Bridging Component Models and Integration problems

Natan Gelber
Natan Gelber

Bridging Component Models and Integration problems

2006
Abstract

Component based software engineering promotes the use of software component as a cornerstone for new software. Taken from other manufacturing fields, the vision of selecting the components needed and building them as Lego to create a new software seems to offer enormous advantages for software development. Worldwide, industry and scholars are trying to remove the obstacles obstructing this course of action. This report names some of the problems that software integrators and developers face. It concentrates on low-level communication and other aspects of integration problems. It is based upon practical and theoretical research. The conclusion of the report is that while each problem can be solved in and of itself, these solutions often lead to the creation of new problems, and therefore leave ample room for further research.
## Contents

1 **Introduction** 1

2 **Problem and Methods** 3
   2.1 Problem .......................................................... 3
   2.2 Method .......................................................... 3

3 **Software Components** 5
   3.1 Component Definition ......................................... 5
   3.2 Component Model .............................................. 5
   3.3 CORBA Component Model (CCM) ............................ 6
   3.4 Enterprise JAVA Bean (EJB) ................................. 8
   3.5 .NET ............................................................ 10
   3.6 Summary ........................................................ 11

4 **Commonality and Variability in Components Model** 12
   4.1 Commonality in Components Model ......................... 12
   4.2 Variability in Components Models ......................... 14
   4.3 Building Bridges ............................................. 15

5 **Component and Platform Bridges** 16
   5.1 Bridge 1 IIOP.NET ............................................. 16
   5.2 Bridge 2 Borland Janeva .................................... 17
   5.3 Bridge 3 J-Integro Expresso ............................... 18
   5.4 Summary ........................................................ 19

6 **Integration more then building bridges** 20
   6.1 Finding components ........................................... 20
   6.2 Interoperability between components .................... 20
   6.3 Component Hidden interfaces .............................. 21
   6.4 Architectural mismatch or Assumptions Conflicts ........ 21
   6.5 Security or can i trust the component, QoS ............. 22
   6.6 Design for Evolution or thinking two steps ahead ....... 22
   6.7 Source Control ............................................... 23
   6.8 Summary ........................................................ 23

7 **Discussions** 25
   7.1 Reflections .................................................... 25
   7.2 Followup ....................................................... 26

References 27

Index
1 Introduction

This paper is a bachelor thesis in Computer Science at Växjö university. It deals with bridging applications for different component models. In writing it, the writer has integrated both theoretical research and practical one. However, the practical research was done as part of my work in a company which has a nondisclosure agreement, and hence it will not be presented.

Component Based Software Engineering (CBSE) is one of the popular methods currently used for software development. It is based upon connecting and integrating individual components with the purpose of creating new software or upgrading an existing one. The components can be developed by the user, or bought as component of the shelf (COTS). In either case, the user has to find a way to integrate between the various components. Integrating different components presents the user with problems in two levels: the communication / interpretability of components from different platforms and vendors; and the application conflicts raised by integrating components. This report discussed these problems and their possible solutions.

In the second chapter of the report, the theoretical problems concerning integration of different component models will be discussed, focusing mainly on two types of problems: communication between component models and the platforms they reside in and integration between different components. After presenting the background for the problems this report discusses, the method used to analyze these problems in order to try and suggest possible solutions shall be detailed.

Following that, the theoretical framework within which this paper was written will be supplied. Firstly, a definition of a component and component model will be presented. Than, an account of the three main component models currently used will be offered: CORBA Component Model (CCM), Enterprise JAVA Beans (EJB) and Microsoft’s .NET component model. The main characteristics of each of these models will be presented as well as their structure and functions.

The fourth chapter will discuss the commonalities and variability of different component models, identifying the common properties of the three models and pointing out the differences between them. The main goal of this chapter is to identify the key attributes that are needed for an application to function as a bridge between two or more models. At the end of each part of the chapter, a table summarizing the commonalities and variabilities between the various models will be presented.

The next chapter will review the reasons for success of applications in bridging between different models. This chapter inspects the three bridging application currently used in the market: IIOP.NET (an open source project); Borland Janeva - a commercial bridge - and J-inetgro Espresso - a C# based commercial CORBA implementation. It will also survey how the bridge application key attributes detailed in chapter 3 are implemented by these products and compare the different bridging applications.

Even after solving the interpretability problem, integration teams still face many problems in the development process. These problems will be discussed in chapter 6. firstly, the issue of component location will be discussed. Than, the interoperability between components will be reviewed. Following that, the issues of component hidden interfaces and architectural mismatches or assumption conflicts will be covered. Next security and trust issues will be elaborated upon. Lastly the subjects of reusability and source controls will be discussed. All these problems will be discussed both on a theoretical level, and through practical examples. Solutions to each of these problems will be suggested and their advantages and shortcomings discussed.

The last chapter of this paper includes some reflections about the subject as it was
presented in the paper. It analyzes the problems existing with current solutions to the bridging problems. This analysis is done in light of the writer’s own experience in his work and in his theoretical research. Suggestions for further research will also be detailed at the end of this chapter.

It is worth noting, that this paper is of limited scope only, due to time and length limitations. Hence, it does not aspire to cover all possible component models and or bridging applications, but only the ones most commonly used. Also, the writer is aware of the fact that many more problems exist in the field of bridging applications between different component models, however he chose to focus on the ones the seemed most crucial to him, based upon his practical experience. Further research, as suggested in the last chapter, may advance discussion of other issues as well.
2 Problem and Methods

2.1 Problem

This report deals with two types of problems that arise when integrating components. The first type is enabling communication between component models and the platforms they reside in. This is the first step in every integration project: making sure that the components can communicate and in a satisfying way.

To solve this problem one must identify what are the functionalities that a bridge application must have and how they can solve the communication problems. In this report I will review three bridge applications, identify their characteristics and compare between them in order to see the differences between them and the way each of them suits a different purpose.

The second type of problems arise after the communication is established. Having components communicate opens the door to a large range of problems that depend on different aspects of the components. For example, the way the components were built, the way they are delivered to the user, the quality of their documentation as well as what is missing in the documentation. All of that and more may cause problems that integrators face. This report covers seven integrations problems, which I believe are the most important ones. These problems are:

- Finding components
- Interoperability between components
- Component Hidden interfaces
- Architectural mismatch or Assumptions Conflicts
- Security or can I trust the component, QoS
- Design for Evolution or thinking two steps ahead
- Source Control

Every problem will be explained with an example and a suggested solution. However it is important to remember that no solution is bullet proof and that introducing a solution to one problem may create a new one.

2.2 Method

This paper integrates theoretical research, based upon the existing literature, with practical experience gained through a practical internship in development of bridging applications for different component models.

The first stage of writing the paper included mainly learning the existing studies about the three component models, which this report focuses upon: CORBA Component Model (CCM), Sun’s Enterprise JAVA Beans (EJB) and .NET. Following that, I reviewed the currently available applications for bridging between these models.

At the second stage of the research, I developed a test application using two of the component models and a bridge application. The implementation process revealed what are the key requirements from a bridging application along with the properties and functions it should fulfill. It also showed what the main obstacles to bridging between different component models are. Following that, I returned to the existing literature to see what solutions and ideas they suggest in order to solve these problems. I noticed, that while
many of these problems are well documented and have had various solutions suggested to them, others are not, and more importantly, some of these solutions create new problems. These problems and solutions are described in the 6th chapter of this paper.
3 Software Components

This chapter will present the theoretical background for the discussion about bridging applications. It starts with a general definition of what a component is and what properties it should include. Later, a description of a component model, how it is built and what can it offer to the components is supplied. The following three sections describe the most common component models today: the CORBA Component Model (CCM), Enterprise JAVA Bean (EJB) and .NET assembly.

3.1 Component Definition

A component is a software entity that gives a certain service through a known interface, and meets certain requirements and qualities. Sommerville[1] argues that a component is an independent entity that functions as a fundamental part of a system. He then elaborates upon 5 properties that a component should bear:

1. Standardized: a component needs to follow a standard component model, which defines the interface, metadata, documentation, composition and deployment.
2. Independent: The component cannot be dependent on other components for deployment and development.
3. Compassable: All communications to the component are done through the public interface and the component must also publish some external information, such as methods and attributes.
4. Deployable: A component is a standalone entity that can be used without compilation. In most cases the component is a binary entity.
5. Documented: A component must be documented so its user will be able to judge if the component meets his needs and have the knowledge needed to use it.

On top of these five qualities, it is worth noting that similarly to object oriented, support components data encapsulation includes a strict separation between the implementation of the component and the interface of the component. The interface gives only a functional view of the component while the capturing of the nonfunctional attributes is done through a contract. A contract of a component will list the global constraints the component has, the component’s preconditions (client needs) and post-conditions (component answers)[2]. Contracts can also be used for defining a group of components and their interactions. A component gives one service or more, as defined in its interface. A set of components can be gathered to a framework of components that give services and can be deployed as one. It is important to note, that a component can be a small unit like a class or a package, but it can also be a whole system that is used as a component by other systems.

3.2 Component Model

The services a component provides, its description and the communication specifications, are described in different contexts. So are other issues, such as the deployment, packaging, configuration and customization of a component. All this is defined in the model the component belongs to. The model also defines other aspects of the component such
as the way a component is identified and where this identification reference is kept. A component model can be divided to three main parts:

1. The interface which describes the component, the operation it performs and services that are given by it. A component has a single public interface that is accessible to other components and clients. This interface can contain more internal interfaces to different parts of the component. The component interface is used to capture the functionality of the component it describes (methods with input that return output), but it cannot always capture the semantics in between. A component model will support the development of the component by introducing a language that can be used to describe its interface; this language is known as IDL - Interface Definition Language.

2. Deployment: This part defines the way components are packaged, and the way to document components. A component model contains information and instructions on how to document the component so it can be successfully deployed.

3. Usage: This part defines how to use the component, the naming scheme given by the model, and the method by which a component can be configured. The naming scheme is one of the most important parts of the component model. It should support both local and remote searches for components and should enable remote invocation of methods and components. Managing components is also an important part of the component model. The components manager is responsible for configuration and replacement of components during runtime. The manager can assist in communication with the external services the model offers.

The model also provides the component with a set of services that it can use. These services are commonly divided into two categories: platform services that allow communication between components of the same model, and horizontal services, which are services used by different kinds of components. Examples for platform services are: naming, interface definition, exception handling etc. Horizontal services are, for example, transaction management, security, and so on.

The communication between the component and the model infrastructure is usually done through introducing a container that the component exists in. The communication between the component and the outer world is done through this container.

This section described what a component model is and how it can enhance components with more functionalities and simplify the use of components by introducing a set of tools that support component creation and description (IDL); component deployment (packaging, description) and component usage (component life cycle control, external services).

### 3.3 CORBA Component Model (CCM)

CCM was developed and maintained by OMG, and introduced as a component model to CORBA middleware. CCM designers use JAVABeans as a source of inspiration, which explains the similarities between the two models[3]. The CCM is an extension to the CORBA object model and gives a better common ground for developing and integrating CORBA servers and clients from different vendors, which operate on different platforms than the CORBA object model.

According to Jubilees and Favre[3], the CORBA component is an independent entity with its own data and logic that is supplied with a set of connections or interfaces describing the communication to and from it. The component is designed to be reusable in
application as is or with changes applied to it. The component interfaces (and the component itself) are described in CIDL: Component Implementation Definition Language.

The CCM provides two levels of components that differ in their functionality. The basic components are a component version of the CORBA object model. The extended components provide more functionalities to the component[4]. A component contains connection points, also known as port, between the component and its environment, clients and others that want to use these ports to connect and communicate with the component. OMG CORBA component specification names 4 ports that exist in the extended component and one shared port, the attributes, as one can see in figure 3.1.

![Figure 3.1: CCM Component structure](image)

**Facet** A facet is a multiple object reference that the component supports. Each component has one interface called "equivalent interface" that can be referenced by others (clients and components). The equivalent interface then serves as a port to multiple component interfaces.

**Receptacle** Is the point of connection. Here the client connects to the component. A receptacle will be the place where the different types of connections the component supports will be defined, as well as the ways to establish the connections and manage them.

**Event source** This is the interface that describes the events the component can publish or subscribe. This interface follows the CORBA notification guidelines concerning CORBA events.

**Event sinks** This is a special type of facet that is an "event consumer". The event consumer interface is then accessible to clients or others (such as configuration com-
ponents). A component that publishes its sink can then be reached by events from other components.

**Attribute** Attributes are used for configuration of the component and contain the relevant information dealing with component configuration.

A component’s life cycle is controlled by a special component factory, called home. Homes are responsible for component creation, initialization and destruction. Like the components themselves, the homes are CORBA type. The home-component relationship is one to one but the component-home relationship is one to many. The definition of a home is done separately from its components. This allows for the transferring of components between different homes.

Components can access CORBA services with help from containers. An external layer surrounds the component and manages the component’s access to the CORBA’s services. The container is a part of the framework the component works in, intercepts the calls done by the components and, if needed, invokes a service provided by the framework.

The implementation of the components is done by segments of an executable that implement the CIDL description, and at least one port. Because CORBA and CCM packaging are specified as recommendations rather than set in the implementation, it leaves CCM packaging up to the implementer. zip files are commonly used to archive CCM components. The software or component descriptions are represented in XML, which follows CCM DTD standards and describes the package and its use. The package contains one or more descriptor files, component descriptor file and property descriptor file. CCM, XML, and DTD are an extension of the OSD (Open Software Description) that is done by the W3C [4]. A set of components can be comprised into an assembly. The assembly descriptor will then also contain information about the services provided by the assembly alongside information about the services that are used and needed by it.

Deployment of components, packages, and assemblies is done with the deployment tool. This can be part of the CORBA ORB, or given by the supplier. The deployment task is to deploy the components according to the topology written in the descriptor on one or more servers or as described by OMG CCM specification:

"The aim of deployment is to install and "hook-up” a logical component topology to a physical computing environment"[4].

CCM is a component model specification developed by OMG (Object Management Group). It is inspired in it is design by the early JAVA-Bean component. CCM supports two kinds of components - basic and extended - that support two different levels of services offered by the model. The component life cycle is controlled by a container and the component home that is responsible for initiation and destruction of the component. Packaging is available through various archive formats such as zip. The description of the package

### 3.4 Enterprise JAVA Bean (EJB)

EJB is a server side component architecture developed by Sun Microsystems. It is an extension of the JAVABeans technology that supports server side components and is therefore different from JB.

"Server components are reusable, prepackaged pieces of application functionality that are designed to run in an application server"[5].

EJB beans are divided into three types:
**Entity:** Entity beans are used to visualize real-world objects such as persons, items, places etc. It can be stored in a database.

**Session:** Session beans are used to run and manage tasks. It is a complement to the entity bean that describes a data type given by the session through its methods and functionality.

**Message Driven:** Is a communication bean that acts as a mediator for the communication between different beans.

The enterprise bean exists on the application server, within a container server object. The container supplies a process or a thread that the beans run in. A container can handle multiple beans and manages the bean’s life cycle and special requirements such as security, transaction, data persistence and state management. These services are transported to the bean and controlled solely by the container [5]. EJB components are built to enable portability to different platforms, as long as they run an application server that applies the EJB contracts. The use of a container allows the developers to concentrate on the business aspects/logic, rather than the middleware. The communication and other facets of the middleware are then handled by the container and server.

![EJB Component Diagram](image)

**Figure 3.2: EJB Component [5]**

As figure 3.2 show us each bean has two external identifiers - the EJB object that is the interface of the bean and which intercepts all client’s calls; and the EJB home, that is the bean identifier. The EJB object provides access to the methods in the bean and delegates the call to the EJB container that contains the bean. The EJB home is the naming identifier, as given in JNDI (JAVA Naming and Directory Interface), and it delivers an interface to the client, through which the bean’s life cycle in controlled (creation and destruction). EJB supports other naming schemes as well, such as the URI scheme. A client searching
for the bean will find the EJB home and utilize its creation methods so it can instantiate an EJB object.

An EJB bean contains two interfaces - local and remote. The remote interface is available for the client and the local interface is for the server running it. Each enterprise bean has certain rules concerning its life cycle and services needed and provided. These rules are specified to the Deployment Descriptor Object during deployment time. This means that during run-time the DDO is read and the enterprise bean is configured according to it, with minimal interference in the program execution. EJB packaging scheme is in jar file format (JAVA archive). The first thing read from the EJB 1.1 jar file is the deployment descriptor. The deployment descriptor of EJB 1.1 is an XML file that support the EJB DTD standard.

EJB is the JAVA component model for application server components. It is not an extension of the JAVA-Bean, though the design is slightly similar; EJB offers a larger range of services and possibilities. There are three types of beans, each for a unique type of assignment: entity beans reassemble objects as the mapped real world entities and entities that can be stored in databases; session bean for state and task management; and message beans for network communication. EJB as CCM use containers for controlling beans’ life cycle. The EJB home is the public interface users can access. As all JAVA packages, the packaging of EJB is done with jar(JAVA archive format) archive, with a component description stored in a XML file.

3.5 .NET

.NET is Microsoft’s latest component technology. A component in the .NET terminology is called an "assembly" and it is a collection of a MSIL (Microsoft Intermediate Language) code, manifest and meta-data[6]. Figure 3.3 show how an assembly is constructed one can
easily see the an assembly us build of three subcomponents:

**MSIL:** MSIL (also referred to as CLR) code is an intermediate representation between text and machine code like JAVA and Perl byte code. .NET translates to MSIL. All of .NET’s languages are translatable. This means that C# code and VisualBasic.NET code will both look the same in MSIL. The MSIL can then, during runtime, translate the source code to machine code in a similar manner to that of the operation of JAVA virtual machine.

**Manifest:** This is a description of the assembly. It is generated by the compiler and contains information about the assembly such as version number, local information, types visibility (public and private) and security of the assembly.

**Meta data:** This a description of all types and interfaces existing in the assembly. The meta-data is generated by the compiler.

An assembly is not bound to one physical file and can be part of several files. There can, however, be only one DLL file that contains the assembly manifest. In .NET components one do not have a separation between data and logic, and the component and its configuration are stored in the component itself [6]. According to Wang and Qian a .NET component is

”a single pre-compiled and self-described MSIL module built from one or more classes or multiple modules deployed in a DLL assembly fill [7]”

Wang and Qian [7] also add resources, such as jpg files, to the MSIL, manifest and meta-data as a basis for the assembly.

.NET components can be visible (for example widget for gui programming), or invisible components for business logic (both Sun’s Bean components types combined into one). It can run locally or as a remote call. .NET components are also transparent to COM and COM+ components, and can call and be called by components from these legacy models. .NET components exist inside the .NET framework. The framework supplies the component with the service it requires, such as security, distribution etc. A loader is then responsible for loading the component by checking the component version. Each component registers its version in the manifest. The loader can load a remote component or a local one, depending on the component configuration information[3].

### 3.6 Summary

This chapter defined what a component is. This definition will serve as the basis upon which this report establishes itself. Following that a definition of a component model was presented and the three most common component models were described: The CORBA Component Model (CCM), Enterprise JAVA Bean(EJB) and .NET assembly.

The next chapters will discuss the common properties that these three models have, compared with the general definition from section 3.2. I will than inquire which properties should a bridge application have in order for components from CCM, EJB and .NET to be able communicate. I shall do this through an analysis of each model and model component vis a vis the definition presented in sections 3.1 and 3.2.
4 Commonality and Variability in Components Model

This chapter identifies the commonalities and variability’s between the different component models. By finding the common attributes to the models one may be able to identify the key attributes that are important for bridging the models. First, we will look at component models in general while focusing on an example component model CM which contains the main features of a component model as described in the previous chapter. This model will be compared to the three models describe in chapter 3. Following that, a description of the variability between the three models will take place, through identifying those properties in the models that may cause a bridge application to fail. The last section will point to the key functionalities that must exist in a bridge application so that it can bridge between components.

4.1 Commonality in Components Model

The first step is to describe our example component model - CM. It contains the general attributes a component model needs to have:

- A component type that is standard for this model.
- A way to define the component and its interface
- A set of services that support the component
- A way to register and locate components (locally and distributed)
- A remote communication support
- A remote object invocation
- A generic way to deploy objects

At this stage it is easy to see that each of the three models discussed in chapter 3, the CCM, EJB and .NET assemblies, contains all the aforementioned points. By examining each of these points in an elaborated manner one can achieve a more concrete view of the model and specify more detailed description of the model.

- Model CM contains a language to define the component. (interface definition language).
- Model CM contains a standardized definition of the services that are available.
- A Model CM component communicates through a standard communication protocol.
- A Model CM component can have a unique identification that may be used to identify the component, both locally and remotely.
- Model CM supports implementation of its components in X programming languages and Y platforms.
- Model CM uses the packaging service to package and deploy its components.
- A Model CM component implements at least one interface.
- Model CM services support various tasks such as transactions, security, events etc.
• Model CM supports fault tolerance and several advanced communication methods such as messages, events and transactions.

• Model CM component description is separated from its implementation. This interface can later be compiled using the language model CM uses.

Additionally, these points show that all three models - CCM, EJB and .NET - follow CM guidelines. There are a few common issues, both in the conceptual level and in the implementation level, to all three models. By looking at the common parts of the models, as described above, one can see in greater detail the commonalities between the models and their implementation. Here is a list of the common parts of the three component models. This list will be followed by a table, summarizing the commonalities.

**Communication protocol:** All three models support communication using known communication protocols such as TCP/IP protocol stack or IIOP. CCM supports the IIOP protocol stack, EJB support TCP/IP and RMI-IIOP communication, and .NET supports TCP/IP communication.

**Exception and Exception management:** Though it is a programming-language dependent property, this quality exists in all three models. CCM CIDL allows the programmer to define exceptions in the interface. EJB and .NET are based on languages that support exceptions (JAVA and C#).

**Naming scheme:** All three models use a naming scheme that supports both the registration of components and the search for components. Each model supports its own scheme, but there are several shared points between the models, such as the use of IP-address and DNS-name (URI) for locating components. Local registration varies more within the different component models.

**Interface specification:** Exists in all three models but is implemented differently with diverse design goals. CCM uses CIDL as an independent interface programming language. CIDL can later be translated to all programming languages that are supported by CORBA. EJB uses JAVA interfaces to describe component interface. JAVA support also gives the component access to JAVA libraries as defined in the component interface. .NET components are defined with .NET languages (C# and VB.NET) and use the interface definition which these languages provide. The interface is defined and exists separately from the implementation.

**Transaction:** All three models support transactions, including support for an external or internal transaction server.

**Events and Messages:** All three models support the use of messages and events; this is done transparently for the components and is implemented by the model, except for events in CORBA, which need to be entered explicitly via hard code by the programmer.

**Security:** Security has a central role in distributed systems and in the component models. All three models support authorization, encryption and the use of secure communication protocols. They also include security as a service provided by the underlying framework. This way, the component can communicate with the system security services without the need to introduce new component security services.

**Packaging:** Is supported by all three models, but implemented differently. All three models follow the same tactic - an archive standard that archives the components and a file that describes the content of the package. This information can be transferred to other components and an executable file, or to support various files such as media, images, database etc. CCM and EJB use an XML file to describe the package. In .NET it is the assembly meta-data. The actual packaging differs from model to model - EJB supports jar; CCM supports various kinds (but zip is the common one); .NET uses zip format as well.
Remote Communication: As distributed object systems, all three models support remote procedure communication, but the implementation differs in each model. CCM is based on Remote Procedure Call. EJB is based on Remote Method Invocation. .NET is based upon MS-RPC.

Component definition: EJB and .NET define a component with their languages, which is then ready to be deployed. CCM CIDL defines a component interface that can be implemented in various languages before deployment.

<table>
<thead>
<tr>
<th></th>
<th>CCM</th>
<th>EJB</th>
<th>.NET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication Protocol</td>
<td>IIOP</td>
<td>TCP/IP, IIOP</td>
<td>TCP/IP</td>
</tr>
<tr>
<td>Exceptions</td>
<td>Exist</td>
<td>Exist</td>
<td>Exist</td>
</tr>
<tr>
<td>Naming scheme</td>
<td>IOR, URI</td>
<td>URI, JNDI</td>
<td>URI, 128 byte unique name.</td>
</tr>
<tr>
<td>Interface specification</td>
<td>Exist</td>
<td>Exist</td>
<td>Exist</td>
</tr>
<tr>
<td>Transaction Support</td>
<td>Exist</td>
<td>Exist</td>
<td>Exist</td>
</tr>
<tr>
<td>Events and Messages</td>
<td>Exist</td>
<td>Exist</td>
<td>Exist</td>
</tr>
<tr>
<td>Security Support</td>
<td>Exist</td>
<td>Exist</td>
<td>Exist</td>
</tr>
<tr>
<td>Packaging</td>
<td>Exist</td>
<td>Exist</td>
<td>Exist</td>
</tr>
<tr>
<td>Remote Communication</td>
<td>RPC</td>
<td>RMI</td>
<td>RPC</td>
</tr>
<tr>
<td>Component Definition</td>
<td>Platform independent</td>
<td>Platform dependent</td>
<td>Platform dependent</td>
</tr>
</tbody>
</table>

Table 4.1: Summary of the commonalities between EJB, CCM and .NET assembly

4.2 Variability in Component Models

Once one have identified the commonalities, the next step is defining the properties that differ between the models and summarize them in the form of a table, table 4.2.

Container: A container that delegates between the component and its framework exists only in EJB and CCM. .NET does not have a container concept, and the component is connected directly to the framework.

Garbage Collection: EJB and .NET are both run on a virtual machine with garbage collection, while for CCM garbage collection is optional.

Data - Type: All models contain a detailed data type definition of both simple and complex types, but those are model specific. The data types in .NET and EJB are based upon the virtual machine those models run on. CCM uses CORBA data types.

After identifying the commonalities and variability’s between the models, one can see which properties and functionalities a bridge application should have if it is to be able to bridge the different models.
### Table 4.2: Summary of the variability of CCM, EJB, and .NET

<table>
<thead>
<tr>
<th>Container</th>
<th>CCM</th>
<th>EJB</th>
<th>.NET</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Container as a moderate</td>
<td>Container as a moderate</td>
<td>Component exist in the context of the</td>
</tr>
<tr>
<td></td>
<td>between the component and the frame work</td>
<td>between the component and the frame work</td>
<td>framework</td>
</tr>
<tr>
<td>Garbage Collection</td>
<td>Not mentioned</td>
<td>Part of the Virtual Machine</td>
<td>Part of the Virtual Machine</td>
</tr>
<tr>
<td>Data Type</td>
<td>CORBA</td>
<td>JAVA</td>
<td>.NET</td>
</tr>
</tbody>
</table>

4.3 **Building Bridges**

Integration of component models is a question of exchanging and manipulating data, usually in the form of objects, between applications built of more than one component model. This integration is based upon communication, identification and invocation. Hence, each part of the application should be able to identify, invoke and communicate with other parts of the application, including those belonging to other types of component models and platforms.

A fictitious application `myApp` is built from two components `A` and `B`. Component `A` belongs to component `A` model and `B` to component `B` model. For `A` to succeed invoking methods in `B` it needs to:

1. Find component `B`
2. Connect to it and invoke the object `B`
3. Allow and/or use `B`’s method to manipulate the data access to object through some kind of stub
4. Get the response

For `A` to find `B`, they both need to use some kind of naming scheme that is not model specific. For `A` to invoke object `B` it has to connect to `B`’s public interface and use these methods to manipulate the data. The result of the manipulation should obviously return to `A` if needed. In light of the above, a bridge that connects `A` and `B`, should fulfill 4 tasks:

1. Use a universal naming scheme and be able to search in different model naming servers.
2. Connect using a communication protocol that is used by both models.
3. Enable remote invocation of objects using the public interface.
4. Be able to translate from one model data type to the other without the lose of data.

Tasks 1 - 3 exist in the commonalities table and are shared by all models today. The main problem is the fourth point, which demands a mapping of all data-types, from the simplest (integer, float, char), through the more complex (array, strings, structures) and to objects and pointers. The use of pointers also raises the question of memory owning and releasing memory address (client pointing to a memory address at the server). Can an object running on a platform with garbage collection free memory from a remote object that runs on a platform that demands an explicit allocating and freeing of memory, and vice versa? Another question deals with exceptions, who should catch server exception, the server that throw an exception or the client that trigger the action?
5 Component and Platform Bridges

At the end of the last chapter 4 functionalities that a bridge application must include in order to bridge different components had been identified. This chapter shall look at three bridging applications that bridge the three component models described in the third chapter, and will then ask whether each application contains all four functionalities from chapter four and how they are implemented in each bridge. A table summarizing the three applications will be presented at the end of this chapter.

5.1 Bridge 1 IIOP.NET

IIOP.NET is an open source bridge application that allows communication between .NET, CORBA and JAVA through IIOP channel. IIOP.NET consists of the following main parts:

1. IIOP channel that introduces IIOP into .NET remote.

2. IDL compiler that creates client and server stubs form (DLL) from CORBA IDL specifications

3. IDL generator that generates IDL definitions from .NET meta-data.

IIOP.NET also contains additional classes that assist communication with CORBA. These classes are responsible for: Exception handling : omg.CORBA.Exceptions, Server (CORBA, EJB) exception can be catched by the client(.NET), CORBA naming: omg.COS.CosNaming that support CORBA ORI naming.

IIOP.NET supports communication between the .NET client and EJB and CORBA servers and EJB and CORBA client and .NET server as figure 5.1 show. The code concerning is more or less as the .NET client should connect a .NET server. IIOP.NET is
a .NET assembly (DLL file) that can be linked and used during runtime by .NET applications (using reference or other linking methods). The development of an application requires the generation of a client and/or server stub (with the IDL generator or compiler), linking the application to the DLLs and to the IIOP.NET DLLs (that is the channel). Analyzing the bridge according to the criteria set before (4.3) one can see that:

1. IIOP.NET supports both CORBA naming scheme and the use of URI.
2. IIOP.NET uses IIOP protocol for communication
3. Object and method invocation is done by the stub/proxy generated.
4. Data type conversion is handled by the IIOP.NET.
IIOP.NET is a bridge application for bridging .NET with CCM and EJB. It allow a transparent communication between the models by using their own communication libraries (IIOP in CCM and EJB and .NET Remote in .NET) and translating from one type to the other. As mentioned above it answers the four criteria set for a bridge application in 4.3.

5.2 Bridge 2 Borland Janeva

Borland Janeva is a bridge product that allows for integration of .NET applications, mostly client applications, with existing CORBA and EJB server applications[9]. Janeva is a set of libraries and a compiler that can compile CORBA IDL or Java remote interface to C# (.NET) assemblies. The compiler generates the client stub according to the interface definition. Janeva libraries are integrated with the .NET framework (they can even be integrated to Microsoft Visual Studio IDE or Borland C#Builder IDE). Janeva works as an ORB on the client side. It generates a client stub and moderates client communication with the CORBA or EJB server as figure 5.2 show. This is done transparently to the client. The client uses .NET remote calls that the Janeva converts to IIOP calls. Janeva libraries support all .NET languages and the .NET virtual machine. This means that Janeva libraries convert between the types (JAVA/CORBA to .NET and .NET to JAVA/CORBA).

Figure 5.2: Borland Janeva architecture[10]
Janeva runtime libraries can access name services to locate the server object or use low-level server services such as transactions and events, and more. This allows the .NET client to use the variety of services that are provided by EJB and CORBA application servers. Comparing Janeva with the four criteria a bridge application (chapter 4.3) should fulfill reveals that:

1. Janeva can search for and locate components in various schemes.
2. Janeva uses IIOP to connect to CORBA and EJB server and .NET remote to connect to the client.
3. A client stub generated by the compiler gives the client the access to the remote interface.
4. Janeva can convert data types between .NET platform and CORBA/J2EE.

5.3 Bridge 3 J-Integro Expresso

This is an ORB written in C# and mapped into the .NET platform that works as a middleman between .NET clients and CORBA and EJB servers. Using espresso is the same as using any other CORBA ORB product. The main difference between espresso and other CORBA ORB products is that it is mapped to C# (the OMG mapping of C# is still under construction). This is done as close as can be to JAVA specifications to allow interoperability with Java products. This differs from IIOP.NET and Borland Janeva, which are bridges that only enable communication between different models/platforms. By having a C# ORB the IDL specification can generate a client stub and server stub. This gives access to CORBA reference data type, such as representation of pointers and in and out parameters in method calls. IIOP.NET and Janeva can only access the IDL method calls and cannot generate a stub. It is up to the developer to create the stub that uses these methods. Figure 5.3 show the range of services and application that espresso can be used to connect to. As with the two bridges mentioned before, comparing Espresso with the four criteria assigned to a bridge application in chapter refBB is showing that.

Figure 5.3: Espresso architecture[11]
1. Espresso uses CORBA’s own naming scheme (IOR) and can use the .NET scheme and URI that is used by all three models

2. The communication is done through the IIOP channel

3. Method invocation is done by client stub server skeleton, as in any CORBA server-client application.

4. Data type conversion is done by the ORB. Espressos according to its documentation[11] converts from JAVA to CLR and CLR to JAVA both simple types and com

5.4 Summary

This section presents a table that summarizes the key points discussed previously regarding the three bridges, their commonalities and variability’s. As one can learn from the table

<table>
<thead>
<tr>
<th></th>
<th>IIOP.NET</th>
<th>Janeva</th>
<th>Espresso</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type Bridge</td>
<td></td>
<td></td>
<td>CORBA ORB</td>
</tr>
<tr>
<td>Platform</td>
<td>.NET</td>
<td>.NET</td>
<td>.NET</td>
</tr>
<tr>
<td>Framework</td>
<td>framework</td>
<td>framework</td>
<td>framework</td>
</tr>
<tr>
<td>Communication protocol</td>
<td>IIOP, .NET.Remoting</td>
<td>IIOP, .NET.Remoting</td>
<td>.NET.Remoting</td>
</tr>
<tr>
<td>CLS to IDL compiler</td>
<td>Exist</td>
<td>Exist</td>
<td>Exist</td>
</tr>
<tr>
<td>JAVA to .NET conversion</td>
<td>Exist</td>
<td>Exist</td>
<td>Exist</td>
</tr>
<tr>
<td>CORBA to .NET conversion</td>
<td>Exist</td>
<td>Exist</td>
<td>Exist</td>
</tr>
<tr>
<td>Services</td>
<td>Extern</td>
<td>Extern</td>
<td>Intern</td>
</tr>
</tbody>
</table>

Table 5.1: Summary of the three bridges

5.1 all three bridge application contain the functionalities mentioned in 4.3 and offer almost the same kind of service of to the users.
6 Integration more then building bridges

Integrating components for the purpose of building new systems, or enchanting an existing system’s abilities, is not just a question of bridging components. Integrating components, or developing software through integrating components, are the main methods used today in software development. The use of components enhances reusability and promotes stability, faster development periods and security, but it also has a dark side: interoperability problems, unpredictable systems, development delays and more. One can ask why a method that promotes one thing introduces the opposite and whether these obstacles exist in every development project that is based upon component integration. In a perfect situation a development process of components-based software engineering would have been based upon:

1. Identifying system requirements
2. Locating components that answer those requirements
3. Integrating the components
4. Test and deliver

This process assumes that all components can be integrated as is, or with minimal configuration that has no side effects that can damage or risk the system or the components it is being integrated with. Unfortunately this is not the situation the software developer faces. A development process based on components and components integration is not a trivial process and the development team faces problems in different stages of the process. This chapter will discuss some of the problems that occur during the integration process.

6.1 Finding components

Using component based software engineering adds some tasks to the development process; identifying the system requirements and trying to locate matching components is one of them.

Example: A system supporting money transfers needs a way to compare the currency value both locally and worldwide. Using a component that is already developed for this purpose can simplify part of the development. But where can one find such a component?

Solution: Several platforms offer different solutions for locating appropriate components. Web-service components are described and made searchable through WSDL description. In the .NET platform components are registered at the registry and there is an assembly file that contains the component’s information.

6.2 Interoperability between components

This problem concerns communication problems between components. The components can be distributed or placed on one machine; a possible problem will be that the components do not use the same communication protocol and/or use different ways to package data.

Example: A system is being built by integrating two remote components. Both components are written in the same programming language - C++ - and use the same method for invoking remote objects: RPC. But component A assumes that the data sent to it is packaged with a header of 8 bits, while component B uses a 16 bits header.

Solution: A solution to the problem is introducing a bridgethat converts data packages
from one format to the other. This solution applies for both black and white box components: while in a black box there is a need to warp the component with the converter layer, in a white box there is a possibility to change the source code of the component so it meets the package format. This introduces a bridge that delegates the communication between the components.

6.3 Component Hidden interfaces

Component based software development assumes that the user of the component - the development team - has some knowledge about the component, namely, that documentation is supplied by the component vendor. This information usually contains the public interface of the component, the list of methods/services the component has to offer and notation about the methods’ required parameters and the methods’ provided answers. The component itself can be built of several internal components with one external interface. The relationship and dependencies between the internal parts of the component are not known to the end users.

This can lead to unexpected behavior of the component because of an unforeseen reaction of one or more of the internal parts. This behavior can be a result of an internal problem or integration between the component and other components. One must remember that the more complex the component is, the more difficult it is to integrate it. A component that is built of one class is easily integrated and predicted, but the larger the component is, the more internal and external dependencies are involved, and the harder it is to predict the component’s behavior.

Example: A software built from two components A & B is a multithread application, and both components have an internal function that allows the main thread to allocate more memory to itself by cleaning the stack if no free memory is available.

Solution: No general solution exists yet. The integrators, unaware of the internal parts functionalities, can not predict all the problems caused by the situation. Some techniques such as Vows offer a general view of the system stability by "bombarding it with wrong data" [12]

and watching the result. One should also note that the hidden interfaces can be deliberately malicious, such as Trojan codes inserted into an application. A technique used to minimize damage from hidden interfaces and side effects is to have the integration team develop a wrapper that wraps the component with the interface the integration team desires, and have communication with the component go through the wrapper. This way the component’s side effects are limited to the wrapper it resides in and have a limited effect on the system as a whole. This solution introduces an overhead in development and can cause problems when upgrading or changing components.

6.4 Architectural mismatch or Assumptions Conflicts

According to David Garlan, Robert Allen and John Ockerbloom [13] the assumptions that the integrators have about a component and its use are not always identical to the ones the component developers had. Such assumptions can be about the component’s nature, the components connectors’ nature, the global architecture nature or the constructions process.

Example: Component nature assumptions and component connector’s assumptions: This type of assumption can lead to interoperability problems as mentioned above.

Global architecture assumption: David Garlan, Robert Allen and John Ockerbloom
describe in their paper the problem they faced when they realized that their database component was designed under the assumption that it should work as a center of a star style ring architecture. Furthermore, the database component assumed that no concurrency should be possible. The Aesop system that Garlan, Allen and Ockerbloom developed was supposed to work in concurrency with a multiple number of threads and communicate with the database. Having a database that forbids concurrency slowed the project, and knowing about this assumption could have led them to choose another database component or a different design for the system.

Construction assumption: These are assumptions concerning the order of instantiation and execution of components.

Solution: What Garlan, Allen and Ockerbloom [13] propagate is a way to document assumptions so that the component documentation will include more than just the public interface and the list of methods. They suggest including the assumptions the component developers had about the pre and post conditions of the component activity, as well as the component non-functional assumptions.

6.5 Security or can I trust the component, QoS

As mentioned in the discussion about hidden interface parts, a black-box component can - and usually does - contain hidden parts. These parts’ true functionality may never be revealed to the component user. The question is, can the user trust the component? Should every component be trusted? If not, how can a user find trustworthy components? Who guarantees component trustworthiness? Who should stand for the component QoS?

Example: Having bought a component to encrypt and decrypt e-mails so that a company can secure their exchange, the company discovers that the information from the encrypted e-mails leaked to their competitors. Can the problem be the component?

Solution: One solution is to promote the use of component certificates: buying components from known vendors reduces the chance of getting an unsecured component. But even that is not bullet proof.

6.6 Design for Evolution or thinking two steps ahead

The reason components-based software engineering flourishes is that it promotes reusability. Reusability can lead to reduced costs in developing new products, as well as shortening the time it takes to make and release the component. Components are reusable and the design of the component and the system using it should be made for clean and trouble free component evolution. The goal is to succeed in replacing a component on the fly during run time with minimal overhead recompilation, and a component that can easily integrate with target applications.

Example: A fiction database newData is a new database. It’s a SQL base network database and can communicate using several network protocol such as TCP/IP (v4), IBM NetBios, Novel Netware etc. But the designers and architecture didn’t took in consideration IPv6 which will made the database not compatible with IPv6 network.

Solution: The above example is quite simple and cover the problem just on the surface. Reusability and design for evolution is a complicated task especially when the component is complex component such as operating system. Using technique that promote reusability such as Object Oriented programing can ease the design (reusing code and design patterns) and produce better reusable code. Another question is how to make the component reusable by allowing changes (as reconfiguration or upgrade) with minimum overhead as possible.
6.7 Source Control

When using a black box component the user puts his software in the hand of another development team. Quite often the integrators don’t have access to the source permission to produce some re-engineering techniques on the component (this depends on the component’s license terms). Users then finding an unexpected feature or missing an expected one have to contact the component’s vendors to implement changes.

Example: Coppit and Sullivan [14] describe a model of component based software engineering POP. They tested the model while developing the software Galileo. Galileo is constructed of several large components: Microsoft Word, Microsoft Access and Visio drawing tool. When some of the functionalities in the Visio tool did not work as described (multi-pages support) Coppit and Sullivan decided to contact Visio’s vendor (at that time: Visio Corporation) to fix the problem. Failing to convince the vendor themselves, they then involved their customer in pressing the vendor to fix the problem in the next release.

Solution: There are several ways to solve this kind of problem. One of them is the open source community, though this is still not a stable or accepted business model. Close code vendors can, as Microsoft does, frequently release updates both to fix software bugs and add new features demanded by their clients.

6.8 Summary

The main problem regarding component integration is the lack of information the component user / integrator has. The component documentation is insufficient and lacks information about the semantic of the component, the component non functional qualities and the component’s internal parts. At the same time, over documentation can be ambiguous and not give a better understanding of the component and its usage. Shaw’s[15] proposal of documenting the component functionalities and allowing the use to expend the documentation when he meets new and undocumented features can be seen as a step in the right direction . The components mentioned in this chapter were developed for reuse, and as such obviously have more problems ones no developed for reuse.

Component certificates can give further assurances about the component’s nature and its functionalities, and possibly eliminate the fear from unwanted malicious functions. But the question still remains of who gives the certificate, which sets the certificate’s standards and what aspects of the component the certificate covers.

The use of black box components enhances reusability and reduces release to market costs and time, but it also introduces a range of new problems concerning component integration /composition. Some of these problems were mentioned in chapter 4. The main cause for these problems is the lack of knowledge about the component. A black box component is delivered as a binary executable application that is linked to the software through an interface. The interface describes the public methods and their pre and post conditions. As Garlan ,Allen and Ockerbloom [13] said, there is no information regarding the assumptions taken when the component was developed. There is no documentation about hidden and internal interfaces in the component. There is no documentation about the component’s semantics and qualities.

Another problem is that a reaction and component’s side effects also depend on the components it is being integrated with. This means that even if one could document the component’s side effects, it would only match this component in a specific environment with a specific set of components. As mentioned by Voas[12], introducing a new component or a new environment or using the component in another system can give different result. Voas [12] offers to test how the system reacts while being bombarded with wrong
data. The results can show how the components building the system and their hidden parts react when exposed to abnormal input. This kind of test can provide speculations about the system’s stability and help predict its reactions. Stafford and Wallnua[16] promote Prediction-Enabled Component Technology. Employing ideas from software architecture, architecture analysis and component certification, this method should assist in the prediction of system qualities at the design phase.

This chapter reviewed some of the problems concerning component integration and adaptation. It is not a complete list; rather it is designed to focus on the problems that the author of this report has encountered during his implementation project and in the research for this report.
7 Discussions

7.1 Reflections

Software development is often compared to the art of cooking, as it is a process that includes several steps and delivers a result that is a combination of one or more components. One can make pancakes by getting and connecting the right ingredients in the right measurements. Mixing egg, flour, sugar and butter can result in a pancake. Exchanging one of the components with another (brown sugar with white one) will still result in a pancake. Having a component that is corrupt (bad milk) can be identified early and be replaced. All this costs time. When time is a problem one may buy a "Bake and Shake" - two component solutions, a fine grain component (the powder) and milk. This may lead to similar results but it is less receptive of changes. It is also much harder to track possible failures and identify their sources than it was when you were working from scratch.

In the practical part of the report i developed a small application built of three components. A CORBA server and ORB, an ASP.NET web site and an IIOP.NET bridge application. One can add a forth component: the web browser used by the client. As all the components were white box and the developer had full control over the application, the main problems the developer encountered concerned not the combination of functions of various components, but rather the combination of logic. these problems were discussed in chapters 4 and 5.

This is not the situation in other integration projects. Chapter six raised a number of problems that are met by integrators during the integration process. Those problems concerned several aspects of the integration process from locating and evaluating a component at the design process to upgrading a component after delivery. Writing reusable software is a famous issue. The main research and development is done on reusable source code such as libraries (C++ STL), packages and frameworks, but this assumes the code will always recompile; composing and integrating components demands reusability of binary entities without recompilation. Plugin based systems can be one way of implementing this, but this method is probably not appropriate to all application types.

While researching this report I found a point that in my opinion is crucial for solving some of the complications that software integrators and developers face: documentation. There is a call for better documentation of the component. According to some [1],[14] [15],this documentation should include more details than the public interface/list of services the component has to offer as derived from the component’s IDL. Capturing the functional and non-functional attributes a component has, the assumptions the component development team had about the component and about the environment it would integrate in are part of the information the user needs. Shaw [15] calls for the use of credentials as a dynamic way to document component features by the user, thus, in time, enlarging our knowledge base. All this deepens the scope and size of documentation and can hence lead to documentation that is too large to use. This can lead to documentation posing a problem not a solution. UML as IDL captures the requirements and provides functionalities, but succeed in capturing the non functionalities and ileitis (quality, security, reliability etc) of a component.

Software architecture is based upon building the system from components, analyzing the architecture and predicting the system’s qualities. A system is thus an integration of several components, each of whom can also consist of internal components. One might think that the larger a component is, the more complex it is to integrate, and the more problems the integration process will encounter. But size (in lines of code or in number of internal subsystems) is not the main factor making a component hard to integrate.
One of the most common applications today is an e-shop site. Built from at least three components - database, web-server, website - using this kind of application has become so popular that there are vendors who produce an all-in-one package. This is a large application and should be hard to integrate and developed, but being so popular it was developed so often a stable architecture emerged and the components became more stable and more reusable. This indicates that the less reusable a component is, the harder it is to integrate it, and integrating a component for the first time is probably the most complicated integration process. Mohagheghi, Conradi, Killi, Schwarz [17] conducted a study at Ericsson where they used data mining methods for comparing reuse vs. defect density. This study showed that the more a component was reused to fewer defects it has, even if it is a large and complex component. Both the e-commerce example and Mohagheghi, Conradi, Killi, Schwarz point to on the right path. Reuse of the component gonna make them in time easier to integrate and develop. Matching architecture style and design pattern will evolve and the process will become simpler as it is to build an e-commerce application today. Going back to the "Bake and Shake" analogy - the more customers complain about the product, and the more they come up with ideas to improve it, the better it will become. In the end, it may even taste like a pancake.

7.2 Followup

Component based software engineering promotes the use of software components as the foundation of new software. Taken from other manufacturing fields, this vision of choosing needed components and combining them like Lego bricks to create new software seems to contain enormous advantages for developers. Worldwide, researchers and scholars are trying to remove the obstacles standing in the path of this vision.

This report named some of the problems that software integrators and developers face. It focused mostly on low-level communication as described in 4 and 5. It also touched upon several other aspects of integration problems as described in chapter ref6. Chapter 4 and 5 concentrated on ways to allow communication between two (or more) component models. Communication as data package exchange, locating remote objects, using some naming scheme and invoking methods and remote objects of the other model - the caller. As described at the end of chapter 4, there are four functionalities that a bridge application should have so it can fulfill its task. A bridge should:

- Be able to resolve names across all the models it bridges
- Use a common communication protocol to all models
- Assist in invoking objects across models.
- Convert data types.

All the bridges mentioned in chapter 5 have those functionalities and by that fulfill the criteria for being a bridge application. By that all three bridges solve the interoperability problems between the three component model mentioned in this report CCM, EJB and .NET. This show that the difficult part in software integration and CBSE is not the communication level but the high-level application conflicts as mentioned in chapter 6. Those problems and more are the obstacles in the integrator’s way. The more studies done on the subject, the more are likely to come as every solution lead to new problems.
References


