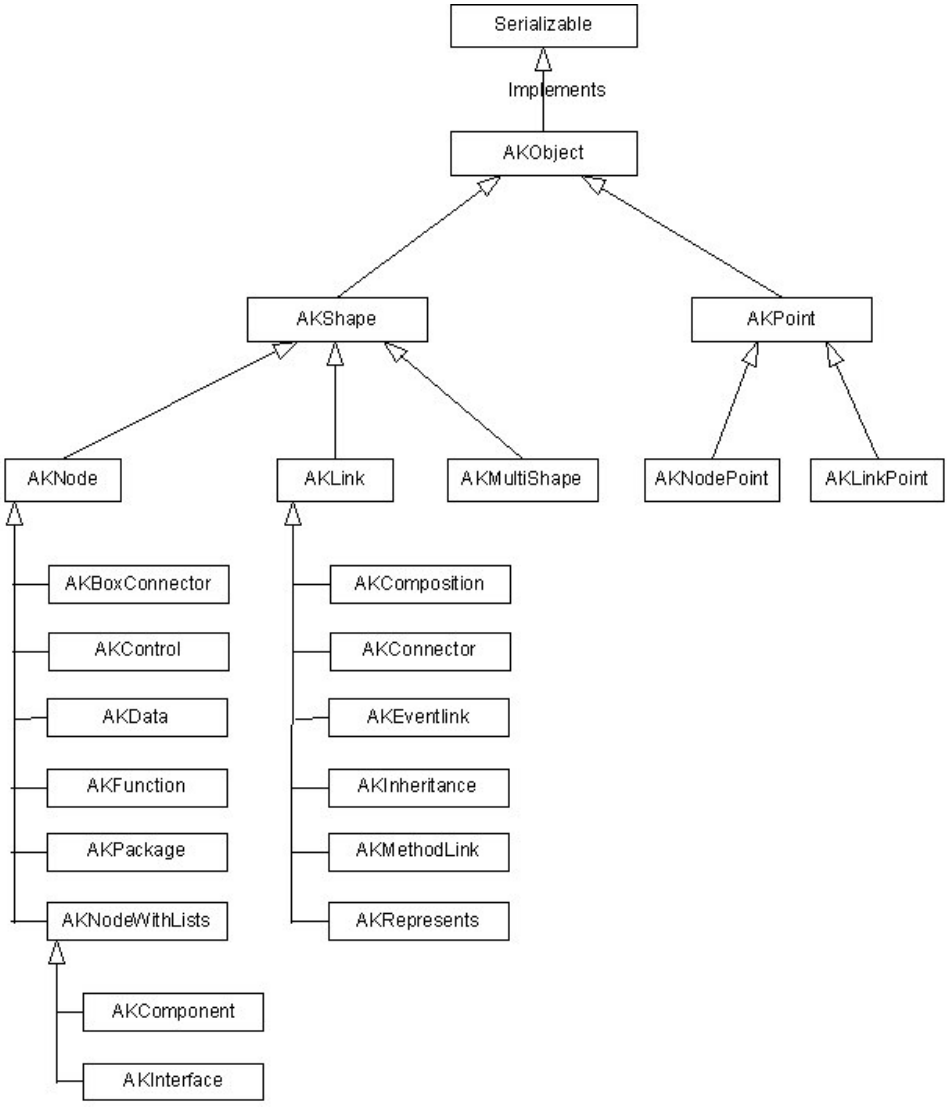


Figure 18. Class hierarchy of COSEML symbol classes (Adopted from [11])

To keep the tree structure of the decomposed system, a quadruply-linked-list structure is used. Figure 19 depicts this structure. On the left, there is a linked-list (rows), holding pointers to the first and the last elements of the rows. This list has an element for every row. The tree on the right consists of nodes representing the abstractions. During the decomposition of the system, new abstractions are added to the model and this tree grows in parallel. Different tree insertion algorithms (insertLeft, insertRight, insertChild) are used for these operations.

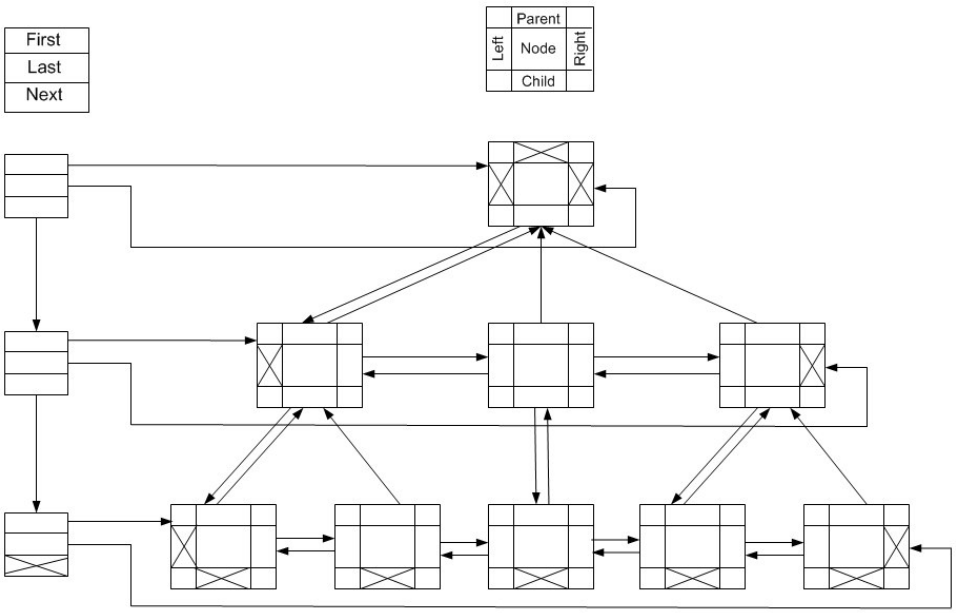


Figure 19. Tree structure of COSECASE (Adopted from [11])

3.5. Implementation for Connection-Related Constraint Checking

The rules organizing the connections among symbols for constraint checking are not hard coded. For flexibility, the rules are kept in a script file. The rules for constraint checking are defined in Table 3.

Abbreviations for links:

Cm : Composition	C : Connector
I : Inheritance	E : EventLink
M : MethodLink	R : Represents

Table 3. Rules for Constraint Checking

	Package	Data	Function	Control	Component	Interface
Package	Cm, I, C	Cm	Cm	Cm	R	
Data		Cm I C	C	C	R	
Function		C	Cm I C	C	R	
Control		C	C	Cm I C	R	
Component					Cm E M	Cm E M
Interface					E M	E M

Here a rule indicates an allowable connection. The first rule for example, in the table suggests that a Package can be connected to another Package through a Composition link. In the table, the rules are read from "Row to Column". The direction of the link is from the element represented in the row header, to the element represented in the column header. For example the *Represents* relation is from a Package to a Component, not from a Component to a Package.

These rules are not strict. As explained above, they are stored in a script file that is in text format, so that they can easily be changed. For determining the rules in Table 3, the following assumptions are made:

- Inheritance can only be between similar abstractions. For example, a Data abstraction can only inherit from a Data abstraction, not from a Package or a Function abstraction.
- Except for the Package, Composition can only be used between similar abstractions. For example, a Composition link cannot be drawn from a Data abstraction to a Function abstraction.
- Represents relation can be used from all abstractions (Note that component and Inheritance are not abstractions, they are not used at abstraction level, see Figure 16) to only Component.
- EventLink and MethodLink links can only be used at component level, they cannot be used among abstractions.

3.5.1. Script File Format

The script file holds all connection rules for each of the links. The format to keep the rules for a link is presented in Table 4.

Table 4: Script File Format

<LinkName>; <SymbolName>: <SymbolName-1> <SymbolName-2> … <SymbolName-N> <SymbolName>: <SymbolName-1> <SymbolName-2> … <SymbolName-N> <SymbolName>: <SymbolName-1> <SymbolName-2> … <SymbolName-N> <SymbolName>: <SymbolName-1> <SymbolName-2> … <SymbolName-N> <SymbolName>: <SymbolName-1> <SymbolName-2> … <SymbolName-N> <SymbolName>: <SymbolName-1> <SymbolName-2> … <SymbolName-N>

Where :

- LinkName is one of Composition, Connector, Inheritance, EventLink, MethodLink and Represents
- SymbolName is one of Package, Function, Data, Control, Component and Interface.

Here, the first line keeps the name of the link about which the rules are given. The link name is appended by a semicolon. In the following lines, the rules for each symbol about this link are given one by one. Let's take one line and examine it.

<SymbolName> in Table 5, corresponds to the "From" end of the link and the list of the symbol names on the right side of the colon correspond to the "To" end of the component. Shortly, this means link <LinkName> can be used from <SymbolName> to <SymbolName-1> or <SymbolName-2> … etc.

The Composition link's script according to rules given in Table 3 is presented in Table 5. For example, the second line in this script means that a Package can be connected to another Package or Data or Function or Control abstractions through the Composition link.

Table 5: The Script for the Composition Link

```
Composition;
Package:Package Data Function Control
Data:Data
Function:Function
Control:Control
Component:Component Interface
Interface:
```

This includes all the connection rules for the Composition. The script file consists of such blocks for each link type. Complete script file is listed in Appendix .

3.5.2. Implementation of Connection-Related Constraint Checking

There is need for a data structure to hold the rules that are primarily kept in the script file. As described in the previous section, the script file consists of consecutive blocks for each link type and the format is the same for each block, so instances of the same data structure can be used for each link type. Figure 20 depicts the data structure for a link type holding the connection rules.

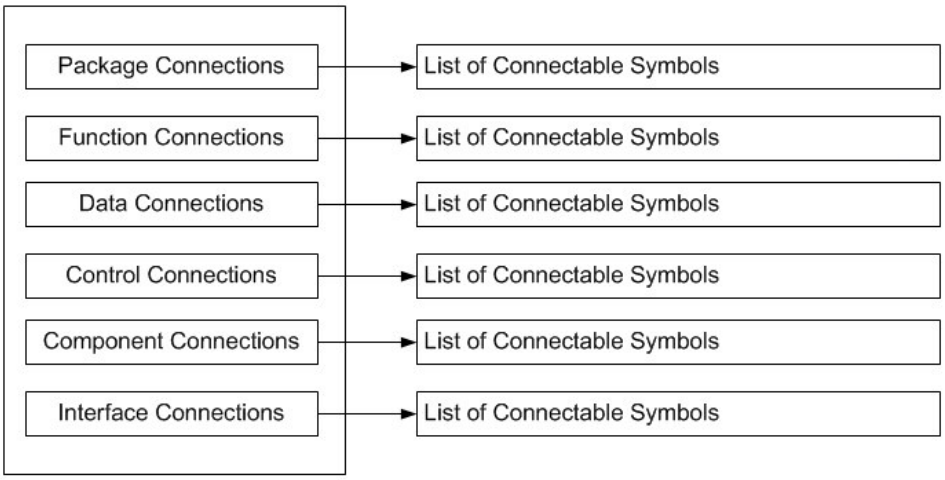


Figure 20. Data Structure for Composition Rules

This structure holds rules for one type of link such as Composition or MethodLink. For example, consider the Composition link. The instance of this data structure for the Composition link is presented in Figure 21.

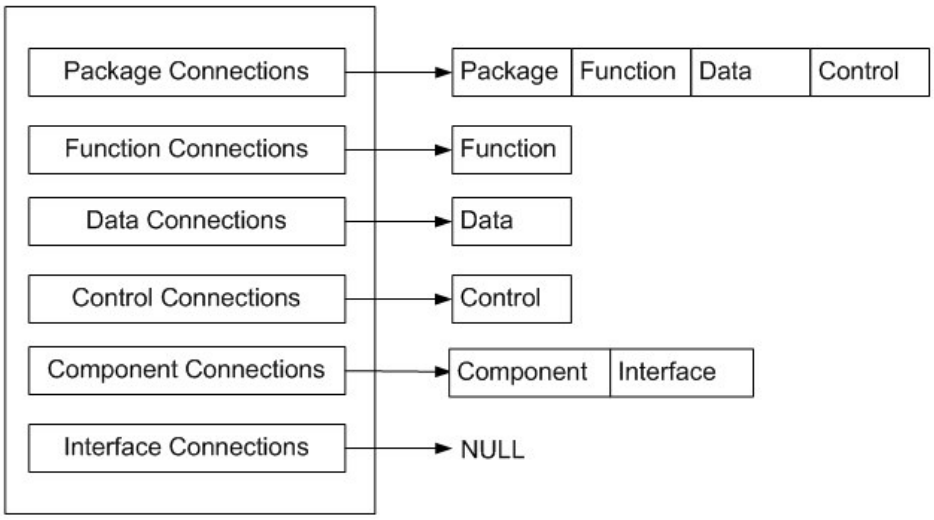


Figure 21. Data Structure for Composition Related Rules

Entire rules are held in an array of such structures for each link type. There are six kinds of links in COSEML, so the array consists of six rows.

At the beginning of the execution, COSECASE reads the connection rules from the script file and constructs the above data structure. When the user wants to establish a connection between two symbols, this connection is checked whether it satisfies the rules with the following algorithm :

```
Boolean canConnect(Node fromNode, Node toNode)
{
    /*Determine the node(nodeToSearch) at the "to" side of the link */
    if ( toNode instanceof  Package )
          nodeToSearch = "Package"
    else if ( toNode instanceof  Function )
          nodeToSearch = "Function"
    .
    .
    .
    else if( nodeToSearch instanceof Interface )
         nodeToSearch = "Interface"
```

```
    /* check if nodeToSearch is in the list of connecteable nodes */
     if( fromNode instanceof Package )
     {
          if ( nodeToSearch  is in  Package Connections)
               return TRUE
          return FALSE
     }
     else
     if( fromNode instanceof Function )
     {
          if ( nodeToSearch  is in  Function Connections)
               return TRUE
          return FALSE
     }
      .
      .
      .
     else
     if( fromNode instanceof Interface )
     {
          if ( nodeToSearch  is in  Interface Connections)
               return TRUE
          return FALSE
     }
}
```

If the attempted connection satisfies the connection rule implemented with the above algorithm, the connection is established, otherwise an error message is displayed to the user.

3.6. Design and Implementation for Cycle Checking

Before discussing the algorithm to detect a cycle, we should determine which links are allowed to form cycles and which are not allowed. Let us assume that we have two COSEML graphical modeling elements: A and B. The COSEML association links are discussed for the allowance of cycles according to this assumption:

- *Inheritance:* Inheritance is not a relation that is defined in COSEML. It is imported from Object-Oriented paradigm. In Object-Oriented paradigm Inheritance is not allowed to form cycles so also in COSEML Inheritance is not allowed to form cycles.
- *Composition:* Composition is also a relation that existed before COSEML. In Object-Oriented paradigm Composition is not allowed to form cycles so also in COSEML Composition is not allowed to form cycles.

- *Connector:* Connector is used to represent Data and Control flow across system modules. If there is a data flow from A to B, there can also be a data flow from B to A, so Connector networks are allowed to form cycles.

- *Represents:* Represents is used to show that an abstraction is implemented by a Component. If an abstraction is represented by a component, then the component cannot be represented by the abstraction, so cycles are not allowed to for "Represents" links.

- *MethodLink:* MethodLink is used to represent a method call among Components and Interfaces. Let us assume A calls B's method. B can also call A's method. So MethodLink is allowed to form cycles.

- *EventLink:* EventLink is used to represent an event flow among Components and Interfaces. Let us assume that there is an event flow from A to B, then there can also be an event flow from B to A. So EventLink networks are allowed to form cycles.

3.6.1. Implementation of Cycle Detection

The algorithm to detect the cycle in the COSE model is a simple graph cycle detection algorithm. When the user establishes a connection between any two symbols, this connection is checked for whether it has formed a cycle or not.

If the new connected link forms a cycle, the cycle must pass through this new link, also through the nodes at the "From" and "To" ends of the link. In the implementation of cycle detection, the node at the "From" end of the link is set as visited and the nodes are traversed recursively by following the same type of link. Starting from the node at the "From" end of the link. This traversing operation continues until there is no node to visit or the visited node (node at the "From" end of the link) is found. If the traversal finishes by finding the visited node, then there is a cycle, otherwise there is no cycle.

The pseudo code of this algorithm is as follows in Table 6:

Table 6 : Pseudo Code for Cycle Detection Algorithm

```
Boolean CheckCycle (TNODE  currentNode)
{
    if   currentNode is visited   then   return TRUE;   //  cycle detected
    nodeList  =  list of nodes reachable from currentNode
    for each node in nodeList   do
    {
          returnVal  =  CheckCycle( node );
    }
    return returnVal;
}
```

When the user establishes a connection between any two symbols, the node at the "From" end is set as visited and the CheckCycle function is called with this node given as the parameter.

3.7. Comparison with Existing Case Tools

Component Oriented approach is a newly maturing software development methodology and currently there are no tools developed for direct comparison with COSECASE and its constraint checking facility. Although there are no tools supporting Component Oriented methodology, there are some tools supporting Component Based development [11]. The most significant of them is Rational Rose. Latest versions of Rational Rose provide means to represent a component, its interface within the Object Oriented model. It does not support the facility of representing a hierarchical Component Oriented model. As explained in chapter 3, Rational Rose provides constraint checking feature. Because Rational Rose does not provide a component oriented view, its constraint checking notion is different than COSEML's. While COSEML checks the constraints on the links between the symbols, Rational Rose checks the component model for different constraints.

Enterprise Architect (EA) from Sparx Systems is another UML tool, which has Component Based development modeling support. EA's component diagram is similar to that of Rational Rose but it also includes some other features than Rose. EA provides a constraint notion in components but it's quite different than the constraint checking concept of COSECASE. In EA, some constraints can be associated with the components. These are conditions under which the element must exist and function. Typical constraints are pre- and post- conditions, which indicate things that must be true before the element is created or accessed and things which must be true after the element is destroyed or its action complete.

Microsoft Visio is a drawing tool, which includes graphical editing functionalities of UML. It also provides some other drawing facilities in component diagrams, which are not present in UML. Microsoft Visio does not have an integrated support for different views, it supports only drawing facilities. Visio also includes a constraint notion in UML diagrams as well as in UML Component diagrams. But its constraint facility is quite different than the one implemented in this thesis. It provides three types of constraints: "Box Constraint", "2-element Constraint" and "OR Constraint". Their definitions given by Visio are as follows:

- **Box Constraint :** A Box constraint is a specification for conditions and propositions that must be maintained as true for the system to be valid. Constraints are expressed as text within braces ({ }) and may be written in a predefined language, such as Object Constraint Language (OCL) or in natural language.

- **2-Element Constraint :** A 2-element constraint applies to two elements, such as two classes or two associations. The constraint is shown as a dashed arrow from one element to the other with the constraint string label in braces ({ }).

- **OR Constraint :** An OR constraint indicates that any instance of a class may participate in only one association at one time. The constraint is shown as a dashed line connecting two or more associations, which must have a class in common. The line is labeled by the constraint string, OR, in braces ({ }).

Visio's constraint facility is only at drawing and textual level. It does not enforce the user for any action or forbid the user from any action with respect to these constraints. The user writes the constraint textually and connects this constraint to components. Table 7 presents a comparison of these CASE tools.

Table 7. Comparison of CASE Tools

	Rational Rose	**Enterprise Architect (EA)**	**Visio**	**COSECASE**
Modeling Category	Object Oriented	Object Oriented	Object Oriented	Component Oriented
Component Oriented Views	Components, Interfaces	Components, Interfaces	Components, Interfaces	Components, Interfaces, Component Composition, Abstract Component Decomposition, Connectors
Constraints	Has constraint checking engine working at the background	Constraints can be associated with the components.	Box Constraints, 2-element Constraints and OR Constraints can be connected to modeling elements	Constraint checking on the relations between COSEML symbols.
User restriction	Constraint checking engine restricts users actions according to UML modeling constraints	Does not restrict the user. Many things, which should not exist in UML models, can be drawn with EA.	Only a drawing tool, does not restrict user.	Restricts the user on setting relations among COSEML symbols. Some relations are not permitted.

CHAPTER 4

ENHANCEMENTS

Besides implementing connection-related constraint checking, this research also includes the implementation of some additional facilities and completion of some incomplete facilities. One of these facilities is about the connection handles of the graphical modeling elements and another facility is the “Delete” function.

4.1. Displaying Connection Handles

In the previous version of the case tool, the connection handles of the graphical modeling elements were always displayed. This is not a correct and user-friendly implementation. A screenshot of a model drawn with the previous version of the case tool is presented in Figure 22.

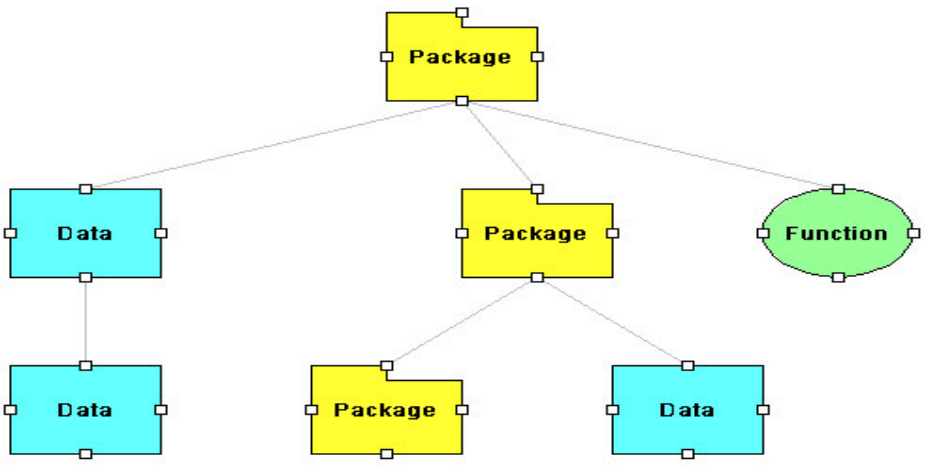


Figure 22. Screenshot of a model drawn with previous version of the case tool

As seen in the figure, all connection handles are displayed. With all connection handles displayed, the model does not look nice. Also connections should correspond to “readiness to connect” indication for the related component.

A good solution to this problem is to hide all the connection handles. While the user is moving the cursor over the model, only the connection handles of the symbol to which the mouse pointer approaches are displayed. For example if the mouse pointer approaches a symbol by less than 10 pixels, the connection handles of that symbol are displayed and all the others are not.

This logic was used for the display of the connection handles. On each mouse move event, the symbols are checked for whether their connection handles will be displayed or not. A screen shot of the same model with the current version of the COSECASE is presented in Figure 23.

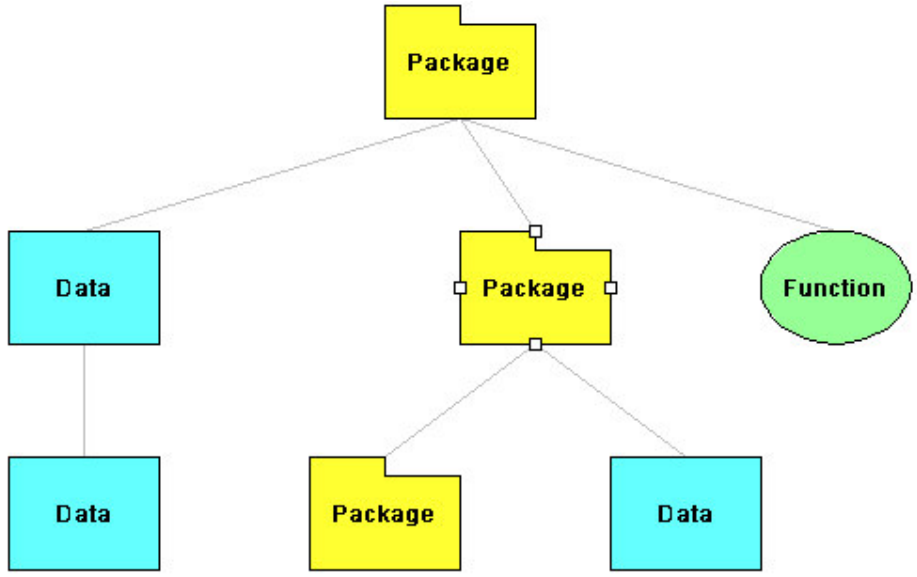


Figure 23. Screenshot of a model drawn with current version of the case tool

Here, the mouse pointer is close to the Package symbol that is in the second level and the handles of this Package symbol are displayed and the others are hidden. This model looks nicer than the previous one, and also is more correct.

4.2. Implementation of Delete Facility

In the previous version of COSECASE, the Delete facility was present in the user interface but it was not working, user was not able to delete the symbols from the model. The implementation of Delete facility was a must. The deletion of the leaf nodes is quiet simple; one single node is deleted from the model, but the deletion of inner nodes is more sophisticated. The choice was to delete the node and then recursively delete all the child nodes of the node. Figure 24 depicts the Delete operation in COSECASE. The node to delete is shown in red rectangle.

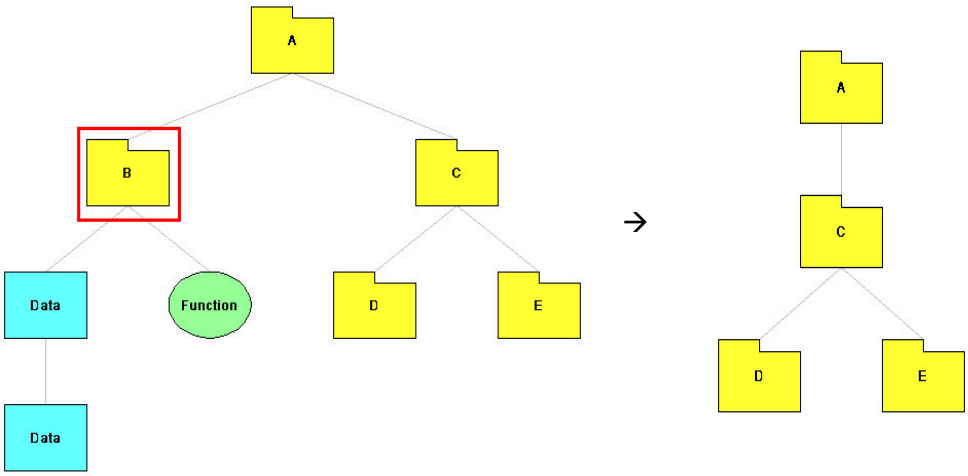


Figure 24. Delete operation in COSECASE

Delete is implemented recursively. When a node is deleted by the user, all the child nodes of that node are deleted and this recursive operation continues until the leaf nodes are reached.

CHAPTER 5

CONCLUSIONS

The graphical editor, COSECASE, for COSEML has been implemented before. It enables the user to draw the component oriented software model of a system. In this thesis, a constraint checking facility is implemented for COSEML.

Connection-related constraint checking is implemented as the collection of rules organizing the relations (Inheritance, Represents, EventLink, etc.) among the COSEML symbols. Enabling the user to set any relation among COSEML symbols brings some inconveniencies; they are explained in the previous chapter. COSEML needed to be extended to include a constraint checking facility, which is implemented in this thesis.

Consequently a designer will have more semantic control over the icons. With the example set of constraints, the tool usage became more instructive: the logically wrong drag/drops are not accepted by the tool. The overall usability of the tool feels to be increased.

5.1. Future Work

The constraint checking algorithm implemented in this thesis is designed for only to control the connections among symbols. A more generic algorithm can be developed for the implementation of different constraints. Maybe a rule-based engine can be implemented for a more effective constraint checking facility.

COSECASE gives the whole structural view of the system with nodes and connectors among them. Different views support can bee added to the tool. For example, a “Data View” displays only the Data abstractions or a “Function View” displays only the Function abstractions, so that the user can see the system from different views.

COSECASE, for now, does not have any integration with any component architecture. It can be integrated with component architecture and some work may be done on application generation from the model.

REFERENCES

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

[2] Kirby McInnis, "Component-Based Design and Reuse" *Article*
http://www.cbd-hq.com/articles/1999/990715_cbdandreuse.asp
Last access : 29.08.2003

[3] Alan Radding, "Hidden Costs Of Code Reuse" Article
http://www.informationweek.com/708/08iuhid.htm
Last access : 29.08.2003

[4] Alan Brown, "Building Systems from Pieces with Component-Based Software Engineering" *White Paper, Macmillan Technical Publishing, July 1999*

[5] Tom Spitzer, "Component Architectures" *Article, DBMS* and *Internet Systems, 1997*

[6] Object Management Group, "A high-level technical overview"
http://www.omg.org/gettingstarted/corbafaq.htm
Last access : 29.08.2003

[7] S. Williams and C. Kindel, "The Component Object Model: A Technical Overview" MSDN Library, Microsoft Corporation, Oct. 1994.

[8] "DCOM Technical Overview" MSDN Library, Microsoft Corporation, November 1996.

[9] Graham Hamilton, "JavaBeans API Specification" Version 1.01-A August 8, 1997. Sun Microsystems.

[10] Vedat Bayar, "A Process Model for Component Oriented Software Development", Middle East Technical University. M. S., Department of Computer Engineering. November 2001

[11] Aydın Kara, "A Graphical Editor for Component Oriented Modeling", Middle East Technical University. M. S., Department of Computer Engineering, April 2001.

[12] Roman Bartak, "Constraint Programming: In Pursuit of the Holy Grail" Malostranske namusti 2/25, 118 00 Praha 1, Czech Republic.

[13] Jean Louis SOURROUILLE, Guy CAPLAT, "Constraint Checking in UML Modeling"

[14] Oracle9*i* Database Concepts Release 2 (9.2) Part Number A96524-01 http://www.csis.gvsu.edu/GeneralInfo/Oracle/server.920/a96524/c22integ.htm Last access : 29.08.2003

APPENDIX

THE SCRIPT FILE

The connection rules of the COSECASE are kept in a script file for flexibility. The initial connection rules among COSEML symbols in this thesis is as follows :

Abbreviations for links:

Cm : Composition C : Connector
I : Inheritance E : EventLink
M : MethodLink R : Represents

	Package	Data	Function	Control	Component	Interface
Package	Cm, I, C	Cm	Cm	Cm	R	
Data		Cm I C	C	C	R	
Function		C	Cm I C	C	R	
Control		C	C	Cm I C	R	
Component					Cm E M	Cm E M
Interface					E M	E M

In the table, the rules are given from “Row to Column”. The direction of the link is from the element represented in the row header, to the element represented in the column header.

The segment in the script file to keep the rules for a link is as follows :

<LinkName>;

<SymbolName>: <SymbolName-1> <SymbolName-2> ... <SymbolName-N>

<SymbolName>: <SymbolName-1> <SymbolName-2> ... <SymbolName-N>

<SymbolName>: <SymbolName-1> <SymbolName-2> ... <SymbolName-N>

<SymbolName>: <SymbolName-1> <SymbolName-2> ... <SymbolName-N>

<SymbolName>: <SymbolName-1> <SymbolName-2> ... <SymbolName-N>

<SymbolName>: <SymbolName-1> <SymbolName-2> ... <SymbolName-N>

Here, the first line keeps the name of the link about which the rules are given. The link name is appended by a semicolon. In the following lines, the rules for each symbol about this link are given one by one.

The script file according to the rules given in the table and according to the given format is as follows :

```
Composition;
Package:Package Data Function Control
Data:Data
Function:Function
Control:Control
Component:Component Interface
Interface:

Connector;
Package:Package
Data:Data Function Control
Function:Data Function Control
Control:Data Function Control
Component:
Interface:

Inheritance;
Package:Package
Data:Data
Function:Function
Control:Control
```

Component:
Interface:

Event;
Package:
Data:
Function:
Control:
Component:Component Interface
Interface:Component Interface

Method;
Package:
Data:
Function:
Control:
Component:Component Interface
Interface:Component Interface

Represents;
Package:Component
Data:Component
Function:Component
Control:Component
Component:
Interface: