Department of Informatics,
School of Economics and Commercial Law
at Gothenburg University

Integration of Heterogeneous Systems

- A Component-Based Approach -

Abstract:

This master thesis concerns principles and technologies for software component integration. It provides an overview of Component-Based Software Development (CBSD), which is a new paradigm for information systems development. It focuses on the comparison of three different techniques: Enterprise JavaBeans, COM and CORBA. To investigate different principles and techniques, we have performed a literature study and made some qualitative interviews with people that have experience from integration tasks/questions. We have also developed a small prototype using Enterprise JavaBeans. We have found design patterns to be useful principles for the integration of software components. Our comparison of the three techniques shows that they have much in common, but that they also differ in some important aspects.

Master of Science Thesis June, 2001

Anna Eriksson
Ali Reza Feizabadi
Baranoush Zamani

Supervisors:
Maria Bergenstjerna, Department of Informatics
Andreas Jirvén, Ericsson Microwave Systems AB

*We are grateful to **Maria Bergenstjerna**, our supervisor at the Informatics institute in Gothenburg University, for her support, guidance, and encouragement.*

*We would also like to thank **Andreas Jirvén**, our supervisor at Ericsson Telecom Management. He initiated this master thesis project and has helped us through our investigation. We are grateful to his support, advice and feedback.*

*A special thanks goes to **Johanna Kjellberg** System & Integration Manager at Ericsson Telecom Management for her enthusiasm to this work and all resources we got to accomplish it.*

Thanks are extended to the interview persons of this investigation.

Table of Contents

1 INTRODUCTION 3

1.1 PURPOSE 3
1.2 PROBLEM DEFINITION 4
1.3 BACKGROUND 4
1.4 DISPOSITION 5

2 METHOD 7

2.1 LITERATURE STUDY 8
2.2 INTERVIEWS 9
2.3 PROTOTYPE 9

3 THEORY 10

4 COMPONENT-BASED SOFTWARE DEVELOPMENT 16

4.1 TWO EXTREMES AND THE THIRD WAY 17
4.2 THE BENEFITS OF COMPONENT-BASED DEVELOPMENT 18
4.3 INTRODUCTION TO SOME FUNDAMENTAL CONCEPTS 20
4.3.1 Components 20
4.3.2 Interfaces 25
4.3.3 Contracts 26
4.3.4 Design Patterns and Frameworks 27
4.4 ACTIVITIES OF THE COMPONENT-BASED DEVELOPMENT APPROACH 29

5 INTEGRATION PRINCIPLES 32

5.1 INTRODUCTION 32
5.2 DESIGN PATTERNS 34
5.2.1 Adapter 35
5.2.2 Bridge 36
5.2.3 Proxy 37
5.2.4 Mediator 38
5.2.5 Facade 39
5.2.6 Other patterns 40

6 INTEGRATION TECHNIQUES 44

6.1 DOMINATING TECHNIQUES 44
6.1.1 Enterprise JavaBeans 44
6.1.2 Component Object Model 68
6.1.3 CORBA (Common Object Request Broker Architecture) 82
6.2 OTHER TECHNIQUES 103
6.2.1 Jini 103
6.2.2 Message Broker 107
6.2.3 Intelligent Agents 112

7 INTERVIEWS 115

7.1 INTERVIEW NO. 1 115
7.1.1 Integration Difficulties and Principles 115
7.1.2 Component-Based Software Development 115
7.1.3 Integration Techniques 116
7.2 INTERVIEW NO. 2 116
7.2.1 Integration Difficulties and Principles 117
7.2.2 Component-Based Software Development 117
7.2.3 Integration Techniques 117
7.3 INTERVIEW NO. 3 118
7.3.1 Integration Difficulties and Principles 118

7.3.2 Component-Based Software Development *118*
7.3.3 Integration Techniques *119*
7.4 INTERVIEW NO. 4 119
7.4.1 Integration Difficulties and Principles *120*
7.4.2 Component-Based Software Development *120*
7.4.3 Integration Techniques *120*

8 PROTOTYPE 121

8.1 OBJECTIVES 121
8.2 REQUIREMENTS 121
8.3 TOOLS/INSTALLATION 121
8.4 TRAINING........ 122
8.5 MODELLING 122
8.6 PROGRAMMING 123

9 RESULTS 125

9.1 RESULTS FROM LITERATURE STUDY 125
9.1.1 Some points from the literature analysis *125*
9.1.2 Comparison of the dominating techniques *126*
9.2 RESULTS FROM INTERVIEWS 132
9.2.1 Integration Difficulties and Principles *132*
9.2.2 Component-Based Software Development *132*
9.2.3 Integration techniques *132*
9.3 PROTOTYPING EXPERIENCE 134
9.3.1 Development Tool Demanding *134*
9.3.2 Development Tool Dependence *134*
9.3.3 Interoperability with CORBA *135*
9.3.4 Inherent Technique Complexity *135*
9.3.5 Support for basic information management services *135*
9.3.6 Performance *135*

10 DISCUSSION 136

10.1 A CRITICAL VIEW ON OUR WORK 136
10.2 CONCLUSIONS 136
10.2.1 Question 1 .... *137*
10.2.2 Question 2 .... *137*
10.2.3 Question 3 .... *138*

11 APPENDIX 140

11.1 INTERVIEW QUESTIONS 140
11.2 THE ZACHMAN FRAMEWORK 141

12 REFERENCES 142

12.1 BOOKS 142
12.2 WEB REFERENCES 143

1 Introduction

One of the major problems with integrating complex information systems is the heterogeneous and different subsystems. According to the Hypertext Webster online dictionary (2001), heterogeneous means "*consisting of elements that are not of the same kind or nature*".

Most large-scale information systems solutions in business, science and administration today tend to share some common characteristics. An essential one among them is, becoming strongly heterogeneous in their technical and organizational context. We mean by that, diversity and dissimilarity in type, structure, logical, and technical dependencies of their consisting parts. (Kutsche & Sünbül, 1999)

It leads to that they require the capability of handling syntactical and semantic heterogeneity in the process of component integration and federation of many different, but logically related information resources.

Often the communication between the system parts is handled in an "ad hoc" way. The integration is accomplished for the particular case at hand, without consideration of wider application. This leads to problems with inflexibility, inertia of change, and difficulties to have a general view over the system. The consequence is that when one part of a system is to be changed or replaced, other parts of the system also will have to be changed or they will be affected in undesirable ways. The costs in this context could sometimes be so high that it, in practice, becomes impossible to accomplish the change.

1.1 Purpose

Our purpose was to study different principles and techniques for integration of software components that support simple replacement of system parts, with minor modifications in the remaining system. By literature studies we have investigated the existing principles, techniques and research in this area. We have performed some interviews with persons who have experience of integration questions and techniques. To become more familiar and gain a better understanding of integration techniques, we have developed a simple prototype in one of the studied techniques.

Telecom Management at Ericsson Microwave Systems has a need for an integration strategy and architecture to make their software solutions flexible and easy to maintain and upgrade. Our intention has also been to be able to make a suggestion of a suitable approach for this. The goal of this thesis project has been to increase the knowledge on various integration concepts, principles and technologies.

1.2 Problem definition

In this thesis we have tried to answer the following general questions:

- What dominating general principles exist for software component integration and what are their benefits and drawbacks?

- What different integration techniques exist for this purpose and what are the main differences between them?

For the specific operating support system at Ericsson Telecom Management, we had the following question:

- How can their third-party products be integrated successfully, supporting simple replacement and maintenance?

1.3 Background

Today, organizations face a dynamic and heterogeneous world, in which fast responses to a turbulent environment, flexibility and a strategic plan for IT management are essential criteria for survival.

A survey of information systems development during the last decades shows the application of the following strategies (Magoulas & Pessi,1998):

- The dominant principles during the sixties, trying to concentrate all information supply to one integrated system,
- Efforts during the seventies to decentralize information systems to each organizational part,
- Activities during the eighties developing PC- based systems.

These efforts are essential reasons for the present situation, which can be described as a complex and problematic world of information systems. This world is characterized of “information islands” caused by isolation, “information labyrinths” composed of conflicting architectures and “rigid structures” caused by inflexible architectures. This has led to high costs, low information quality, poor information processing and availability and unchangeable structures. (Magoulas & Pessi, 1998)

The previous slow and almost predictable market variations have been replaced with unstable and quick changes, causing new competition situations where time is a critical factor.
A complex and fast-changing environment causes external insecurity for the organization. (Christiansson & Jakobsson, 2000)

Surviving in this chaotic world and managing these problematic situations requires well-designed, flexible and maintainable information systems, which contributes to making the right decisions at the right time. This, in its turn requires well-defined integration principles to compose a collection of interacting system parts, which are easy to configure and replace. But, this is not an easy and problem free process.

Telecom Management is a sector in Ericsson Microwave Systems AB. Here products are developed, services and solutions for both the mobile and stationary nets as well as the data net in the area of telecom management. The activity constitutes a strategically important section in the business developing in Ericsson and it is found to be in an expansive level.

These products are selling over the whole world market and are often an entire solution from Ericsson to telephone operators. In some cases, Ericsson has chosen as a strategy, to buy software from other companies to build bigger and more complex solutions.

Ericsson's customers use these solutions to manage their telecommunication's networks. These operating support systems handle, among other things, alarms from the network. Some parts of the systems need to communicate with each other. Today, the integration between two software components happens in a unique way. It is desired that this integration happens in a reusable way. In other words, it is desired to develop a communication-mechanism, which can be similar in all interfaces between different components.

Integration is the key word for this work. In this work we try to research the principles, which can be used in an integration work. We hope that our work can be a source of information that can give a higher view over integration principles, techniques and other involved questions in this area.

1.4 Disposition

This master thesis has the following structure:

In chapter 2, **"Method"**, we describe the research methods we have used to gather information about the problem area.

In chapter 3, **"Theory"**, we discuss systems theory, which we have used as the theoretical foundation for this thesis.

In chapter 4, **"Component-Based Software Development"**, we give an overview of CBSD, which is a new paradigm for systems development, and the basis for the three dominating techniques, which will be described in depth in this thesis.

In chapter 5, **"Integration Principles"**, we describe what integration principles could be suitable for software component integration.

In chapter 6, **"Integration Techniques"**, we present the three dominating techniques for software component integration and give a brief overview of some alternative techniques.

In chapter 7, **"Interviews"**, we give an overview of each interviewee's opinions and experience of software component integration issues.

In chapter 8, **"Prototype"**, we describe the process of developing our prototype.

In chapter 9, **"Results"**, we present our results from the literature study, from the interviews and from the prototyping process.

In chapter 10, **“Discussion”**, we present our conclusions for the three questions in the problem definition and give a critical view over our chosen research method.

There is also an **Appendix** and a **References** section at the end of this report.

2 Method

Figure 2.1 presents an overview of our research within the area of software component integration. As a foundation for our work we have chosen to use systems theory. The systems theory provides us with basic concepts such as systems, parts, relations, etc. These concepts can be mapped to the component-based software development-paradigm, which we have used as a framework for our thesis.

We have studied the component-based software paradigm and searched for simple principles or mechanisms suitable for integrating components. We have performed theoretical and empirical studies of the three dominating techniques within the area of component-based software integration.

The theoretical studies of integration techniques concern the architectures of EJB, CORBA and COM and the empirical part of our research consists of interviews with people experienced within the field of component integration, and the development of a simple prototype using Enterprise JavaBeans.

We have also briefly studied some alternative techniques for software component integration, namely JINI, message brokers and intelligent agents.

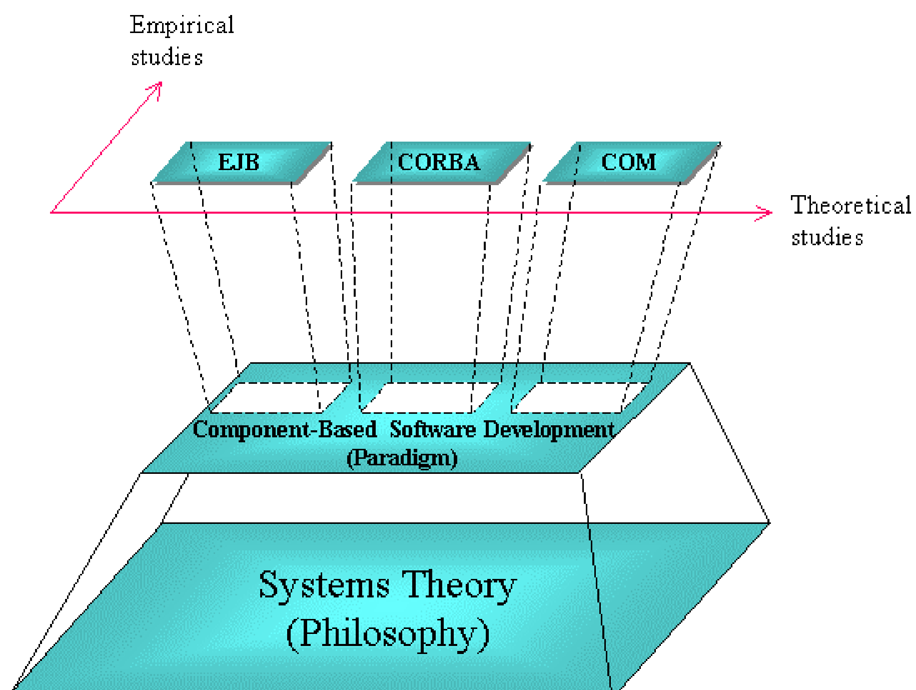


Figure 2.1 A model showing our chosen path of research

2.1 Literature study

We have chosen a broad explorative attempt for the literature study.

The major purpose of explorative studies is to gather as much knowledge as possible within a specific problem area and try to illuminate the problem area comprehensively. Since these studies often aim at reaching knowledge that can lay the foundation of further studies, richness of ideas and creativity are important features. (Patel & Davidson, 1991).

In an explorative study, the researcher searches for knowledge in a rather unsystematic way. The researcher starts out from his or her own understanding and experience, reads others work and makes preliminary inquiries to gain knowledge within the area of interest. Here, the researcher is foremost interested in describing reality and discovering new aspects of reality instead of trying to verify or falsify already existing hypotheses or theories about reality. (Patel & Davidson, 1991)

The literature study would help to increase our knowledge of different concepts, principles and technologies for system integration. We started to scan a broad area, to get an overview of the area, and to be able to make a demarcation in our work. When we had become more familiar with general principles and concepts, we studied in depth the dominating techniques and more briefly some other techniques. Figure 2.2 depicts the areas we have covered during our literature study.

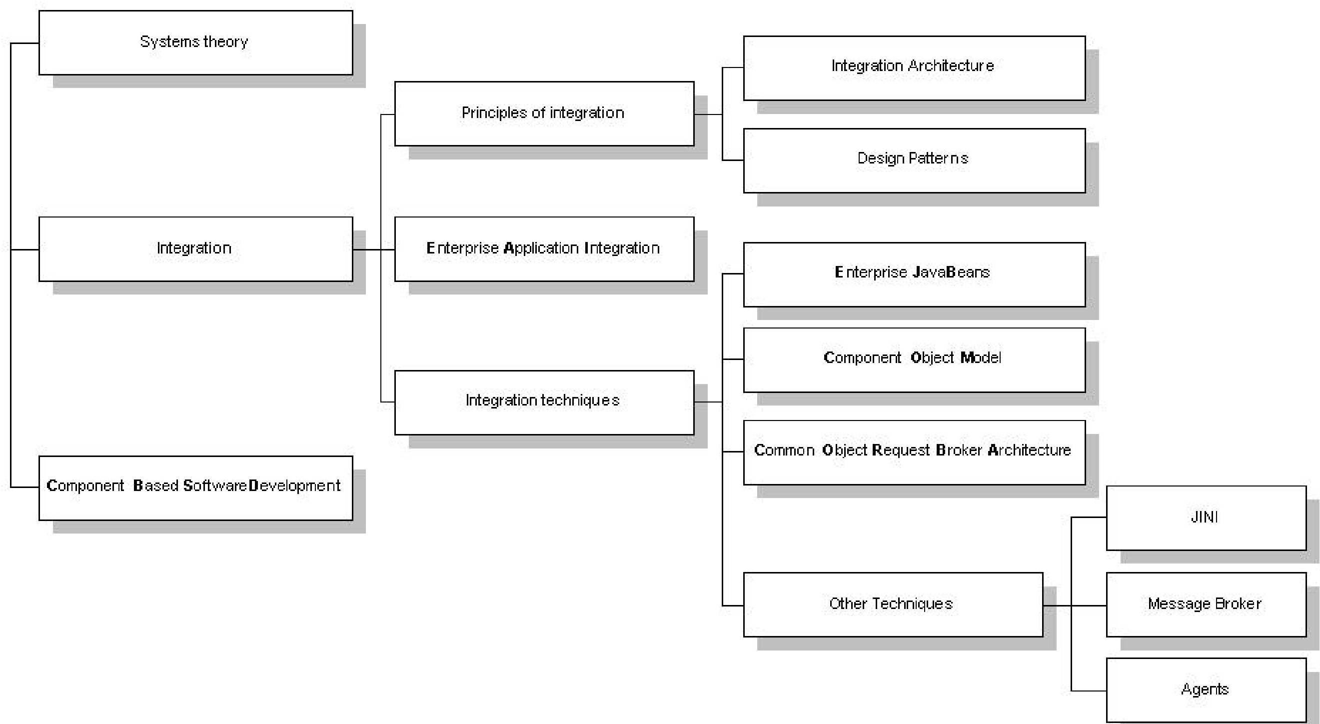


Figure 2.2 Our literature study model

2.2 Interviews

We have performed four interviews. The purpose with the interviews was to get some experts' point of view regarding key concepts in the area of software integration and its related technologies and to derive advantages from their experience.

Our criteria for choosing the interview persons, was that they have been involved in integration questions, either in business or in the academic world. The interviews were semi-structured, and took about one hour per person. We have chosen the semi-structured model for the interviews, because of the nature of the problem domain, which makes it difficult using structured questions with specific boundaries. It also gave us the opportunity to initiate some deeper discussions with the interview persons.

The interview questions can be found in the Appendix.

2.3 Prototype

A prototype is a test version of a future system that does not have to be identical with the final product in all respects. It is important that it is similar to the final product in the areas under examination. Certain functionality could consciously be left out. (Wallén, 1993)

We have developed a simple prototype to become familiar with the degree of difficulty of software integration tasks and to get a better insight into a new integration technique using a server-side component architecture, called Enterprise JavaBeans.

An advantage of building a prototype is that it gives a possibility to concretely experience the problems of performing an integration task.

3 Theory

We have decided to use systems theory as a comprehensive theory for our work. Systems theory is generally a concept, which basically was used in 1920 by *Von Bertalanffy* who was a biologist. Systems theory was originally practiced in natural sciences and more in biology but in the last years it has also been practiced in sociology and behavioral researches. It has been even more usual in some other sciences such as business economics, management, political science etc. (Norrbom, 1973)

Systems theory has an approach to focus on an *overall* and a *comprehensive* view as an important aspect. It pays attention to different valuation and target understanding, which can lead to different solutions for a problem. In this way it is very important to find a system with similar structure and also the general positions such as control and feedback. (Norrbom, 1973)

This is not so easy as it sounds because "system" as a concept has various meanings for each of us and therefore systems theory is a very subjective concept.

Before we talk more detailed about system theory we will describe the meaning of the above sentence in a more philosophical way.

In real life, when we want to get information about something, the first thing we do is to get a clear *definition* of some concepts. In this way, we often try to describe a concept in a *direct* way, for example:

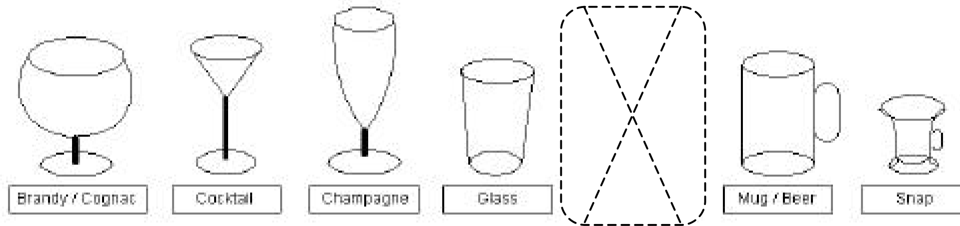
Main question:	**What is a car?**
Answer: *(A direct definition!)*	A car is a vehicle, which has 4 wheels, a steering wheel, a clobber, a coach . . . etc.

In other words, in a *direct definition*, there is more focus on structures. A direct definition is a very usual way to describe objects, problems or concepts, often simple things.

When we meet concepts, which are more complex, a direct definition is often not enough to get a clear understanding over the problem or concept. In these cases there is another useful way to define the problem. That is an *indirect definition*.

In an indirect definition we do not try to describe the concept or problem but we try to describe other things, concepts, objects or problems, which are related to the demanded concept but are still not the concept, which we actually will define. When we have done that, we can make the conclusion that the problem, object or concept actually is something else, something, which is *not* the things we already have defined!

For example:
Main question: **What is a Wineglass?**

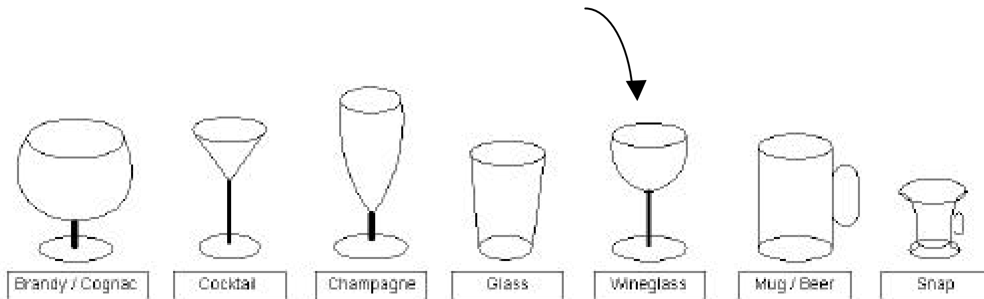


Help questions:

Do you know a usual glass?	Yes
Do you know a mug?	Yes
Do you know a snap?	Yes
Do you know a Cocktail glass?	Yes
Do you know a Brandy glass?	Yes
Do you know a Champagne glass?	Yes

Answer: *(An indirect definition!)* The one that is *not* one of them, that is a wineglass!

Conclusion: Aha! Then I know what a wineglass is!



An indirect definition is sometimes more powerful than a direct definition, and it is often used for more complex concepts but even an indirect definition is not always enough to define those concepts, which are really complex.

Some concepts, such as “Architecture”, “Democracy”, “System”, etc. are very difficult to describe because they can be “translated/understood” in different ways by different persons.

In these cases there is a third way to define concepts. This third way is nothing more than a mix of *both* a direct and an indirect definition.

In these complex cases, if we do not use the third way to define the concepts, there is always a risk to get an “unreal” view over the concept and therewith get an understanding that is not truthful.

For example:
A democratic society is sometimes defined as a society that fulfils a number of different conditions (right to vote, etc.). But the fact is that it can even exist some societies, which have a

lot of these conditions but yet cannot be counted as democratic societies. They are rather authoritarian (dictator) societies than democratic societies.

So for defining a "Democracy" it is not enough to describe it *just* directly or *just* indirectly.

The best way is to first describe it directly, and beside it, also try with an indirect definition, i.e. describe an authoritarian system and the factors, which exist in an authoritarian system.

Then, we can finally make the conclusion that a Democracy is something, which *has* these conditions and at the same time *has not* those other factors. Thus, a mix of both a direct and an indirect definition.

System as a concept is another such complex concept. In other word, different persons can translate it in different ways. (Johnson, Kast & Rosenzweig, 1973)

We can describe a system in a direct and structural way and say that a system is a structure of a number of elements/components, which work together.

But we should not forget the fact that a system can even be a subsystem (a part of another system) or a super/mother system, which contains a number of other systems.

With a system structure, we mean the pattern, which is built by components and the relationships in a system. At least, the different forms of system's structures can be divided into three main groups: *Hierarchical*, *Multilateral* and *Temporary* (which depends on time) structures.
The figure 3.1 depicts just a hierarchical structure. In this work we will not describe the other forms of system's structures, which was named above. We just want to point out that giving a structural (direct) definition of systems, is not always enough to describe a system well.

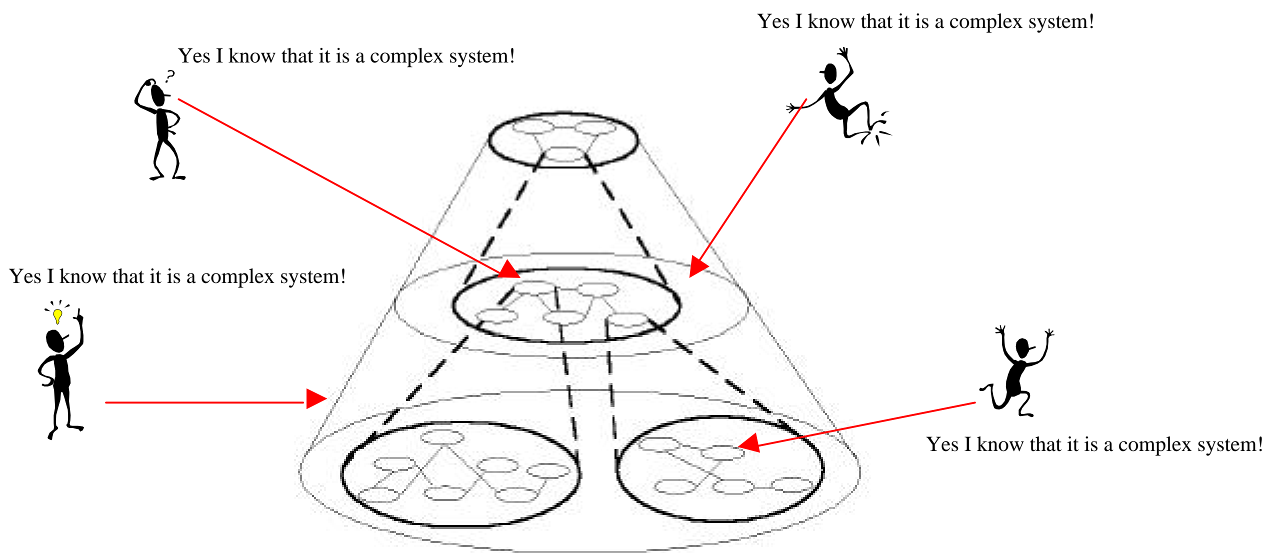


Figure 3.1 *In this hierarchical system structure (and even in the other structural forms), different observers have different focus and view over the system!*

As we see here a direct definition of system is not enough to get a good understanding over this concept.

The most difficult problem in this way is to take an acceptable restriction over the elements, which are included in an arbitrary system so “my system” even *is* “your system”, in other word we have a common view over a system! (Norrbom, 1973)

Missing this common view creates big problems because it can lead to a misunderstanding where we suppose that we are talking about the same thing and in the same language but actually we mean different things.

This problem grows much more when the problem area does not contain just one system but a number of systems. (Johnson et al., 1973)

A solution to work out this problem is that we also use an indirect definition. In other words, we try to get an understanding over the functions, inputs, outputs of a system and even try to get a better view over *“how a system can be influenced by other systems”* and *“how it influences other systems”*. In this way we can understand: “*Who* has control over *what*?!”. (Johnson et al., 1973)

This helps us to demarcate the area of the system and therewith it is more truly that we get a common view over system where “your system” also *is* “my system”!

One way to get a more clear view over a system is getting a *“Root definition”*.
With a root definition we try to find answers to three main questions: *What, How and Why!*
There we try to describe *What* is to be done by the system, *How* the system will attempt to do this and *Why* it is to do this.

Unfortunately even a root definition is not enough to define a system.

A popular way to define an information system in this mixed way is CATWOE, which was presented in 1976 by Smith & Checkland.

With CATWOE we try to research the construction of a system with help of some important points:

The **C**ustomers of the system	The beneficiaries or victims of the systems activities, who is advantaged or disadvantaged by the system.
The **A**ctors	The persons who carry out or cause the system’s activities to be carried out.
The **T**ransformation processes	The core transformation process of the human activity system. This might be defined in terms of the input and output of the transformation.
The **W**eltanschauung	The basic beliefs or view of the world implicit in the root definition that give coherence to this human activity system and make it meaningful.

The **O**wners	The persons who have the power to modify or demolish the system.
The **E**nvironmental constraints	The constraints on the system imposed by its environment or a wider system that taken as given in the root definition.

(Lewis, 1994)

Analysing these six important points is not always too easy. It can sometimes be very difficult to find out these points, especially when we talk about enterprises, which are very complex where some parts of the company are customers/suppliers to some other parts of the same company or some parts of the company buy products from outside companies to prepare products for selling to other parts of the same company or vice versa.

In its entirety, the figure below depicts a system in a very compact and informative way.

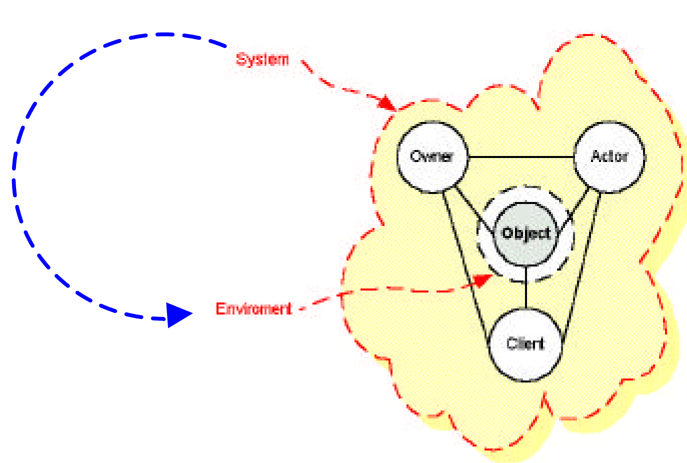


Figure 3.2 Each environment contains of many systems!

Finally, systems theory puts the focus on a total view over the whole system with all its components, and also on the relationship between these components. In this way, we mean that a system is not just a collection of components. According to system theory, a system is defined as "*an array of components designed to accomplish a particular objective according to plan*". (Johnson et al., 1973)

Thus, choosing a collection of right components is very important, and is a difficult task to perform. But it is also important that these ***components*** also work with each other in a harmonically way, i.e. there is an ***interplay***. Therefore it exists some ***design*** and each design has some ***purpose(s)***.

In the journey from an abstract world of system concepts to the concrete world of information systems, we take the advantages of using the system theory as our guiding-star in this thesis.

Using this theoretical framework gives us the power to recognize the connectivity between key concepts such as software systems, components, architecture and integration in a software engineering approach with the concepts of systems, parts, relations, etc. that has already been discussed in the systems theory.

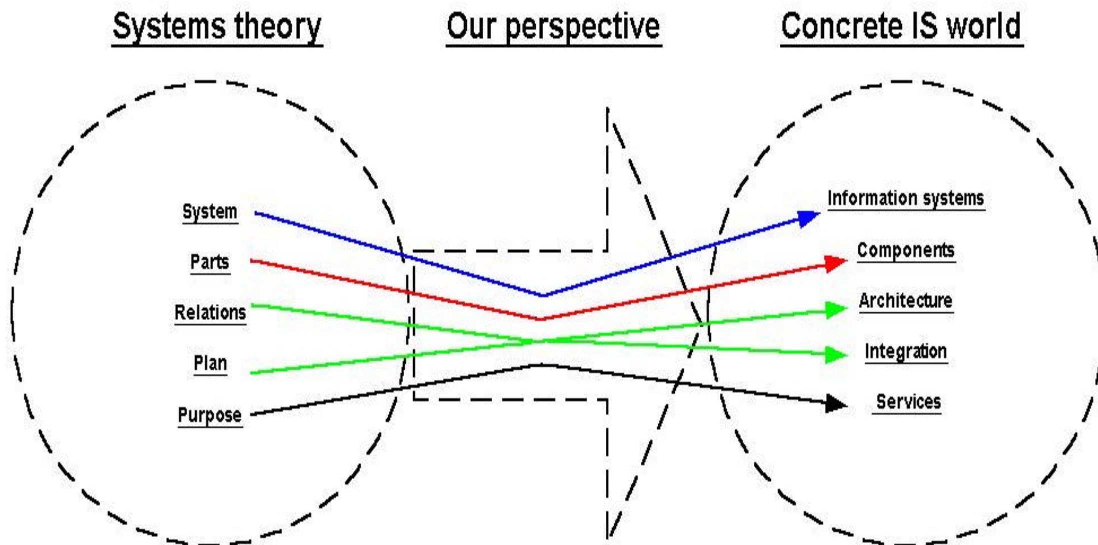


Figure 3.3 Our model of application of systems theory in the information system context

4 Component-based software development

”The best way to attack the essence of building software is not to build it at all. Package software is only one of the ways of doing this. Program reuse is another.”

F.P. Brooks, JR

In this section, we will give an overview of a new paradigm for system development, which is the basis for the three dominating techniques described in depth in this thesis.

In recent years, a new paradigm in the development process has been recognized. This is a paradigm motivated by the decreasing lifetime of applications, the need for more rapid product development, and economic pressures to reduce system development and maintenance costs. It focuses on the realization of systems through integration of pre-existing components and shifts the development emphasis from programming software to composing software systems. The final products are not closed monolithic systems, but are instead component-based products that can be integrated with other products available on the market. The developers are not only designers and programmers, they are integrators and market investigators. (Larsson, 2000)

There are a number of factors, which are driving the approach of component-based software development. According to the Software Engineering Institute they are (Brown, 1996):

- The developments of the World Wide Web and the Internet have increased understanding and awareness of distributed computing. The WWW encourages users to consider systems to be loosely coordinated services that reside somewhere in “cyberspace”. In accessing information it becomes unimportant to know where the information physically resides, what underlying engines are being used to query and analyse the data, and so on.

- Both the use of object-oriented software design techniques and languages, and the move from mainframe-based systems toward client/server computing, lead developers to consider application system not as monolithic, but rather as separable, interacting components. In some applications, for example, compute-intensive storage and search engines are separated from visualization and display services.

- The rapid pace of change in technology has brought many advantages, but also a number of problems. Organisations are struggling in their attempts to build systems in such a way that they can incrementally take advantages of technology improvements over the system’s lifetime. Such flexibility to accept technology upgrades is seen as a key to gaining competitive advantages.

- The economic realities of the 1990’s have to lead to downsizing, restructuring, and rescoping of many organizations. In many situations it is infeasible to consider constructing each new system from scratch. To stay in business, organizations must learn to reuse existing practices and products, and when appropriate to build upon commercial packages in as many parts of a system as possible.

- Finally, and perhaps most importantly, a revolution has occurred in the business environment in which organizations operate. A key to the success of many organizations is to maintain some measure of stability and predictability in the markets in which they operate, in the technology employed to support its core businesses, and in the structure of the organization itself. Unfortunately, the past few years has seen an inexorable rise in rate of change faced by organizations in all of these areas. Strategic advantage can be gained by those organizations that can deal with these changes most effectively. The ability to manage complexity and rapidly adapt to change has become an important differentiator among competing organizations.

Building better software in less time is the goal of every IS shop. Every significant advance in software development, whatever its source, aims at that goal. In the last forty years, advances such as high-level languages, database management systems, object technology, and more have helped us to improve the way we create software. The next major innovation in software development is component-based development. Component-based development means assembling applications from reusable, executable packages of software called components that provide their services through well-defined interfaces. The components may have been written internally or purchased from third parties, but in either case, the result is the same: more reliable applications produced in less time. Component-based development still requires programming--it's technology, not magic--but by building on what's already available, it makes the process of software development faster and easier. (Chappell, 1997)

4.1 Two extremes and the third way

Traditional software development can be described by two extremes.

- On one side we have the custom construction of a tailored solution fitting a specific customer's exact needs.

- On the other side we have the development of application packages that is made from general demands with an entire customer-segment in focus.

The custom-made approach has the advantage that the software system can support the customer's way of making business, if this is a unique way the customer can get a competitive edge. A drawback concerning custom made software systems is the cost for developing and time to market, the whole cost should be covered by the increased profit which using the system should result in. If several customers split this cost, as with developing application packages, the cost tends to be lower. Some other disadvantages with custom made software systems are the communication problem between the software engineer and the customer as mentioned above and the new systems' ability to communicate with other existing and yet to come software systems. (Christiansson & Jakobsson, 2000)

These disadvantages do not apply when acquiring application packages. There are however drawbacks with the use of application packages as well. One disadvantage is that when acquiring an application package one may need to reorganize ones way of making business to fit the application package. Another drawback is if competitors use the same application package that is not in itself a competitive edge. Yet another drawback is that when a company changes its way of making business it is very difficult to change an application package at the same time.

Choosing a standard solution may force drastic changes in the culture and operation of an organization. An example is the Australia Post, which decided in 1996 to use a new integrated solution (SAP's R/3). The Australia Post has a large organization with a federated structure, which means that each state has its own head office reporting to a central head office. R/3 makes it possible to keep track of each individual transaction, down to the sale of a single stamp but it supports only a monolithic hierarchy of access authorization. It means that it is not possible to grant the national head office access to the accounts in-the-large without also granting access to every individual transaction. This undermines organization's traditional autonomy in making locale decisions and clashes with the concept of a federated organization. (Szyperski, 1999)

The development of software systems should be imprinted with the use of situation-adaptation, which means that the development should be adapted to the present and unique situation. This adaptation can result in a combination of the two above described extremes. The concept of component software represents a middle path that could solve this problem. Although each bought component is a standardized product, with all the advantages that brings, the process of component assembly allows the opportunity for significant customization. The components of different quality will be available at different prices and this makes it possible to set individual priorities when assembling based on a fixed budget. Some individual components can be custom-made to suite specific requirements or to win some strategic advantages. (Szyperski, 1999)

4.2 The Benefits of Component-Based Development

- Component-based development can produce applications more quickly. Assembling an application from components, then writing only the code required for new features is much faster than writing the entire application from scratch. As hardware designers have known for years, building on existing components is a very effective way to increase productivity.

- Component-based development can result in more reliable software with higher quality. Creating an application that's largely constructed from existing components means that much of the application's code has already been tested. While testing of the complete application is still required, component-based applications can be more reliable than those developed using traditional techniques simply because the code within each component is already known to work.

- Component-based development lets developers focus more on business problems. Building a component-based application using, say, Visual Basic is significantly easier than building an object-oriented application in C++. While this does not mean that programmers are no longer needed - far from it - component-based development lets those programmers spend more of their time addressing the business problem they're solving, rather than worrying about low-level programming details.

- Component-based development can be cheaper than traditional development. Because component-based development allows building more reliable applications in less time, it can save money.

- Component-based development allows easy mixing and matching of languages and development environments. Components written in one language can be easily used from

another language or even another machine. The component model provides a standard packaging scheme that makes this transparency possible.

- Component-based development offers the best of both alternatives in the build vs. buy decision. Rather than buying into a complete and perhaps less-than-perfect packaged solution, an organization can purchase only the required components, and then combine them into a customized solution. Doing this lessens risk, because the purchased components have already been extensively used and tested, while at the same time allowing the organization to build a customized solution to meet its unique needs. This is an especially attractive approach for server components. In fact, we may see server applications packaged entirely as groups of components in the not-too-distant future, with each component complementing the others to provide a complete, highly customisable service. (Chappell, 1997)
- One of the key problems in the development of modern software systems is planning for change: open systems must be flexible in that they must be easy to adapt to new and changing requirements. Increasingly systems developers have come to the consensus that the best way of dealing with open requirements is to build systems out of reusable components conforming to a "plug-in architecture". The functionality of an open system can then be changed or extended by substituting components or plugging in new components.
- Component software also puts an end to the age-old problem of massive upgrade cycles. Traditional fully integrated solutions required periodic upgrading, usually a painful and expensive process of migrating old databases, ensuring upwards compatibility, retraining staff, buying more powerful hardware, and so on. In a component-based solution, evolution replaces revolution, and individual upgrading of components as needed and out of phase can allow for much smoother operations. This requires of course a different way of managing services. (Szyperski, 1999)
- Using CBD means reducing the gap between the analysis- and design phases. There is also, according to Langefors, a gap between the design and implementation phases when proceeding from the infological to the datalogical part of the system development process. This gap will be reduced, since a component always has a specification, one or more implementations and executable forms. This means that one and the same component can be described and identified in all phases of the lifecycle of an information system through its three forms. Depending on which phase of the lifecycle that the information system is in, the component form that is most suitable for the moment is used. For infological problems the specification is used, and for datalogical problems the implementation and the executable form is used. (Christiansson & Jakobsson, 2000)

4.3 Introduction to some fundamental concepts

4.3.1 Components

There are several established definitions, which define this central concept in various ways from similar and different points of view. We look at three of them:

“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.”
ECOOP (European Conference On Object-oriented Programming, 1996)

“A physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces. A component represents a physical piece of implementation of a system, including software code (source, binary or executable) or equivalents such as scripts or command files.”
OMG (CORBA)

”A component is a reusable piece of software in binary form that can be plugged into other components from other vendors with relatively little effort.”
Microsoft (COM)

Microsoft’s definition describes only a property of components, which is obviously true, but is too weak for a complete definition of this concept. It even holds for compiled libraries (e.g., .o- and .dll – files). (Kiziltan et al., 2000) The definition from OMG is more sufficient, but we have found the one from ECOOP superior because it covers the most characteristic properties of components. It has a technical part with aspects such as independence, contractual interfaces, and composition. It also has a market-related part, with aspects such as third parties and deployment. Besides the specification of provided interfaces, this definition of components also requires components to specify their needs. In other words, the definition requires specification of what the deployment environment will need to provide, such that the components can function. These needs are called context dependencies, referring to the context of composition and deployment.

From the similar point of view, Clemens Szyperski defines three characteristic properties of components as below (Szyperski, 1999):

- A component is a unit of independent deployment
- A component is a unit of third-party composition
- A component has no persistent state

This means that a component is well separated from its environment and other components, and is sufficiently self-contained. It should come with clear specifications of what it requires and

provides. Components cannot be distinguished from copies of their own. In any given process, there will be at most one instance of a particular component.

The last statement, "*A component has no persistent state*" is the focus of a hot debate between different schools of thought. Persistence is a key issue in for example Enterprise JavaBeans component architecture. By this statement, Szyperski makes a distinct separation between concepts of objects and components. He states that the notions of instantiation, identity, and encapsulation lead to the notion of objects. In contrast to the properties characterizing components, an object's characteristic properties are:

- It is a unit of instantiation (it has a unique identity).
- It has state that can be persistent.
- It encapsulates its state and behavior.

According to him, a component comes to life through objects and therefore would normally contain one or more classes or immutable prototype objects. In addition, it might contain a set of immutable objects that capture default initial state and other component resources. However, there is no need for a component to contain only classes or any classes at all. A component could contain traditional procedures; or it may be realized in its entirety using a functional programming approach, an assembly language, or any other approach. Objects created in a component, or references to such objects, can become visible to the component's clients, usually other components. A component may contain multiple classes, but a class is necessarily confined to a single component, since partial deployment of a class wouldn't normally make sense. (Szyperski, 1999)

Szyperski (1999) gives an example in his book Component Software, beyond Object-Oriented programming to describe this more clearly. A database server could be a component. If there is only one database maintained by this class of server, then it is easy to confuse the instance with the concept. For example, you might see the database server together with the database as a component with persistent state. According to the definition described previously, this instance of the database concept is not a component. Instead, the static database server program is and it supports a single instance: the database object. According to him this separation of the immutable plan from the mutable instances is key to avoid massive maintenance problems. If components could be mutable, that is, have observable state, then no two installations of the same component would have the same properties. The differentiation of components and objects is thus fundamentally about differentiating between static properties that hold for a particular configuration and dynamic properties of any particular computational scenario. Drawing this line carefully is essential to curbing manageability, configurability, and version control problems.

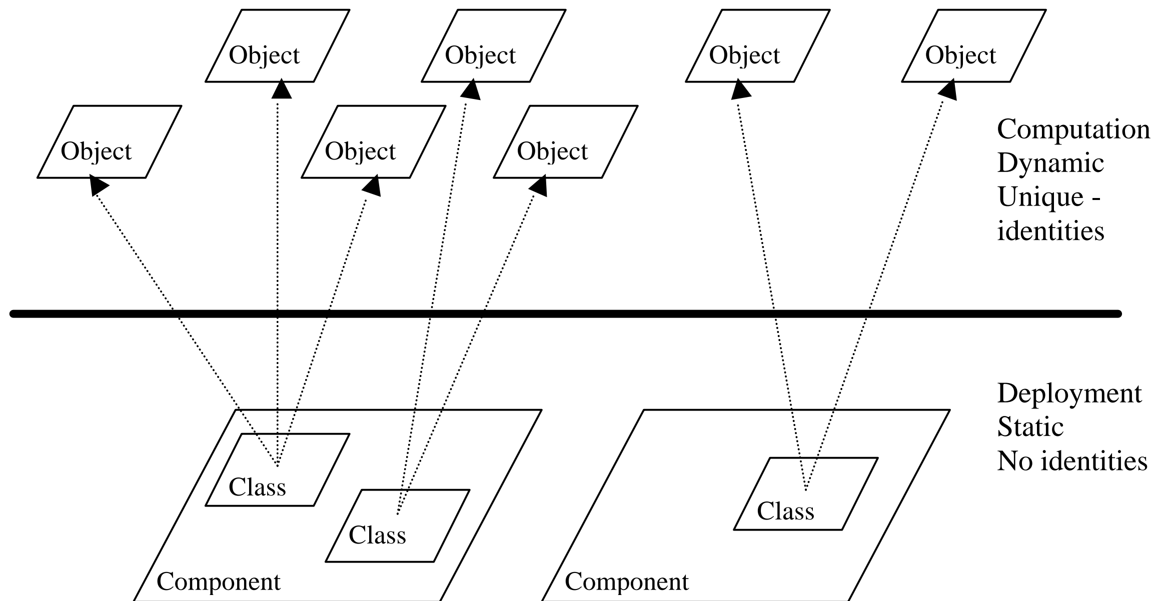


Figure 4.1 Components are the deployable static units that, when activated, can create interacting objects to capture the dynamic nature of a computation (Szyperski, 2000)

The same school of thought considers the object technology (OT) insufficient and non necessary for component-based software development (CBSD). They mean although OT was a useful and convenient starting point for CBSD, by itself:

- OT did not express the full range of abstractions needed by CBSD

  And

- It is possible to realize CBSD without employing OT.

To illustrate the insufficiency of OT for CBSD, they consider the role of a component as a replacement unit in a system. The earlier definitions of component addressed at least one characteristic that relates to replaceability - explicit context specification. Concretely, explicit context specification might be implemented via a "uses" clause on a specification, i.e., a declaration of what system resources are required for the component to work. OT does not typically support this concept. (Brown & Wallnau, 2000) To illustrate the non-necessity of OT, they argue that there is no need for a component to contain any classes. It could contain procedures or other executable program segments, which don't behave as objects. In addition they consider the lack of a visibility scope, which can enclose several objects, as a fundamental weakness of most Object-Oriented programming languages. Furthermore, object-orientation binds the implementation to a particular class library and language. Take Smalltalk as an example. In Smalltalk the programmer is bound to the Smalltalk language and the classes provided in the environment. Components should not be bound to a particular language and they communicate through independent interfaces. (Larsson, 2000)

Diversity of perspective appears even in consideration of components as replacement units in component-based systems. The concept of replacement unit has different meanings depending upon which of two major perspectives are adopted. The first perspective (CBSD with off-the-shelf components) views components as commercial-off-the-shelf commodity. In this perspective, CBSD requires industrial standardization on a small number of component frameworks. The second perspective (CBSD with abstract components) views components as application-specific core business assets. This perspective places much less emphasis on standard component infrastructures or component marketplaces, and instead emphasizes component-based design approaches. (Brown & Wallnau, 2000)

Furthermore, components can be considered from three different views of architecture:

- Run-time: This includes component frameworks and component models that provide run-time services for component-based systems.

- Design-time: This includes the application-specific view of components, such as functional interfaces and component dependencies.

- Compose-time: This includes all that is needed to assemble a system from components, including generators and other build-time services (a component framework may provide some of these services). (Brown & Wallnau, 2000)

However the majority agree on this point that a component package may consist of:

- A list of provided interfaces to the environment.

- A list of required interfaces from the environment.

- External specifications.

- Executable code.

- Design (i.e. documents and source code).

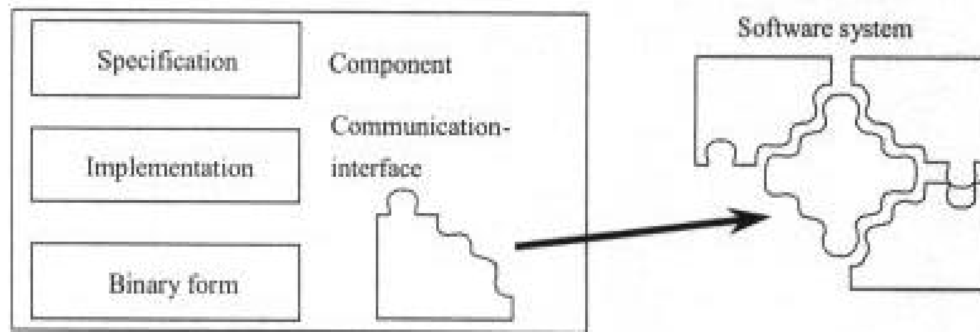


Figure 4.2 Component package (Christiansson & Jakobsson, 2000)

Components today can be grouped into four categories, and various component technologies address each of those categories. The categories and their relevant technologies are:

- In-process client components. This style of component must run inside a container of some kind - it can't run on its own. Popular choices for containers today include PowerBuilder, Visual Basic, and web browsers such as Netscape Navigator. A developer is usually able to manipulate this style of component visually, allowing her to build an application from components by dragging them onto a form, then writing the code needed to make them work together. The leading technologies for creating in-process client components are Microsoft's ActiveX Controls (which are based on COM), IBM's OpenDoc, and Sun's JavaBeans.

- Standalone client components. Any application that exposes its services in a standardized way to other software can be used as a component. Many Windows desktop applications do this, including Excel, Word, CorelDraw, and more. By far the most common technology used for this purpose today is Automation, which is another application of Microsoft's COM.

- Standalone server components. A process running on a server machine that exposes its services in a standardized way can also qualify as a component. This kind of component is accessed via remote procedure calls or some other kind of network communication. The most visible technologies supporting this option are Microsoft's Distributed COM (DCOM) and the Object Management Group's Common Object Request Broker Architecture (CORBA). Sun's Java environment also supports this option using Remote Method Invocation (RMI).

- In-process server components. Given a suitable container, in-process components can also be useful on servers. An obvious choice for a server-side container is a transaction server. The Microsoft Transaction Server (MTS) is an example of this kind of container, and it requires developers to create transaction programs as ActiveX (i.e., COM-based) components. Sun has Enterprise JavaBeans, a model for in-process components targeted to run in various kinds of server-side containers.

(Chappell, 1997)

4.3.2 Interfaces

The online Merriam-Webster dictionary (2001) defines an interface as "*the place at which independent and often unrelated systems meet and act on or communicate with each other*"; "the means by which interaction or communication is achieved".

In the world of software systems, components express themselves through interfaces. An interface is the connection to the user that will interact with a component. Szyperski (1999) defines interfaces as means by which components connect. In fact they are access points to components. Technically, an interface is a set of named operations that can be invoked by clients. According to him, each operation's semantics is specified, and this specification serves both:

- Providers implementing the interface

- Clients using the interface

(Szyperski, 1999)

As, in a component setting, providers and clients are ignorant of each other, the specification of the interface becomes the mediating middle that lets the two parties work together. The interface of a component is important for composition and customization of components by users, allowing to find suitable components and to understand their purpose, functionality, usage and restrictions.

Components can export functionality by implementing one or more interfaces, and can import functionality by using interfaces from other components. Hence, export interfaces correspond to the services a component provides, and import interfaces corresponds to the services a component need in order to implement the exported services. (Kiziltan et al., 2000)

A component may either directly provide an interface or it may implement objects that, if made available to clients, provide interfaces. Interfaces directly provided by a component correspond to procedural interfaces of traditional libraries and indirectly implemented interfaces correspond to object interfaces. The procedural interfaces are always direct, the definition and its implementation belongs to the same component. An object interface, on the other hand introduces an indirection called method dispatch, which can lead to the involvement of a third party of which both the calling client and the interface-introducing component are unaware. This is the case when the object implementing an interface belongs to a component different from the component with which the client seems to interact through the interface. (Szyperski, 1999)

Szyperski gives an example to make this clearer: (Look at the figure 4.3) A word processing client calls on services of a grammar checker that is acquired indirectly from a component mediating between clients and providers of text services. First, the grammar checker knows about the text service mediator. Secondly, the grammar checker registered itself as the default checker with the mediator. The mediator knows only about the abstract checker interface. Thirdly, the word processor knows about the mediator. Fourthly, the word processor acquires a reference to the current default checker from the mediator. Just as the mediator, the word processor, knows only the abstract checker interface.

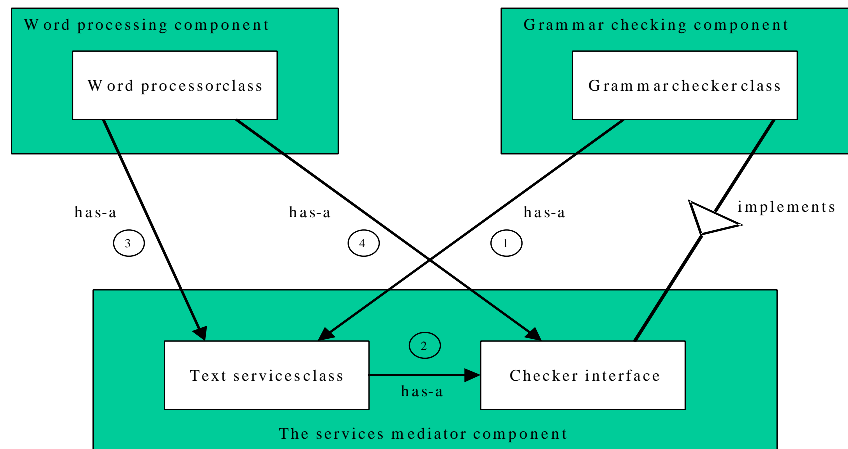


Figure 4.3 Illustration of an indirect interface (Szyperski, 1999)

4.3.3 Contracts

"*In general, the approach to component interfaces is syntactic*", (Kiziltan et al., 2000) which is not sufficient to solve problems regarding semantic issues, related to context dependencies and interactions. "*Therefore, at a minimum, component's interface(s) should provide a contract that states both the services a component offers and the services that it requires in fulfilling its commitments*". (Kiziltan et al., 2000) Just as a contract in real life, it binds two partners, the provider of services and the consumer and defines certain obligations and benefits for both parts in an agreement.

Specifying pre- and post conditions for the operation can capture the two sides of the contract. The client has to establish the precondition before calling the operation, and the provider can rely on the precondition being met whenever the operation is called. The provider has to establish the post condition before returning to the client and the client can rely on the post condition being met whenever the call to the operation returns. (Szyperski, 1999)

Ideally, a contract should cover all essential functional and non-functional aspects, but with pre- and post conditions not all aspects of an interface can be specified. Aspects like safety and progress conditions, complexity bounds, time and space requirement are examples of issues, which should be specified in a contract. In fact the simple pre- and post conditions on operations can be used only to establish partial correctness; a partially correct operation either terminates correctly or does not terminate at all. To achieve total correctness, termination is required and this requirement can be added as a convention to all contracts. (Szyperski, 1999)

Dijkstra's weakest preconditions for an operation and a given post condition is a popular notation for totally correct conditions:

The predicate Wp(C, R) is true in the state s if and only if every execution of C, which begins in s, leads to a state, in which R is satisfied.

$Wp(i:=i+1, i>0) \equiv i>-1$ (Holström, 1990)

4.3.4 Design Patterns and Frameworks

The basic idea of using patterns is borrowed from work done in building architecture to describe qualities for good architectural designs. In the seventies, the architect Christopher Alexander started using pattern languages to describe the events and forms that appeared in cities, towns, and buildings in the world at large. He saw similarities in architectural structures, the problem of constructing them and the solution to construct them well. He showed that many of the problems in architectural design could be represented as design patterns. Alexander wrote "*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice*". (Larsson & Sandberg, 2000) Each building is unique, yet all buildings share many features.

Software development presents an analogous situation. Independently developed software systems often share common elements of an architectural structure. The connection between Alexander's patterns and software architecture has led many in the software community to argue for a higher-level organizing principle in software than that of objects and classes and thereby much of these discussions logically centres on software component design, where it is natural to discuss interactions between entities.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides define design patterns as "*descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. …A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and their instances, their roles and collaborations, and the distribution of responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether or not in can be applied in view of other design constraints, and the consequences and trade-offs of its use. Since we must eventually implement our designs, a design pattern also provides sample ... code to illustrate an implementation. Although design patterns describe object-oriented designs, they are based on practical solutions that have been implemented in mainstream object-oriented programming languages....* " (Gamma et al., 1995)

Another concept closely related to patterns is frameworks. A framework defines a certain group of participants and relations between them as a highly reusable design for an application, or part of an application, in a certain domain. (Kiziltan et al., 2000) According to Gamma et al. the framework dictates the architecture of an application. It will define the overall structure, its partitioning into classes and objects, the key responsibilities thereof, how the classes and objects collaborate, and the thread of control. (Gamma et al., 1995)

A framework is different from a design pattern. While design patterns are microarchitectures, which describe the abstract interaction between objects collaborating to solve a particular problem, a framework is a concrete powerful solution that can be described in source code. Only examples usage or applications of a design pattern can be described in source code. Furthermore, a framework

is often built by the application of a number of design patterns, and thus, patterns describe microarchitectures often used in frameworks. (Larsson & Sandberg, 2000)

Gamma et al. (1995) list the following differences between patterns and frameworks:

- Design patterns are more abstract. Frameworks are partial implementations of subsystems, while patterns have no immediate implementation at all; only examples of patterns can be found in implementations.

- Design patterns are smaller architectural elements than frameworks. A typical framework contains several design patterns, but the reverse is never true.

- Design patterns are less specialized. Frameworks always have a particular application domain.

A problem with using frameworks in practical development is to determine the level of detail, at which the framework should be defined. Finding the proper boundaries and granularity for a framework is a crucial and hard task. A high level of rigidness, limits the use of the framework and would risk to rule out certain target domains. A high level of flexibility, on the other hand, increases complexity and makes it inefficient in usage.

What is the relationship between components and these architectural elements? Actually as the figure 4.4 shows, one or more design patterns can be applied to build a component, but also, as a realization of a design pattern, one or more components can be used. Furthermore, components can be used as parts in for example a framework and a framework can be viewed as the glue code that makes components work together. In fact, technologies like JavaBeans, COM/DCOM or CORBA, are different specialized frameworks making it possible to connect components. (Larsson & Sandberg, 2000)

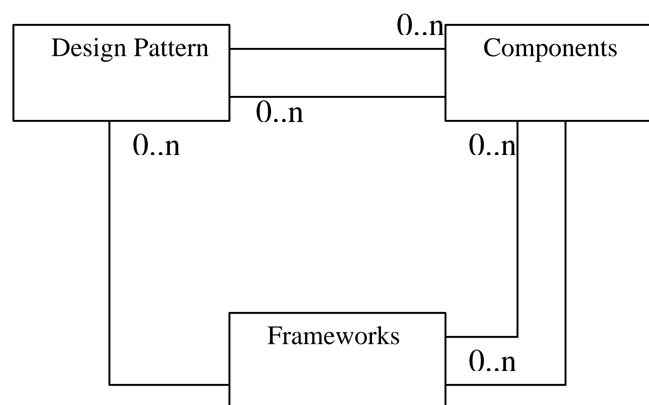


Figure 4.4 Relationships between Patterns, Frameworks and Components (Larsson & Sandberg, 2000)

4.4 Activities of the Component-Based Development Approach

Development with components differs from the traditional development. In component-based development, the notion of building a system by writing code has been replaced with building a system by assembling and integrating existing software components. In contrast to traditional development, where system integration is often the tail end of an implementation effort, component integration is the centrepiece of the approach; thus, implementation has given way to integration as the focus of system construction. Because of this, integrability is a key consideration in the decision whether to acquire, reuse, or build the components.

As depicted in figure 4.5, four major activities characterize the component-based development approach:

- Component qualification (sometimes referred to as suitability testing)
- Component adaptation
- Assembling components into systems
- System evolution

The vertical partitions shown in figure 4.5 describe the central artefact of component-based systems – the component – in various states. These are:

- *Off-the-shelf components* have "hidden interfaces" (not fit for use). They come from a variety of sources; some developed in-house (perhaps used in a previous project), others specifically purchased from commercial vendor.
- *Qualified components* have discovered interfaces so that possible sources of conflict and overlap have been identified.

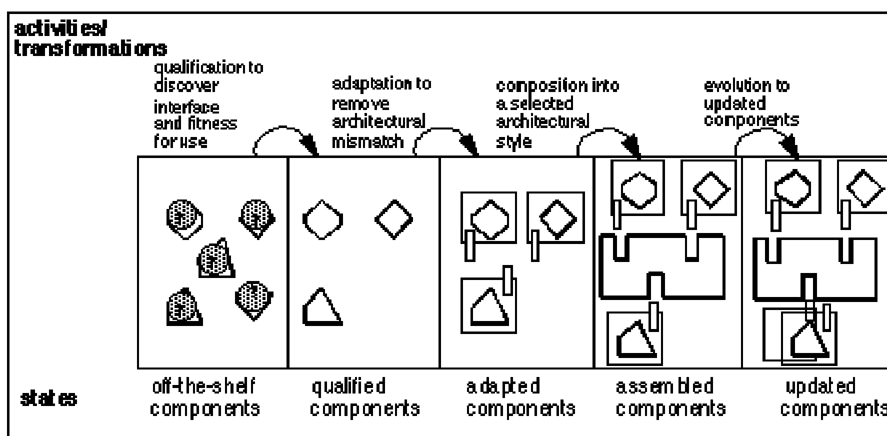


Figure 4.5 Activities of the Component-Based Development Approach (Brown, 1996)

- *Adapted components* have been amended to address potential sources of conflict. The figure implies a kind of component “wrapping”, but other approaches are possible, like the use of mediators and translators.

- *Assembled components* have been integrated into an architectural infrastructure. This infrastructure will support component assembly and co-ordination.

- *Updated components* have been replaced by newer versions, or by different components with similar behaviour and interfaces. Often this requires wrappers to be rewritten, and for well-defined component interfaces to reduce the extensive testing needed to ensure operation of unchanged components is not adversely effected. (Brown, 1996)

And now a description of the major activities in component-based development according to Carnegie Mellon, Software Engineering Institute (2000):

- *Component qualification* is a process of determining "fitness for use" of previously developed components that are being applied in a new system context. Component qualification is also a process for selecting components when a marketplace of competing products exists. Qualification of a component can also extend to include qualification of the development process used to create and maintain it (for example, ensuring algorithms have been validated, and that rigorous code inspections have taken place). There are two phases of component qualification: *discovery* and *evaluation*. In the *discovery phase*, the properties of a component are identified. Such properties include component functionality (what services are provided) and other aspects of a component's interface (such as the use of standards). These properties also include quality aspects that are more difficult to isolate, such as component reliability, predictability, and usability. In some circumstances, it is also reasonable to discover "non-technical" component properties, such as the vendor's market share, past business performance, and process maturity of the component developer's organization. Discovery is a difficult and ill-defined process, with much of the needed information being difficult to quantify and, in some cases, difficult to obtain. There are some relatively mature *evaluation techniques* for selecting from among a group of peer products. For example, the International Standards Organization (ISO) describes general criteria for product evaluation while others describe techniques that take into account the needs of particular application domains. These evaluation approaches typically involve a combination of paper-based studies of the components, discussion with other users of those components, and hands-on benchmarking and prototyping. One recent trend is toward a "product line" approach that is based on a reusable set of components that appear in a range of software products. This approach assumes that similar systems (e.g., most radar systems) have similar software architecture and that a majority of the required functionality is the same from one product to the next. The common functionality can therefore be provided by the same set of components, thus simplifying the development and maintenance life cycle.

- *Component adaptation*: Because individual components are written to meet different requirements, and are based on differing assumptions about their context, components often must be adapted when used in a new system. Components must be adapted based

on rules that ensure conflicts among components are minimized. The degree to which a component's internal structure is accessible suggests different approaches to adaptation:

- *White box,* where access to source code allows a component to be significantly rewritten to operate with other components.
- *Grey box,* where source code of a component is not modified but the component provides its own extension language or application programming interface (API).
- *Black box,* where only a binary executable form of the component is available and there is no extension language or API.

Each of these adaptation approaches has its own positives and negatives; however, white box approaches, because they modify source code, can result in serious maintenance and evolution concerns in the long term. Wrapping, bridging, and mediating are specific programming techniques used to adapt grey- and black-box components.

- *Assembling components into systems*: Components must be integrated through some well-defined infrastructure or frameworks. These architectural styles provide the bindings that form a system from the disparate components.

- *System evolution*: Component-based systems seem relatively easy to evolve and upgrade since components are the unit of change. To repair an error, an updated component is swapped for its defective equivalent, treating components as plug-replaceable units. Similarly, when additional functionality is required, it is embodied in a new component that is added to the system. However, this is a rather simplistic view of system evolution. Regarding today's component technology, replacement of one component with another is still a time-consuming and arduous task since the new component will never be identical to its predecessor and must be thoroughly tested, both in isolation and in combination with the rest of the system. Wrappers must typically be rewritten, and side effects from changes must be found and assessed.

5 Integration principles

In this section, we will study what integration principles could be suitable for software component integration.

5.1 Introduction

According to systems theory a system is defined as " *an organised or complex whole; an assemblage or combination of things or parts forming a complex or unitary whole*." With the same spirit integration is defined as the "*act or process of making whole or entire, to bring parts together into a whole*". (Johnson et al., 1973) The *vitalist* theory of deduction or philosophical reasoning proposes the following points regarding principles of integration:

- The whole is primary and the parts are secondary.
- Integration is the condition of the interrelatedness of the many parts within one.
- The parts so constitute an indissoluble whole that no parts can be affected without affecting all other parts.
- Parts play their role in light of the purpose for which the whole exists.
- The nature of the part and its function is derived from its position in the whole and its behaviour is regulated by the whole to part relationship.
- The whole is any system or complex or configuration of energy and behaves like a single piece no matter how complex.
- Everything should start with the whole as a premise and the parts and their relationships should evolve.

(Johnson et al., 1973)

Now it will be helpful to define systems more precisely as "*an array of components designed to accomplish a particular objective according to plan*". (Johnson et al., 1973) Note that there are three significant points in the definition:

- Purpose or objective which, the system is designed to perform.
- Design or an established arrangement of the components.
- Plan, which inputs of information, energy, and materials must be allocated in accordance with.

Regarding these points, the definition of integration - *to make into a whole; unify, to join with something else, and unite* – holds true when considering the phrase "system integration". System integration involves taking a number of disparate components and engineering each component

to integrate with all others. The Institute for Telecommunication Science defines system integration as "*The progressive linking and testing of system components to merge their functional and technical characteristics into a comprehensive, interoperable system. Integration of data systems allows data existing on disparate systems to be shared or accessed across functional or system boundaries*". (Institute for Telecommunication Science, 1996)

Let us transfer the discussion to the world of information systems. An information system according to Andersen is a system for gathering, processing, storing and transferring information. This system can be performed in part or as a whole by computers. The part of the information systems that is being performed by computers is called software system. Langefors describes a software system as constituted by two parts, the informal and the formal part. The informal part contains issues such as required information and how the software solution addresses user-issues. The formal part contains issues such as data storage and computer instructions. (Christiansson & Jakobsson, 2000) Although almost everything mentioned above regarding definitions and principles has correspondence in the world of software systems, the essence of the integration concept in this world is to challenge some of the principles above, especially the third one, which is "*the parts so constitute an indissoluble whole that no parts can be affected without affecting all other parts*". Up to now the traditional way of system development have led to information systems, which are characterized of hard coupling, unchangeability, inflexibility, and isolation (information island phenomenon). (Magoulas & Pessi, 1998) Every effort of modification of a system to satisfy and manage new needs and situations is a very complex task, demanding long time and high level of costs.

The selection of a particular architectural style, or the invention of a custom style, is perhaps the most important design decision affecting the integration process. In fact, the architecture will drive the integration effort.

According to Bass, Clements, and Kazman the software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them. By "externally visible" properties, means those assumptions other components can make of a component, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on. (Carnegie Mellon, Software Engineering Institute, 2001) There are four essential points in this definition:

- Architecture defines components. The architecture embodies information about how the components interact with each other. This means that architecture specifically omits content information about components that does not pertain to their interaction.

- The definition makes clear that systems can comprise more than one structure, and that no one structure holds the irrefutable claim to being the architecture. By intention, the definition does not specify what architectural components and relationships are. Is a software component an object? A process? A library? A database? A commercial product? It can be any of these things and more.

- The definition implies that every software system has architecture, because every system can be shown to be composed of components and relations among them.

- The behaviour of each component is part of the architecture, insofar as that behaviour can be observed or discerned from the point of view of another component. This behaviour is

> what allows components to interact with each other, which is clearly part of the architecture.
>
> (Carnegie Mellon, Software Engineering Institute, 2001)

Generally, integration involves a relationship between entities. Regarding this and the definition above, in the component based software systems, integration is very much a matter of architecture. In fact there are integration architectures. (Isazadeh et al., 1995)

A software integration architecture is a general pattern that serves as a blueprint to define the basic layout of an integrated system. It has to deal with bottom-up integration of existing components as well as the development of new components for top-down integration (pre-facto integration). Integration architecture describes the general strategy of system decomposition, data storage, and communications. The architecture should not be concerned with the internal structure of the components. A software component within the integration architecture, therefore, should be monolithic, i.e., allow no access to its internal structure. (Isazadeh et al., 1995)

According to our interpretation of studying Isazadeh et al.(1995), Christiansson & Jakobsson (2000) and Szyperski (1999) there are the following major issues concerning integration architectures:

- Design and composition**:** what is to be put together and how?
- Managing dynamic situations
- Analysing and managing complex situations
- Communication

We believe that one of the difficulties in integration efforts is the lack of a higher-level and explicit representation of interaction between components, which can be provided by design patterns. With help of design patterns you can collect and abstract common patterns of behaviours and make them explicit and general and thereby get a clear and good high-level view over the interaction between the involved components. This means that the integrator gets more power to manage dynamism, complexity and solving communication issues in the composed system.

5.2 Design patterns

As we earlier mentioned, a design pattern is a description of a suggested solution to a general design problem that may occur in many different situations. Their purpose is to serve software engineers in designing, by using well-proven solutions to their design problems, and to give them a common design vocabulary, that facilitates communication of a specific design. (Hnich, Jonsson & Kiziltan, 1999)

A design pattern is often described as constituted of four elements: a *pattern name*, a *problem* element which describes in which problem contexts the pattern is applicable, a *solution* element which describes the suggested solution in terms of elements, their relationships, responsibilities

and collaborations and a *consequences* element, which describes the results and trade-offs of applying the pattern. (Gamma et al., 1995)

Gamma et al. (1995) describe 23 different design patterns for use in design of object-oriented software. They divide these design patterns into three categories: creational, structural and behavioural patterns. The creational design patterns describe how to abstract the instantiation process of objects, the structural design patterns are concerned with how classes and objects are composed to form larger structures and the behavioural design patterns are concerned with algorithms and the assignment of responsibilities between objects. We believe that some of the structural and behavioural patterns could be applied to the area of component-based development, for use in integrating components. Here follows a description of each of these patterns.

5.2.1 Adapter

An adapter, as explained by Gamma et al. (1995) converts the interface of a class into another interface that the clients expect, thereby making it possible for classes with incompatible interfaces to communicate. Another name for this pattern is *wrapper*. The clients of a class that uses an adapter, will make requests to the adapter instead of the class. The adapter translates the request to a form that can be understood by the class and then sends this request to the class.

This pattern could be applied at a component-level, where the adapter converts the interface of a component (see figure 5.1). By using an adapter, no changes need to be done in the existing component or its clients, to enable communication between them. The term client here could represent other components.

The adapter could be used when using COTS-components that have interfaces that do not match the interfaces that the existing components expect.

Adapters can vary in the amount of work they do to adapt a class/component to its clients. In some cases, the adapter simply needs to change the names of the operations and sometimes a completely different set of operations may need to be supported. This variation depends on the difference between the component's interface and the expected interface. (Gamma et al, 1995)

Introducing an adapter to a system's architecture means that every request to the wrapped component will be performed in two steps, and might result in decreased performance or reliability. (Gamma et al., 1995)

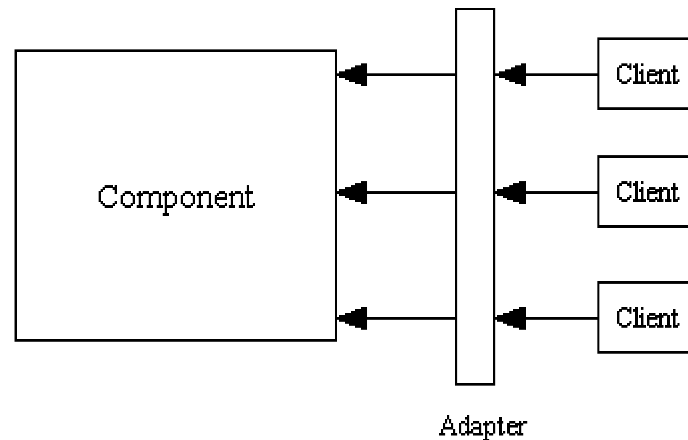


Figure 5.1 Adapter pattern for components

5.2.2 Bridge

Gamma et al. (1995) defines the intent of a bridge as being to decouple an abstraction (in the form of an interface) from its implementation so that the two can vary independently. This means that an implementation will not be permanently bound to an interface. The implementation of the abstraction can be switched at run-time. An abstraction class forwards client requests to an implementor class.

This could be used at the component level, to enable a looser coupling between a component interface and its implementation (see figure 5.2). This is quite similar to an adapter. The major difference is that an adapter is used to make two existing independently designed classes/components communicate with each other while the bridge is used at design time of a single class/component to make it possible to have a stable interface and in the same time enable changes of the implementation as the system evolves. An adapter makes things work after they're designed, while a bridge makes them work before they are. (Gamma et al., 1995)

When changing the implementation of a component that uses a bridge, there is no need to recompile the abstraction or its clients. The clients will not need to be aware of implementation details. The Bridge pattern makes it possible to extend the abstraction and its implementation independently of each other. There is no need for a one-to-one correspondence between the interface and implementation methods. The implementation can provide more primitive operations, and the interface can define higher-level operations based on these primitives. (Gamma et al., 1995)

The choice of implementation can be done in three different ways. One solution is to let the abstraction know of all the existing implementations and let the client choose one of them by sending a parameter specifying its choice. Another solution is to let a default implementation be used that can be changed depending on the usage of the implementation. The last approach is to delegate the choice of implementation to a separate component. Then the abstraction calls this component, which returns a reference to the chosen implementation. (Gamma et al., 1995)

Since the bridge pattern introduces one more layer to a system's architecture, there might be a slightly decreased performance.

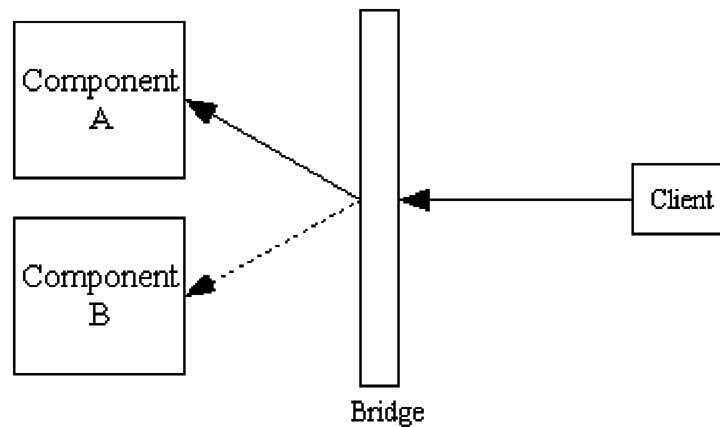


Figure 5.2 Bridge pattern for components

5.2.3 Proxy

A proxy provides a "*surrogate or placeholder for another object to control access to it*" (Gamma et al., 1995). It could be seen as a middleman that manages the communication with a target. It controls access to the target and has an interface that is identical (or a subset) to the interface of the target.

The proxy design pattern is widely used in infrastructures for distributed systems to represent remote components (see figure 5.3). This pattern makes clients communicate with a representative rather than with the component itself.

A proxy could be used to avoid hard coding physical location into the client. It can also be used to provide access control to the component and may for example give different clients different access rights. When used for access control the proxy might provide an interface that is a subset of the component's interface. (Gamma et al., 1995)

The benefit of using a proxy for remote communication is that it decouples clients from the location of remote server components. The proxy becomes responsible for encoding requests and their arguments and for sending the requests to the remote component in a different address space.

The proxy pattern might lead to less efficiency due to indirection and the implementation could become complex.

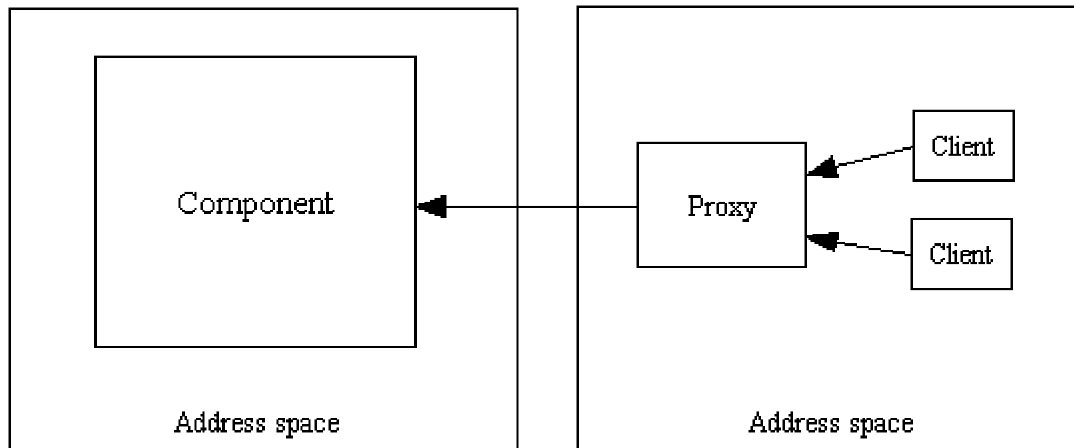


Figure 5.3 Proxy pattern for components

5.2.4 Mediator

The mediator design pattern describes a central connection point that controls the interaction between a set of objects. The mediator functions as a middleman, that provides a looser coupling between the objects, since the objects will not have to explicitly refer to each other. The objects only know the mediator and send all their request to this central hub. (Gamma et al., 1995)

The mediator could at a component-level be a component that manages the interaction between a set of components that have well-defined but complex interdependencies (see figure 5.4). The components will send and receive requests to the mediator. The mediator forwards the requests to the appropriate target component.

A mediator centralizes behaviour management to a central hub. The communication pathways decrease since there will be one-to-many communication between the mediator and the components, instead of one-to-one communication between every component-pair that needs to interact. For example, in a system with N components, where every component needs to interact with every other component, there will be N*(N-1) dependencies without a mediator and 2*N dependencies with a mediator. This means that the components will be more loosely coupled to each other when using a mediator, which facilitates easy replacement and reuse of components.

By letting a central hub manage the interaction between the components, the interaction behaviour is separated from the individual components' behaviour, which could give a more clear view of the system as a whole. A problem with using the mediator pattern is that since the mediator manages all the interaction between the components, this component could easily become very complex and hard to maintain. (Gamma et al., 1995)

Another problem we could see with this pattern is that since the mediator is the only connection point between the components, the architecture becomes heavily dependent on the mediator. If the mediator does not function, no components will be able to send and receive request among

each other. The performance and reliability of the mediator component will have a deep impact on the functioning of the system as a whole.

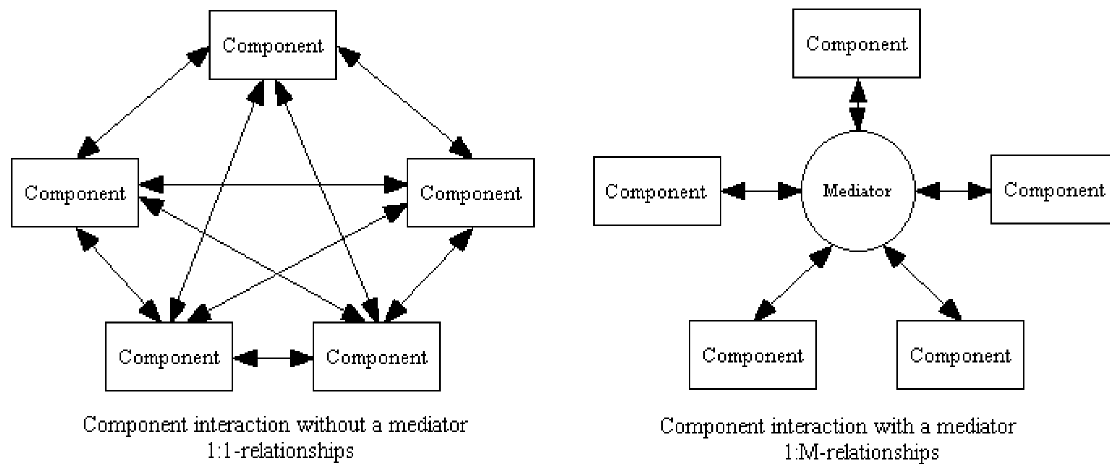


Figure 5.4 Mediator pattern for components

5.2.5 Facade

Facade is a design pattern that gives a high-level interface to a set of interfaces in a subsystem, to make the subsystem easier to use and to shield clients from the inner parts of the subsystem. The purpose of the facade is to provide a simple view of the subsystem that will be enough for most clients. Only those clients, who need more customizability, will have to look beyond the facade. (Gamma et al., 1995)

The facade pattern could be used for components, to provide a simple interface to a component's clients (see figure 5.5). The clients interact with the component by sending requests to the facade, which forwards them to the appropriate part of the component. The facade will be a part of the component's public interface, but there will also be other interfaces that provide a direct entry point to different parts of the component for clients with more specialized needs.

The facade is useful when there are many dependencies between different parts of a component and its clients. A facade provides weak coupling between the component and its clients, which mean changes can be made within the component without affecting the clients. The facade could be used for layering, where the facade functions as a single entry-point to each layer-level. (Gamma et al., 1995)

A drawback of using a facade is that it might not give full possibilities of using all functionality inside a component, so it might not be useful for all clients. Since a facade does not prevent clients from directly making requests to the inner parts of the component, those clients who access the component through other interfaces will still have to be considered when making changes to the component.

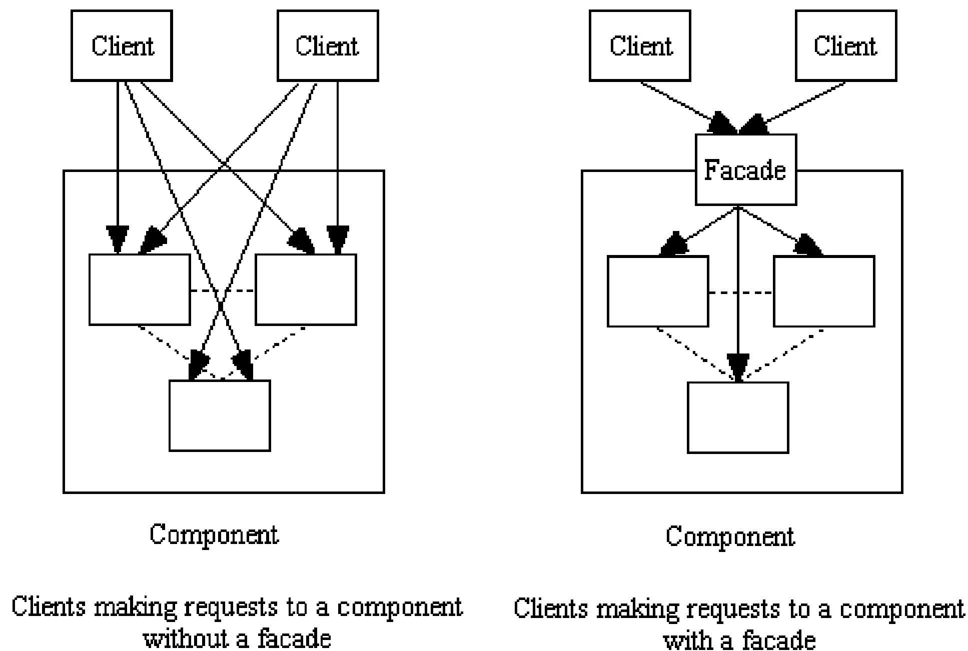


Figure 5.5 Facade pattern for components

5.2.6 Other patterns

Architectural patterns

In their book "Pattern-Oriented Software Architecture", Buschmann, Meunier, Rohnert, Sommerlad and Stal (1996) present 16 patterns divided into architectural and design patterns. Many of the patterns are similar to the design patterns described by Gamma et al. (1995). Two architectural patterns that differ from the patterns described by Gamma et al. and that could be useful for component integration are:

Broker

The Broker pattern can be used to coordinate communication in distributed software systems with decoupled components. It consists of six types of components: clients, servers, brokers, bridges, client-side proxies and server-side proxies. (OpenLoop, 2000a)

A *server* implements services that expose their functionality through interfaces consisting of operations and attributes. The server registers itself with the local broker and receives requests from clients and sends back responses and exceptions through a server-side proxy. The *client* implements user functionality and sends requests to servers through a client-side proxy and receives responses and exceptions. Clients do not know about the location of the servers. A *broker* registers and locates servers. It offers an interface to clients and servers, and transmits the requests from clients to servers and the responses and exceptions from the servers back to clients.

If a request is made to a server hosted by another broker, it forwards the request to that broker (see Figure 5.6). (OpenLoop, 2000a)

A *client-side proxy* constitutes a layer between clients and the broker. It makes remote servers appear as local to the clients, by hiding implementation details like the message transfer between the broker and the client. A *server-side proxy* is analogous to the client-side proxy for the server-side. It mediates between the server and the broker. It receives requests, unpacks messages and calls the appropriate service in the server. A *bridge* is used to bridge communication among different brokers. It encapsulates network-specific functionality. A bridge mediates between the local broker and the bridge of a remote broker. (OpenLoop, 2000a)

The broker pattern provides location transparency, which means that clients do not care where servers are located and vice versa. Server implementations can be changed without affecting the clients as long as the interface is not changed. Changes to internal broker implementation does not affect clients and servers. Different broker systems may interoperate if they have a common protocol for the exchange of messages. The fault tolerance and efficiency compared to non-distributed software will be lower. (OpenLoop, 2000a)

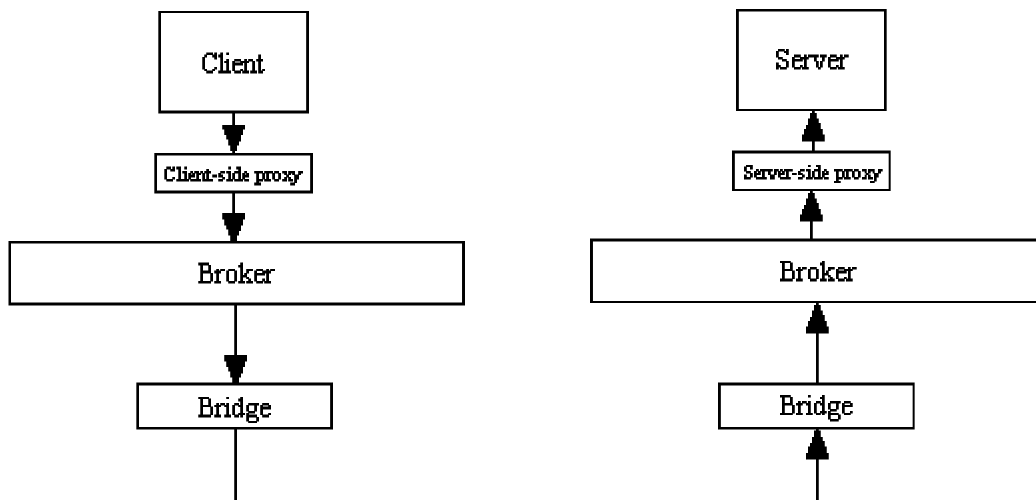


Figure 5.6 Broker pattern (where a client calls a server hosted by a remote broker)

Layer

The Layer pattern structures a system into a number of layers that represent different levels of abstraction. A layer provides services to the layer directly above, and uses services from the layer directly below. This pattern is useful when building a system that consists of a mix of components managing high- and low level issues, where the high-level components make use of the lower-level components (see figure 5.7). (OpenLoop, 2000b)

This way of integrating components keep dependencies more local – for each layer there is only the layer above that uses its services. When changing the interface of a component, only components in the above layer might need to be changed. Layering supports easy reuse of layers. Efficiency might be affected and it could be difficult to decide the appropriate number of layers for a system. (Mathiassen, Munk-Madsen, Nielsen & Stage, 1998; OpenLoop, 2000b)

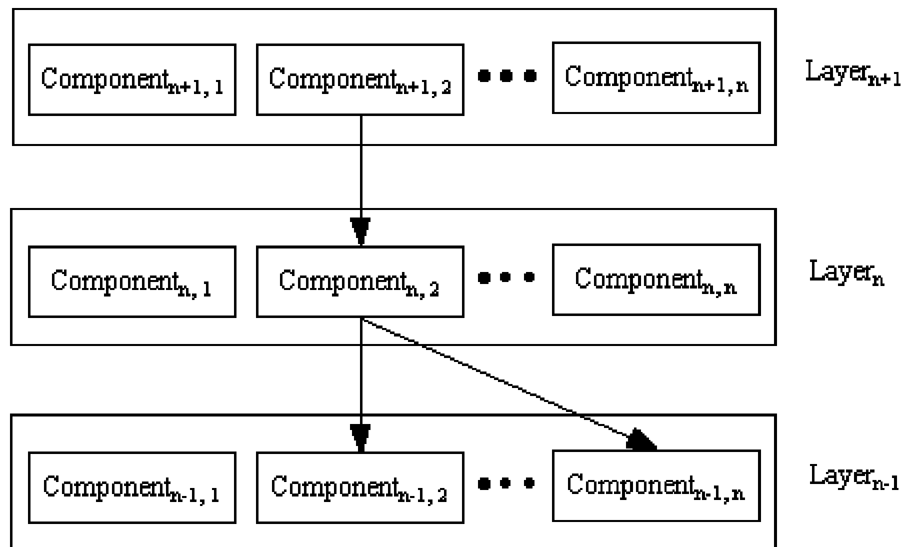


Figure 5.7 Layer pattern

Component Interaction Patterns

Eskelin (1999) describes five patterns, called Component Interaction Patterns: Abstract Interaction, Component Bus, Component Glue, Third-Party Binding and Consumer-Producer. The patterns deal with the problem of assembling components to communicate, collaborate and coordinate in a flexible and effective way.

- ***Abstract interaction*** is a pattern that reduces a component's dependence on its environment by defining interaction protocols between components separately from the components itself. The interactions are specified in terms of abstract interfaces and implemented as a separate component through which all the communication between the components is directed.

- The ***Component bus*** pattern manages the routing of information between participating components, from those that produce information to those that consume it. Each participant can dynamically attach and detach to and from the bus without effecting the integrity of others. When a component is attached to the bus, it registers interest in the information it requires. When a component places information onto the bus, it is delivered to the participants that registered interest for the information. All components communicate with the interface of the bus.

- ***Component glue*** is a design pattern that solves component incompatibilities by using a scripting language to create "glue" code that mediates or adapts between components. Glue code ties components together without increasing coupling and is an alternative to the adapter pattern. Glue code is not implemented as components, unless the same glue code is used in many parts of a system.

- ***Third-party binding*** makes connections among components explicit and separate, by having a third component bind two interacting components together.

- The ***Consumer-Producer*** pattern introduces a producer component that acts as a single, generic interface to different core system services like heterogeneous databases, naming and directory services. The consumers are the clients that use these services. A service provider interface is defined that allows programmers building service providers to conform to the producer and allow consumers to access their service.

6 Integration techniques

In this section, we will study various techniques for integrating software components. We will give a detailed presentation of the three dominating techniques in this area, and also a brief overview of some other techniques, which may get a significant portion of the market in the future.

6.1 Dominating techniques

There are three dominating techniques in the area of software component integration, namely Enterprise JavaBeans from Sun, COM from Microsoft and CORBA from the Object Management Group. In this section, we will present their architectures in a detailed level.

6.1.1 Enterprise JavaBeans

Enterprise JavaBeans defines a set of services and APIs for server-side Java components, promising hardware and operating system independence for component architecture. It is an extension of the JavaBeans-architecture, which will be described first in the following.

JavaBeans

The Java Development Kit (JDK) 1.1 released in February 1997, had a new API called *JavaBeans*. JavaBeans is the portable, platform-independent, component architecture for the Java application environment, which allows applications to be assembled from ready – made components, known as *beans*. It is intended for use in developing or assembling network-aware solutions for heterogeneous environment – within an enterprise or throughout the Internet. (Jia, 2000)

JavaBeans is an attempt to provide a standard for creating objects that have expected and documented behaviour. That is originally an initiative from Sun Microsystems but several companies such as Apple, Borland, IBM, Microsoft and Netscape have been involved in the specification. Software that adheres to the JavaBean specification will publish information about itself onto an Integrated Development Environment (IDE). IDEs usually consist of a compiler, debugger and a graphical development environment. Symantec Café and Java Workshop are examples of IDEs. The IDE can then present the developer with the object’s attributes as well as notify the developer of events that the object will respond to. (Schneider & Arora, 1997)

The JavaBeans initiative is a family of specifications as well as some basic classes that facilitate writing Java Components. A *bean* is a reusable software component that can be developed in a visual tool builder. Creation of beans is done with a package called *java.beans* and a Beans Developer Toolkit. (Schneider & Arora, 1997)

Software components hide implementation, conform to interfaces, and encapsulate data, just like classes do in object-oriented languages. So how do components differ from classes? The answer

is: almost all software components consist of classes. What makes them components is their conformance to a *software component specification*. The *JavaBeans Specification* is the document that describes what a Java class must do to be considered a "Bean." Programmers (and, as importantly, integrated development environments [IDEs]) can depend on any class that advertises itself as a Bean to conform to the rules set out in the specification. If it doesn't conform, the contract has been broken and the Bean is defective. (Johnson, 1997)

One of the requirements to make a class into a Bean is that the class implements the interface "java.io.Serializable". Serializable classes know how to package themselves into streams of bytes to be transmitted through networks or saved to disk, awaiting later reincarnation.

Interfaces and classes defined in the package "java.beans" allow a Beans developer to control how the Beans user (a programmer using a particular Bean) may set and get Beans' properties, hook Beans up to communicate with other components, and ask Beans to describe themselves. The component specification outlines how the component must implement these methods. (Johnson, 1997)

JavaBeans is not a product, program, or development environment. It is both a core Java package (`java.beans`) that Beans may use to provide extended functionality, and a document (the *JavaBeans Specification*) that describes how to use the classes and interfaces in the `java.beans` package to implement "Beans functionality." (Johnson, 1997)

JavaBeans turns classes into software components by providing several new features. Some of these features are specific to Beans. Others, like serialization, can apply to *any* class, Bean or otherwise, but are crucial to the understanding and use of Beans.

There are five features that are common for most JavaBeans:

- Introspection: enables a builder tool to analyse how a bean works.
- Customization: enables a developer to customise the appearance and behaviour of a bean.
- Events: enables beans to communicate and connect together.
- Properties: enables developers to program with beans
- Persistence: enables developers to customize beans and then retrieve those beans with customized features for future use.

Software components have *properties*, which are attributes of the classes. *Customization* is the process of configuring a Bean for a particular task. The new *event handling* scheme in Java 1.1 was created in part to ease communication between Beans. Beans may be dissected by IDEs or by other classes through a process called *introspection*. Beans may be *persisted* (i.e., *serialized*) into byte streams for transmission or storage, and persisted Beans may be *packaged* into "JAR files" to ease downloading and access. Finally, Beans have been designed to *interoperate* easily with legacy component technologies such as ActiveX and LiveConnect, and participate in transactions with Object Request Broker systems such as CORBA. (Johnson, 1997)

Properties and customization

Properties, as noted above, are attributes of a Bean. Visual properties might include colour or screen size. Other properties may have no visual representation: a BrowserHistory Bean, for example, might have a property specifying the maximum number of URLs to store. Beans expose ***setter*** and ***getter*** methods (called "accessor methods") for their properties, allowing other classes or IDEs to manipulate their state. The process of setting up a Bean's properties at design- or runtime is called ***customization***. (Johnson, 1997)

The value of an *indexed property* is an array. Indexed properties' accessor methods receive and return arrays of values instead of scalars. Accessor methods may throw checked exceptions to report error conditions.

Sometimes it's useful for an action to occur when a certain property of an object changes. *Bound properties* cause events to be sent to other objects when the property's value changes, possibly allowing the receiver to take some action. So, a SpreadSheet Bean might be configured to tell a PieChart Bean to redraw itself whenever the spreadsheet data changes.

Sometimes, certain values for properties are illegal, based on the state of other Beans. A Bean can be set up to "listen" to these *constrained properties* of other Beans, and "veto" changes it doesn't like.

When a developer is connecting Beans together to create an application, the IDE can present a property sheet containing all the Beans' properties and their current values. (A property sheet is a dialog box used to set and/or view properties, like what you get by selecting Options… on a menu.) The developer sets the properties graphically, which the IDE translates into calls to the Beans' setter methods, changing the Beans' state. This *customizes* the Beans for the particular application. The relevant classes for manipulating properties and customization are in the java.beans package.

Event handling

A pre-condition for all interactions between Beans is some way of communication. JDK 1.1 brought a new *event model* that classes use to communicate. In this event model, a class registers interest in the activities of another class by way of a *listener interface*. In effect, the *target* object (the interested party) tells the *source* object (the object of interest), "Let me know whenever so-and-so happens." When the so-and-so occurs, the source object "fires" an event at the target by invoking the target's event handler with a subclass of EventObject as the argument.
(Johnson, 1997)

Events can be used to implement bound and constrained properties. In the PieChart and SpreadSheet example above, the PieChart "registers" interest in any change to the SpreadSheet's (let's say) *DataList* property. When the SpreadSheet is going to change its *DataList* property, it passes a DataListChangedEvent (subclassed from EventObject), indicating what changed, to every interested listener's event handler method. The target (PieChart) then examines the event, and takes appropriate action. In the case with constrained properties, it works similarly; but in that case, the target *vetoes* the change by throwing an exception.

The EventObject class can be extended to create *user-defined events*. Classes can now define and use new event types to send messages to one another. This means that Beans running inside the same container may communicate by passing messages around. This helps uncouple dependencies between objects, which we know is an advantage. User-defined (and other) events are derived from the class java.util.EventObject. (Johnson, 1997)

Introspection

It is the name of the process of programmatically analyzing a class's public methods and members. This process is also sometimes called *discovery*. The new *reflection* mechanism in the Java core, which can dissect an object and return a description of its contents, makes introspection possible.

The IDE discovers a Bean's properties in one of two ways: by asking the Bean for a description of its properties, or by dissecting the Bean by introspecting it. A typical IDE will start by asking a Bean for a BeanInfo object, which describes the Bean's properties, among other things. The IDE will then use the BeanInfo object to construct a property sheet. (This is assuming the Bean doesn't provide a customizer of its own.) If the Bean doesn't know how to return a BeanInfo object, the IDE then introspects the Bean, and scans the list of methods for names beginning with *set* and *get*. It assumes (by convention) that these methods are accessors for properties, and creates a new property sheet based on the accessor methods that exist and the types of the arguments those methods take. So, if the IDE finds methods like setColor(Color), Color getColor(), setSize(Size), and Size getSize(), then it will create a property sheet with the properties *Color* and *Size*, and appropriately-typed widgets for setting them. This means that if a developer simply follows the conventions for naming accessor methods, an IDE can determine automatically how to create a customization property sheet for the component. (Johnson, 1997)

The reflection mechanism that performs introspection is in the package “java.lang.reflect".

Persistence and packaging

It's often useful to "freeze-dry" an object by converting its state into a blob of data to be packed away for later use -- or transmitted through a network for processing elsewhere. This process is called *serialization*.

One of the simplest uses for serialization is to save the state of a customized Bean, so that a newly-constructed Bean's properties can be correctly set at run time.
Also, serialization is a mainstay of component technology, making possible distributed-processing schemes such as CORBA.

Where should a group of freeze-dried Beans that have been "pickled" in this way be kept? Well, in a JAR, of course! The JavaBeans specification describes a *JAR* file as a structured ZIP file containing multiple serialized objects, documentation, images, class files, and so on, with a *manifest* that describes what's in the JAR. A JAR file, containing many compressed small files, can be downloaded all in one piece and decompressed on the client end, making applet downloading more efficient. (Johnson, 1997)

The java.io package provides object serialization. The JavaBeans Specification describes the format of JAR files.

Interoperation

There are a lot of standards in the area of component technology. For example there are many existing systems based on OLE (or its latest incarnation, ActiveX), OpenDoc, and LiveConnect. JavaBeans has been designed to (at least eventually) interoperate with these other component technologies.

It's not realistic to expect developers to abandon existing investments in other technologies and reimplement everything in Java. Since the release of Java 1.1, the first Beans/ActiveX "bridge" kits have become available, allowing developers to link Beans and ActiveX components seamlessly into the same application. The Java IDL interface, which allowed Java classes to operate with existing CORBA systems, came later. (Johnson, 1997)

While the Beans/ActiveX bridge and Java IDL are not part of the standard JavaBeans distribution, they round out JavaBeans' capabilities as an industrial-strength, open technology for portable component software.

Enterprise Beans

As a development of JavaBeans the first version of Enterprise JavaBeans was released in 1998. Enterprise JavaBeans extends JavaBeans by letting container-based objects communicate with each other via a network. Containers are software receptacles that know how to communicate with and manage beans and provide services such as persistence management, transactions, concurrency and security. (Jia, 2000) Enterprise JavaBeans specification defines the Enterprise JavaBeans component architecture and the interfaces between the Enterprise JavaBeans technology enabled server and the component.

An enterprise bean is a server-side software component that can be deployed in a distributed multi-tier environment. An enterprise bean can comprise one or more Java objects because a component may be more than just a simple object. Regardless of an enterprise bean’s composition, the clients of the bean deal with a single exposed component interface. This interface, as well as the enterprise bean itself, must conform to the Enterprise JavaBeans specification. The specification requires that the beans expose a few required methods, these methods allow the EJB container to manage beans uniformly, regardless of which container the bean is running in. (Roman, 1999)

The client of an enterprise bean could be anything—perhaps a servlet, an applet, or even another enterprise bean. In the latter case, a client request to a bean can result in a whole chain of beans being called. This is a very powerful idea because a complex bean task can be subdivided, allowing one bean to call on a variety of prewritten beans to handle the subtasks. This hierarchical concept is quite extensible.

Enterprise beans are very similar to two other types of Java components: applets and servlets:

- Applets are portable Java programs that can be downloaded from a Web server into a Web browser and they typically display user interfaces to the end-users. Applets can be deployed in a Web page, where the browser's applet viewer provides a runtime container for the applets.

- Servlets are networked components that can be used to extend the functionality of a Web server. Servlets are request/response oriented, in that they take requests from some client host and issue a response back to that host. Servlets can be deployed in a Web server, where the Web server's servlet engine provides a runtime container for the servlets.

- Enterprise beans are deployed in an application server, where the application server provides a runtime container for the Enterprise JavaBeans.

(Roman, 1999)

The real difference between applets, servlets, and enterprise beans is the domain of which each component type is intended to be a part. Both applets and servlets are well suited to handle client-side operations, such as rendering graphical user interfaces, performing other presentaion-related logic, and lightweight business logic operations. The client side could be a Web browser (in the case of applets), or a Web server (in the case of servlets). In both cases, the components are dealing directly with the end-user. (Roman, 1999)

Enterprise beans are server-side components and they are meant to perform server-side operations. Server-side components need to run in a highly available, faulttolerant, transactional, and multi user secure environment. An application server provides this high-end server-side environment for the enterprise beans, and it provides the runtime containment necessary to manage enterprise beans.

IBM, Sun / Netscape and Oracle provide commercial products that implements the Enterprise JavaBeans specification. The key features of the Enterprise JavaBeans technology are:

- Enterprise JavaBeans components are written entirely in Java programming language.

- Enterprise JavaBeans components contain business logic only, no system level programming.

- The system level services such as security, threading, persistence are automatically managed by the Enterprise JavaBeans server.

- Enterprise JavaBeans component are fully portable across any Enterprise JavaBeans server and any Operating System.

(Sun Microsystems, 2000a)

EJB is intended to support distributed, Java-based, enterprise-level applications, such as business information management systems. Among other things, it prescribes an architecture that defines a standard, vendor-neutral interface to information services including transactions, persistence, and security. It thereby permits application writers to develop component-based implementations of business processing software that are portable across different implementations of those underlying services. (Sousa & Garlan, 1999)

One of the most important and prevalent classes of software systems are those that support business information applications, such as accounting systems and inventory tracking systems. Today these systems are usually structured as multi-tiered client-server systems, in which business-processing software provides services to client programs, and in turn relies on lower level information management services, such as for transactions, persistence, and security. (See Figure 6.1.)

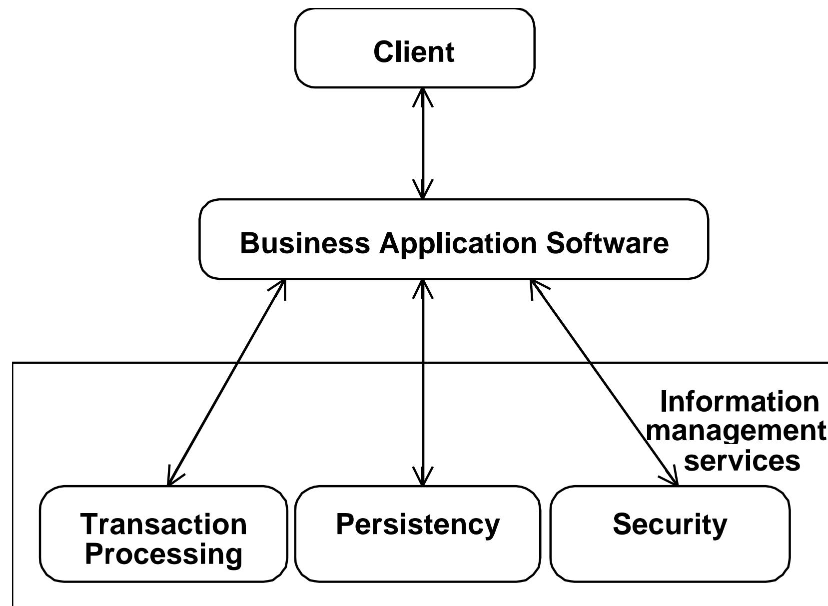


Figure 6.1 A three-tiered business application (Sousa & Garlan, 1999)

Currently one of the problems with writing such applications software is portability: application software must be partially rewritten for each vendor's support facilities because information management services provided by different vendors often have radically different interfaces.

Additionally, clients of application software are faced with a huge variety of interfaces to those applications. While some differences are inevitable, given that different applications must provide different capabilities, one would wish for certain levels of standardization for generic operations such as creating or deleting business entities (such as accounts). To address this problem several vendors have proposed component integration frameworks for this class of system. One of these is Sun Microsystems'Enterprise JavaBeans framework
(Sousa & Garlan, 1999)

Sun's “Specification of the Enterprise JavaBeans Architecture”, defines a standard for third parties to develop Enterprise JavaBeans deployment environments. Understanding Enterprise JavaBeans architecture requires comprehension of four entities: servers, containers, components and clients.

- The EJB server provides a runtime environment for one or more containers. EJB servers manage low-level system resources, allocating resources to containers as they are needed. Enterprise JavaBeans servers provide fundamental services like legacy systems for

containers and the components they contain.

- The EJB container provides a playground where your enterprise beans can run. There can be many beans running within a container. Bean containers are responsible for managing the beans running within them. They interact with the beans by calling a few required methods that the bean must expose. Containers may also provide access to a legacy system. Enterprise JavaBeans containers are the middleman between clients and components at the method call level.

- They handle features like life cycle, transaction and security management for the components. Containers make it possible to manage transactions over different servers and even different machines.

- Enterprise JavaBeans components contain the actual logic. By implementing some interfaces the developer can use container provided services. Several components usually are deployed in the same container.

(Roman, 2000)

The client program is presented with the same interface for all containers. This makes it simple for the programmer to use different containers on different servers and on different systems.

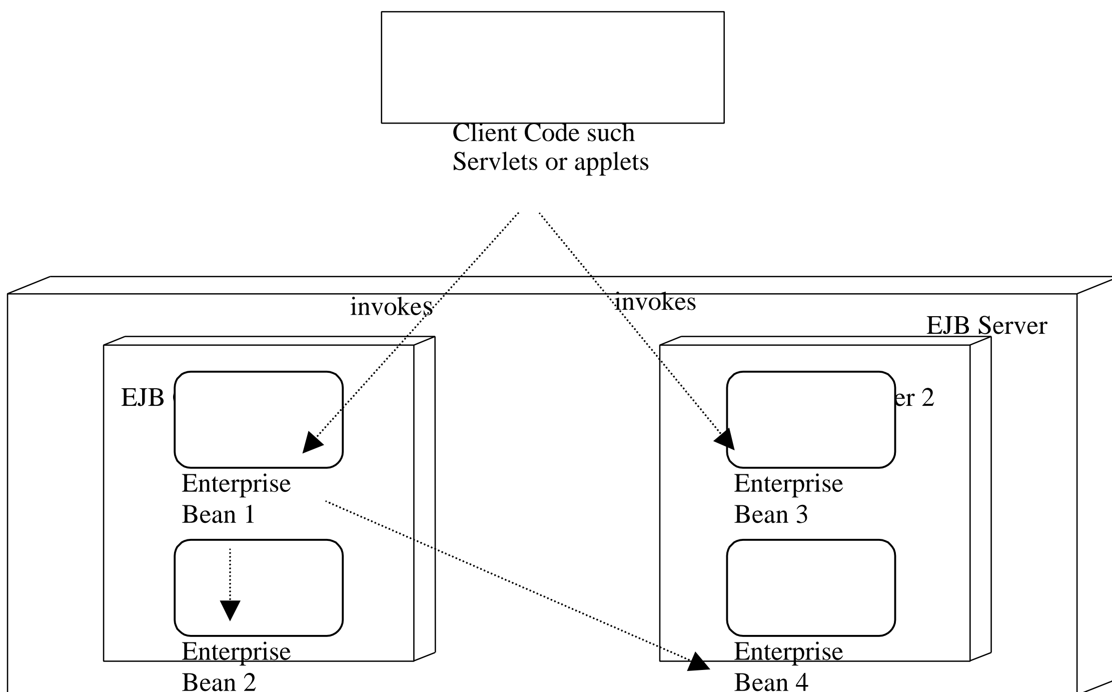


Figure 6.2 The relationship between EJB servers and EJB containers (Roman, 1999)

An application running in one of these environments would access information management services by requesting them of the EJB server, via the EJB API, in the way prescribed by the EJB spec.

Figure 6.3 illustrates a system with a remote client calling an application that implements some business logic, for which Orders and Accounts are relevant operational entities. In the object-oriented paradigm, such entities are termed objects. An object can be viewed as a unit that holds a cohesive piece of information and that defines a collection of operations (implemented by methods) to manipulate it. (Roman, 1999)

According to EJB framework Enterprise Beans must conform to specific rules concerning the methods to create or remove a particular bean, or to query a population of beans for the satisfaction of some property. Hence, whenever a piece of client software needs to access a bean, it can take some features for granted. (Sousa & Garlan, 1999)

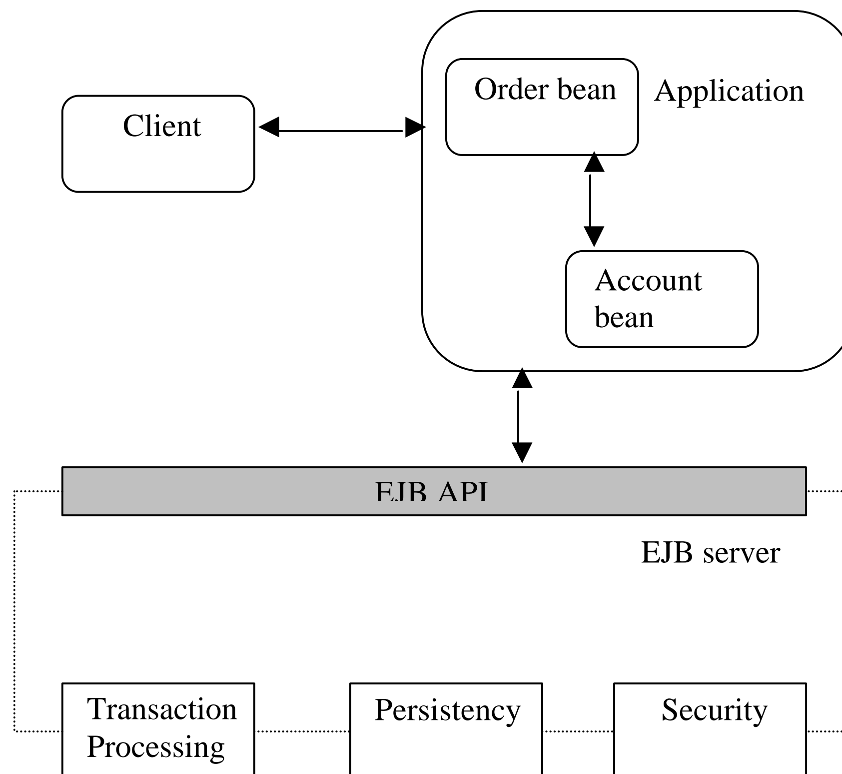


Figure 6.3 The EJB server offering access to information management services (Sousa & Garlan, 1999)

It is the job of an EJB server provider to map the functionality that the EJB spec describes into available products and technologies. In version 1.0, released in March 1998, the EJB spec covers the request of transaction management, persistence and security services. The EJB spec does not regulate how these services are to be implemented, however: they may be implemented by the

EJB server provider, as part of the server, or they may rely on external products, eventually supplied by other vendors. Such products, however, are invisible to the beans.
(Sousa & Garlan, 1999)

The EJB spec refers to the collection of services that both the beans and the client software use as a container (See figure 6.4). A container provides a deployment environment that wraps the beans during their lifecycle. Each bean lives within a container. The container supports (directly or indirectly) all aspects that the bean assumes about the outside world, as defined in the EJB spec. The protocols that regulate the dialog between a bean and its container are termed the *bean contract*. (Sousa & Garlan, 1999)

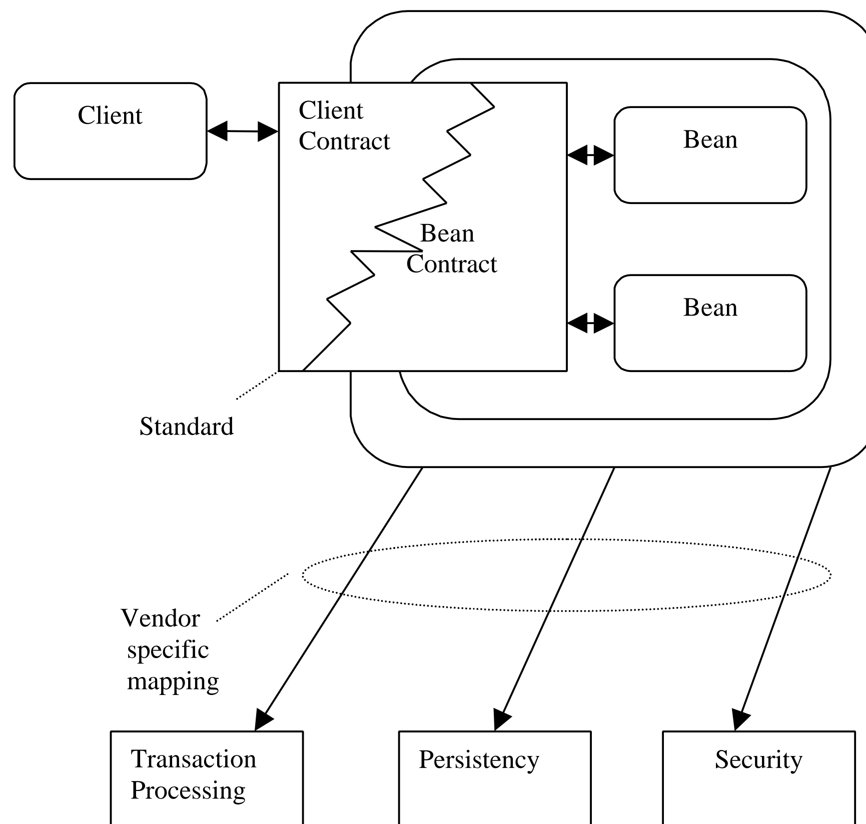


Figure 6.4 The EJB container (Sousa & Garlan, 1999)

The container also supports a set of protocols, termed the client contract that regulates the dialog between client software and a bean. The client contract defines two interfaces that a client uses to communicate with a specific bean.

Both interfaces are created at deployment-time by special-purpose tools supplied by the EJB server provider.

- The Remote Interface, which reflects the functionality of the bean it represents, as it publishes the so-called business methods of the bean. Each bean has one such interface.

- The Home Interface, which contains the methods for creation and removal of beans, as well as optional methods for querying the population of beans (finder methods). There is one such interface per bean class. (Sousa & Garlan, 1999)

Before we discuss further the matter of different types of interfaces, we should look at some fundamental concepts such as enterprise bean class, EJB object, and home object.

Enterprise Bean Class

An enterprise bean component is not a single monolithic file. A number of files work together to make up an enterprise bean. An enterprise bean is a complete component, which is made up of at least two interfaces and a bean implementation class. In other word, you provide your enterprise bean component implementation in an *enterprise bean class*. This is simply a Java class that conforms to well-defined interfaces, obeys certain rules, and contains implementation details of your component. A session bean implementation is very different from an entity bean implementation. For session beans, an enterprise bean class typically contains business-process-related logic, for example logic to compute prices and transfer funds between bank accounts. For entity beans, it contains data-related logic, for example logic to change a customer’s name and reduce the balance of a bank account. The EJB specification defines a number of standard interfaces that bean classes can implement. These interfaces force bean classes to expose certain methods, which will be called by containers to manage the beans and alert them to significant events. The most basic interface is the *javax.ejb.EnterpriseBean* interface. This is a marker interface, which indicates that the class is indeed an enterprise bean class. It extends *java.io.Serializable*, which means that the class can be converted to a bit-blob and shares all properties of serializable objects. However the enterprise bean class never needs to implement the *javax.ejb.EnterpriseBean* interface directly. It implements the interface corresponding to its bean type, which is *javax.ejb.SessionBean* or *javax.ejb.EntityBean*. (Roman, 1999)

EJB Object

When a client wants to use an instance of an enterprise bean class, the client never invokes the method directly on an actual bean instance. The invocation is intercepted by the EJB container and then delegated to the bean instance. There are two important reasons for this. First of all, the enterprise bean class cannot be called across the network directly because this is not network-enabled. The EJB container handles networking by wrapping the bean in a network-enabled object. This object receives calls from clients and delegates these calls to instances of the bean class. For the second, by intercepting requests, the EJB container can automatically perform some necessary management, such as transaction logic, security logic, and bean instance pooling logic. Thus, the EJB container is acting as a layer of indirection between the client code and the bean, which manifests itself as a single network-aware object, called *EJB object*. An EJB object acts as glue between client and the bean and expose every business methods that the bean itself exposes. (Roman, 1999)

Home Object

But, how do clients acquire references to EJB objects? It is the fact that the client cannot instantiate an EJB object directly because EJB objects and the client could exist on different machines and because of the location transparency the client should never be aware of where EJB objects reside. For acquiring a reference to an EJB object, the client code asks for an such object from an EJB object factory, which is responsible for instantiating and destroying EJB objects. The EJB specification calls such a factory a *home object*.

The Remote Interface

As we saw above, the clients invoke methods on EJB objects instead of the beans themselves. For doing this, EJB objects must clone every business method that the beans expose. The question is, how auto-generate EJB objects know which method to clone. The answer is the special interface that a bean provider writes, which duplicates all the business logic methods that the corresponding bean class exposes. This interface calls *remote interface*.

All remote interfaces must follow special rules, which are defined by EJB specification. All of them must derive from a common interface, which is called *javax.ejb.EJBObject*. It lists a number of methods such as, getEJBHome(), getPrimaryKey(), remove(), getHandle(), isIdentical(). All these methods are required that all EJB objects must implement. The implementation performs by the EJB container when it auto-generates the EJB objects. In addition to these methods, the remote interface duplicates enterprise beans' business methods too. When a client invokes any of these methods, the EJB object will delegate the method to its corresponding implementation, which resides in the bean itself. (See figure 6.5) (Roman, 1999)

The Home Interface

Home objects are factories for EJB objects. But the question is, how does a home object know how the EJB object should be initialized? One EJB object might expose an initialization method that takes a string as a parameter, while another EJB object might take an integer. The container needs to know this information to generate home objects. This provides to the container by specifying a *home interface*. It defines methods for creating, destroying, and finding EJB objects. (Roman, 1999)

EJB specification defines some required methods, which all home interfaces must support. These are defined in the *javax.ejb.EJBHome* which the bean's home interface must extend. The *javax.ejb.EJBHome* derives from java.rmi.remote, which means that home objects are fully networked Java RMI remote objects and can be called across the network. (See figure 6.6)

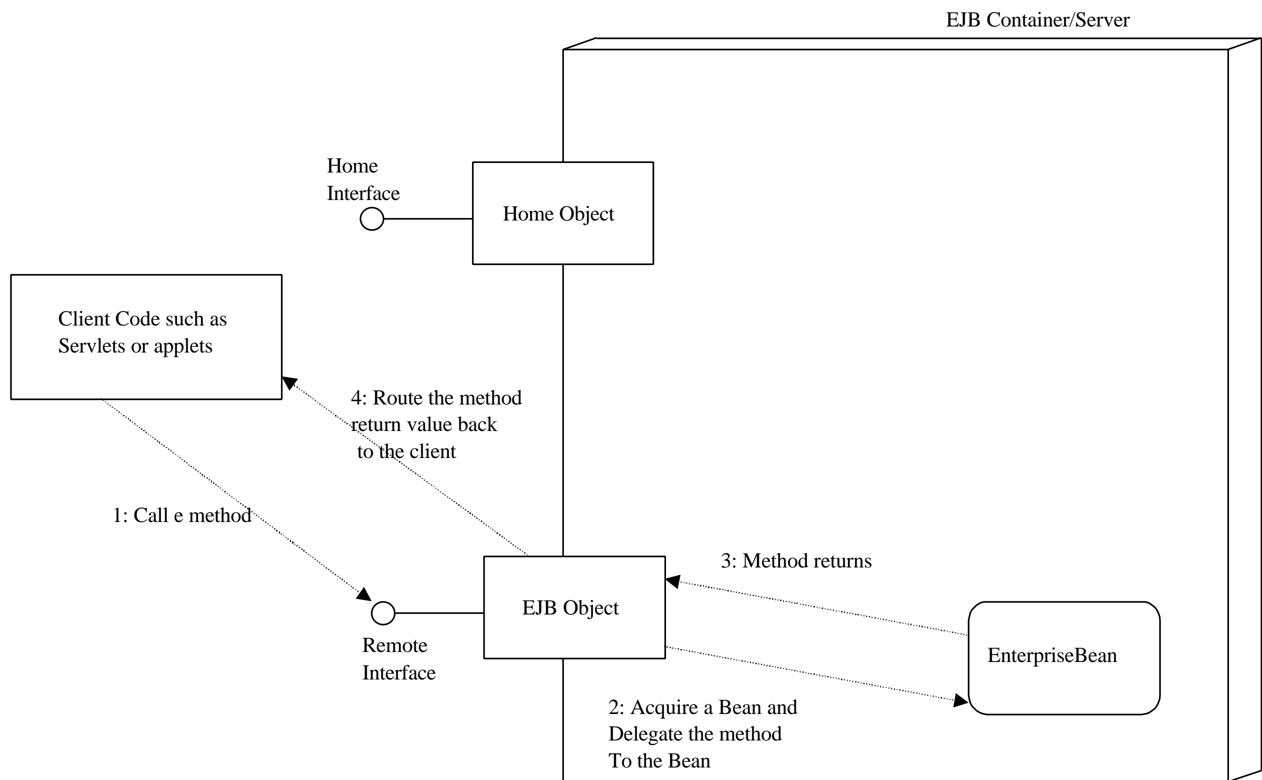


Figure 6.5 EJB Objects and Remote Interface (Roman, 1999)

The home and remote interface is the clients' gate to the server. These interfaces are provided by classes constructed by the container when a bean is deployed, based on information provided by the bean. The home interface provides methods for creating a bean instance, while the remote interface provides the business logic methods for the component. By implementing these interfaces, the container can be a middleman between the client and the beans. (Roman, 1999)

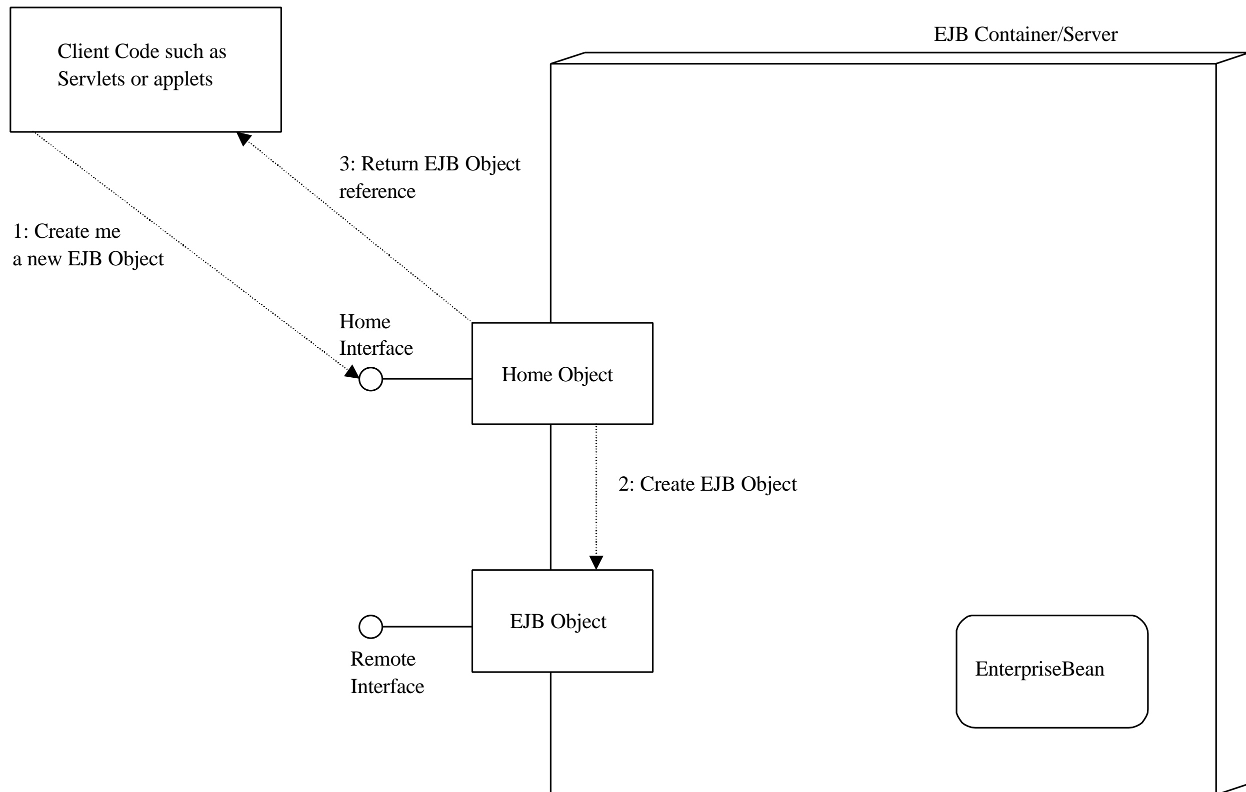


Figure 6.6 Home Objects and Home Interface (Roman, 1999)

Session & Entity beans

EJB supports the two object persistence modes - transient or persistent. EJB defines them as session and entity beans respectively:

- Session beans that provide business logic, are used in the algorithms used by the client programmer. They deal with the logic in the run time processes. For example, a session bean could perform price quoting, order entry, video compression, banking transactions, and more. Session beans are called session beans because they live for about as long as the session (or lifetime) of the client code that's calling the session bean. For example, if client code contacted a session bean to perform order entry logic, the application sever is responsible for creating an instance of that session bean component. When the client later disconnects, the application server may destroy the session bean instance. Session beans can be further defined as stateless or stateful.

    - Some business processes naturally lend themselves to a single request paradigm. A single request business process is one that does not require state to be maintained across method invocations. *Stateless session beans* are components that can accommodate these types of single request business

processes. With stateless session beans, each method call can be handled by any instance of the EJB. An example of a stateless session bean is a credit card verification component. The verifier bean would take a credit card number, an an expiration date, a cardholder's name, and a dollar amount as input. The verified would then return a yes or no answer. Once the bean completes this task, it is available to service a different client and retains no past knowledge from the original client.

- Some business processes are more drawn out and can last across multiple method requests and transactions. A *stateful session bean* is designed to service such processes. To accomplish this, stateful session beans retain state on behalf of an individual client. If a stateful session bean's state is changed during a method invocation, that same state will be available to that same client upon the following invocation. With stateful session beans, all method calls made by a given user are handled by a unique instance of the EJB for that user. An example of a business process that lasts for multiple method calls is an E-commerce Web store. As the user peruses an online E-commerce Web site, he/she can add products to the online shopping cart. This implies a business processthat spans multiple method requests. So the component (the related stateful session bean) must track the user's state from request to request.

- Entity beans that provide business data, represents specific data or collects data from a database. Entity beans are defined to have a persistent primary key, and the bean is stored in a persistent data store. To be able to perform series of calculations on a database an entity bean survives as long as data remains in the database. Entity beans can be further defined as using Container Managed Persistence (CMP), or Bean Managed Persistence (BMP). With CMP, the application server is responsible for reading data from and writing updates to the underlying data store, such as DB/2, Oracle, SQL Server, etc. With BMP, you have to write the code to map the data in the bean to the underlying data store. (Roman, 1999)

When accessing stateless session beans, each user has exclusive access to a particular stateless session bean for a single method call. The container can route subsequent method calls to other stateless session beans. However, stateful session beans require a unique instance per user, which comes with significant performance and memory overhead. Instead of using stateful session beans, you can use fewer stateless session beans to service a larger user community with better response times. You can use an alternative state-handling mechanism, such as HttpSessions, to provide state management when using stateless session beans. (Higham, 2001)

Session beans are meant for temporary copies of data, optionally stored in a persistent data store. Session beans are required for EJB version 1.0. The session bean does actions on behalf of a client request, often accessing persistent data. The session bean is not shared between client requests. The session bean is responsible for storing it's state to a data store. Session beans are optionally transactional. (Orchard, 1998)

Entity beans are defined to be objects reflecting some state in a persistent store. Entity beans are optional in EJB 1.0, but mandatory for EJB 2.0. The entity bean state exists outside the lifetime of client requests. The entity bean state is shared between client requests. The entity bean state can be managed by the entity bean, or it can be delegated to the container. Entity beans are transactional. (Orchard, 1998)

The choice of session or entity beans is made by the EJB developer by inheriting from either EntityBean or SessionBean. The developer then optionally implements various functions. For example, a stateful session bean that has open files must close the files when the bean is being stored to a persistent store. The bean developer provides an implementation of the ejbPassivate method to do any special cleanup before passivation.

Often sessions are equated with stateless business objects, and entities are equated with stateful business objects. That comparison is not completely valid. Stateful and stateless middle-tiers both acknowledge that the data in the middle-tier is representative of data in the persistent storage. The key difference between the stateful and stateless is whether the state is shared between clients or not. A stateless middle-tier will create a copy of the state for every client request, whereas a stateful middle-tier will use only one copy of the state. A stateless middle-tier could easily consist of entity beans that are activated for each client, then passivated immediately after the client request is serviced. (Orchard, 1998)

The debate between stateful or stateless middle-tiers is widespread. In general, database and integration product vendors are supporting stateful middle-tiers, examples being IBM and Oracle. Stateless middle-tiers are advocated by many in the web community, prominent gurus such as Roger Sessions, and by Microsoft.

As usual when there are two entrenched sides, they are both right, as it depends on the type of application. The choice of stateful or stateless objects is one of the largest decisions that system architects will make in application development. The important factors in the state decision are:

- the type of clients,
- whether the connection to the data store supports transactions,
- whether the application has sole access to the data store,
- and the performance and scalability requirements. (Orchard, 1998)

The developer must consider the use of EJB on a case-by-case basis. It's a matter of picking the right tool for the job, and he/she can often use simple JavaBeans instead of EJB. When the developer are deciding whether to use a JavaBean or EJB, he/she should focus on the word "enterprise." If the business logic or data being considered as an EJB will be used by multiple applications within the enterprise, it's a good EJB candidate. For scenarios in which the developer is coordinating multiple systems or applications (distributed transactions), using session beans makes sense because of their built-in transaction management. If the logic or data will only be used in one application, and won't manage distributed transactions, using EJB becomes questionable. (Higham, 2001)

One of the major value propositions of entity beans lies in Container Managed Persistence (CMP), where the application server is responsible for reading data from and writing data to an underlying data store such as DB/2, Oracle, SQL Server, etc. However, the EJB 1.0 and 1.1 specifications require the entity bean to map to a single row in a table. Although there are instances where this might work, if the data the entity bean maps to is contained in multiple rows or multiple tables (requiring a join), you must use Bean Managed Persistence (BMP), which you

have to write yourself. The single-row limitation with CMP will be removed with EJB 2.0, due for release in early 2001.

It is recommend to access entity beans from a stateless session bean. Because the session bean and entity bean both reside on the server, the number of remote calls is drastically reduced. Tests have shown a 40 to 50 percent performance improvement using session beans to access entity beans. (Higham, 2001) Stateless objects beans are often appropriate for Web applications, applications that do not have sole access to the data store, or applications where the bean state must be shared across CPUs - possibly for load-balancing or fault-tolerance. (Orchard, 1998)

Passivation/Activation

An EJB server manages the population of beans that reside in main memory in a way that is transparent to the client software. As the population of beans inside a container grows beyond a certain limit, the container sends some number of the least recently used beans to secondary memory. The EJB spec refers to the beans that are subject to this operation as passivated. Since every call to a bean flows through the interfaces in the container, it is the container that relays the call to the bean, as appropriate. So, whenever a method call is addressed to a passivated bean, the bean is brought back to primary memory by the container. The EJB spec refers to beans that are subject to this latter operation as activated. (Sousa & Garlan, 1999)

Although the passivation/activation matter is transparent to the client calling the bean, it is not so to the bean itself. Before being passivated, the bean is required to release any assigned resources, so as not to lock them during the passivation time. Likewise, upon activation, the bean may have to reacquire the resources in order to serve the client's request. Therefore, in order to allow the bean to perform these actions, the container issues synchronization messages
to the bean just before passivation and right after activation (before the client's call is relayed).

An Example

Here follows a code example of an entity bean called Customer. (jGuru, 1999)

The Home Interface:

```
import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import java.rmi.RemoteException;

public interface CustomerHome
              extends EJBHome {

   public Customer create(Integer
               customerNumber)
               throws RemoteException,
                     CreateException;
```

```
    public Customer findByPrimaryKey(Integer
                        customerNumber)
                throws RemoteException,
                        FinderException;

    public Enumeration findByZipCode(int zipCode)
                  throws RemoteException,
                         FinderException;
}
```

The Remote Interface:

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Customer extends EJBObject {

    public Name getName()
            throws RemoteException;
    public void setName(Name name)
         throws RemoteException;
    public Address getAddress()
           throws RemoteException;
    public void setAddress(Address address)
                 throws RemoteException;

}
```

Customer Bean Class:

```
import javax.ejb.EntityBean;

public class CustomerBean
            implements EntityBean {

    Address    myAddress;
    Name       myName;
    CreditCard myCreditCard;

    public Name getName() {
        return myName;
    }

    public void setName(Name name) {
        myName = name;
    }

    public Address getAddress() {
```

```
        return myAddress;
    }

    public void setAddress(Address address) {
        myAddress = address;
    }

    ...
}
```

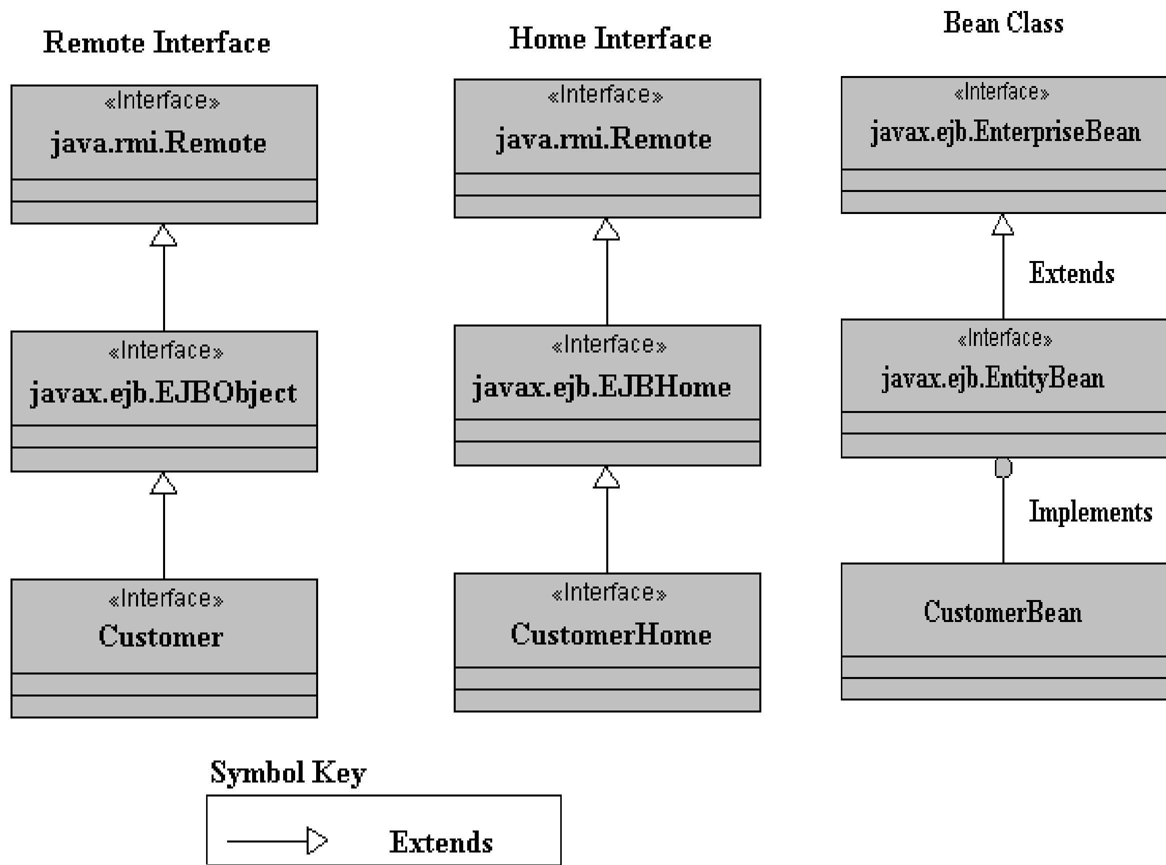


Figure 6.7 Class Diagram of Remote Interface, Home Interface and Bean Class (jGuru, 1999)

Java Naming and Directory Interface (JNDI)

The Java Naming and Directory Interface is a standard for *naming and directory services*. A naming service is an entity that performs the following tasks:

- It associates name with objects. This calls *binding* names to objects. This is very similar to a telephone company's associating a person's name with a specific residence's telephone number.

- It provides a facility to find an object based on a name. This calls looking up or searching for an object. This is similar to a telephone operator finding a person's telephone number based on that person's name and connecting the two people together. (Roman, 1999)

Naming services are everywhere in computing. When you want to locate a machine on the network, the *Domain Name System* is used to translate a machine name to an IP address or when you want to access a file on your hard disk, a File System Naming Service provides the

functionality for translating the file's name to an actual file of data. In general, a naming service can be used to find any kind of generic object. But one type of object is of particular importance: a *directory object*. A directory object is different from a generic object because attributes can be stored with directory objects. These attributes can be used to store information for a wide variety of purposes. For example if a directory object represent a user in a company. The information about this user like password and username can be stored as attributes in the directory object. (Roman, 1999)

A *directory service* is a naming service that has been extended and enhanced to provide directory object operations for manipulating attributes. A directory is a system of directory objects, all connected. Some examples of directory products are Netscape Directory Server and Microsoft's Active Directory. There are many types of directory services, as well as protocols to access them, which are all competing standards for example Internet standard LDAP and Novell's NDS.

Enterprise JavaBeans relies on JNDI for looking up distributed components across the network. JNDI is a key technology required for client code to connect to an EJB component. This is a system for Java-based clients to interact with naming and directory systems. JNDI is a bridge over different types of naming and directory services. It provides one common interface to disparate directories. Users who need to access an LDAP directory use the same API as users who want to access an NIS or Novell's directory. All directory operations are done through the JNDI interface, providing a common framework. The JNDI insulates the application from protocol and implementation details, which allows code to be portable between directory services. (see figure 6.8) (Roman, 1999)

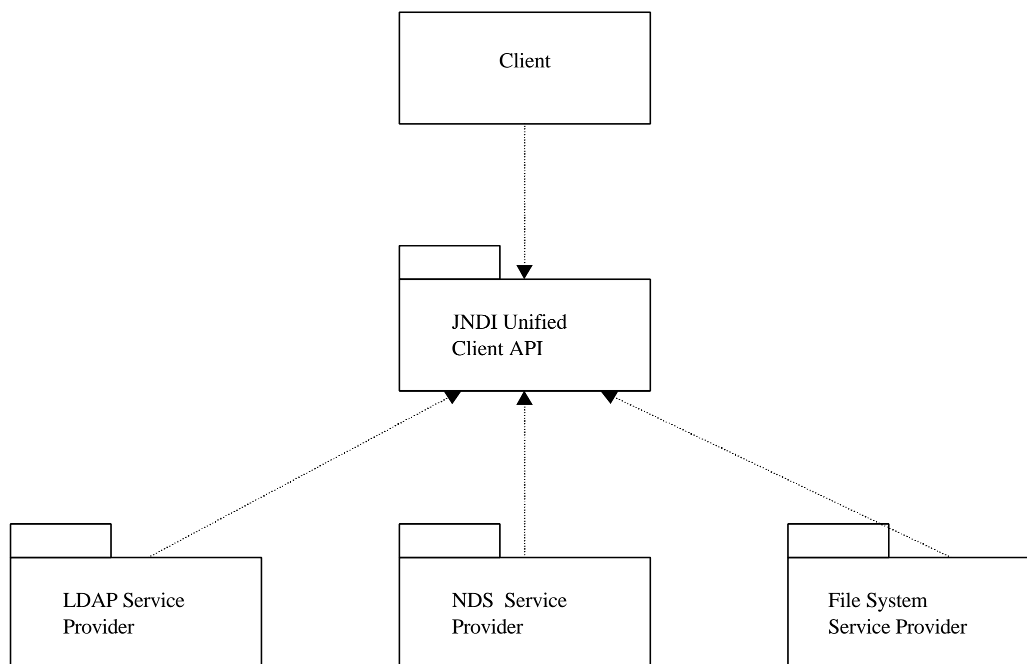


Figure 6.8 The Java Naming and Directory Interface (Roman, 1999)

To use the services of a bean a client first obtains a reference to the bean's class Home Interface using the Java Naming and Directory Interface (JNDI). Using this reference, the client software can call a create method in the class's Home Interface, thus obtaining a reference to the bean's Remote Interface implemented by the container. The Remote Interface then delegates subsequent method calls to the corresponding bean. The fact that the client uses JNDI, to obtain a reference to the Home Interface of the class, is a necessary condition for distribution transparency. Any piece of software, including a bean, may use the client contract to communicate with some bean, if the software does not know (or care) where the target bean is actually being deployed. Such software calls the interfaces in the container holding the target bean using Remote Method Invocation. (Sousa & Garlan, 1999)

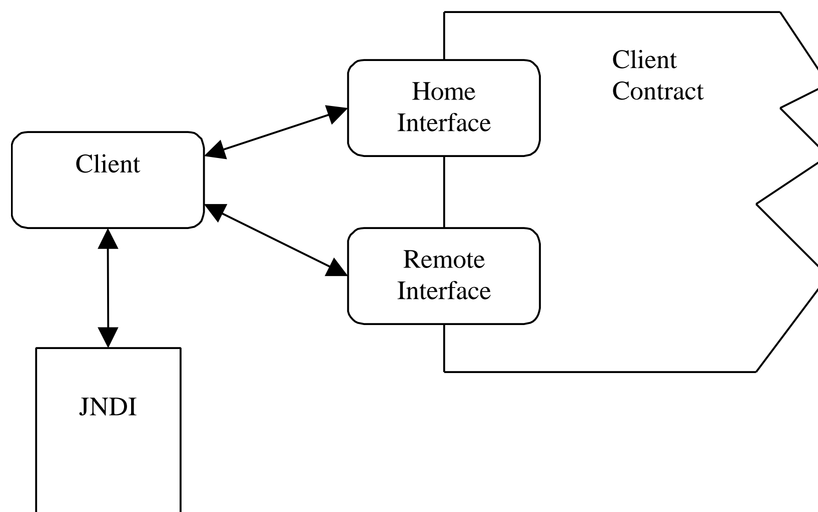


Figure 6.9 Detail of the Client Contract (Sousa & Garalan, 1999)

Other services in EJB

Enterprise Java Beans are named using the Java Naming and Directory (JNDI) API. The naming hierarchy has the EJB server at the top level, layered on the application name, then the container name. The bean is identified by a primary key within the container. There are no naming convention requirements for the primary keys. In addition, CORBA Naming standard (COS) is supported. An EJB Server that supports COS will automatically publish a COS version of the EJB name. (Orchard, 1998)

The Container provides the life-cycle services of an object. All containers have a Home interface that provides Factory methods for creation of objects. Entity beans also have a set of Finder methods for retrieving objects based upon a primary key. Home interfaces have destroy methods for deleting objects.

EJB provides an infrastructure for distributed transactions using two-phase commit protocols. It is based upon the OMG Transaction Service (OTS), specifically the Java Transaction

Service(JTS) implementation of OTS. JTS supports multiple transaction managers propagating transactions to multiple databases. One of the exciting aspects of EJB is that it brings the strength of database transactions to objects. Distributed objects can participate in the begin/prepare/commit sequence typical of transaction protocols. (Orchard, 1998)

The transaction requirements that a bean supports are specified in the transaction attribute associated with the bean. The options are BEAN_MANAGED, NOT_SUPPORTED, SUPPORTS, REQUIRES, REQUIRES_NEW and MANDATORY. These options correspond to the level of transaction support that a bean can operate in or that it requires from the container. Readers unfamiliar with transactions and two-phase commit protocols are encouraged to investigate transaction theory as it can significantly improve the robustness and integrity of data and programs. (Orchard, 1998)

EJB can use CORBA and RMI for messaging. The beans are defined using the RMI interfaces, then the standard RMI deployment is used. EJB Servers can also support CORBA IIOP integration. The EJB server provides a CORBA ORB and tools for creating a CORBA object from the EJB object. It is also possible to support DCOM and COM this way.

The run-time information of the Bean is specified in a deployment descriptor field in an ejb-jar file. The ejb-jar is installed in the EJB Server. It then deploys the bean according to the descriptor fields, such as transaction characteristics. This is a natural extension of the packaging of the Java Beans specification.

To set up an EJB server

Setting up an Enterprise JavaBeans server requires six steps on the server:

1. Enterprise JavaBeans server vendors supply EJB components run on EJB containers. The first step is therefore to download an EJB server. A server that is recommended by SUN is BEA Weblogic EJB server.

2. Construct a remote interface on the EJB server. The remote interface is the client' s view of the EJB. The interface is developed using JAVA RMI syntax and the EJB container provider generates the code for the interface.

3. Construct a home interface by which the container creates new session beans on behalf of the client. Just like for the remote interface the home interface is declared using RMI syntax and implemented by the container service provider tools.

4. Write an Enterprise JavaBean. Here is all the logic constructed.

5. The next step is to compile the three files above and package them into a so-called Enterprise JavaBeans -jar file. This is the file the developer will place in the container.

6. Now, place the Enterprise JavaBeans-jar in the container. The way this is done depends on which server vendor that is used.

After the server side is finished constructing containers and filling them with components the client needs to find a specific EJB. The client accesses the Enterprise JavaBeans by looking up the class that contains the EJB by name using Java Naming and Directory Interface (JNDI). JNDI is an API that provides naming and directory functionality for applications written in Java. The function the client programmer has to use is simply called lookup ("EJB_name") where EJB_name is the name of the Enterprise JavaBean.

The developers of Enterprise JavaBeans focus on making EJB work for business to business integration and their main approach is to make EJB work with different integration techniques. Lots of research is put into making EJB and CORBA work together where CORBA handles the transaction and EJB the business logic. CORBA would then take the part that RMI has today with the biggest advantage that CORBA works with all programming languages. For some implementations a co-operation between EJB and XML can be suitable. These two technologies do not have the same purpose but are complementary. Enterprise JavaBeans handles portable business logic and XML technology defines a standard for portable data.

The Enterprise Java Beans specification provides a great first step in defining how Java developers can build and re-use server logic. EJB is a natural extension of "Write Once, Run Anywhere" to middleware and protocols by providing a robust and well-defined framework for developers and vendors. Developers will be able to focus more on business logic, and less on plumbing, while gaining independence from middleware. (Orchard, 1998)

6.1.2 Component Object Model

Building applications from classes and their objects is relatively a new popular and modern method. One problem is that every programming language has its own notion of what an object is. There are many programming languages (such as C++, Java, Delphi, and even Visual Basic) that support objects in some way, but the differences among them can cause confusion for developers. Those differences also create difficulties for vendors that want to provide object services in a standard way to their customers working in any language.

One way to solve this problem is to define a single object model that can be used across all languages. Once that object model exists, services that the operating system (or applications running on that operating system) provides can all be exposed in a common way, regardless of the language they're written in. This is exactly what Microsoft's Component Object Model (COM) does.

COM is actually the foundation technology for Microsoft's OLE and Active X. COM has its roots in OLE version 1, which was created in 1991 and was a proprietary document integration and management framework for the Microsoft Office suite. Microsoft later realized that document integration is just a special case of component integration.

OLE version 2, released in 1995 was a major enhancement over its predecessor. The foundation of OLE version 2, now called COM, provided a general-purpose mechanism for component integration on Windows platforms. (North, 1997)

Microsoft Component Object Model (COM) refers to both a specification and implementation developed by Microsoft Corporation, which provides a framework for integrating components. This framework supports interoperability and reusability of distributed objects by allowing developers to build systems by assembling reusable components from different vendors, which communicate via COM.

COM defines an application-programming interface (API) to allow for the creation of components for use in integrating custom applications or to allow diverse components to interact. However, in order to interact, components must adhere to a binary structure specified by Microsoft. As long as components adhere to this binary structure, components written in different languages can interoperate. (Microsoft, 1999)

COM can even be described as a language-independent, distributed, object-oriented system for creating binary software components that can interact.

It is important to indicate that COM is not an object-oriented language but a standard. COM defines a language-independent notion of what an object is, how to create objects, how to invoke methods, and so on. This allows development of components that programmers can use (and reuse) in a consistent way, of which languages they use to write the component and its client. COM does not specify how an application should be structured. The language, structure and implementation details are left to programmer. COM specifies an object model and programming requirements that enable COM objects (COM components) to interact with other objects. COM also defines how objects work together over a distributed environment and has added security features to ensure system and component integrity. (Microsoft, 1999)

It defines a binary structure for the interface between the client and the object. This binary structure provides the basis for interoperability between software components written in arbitrary languages. As long as a compiler can reduce language structures down to this binary representation, the implementation language for clients and COM objects does not matter. The point of contact is the run-time binary representation. Thus, COM objects and clients can be coded in any language that supports Microsoft's COM binary structure.

A COM object can support any number of interfaces. An interface provides a grouped collection of related methods. COM objects and interfaces are specified using Microsoft Interface Definition Language (IDL), an extension of the DCE Interface Definition Language standard. (Gopalan, 1999)

Interfaces are considered logically immutable. Once an interface is defined, it should not be changed, new methods should not be added and existing methods should not be modified. Of course this restriction on the interfaces is not enforced, but it is a rule that component developers should follow. Adhering to this restriction removes the potential for version incompatibility. If an interface never changes, then clients depending on the interface can rely on a consistent set of services. If new functionality has to be added to a component, it can be exposed through a different interface.

Every COM object runs inside of a server. A single server can support multiple COM objects. There are three ways in which a client can access COM objects provided by a server:

1. **In-process server:** The client can link directly to a library containing the server. The client and server execute in the same process. Communication is accomplished through function calls. (In this case the sharing of data between the two is simple).

2. **Local Object Proxy:** The client can access a server running in a different process but on the same machine through an inter-process communication mechanism. This mechanism is actually a lightweight Remote Procedure Call (RPC).

3. **Remote Object Proxy:** The client can access a remote server running on another machine. The network communication between client and server is accomplished through DCE RPC. The mechanism supporting access to remote servers is called DCOM.

   (When the server process is separate from the client process, as in a local server or remote server, COM must format and bundle the data in order to share it. This process of preparing the data is called marshalling. Marshalling is accomplished through a "proxy" object and a "stub" object that handle the cross-process communication details for any particular interface).

In the case 2 and 3, COM creates the "stub" in the object's server process and has the stub manage the real interface pointer. COM then creates the "proxy" in the client's process, and connects it to the stub. The proxy then supplies the interface pointer to the client. (Gopalan 1999)

The client calls the interfaces of the server through the proxy, which marshals the parameters and passes them to the server stub. The stub “unmarshals” the parameters and makes the actual call inside the server object. When the call completes, the stub marshals return values and passes them to the proxy, which in turn returns them to the client. The same proxy/stub mechanism is used when the client and server are on different machines. However, the internal implementation of marshalling and unmarshalling differs depending on whether the client and server operate on the same machine (COM) or on different machines (DCOM). Given an IDL file, the Microsoft IDL compiler can create

default proxy and stub code that performs all necessary marshalling and unmarshalling. (Gopalan 1999)

All COM objects are registered with a component database. When a client wishes to create and use a COM object:

1. It invokes the COM API to instantiate a new COM object.

2. COM locates the object implementation and initiates a server process for the object.

3. The server process creates the object, and returns an interface pointer at the object.

4. The client can then interact with the newly instantiated COM object through the interface pointer.

An important aspect in COM is that objects have no identity. In other words, a client can ask for a COM object of some type, but not for a particular object. Every time that COM is asked for a COM object, a new instance is returned. The main advantage of this policy is that COM implementations can pool COM objects and return these pooled objects to requesting clients. Whenever a client has finished using an object the instance is returned to the pool. However, there are even mechanisms to simulate identity in COM but we will not talk about them here. (Gopalan, 1999)

COM has enjoyed great industrial support with thousands of developing COM components and applications. COM even suffers from some weaknesses. The main problems with COM is:

1. COM is hard to use. Reference counting, Microsoft IDL, Global Unique Identifiers (GUID), etc. require deep knowledge of COM specification from developers.

2. COM is not robust enough for enterprise deployments. Services such as security, transactions, reliable communications, and load balancing are not integrated in COM.

But at all, a number of developers see COM as more than a little challenging to understand and use. The reason for this is simple: Using COM's language-independent objects in any real programming language requires understanding a new object model, the one defined by COM. For example, a C++ programmer knows that creating a new object requires using the language's new operator, while getting rid of that object requires calling delete. If that same C++ programmer wants to use a COM object, however, she can't do things in this familiar way. Instead, she must call the standard COM function CoCreateInstance (or one of a few other choices) to create the object. When done with this COM object, she doesn't delete it explicitly, as in C++, but instead invokes the object's Release method. The object relies on an internal reference count that it maintains to determine when it has no more clients, and thus when it's safe to destroy itself. (Platt, 1999)

Writing COM objects requires a developer to provide a significant amount of infrastructure that doesn't vary much from one object to another. For example, every COM object requires an implementation of the Unknown interface, which provides reference counting and polymorphism. Many require a class factory to let the operating system reach into the server and create the object, and code for making the registry entries that tell the operating system where to find this class factory. Objects that want to run in scripting environments such as Web browsers (most objects, these days) need to implement a fairly complex interface called IDispatch to permit runtime binding to method names, and so on.

An interesting thing about COM components is that they are never linked to any application. The only thing that an application may know about a COM object is what functions it may or may not support. In fact, the object model is so flexible that applications can query the COM object at run-time as to what functionality it provides. (Gopalan, 1999)

Garbage Collection is one other major advantage to using COM. When there are no outstanding references (a.k.a. pointers) to an object, the COM object destroys itself.

COM can even be seen relatively straightforward to use from some languages. Visual Basic programmers, for example, must make some COM-specific calls, but VB itself hides many of the details. Today, after the judgment, that Microsoft has got in fraction with SUN, we don’t know clearly if Microsoft can yet have an implementation of the Java virtual machine or not but any way, Microsoft's implementation of the Java virtual machine makes COM's integration with Java even simpler. It allows Java programmers to write ordinary Java code, and then silently performs any necessary translations. But for C++ developers, using COM means understanding a significant number of COM-specific rules and API calls. While COM+ will bring some changes to developers working in VB, Java, and other higher-level languages, it's C++ programmers who will most appreciate it. Because COM+ relies on the C++ compiler to do much of the work of translating between this language-independent object model and the C++ object model, the developer's life becomes easier. In COM, objects and their clients make calls on a standard COM library. In languages like Visual Basic and Java, some or all of these calls are hidden, but C++ programmers use this library directly.

The only language requirement for COM is that code is generated in a language that can create structures of pointers and, either explicitly or implicitly, call functions through pointers.
So Object-oriented languages such as MS Visual C++ and Smalltalk provide programming mechanisms that simplify the implementation of COM objects but even some other languages such as Ada, C, Java, Pascal and even Basic programming environments can create and use COM object. Many of the weaknesses in COM have disappeared in a new version of COM, which is called COM+. (Platt, 1999)

COM +

COM+ is much younger than COM. It came in 1997 with NT 5.0. NT 5.0 is named today as Windows 2000. COM+ combines enhancements to the Microsoft Component Object Model (COM) with a new version of the technology called Microsoft Transaction Server 2.0, along with many new services. COM+ integrates MTS services and message queuing into COM, and makes COM programming easier through a closer integration with Microsoft languages, like Visual Basic, Visual C++, and J++. COM+ will not only add MTS-like quality of service into every COM+ object, but it will hide some of the complexities in COM coding.

Actually, COM+ is many things. COM+ is the merging of the COM and MTS programming models with the addition of several new features.

Some of these features are:

- Object Pooling
- Microsoft Message Queue (MSMQ)
- Role-based Security
- Queued Components
- Threading
- Automatic Transactions
- COM+ Events
- Application and Component Administration

COM was created long ago as a workstation-level component technology; with the release of Distributed COM (DCOM) in Windows NT 4.0, the technology was expanded to support distributed applications via remote component instantiation and method invocations. MTS followed. It was designed to provide server-side component services and to fix some of DCOM's deficiencies, e.g., how it handles security issues, and the complete lack of a component management and configuration environment. COM+ now comes along to unify COM, DCOM, and MTS into a coherent, enterprise-worthy component technology. (Platt, 1999)

Note that programs running on NT or Windows 95 operating systems can be clients of COM+ objects running on Windows 2000, but cannot be servers. In one sense, COM+ is just the Windows 2000 release of COM, with incrementally more new features to solve new problems, the same as previous releases of COM have had.

COM+ takes the COM and MTS programming models to the next level. At the same time, COM+ addresses many of the shortcomings associated with COM and DCOM development, for example, COM+ provides a much better component administration environment, support for load balancing and object pooling, and an easier-to-use event model.

Here's another way to look at COM+: It is the maturation of Microsoft's component architecture. COM has always been difficult for developers to understand, at least initially. Developing COM-based applications requires a change in mindset. It's almost as difficult as moving from C to C++ programming. COM+ doesn't necessarily change this completely, but it does move many of the details into the operating system so developers are free to focus on higher-level problems and their solutions. (Platt 1999)

Today, COM and MTS components place all of their configuration information in the Windows registry. With COM+, however, most component information will be stored in a new database, currently called the COM+ Catalog. The COM+ Catalog unifies the COM and MTS registration models and provides an administrative environment for components. As a developer you interact with the COM+ Catalog using either the COM+ Explorer or through a series of new COM interfaces that expose its capabilities.

One COM+ feature is its support for declarative programming. What this means is that you can develop components in a generic way and defer many of the details until deployment time.
COM+ is a good candidate to implement the middle layer of multi-tier architectures. The distribution support and quality of service provided by COM+ can help to overcome some of the complexities involved in these architectures.

COM+ Programming is based on the following points:

1. COM+ Programming is Interface Programming. This means that clients program on the basis of Interfaces not Classes.

2. Code is not statically linked but is loaded at runtime as and when required.

3. Component implementers specify their requirements declaratively and the framework ensures that these requirements are met as in an MTS or a COM+ system.

COM+ still provides a standard library, and objects and their clients still use it. But in contrast to COM, COM+ hides calls to this library beneath the equivalent native functions in the programming language.

Since COM+ is involved in every method call, it's now possible to insert other objects in that path. A COM+ object that is automatically invoked during access of another COM+ object is called an *interceptor.* One or more interceptors can lie in wait along the path. For example, an interceptor might perform a security check on the client, then cause the method call to fail if the necessary permissions aren't in place.

In COM, the COM library is not directly involved when a client calls a method in a COM object. Instead, the call goes directly from the client to the object. In COM+, this is no longer true. All COM+ method calls pass through the COM+ library, although Microsoft promises that the overhead this incurs will be negligible. For example, C++ programmers can use the standard new operator rather than CoCreateInstance to create a COM+ object. In doing so, they are relying on a C++ compiler that is aware of COM+ to generate the correct code to call the COM+ library.

To accomplish this, the compiler uses the COM+ library at compile time, then embeds calls to this same COM+ library in the generated binary. Microsoft will provide this library, and any language tool that wants to use COM+ must rely upon it. Unlike classic COM, where only COM objects and their clients use the COM library, COM+ also requires compilers (or interpreters, such as those for Visual Basic, and scripting languages, like JavaScript) to rely on a standard library to produce the correct code. Microsoft's competitors in the tools market, already accustomed to working on top of their competitor's operating system, now face the prospect of depending on yet another Microsoft-supplied component for key functions. The benefit, however, is that doing this will make using Microsoft's language-independent object model easier for their customers, too.

Object Pooling

Object pooling allows an application to create objects that can be created and pooled according to application needs. Reusing the elements in a pool of deactivated objects can save on system

resources. You can pool almost any kind of object, and object pooling is easy to implement. (Microsoft, 1999)

Microsoft Message Queue (MSMQ)

MSMQ is a middleware product that enables asynchronous communications between applications. This means that client applications can send messages to a server even when the server isn't running. It also means that the server can receive messages after the client application has gone away. In environments in which client applications and servers can become disconnected for any number of reasons, this capability allows the distributed application as a whole to stay up and running. MSMQ is based on the asynchronous delivery of messages to named queues. It's a product that can be integrated into distributed applications, including those based on MTS. Messages model procedure calls between a client and a server except that either party can do its work in the absence of the other. (Microsoft, 1999)

For example, in a sales order application a client could submit several orders to the queue even when the server application wasn't running. At some time later when the server was started, the server could open the queue and retrieve all the order requests that were submitted by any client. MSMQ also makes it possible to return a message receipt to the caller as if it were the response to a method call. MSMQ also contributes to an application's scalability. As the traffic in a distributed application increases, the managers of the system need to add more computers to increase the throughput of the entire system. To make use of any additional hardware, the system must somehow distribute client requests across all the available servers. This act of using as many of the available processing cycles as possible is known as load balancing. A queue facilitates load balancing because a client doesn't send a request to a specific server. The queue is a single repository that holds all incoming requests. A group of servers monitors the queue and splits up the workload as it arrives. Visual Basic programmers can access MSMQ services through a set of COM interfaces. MSMQ integration with MTS makes it possible for a business object to retrieve a message from a queue inside a declarative transaction. If something goes wrong during the transaction, the message is left in the queue. This means that MSMQ can ensure that all messages eventually reach their destination. MSMQ also provides advanced features that address routing efficiency, security, and priority-based messaging. (Microsoft, 1999)

Role-based Security

COM+ supports both role-based security and process access permissions security. In the role-based security model, access to parts of an application are granted or denied based on the logical group or role to which the caller has been assigned (examples of roles could include administrator, developer, and other actors). Also in role-based security is the ability to have security at the method level, as well as at the component and interface levels. (Platt, 1997)

Queued Components

Queued Components allow you to create components that can execute immediately if the client and server are connected; or, if the client and server are not connected, the components can defer execution until a connection is made. Queued Components is key functionality for line-of-business applications built for enterprises. It augments "real time" transactions with the infrastructure necessary to process asynchronous, queued transactions. This technology is effective for creating applications used by clients "in the field" and disconnected from a network; the client can enter transactions that will be executed when the client is again connected to the network. This asynchronous message queuing uses Windows 2000 Messaging Services. (Platt, 1999)

Threading

COM+ includes a new threading model called the neutral threading. The neutral model supports execution of objects on any thread type. The benefit of neutral apartment threading is that it does not require the developer of objects running in a neutral apartment to make the object thread safe. (Microsoft, 1999)

Automatic Transactions

COM+ supports all Microsoft Transaction Server (MTS) 2.0 semantics and adds "Auto-done," which allows the system to automatically call SetAbort if an exception is triggered or SetComplete if not. In addition, COM+ supports components with a special environment called a context, which provides an extensible set of properties that define the execution environment for the component. (Microsoft, 1999)

COM+ Events

COM+ event is a component-based event system using publishers and subscribers. Publishers send events to subscribers; the publisher through subscriptions identifies subscribers. Publishers can be any COM+ objects that call an event object, and subscribers can be simple COM+ objects that implement the methods on the event interface. (Microsoft, 1999)

Application and Component Administration

In COM+, a new registration database called RegDB stores the metadata that describes components. This database is highly optimized for the type of information that COM+ needs for activation of components and is used instead of the system registry. In addition, COM+ exposes a transactional, scriptable interface called the COM+ catalog, which accesses information in the RegDB. Finally, the Component Services administrative tool provides a fully scriptable user interface for both developers

and administrators to administer components as well as deploy both client-side and server-side multi-tier applications.

Metadata's Importance

Today, a COM object defines the methods it supports in some number of interfaces, each of which can be described using COM's interface definition language (IDL). A tool (the MIDL compiler) then compiles an object's IDL to produce a type library, commonly stored in its own file. Clients of the object can (but don't have to) read this library to learn how to make calls on the object's methods. In COM+, developers no longer need to define interfaces using IDL. Instead, they can just use their programming language's syntax to define the object's interfaces. The compiler for that language then works with the COM+ library to generate metadata for the object. Metadata, essentially a superset of the information in today's type library, goes in the object's same binary file. And unlike optional type libraries, every COM+ object must have metadata.

Even more interesting, COM+ object metadata will be accessible through the generic data access interfaces defined by OLE Database (more commonly called OLE DB). These let clients of the object issue SQL queries against its own metadata, in order to search for methods or parameter types. One obvious problem: Not relying on the current IDL might mean adding syntax extensions to programming languages to accomplish the same function. Microsoft can use whatever interface definition syntax it chooses for Visual Basic, since it owns the language. Java already has standard language constructs for specifying interfaces. But C++, which will benefit most from COM+, currently has no standard way to specify interfaces. Although plans are not yet finalized, Microsoft says it intends to extend the language syntax by implementing an interface definition scheme in its widely used Visual C++ compiler.

Since every COM+ object has metadata, it's also possible to approach marshaling consistently. Marshaling is packaging a method call's parameters in some standard way, allowing these parameters to move effectively between objects written in entirely different languages or running on entirely different machines. COM today provides two very different solutions to perform marshaling. (Gopalan, 1999)

If an object exposes its methods using a vtable interface (also called a custom interface), its client typically relies on a proxy and stub to marshal and unmarshal the parameters for calls to those methods. A stub and proxy can be automatically generated from an interface's IDL definition using the same MIDL compiler that produces type libraries.

The other way is for a COM object to expose its methods using a dispatch interface (or *dispinterface*). In this case, a client need not rely on a proxy and stub for marshaling and unmarshaling. Instead, the client can read an object's type library, and then dynamically perform marshaling as required. This is a more flexible system; however, since not all COM objects have type libraries, it's not always possible in practice today.

But every COM+ object has metadata, the new equivalent of a COM type library. So COM+ can potentially get rid of proxies and stubs altogether, allowing a single consistent type of marshaling. COM+ also does away with the distinction between vtable interfaces and dispinterfaces, an inconvenient artifact of the way COM grew. While Microsoft has indicated that the first release of

COM+ might still need proxies and stubs, the intent is clearly toward dynamic marshaling as the standard approach.

COM+ addresses yet another important but challenging problem in creating a language-independent object model: data types. Different languages support different data types, which causes problems when passing parameters between objects written in different languages. C++, for example, supports structures, while Visual Basic does not. Today, COM supports one set of data types for vtable interfaces (defined with C++ in mind) and yet another more limited set of data types for dispint erfaces (created with Visual Basic in mind). COM+ gets rid of this historical distinction by defining one common set of data types that are usable across all interfaces and then relying on the COM+ library to perform any translations that are necessary.

COM+ brings many other changes to the COM. One of the most important: COM+ eliminates the need for clients to call Release when they are done using an object. Instead, the COM+ library automatically handles reference counting, always one of COM's most error-prone areas. And while COM has always supported interface inheritance, COM+ also allows implementation inheritance between COM+ objects running in the same process. Despite years of arguing that this was not a desirable feature for a component model, Microsoft appears to have yielded to the demands of at least some of their customers and added this feature. (Gopalan, 1999)

COM+ also changes COM's persistence model. Today, the creator of a COM object must typically implement one or more of a fairly large set of interfaces related to persistence. A client of this object then calls various methods in those interfaces to have the object load or save its persistent state. But the COM+ library provides standard support for persistence, removing much of the burden from the COM+ object implementor. And by representing an object's properties in a standard way ("serialization"), COM+ lets you pass objects by value. All that's required is to send this serialized representation of an object's data to another object of the same class.

Another interesting change from COM to COM+ is that support for constructors, makes COM+ objects more like objects in a typical object-oriented programming language. Languages like C++ and Java can define a constructor method that runs when first creating an object. The creator of the object can then pass parameters as needed to this constructor, allowing easy initialization. So COM objects do not support constructors, but COM+ objects do. COM+ constructors even allow passing parameters, better integrating COM+ objects and the objects used by today's most popular object-oriented languages.

DCOM

The Distributed Component Object Model (DCOM) is a protocol that enables software components to communicate directly over a network in a reliable, secure, and efficient manner. Previously called "Network OLE," DCOM is designed for use across multiple network transports, including Internet protocols such as HTTP.

Distributed COM is an extension to COM that allows network-based component interaction. While COM processes can run on the same machine but in different address spaces, the DCOM extension allows processes to be spread across a network. With DCOM, components operating on a variety of platforms can interact, as long as DCOM is available within the environment. (Microsoft, 1999)

Microsoft Transaction Server (MTS)

A transaction defines a set of events that are guaranteed to be committed or rolled back as a unit. Accordingly, every transaction has a beginning, some number of events that are part of the transaction (such as database updates), and an end. Exactly how a transaction's boundaries are demarcated can vary. A common approach in traditional client/server transaction products is to require the client to make a specific call to the transaction coordinator to begin the transaction. The client then makes calls to the components that carry out this transaction's work, and finally ends the transaction with another explicit call to the transaction coordinator. This final, transaction-ending call can indicate that the transaction should be committed, making all of its changes permanent, or aborted, causing all of its changes to be rolled back. It's the job of the transaction processing system to ensure that exactly one of these two outcomes occurs. (Microsoft, 1999)

Microsoft Transaction Server is a component-based transaction processing system/service for developing, deploying, and managing high-performing, scalable, and robust enterprise, Internet, and intranet server applications. MTS was first available in 1996 in the Windows NT operating system (such a separate packet). (Platt, 1999)

MTS is a component runtime environment, in which components live, which watches requests coming into components and participates in processing them, providing security, automatic transaction management and a scaleable environment. Also, it is important to realize that MTS is a stateless component model, whose components are always packaged as an in-proc DLL. Since they are composed of COM objects, MTS components can be implemented in a variety of different languages including C++, Java, Object Pascal (Delphi), Visual Basic and even COBOL!

MTS is actually built on COM and brings in mainframe-like transactional reliability to the PC world following a "write once, run many" strategy. Developers use MTS to deploy scalable server applications built from COM components, focusing on solving business problems instead of on the programming application infrastructure. MTS delivers the "plumbing" - including transactions, scalability services, database connection management, and point-and-click administration - providing developers with the easiest way to build and deploy scalable server applications for business and the Internet. (Gopalan, 1999)

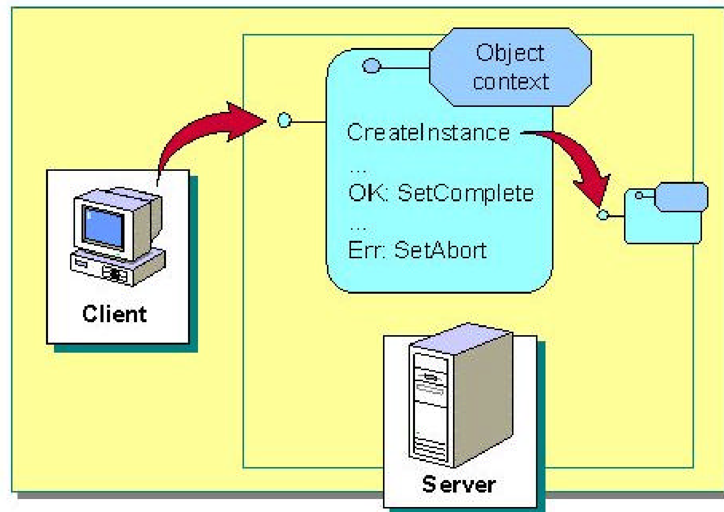


Figure 6.10 Microsoft Transaction Server Model (Microsoft, 1999)

An MTS architecture is mainly built of 5 things:

1. The MTS executive (mtxex.dll)
2. A factory wrapper and a context wrapper for each component
3. The MTS server component
4. MTS client
5. And some other additional services systems such as Service Control Management, COM runtime services, the COM Transaction Integrator etc.

An MTS application is implemented as one or more components. Those COM-Components that are running under MTS are called MTS Component. The MTS Executive manages these components. MTS relies on a class factory object to create specific instances of each COM class.

When a client send a call to MTS component, the Factory Wrapper and the Object Wrapper catch this call and inject their own algorithm, called JITA[1], into the call and send in to actual MTS component. (Microsoft, 1999)

[1] Just In Time Activation

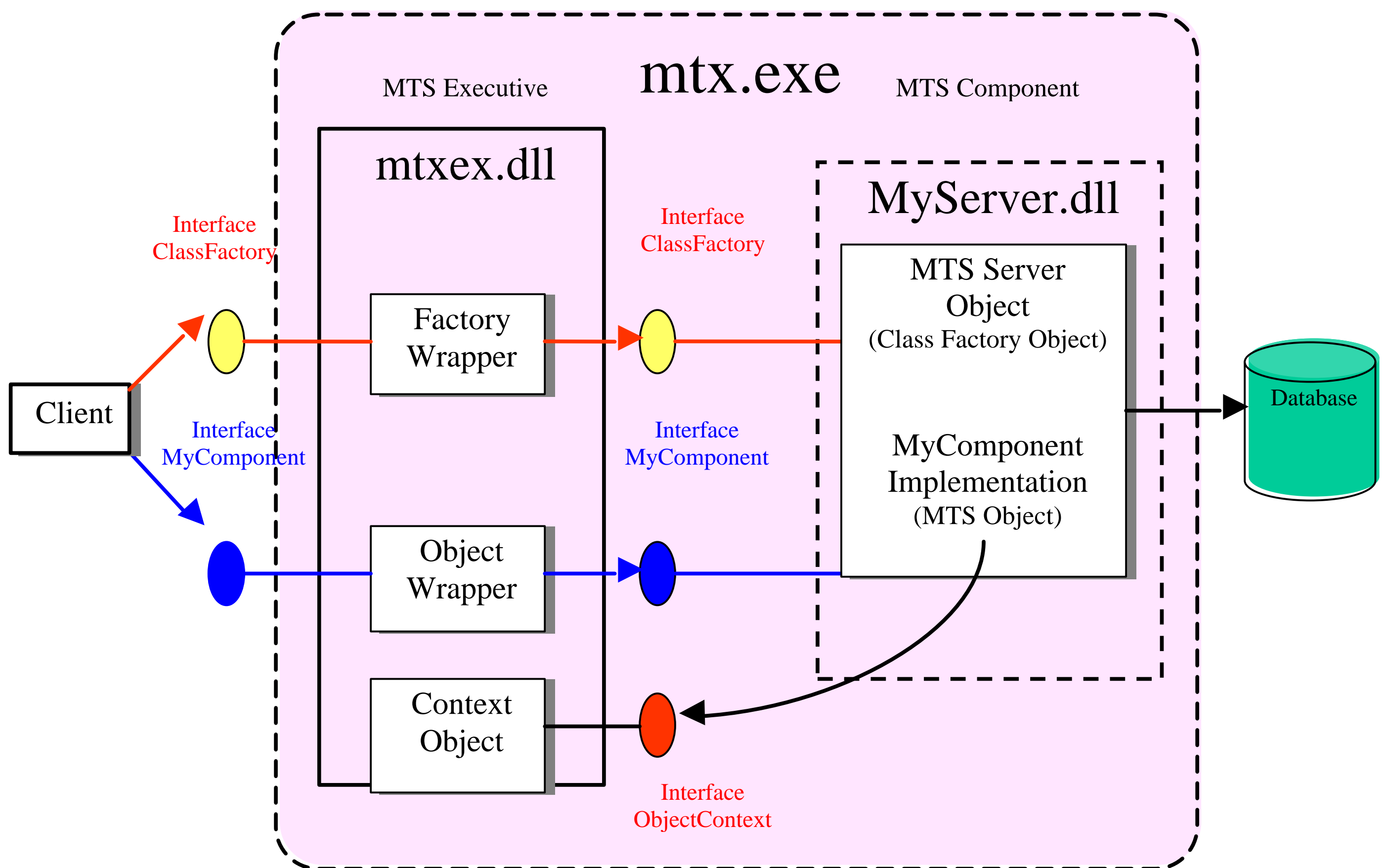


Figure 6.11 A basic MTS architecture (Gopalan, 1999)

To allow MTS to provide extra services, the MTS Executive transparently inserts a wrapper object between each object it manages and that object's client. For example, when a client uses the COM IClassFactory interface to create an instance of a MTS object, that call is actually made on a factory wrapper object implemented by MTS, which in turn passes on the call to the real class factory. Similarly, every method call a client makes on the business methods of a MTS object is first handled by a MTS-supplied context wrapper object. The MTS Executive also supplies a context object for each MTS object. This context object maintains information specific to the MTS object, such as what transaction it belongs to (if any) and whether it has completed its work. Every context object implements the IObjectContext interface, through which an MTS object, can access the services that MTS provides.

MTS objects are typically designed to encapsulate some set of business functionality. For example, an MTS object might allow a client to transfer money between two accounts or build and submit an order of some kind. Because MTS objects are written as distinct components, business functions encapsulated into these objects can be combined in arbitrary ways, allowing them to be flexibly reused. Although it's not strictly required, an MTS object is typically accessed by one client at a time, and it may (but isn't required to) use transactions.

An MTS client needs to do nothing special to create and work with transactional objects; to a client, an MTS object looks like an ordinary COM object. (Microsoft, 1999)

Transactions in Microsoft Transaction Server

When combining transactions with components, the traditional client-controlled model isn't usually the best approach. Instead, MTS introduced a new way to demarcate transaction boundaries. Called automatic transactions, it allows clients to remain unaware of when transactions begin and end—they never need to make explicit calls to begin or end a transaction. Instead, when a transaction begins depends on the value of that component's transaction attribute. This value can be set to one of four possibilities

- If a component is marked as **Requires New**, the MTS Executive will always begin a transaction when its caller first invokes a method in one of that component's objects.
- If a component is marked as **Required**, the MTS Executive will begin a transaction when its caller first invokes a method in one of that component's objects unless the caller is already part of a transaction. In this case, any methods invoked in the object will become part of the existing transaction.
- If a component is marked as **Supported**, the MTS Executive will never begin a transaction when a caller invokes methods in one of the component's objects. If the caller is already part of a transaction, the work this object does will become part of that transaction. If the caller is not part of a transaction, this object's work will execute without being wrapped inside a transaction.
- If a component is marked as **Not Supported**, the MTS Executive will never begin a transaction when a caller invokes methods in one of the component's objects. Even if the caller is part of an existing transaction, this object's work will not become part of that transaction.

When it has finished its work, an MTS object can call either SetComplete, if it wishes to commit that work, or SetAbort, if the work it has done should be rolled back. If neither is called by the time this object's transaction ends, the MTS Executive behaves as though the object called SetComplete-the default behavior is to commit the transaction. These calls are made by a method in the MTS object itself, not by the client (again, the client need not be aware that transactions are being used). If every object participating in the transaction agrees to commit, the MTS Executive will tell the transaction coordinator to commit the transaction. If any MTS object in this transaction calls SetAbort, however, the MTS Executive will tell the transaction coordinator to abort the transaction. (Gopalan, 1999)

6.1.3 CORBA (Common Object Request Broker Architecture)

CORBA is a specification developed by the Object Management Group (OMG) to aid in distributed objects programming.

Object Management Group (OMG)

The Object Management Group is an international non-profit organization that is supported by over 800 members, including IS vendors, software developers and users. Its purpose is to establish industry guidelines and specifications for object management to provide a common framework for application development. This framework is intended to make it possible to develop heterogeneous applications environments across all major hardware platforms and operating systems. OMG produces specifications only, not executable systems. OMG also promotes using object-oriented technology in software development and has established the Object Management Architecture (OMA), which provides the conceptual infrastructure upon which all OMG specifications are based. (Object Management Group, 2001)

OMA Object Model and Reference Model

The *Object Management Architecture Guide* defines an abstract *Core Object Model*. From this model, a concrete *Object Model* that underlies the CORBA architecture is derived. The concrete Object Model provides a presentation of object concepts and terminology. (Object Management Group, 2001)

In the OMA Object model, an object is an identifiable, encapsulated entity that provides one or more services that can be requested by a client. Clients issue requests to objects to perform services on their behalf. The implementation and location of each object are hidden from the client making the request. A client makes the request through the object's interface, which is a description of a set of operations. A client object is an object, which issues a request for a service. A server object is an object providing response to a request for a service. This means that a given object may be a client for some requests and a server for other requests. (Object Management Group, 1997)

The Object Management Architecture Guide also defines a *Reference Model* (see figure 6.12). The Reference Model identifies the components, interfaces and protocols that compose the OMA, where CORBA is one of these components. The Reference Model consists of the following components:

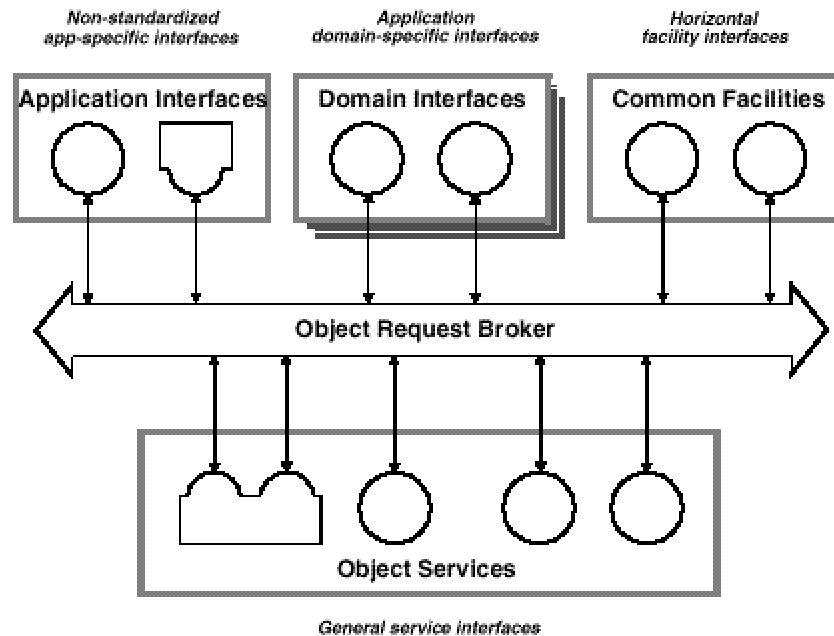


Figure 6.12 OMG Reference Model: Interface Categories (Object Management Group, 1997)

Object Request Broker (CORBA)

The Object Request Broker is the central item in the OMA. It is this component that is commercially referred to as CORBA. The ORB is the middleware that manages the client-server relationships between objects. "*The term middleware is used to describe separate products that serve as the glue between two applications. Middleware is sometimes called plumbing because it connects two sides of an application and passes data between them.*" (The Lycos Tech Glossary, 2001) It provides an infrastructure that allows objects to communicate, independent of the specific platforms and techniques that are used to implement the objects. By using an Object Request Broker, a client can transparently invoke an operation on a server object, which can be on the same machine or across the network; the client will not notice any difference. The ORB is responsible for finding the object implementation for a request, to pass the parameters, invoke the operation and return the results. It is like a telephone exchange, providing the basic mechanism for making and receiving calls. The client does not have to be aware of the object's location, its programming language, its operating system, or other system aspects that are not part of the object's interface. (Object Management Group, 1997)

Object Services (CORBA Services)

The Object Services component provides general, domain-independent services that are likely to be used in any program that is based on distributed objects. There are so far 16 standardized services that handle tasks such as naming of objects, object security, creation and deletion of

objects, locating objects by attributes, etc. Note that since OMA is a specification and not an executable system, OMA only provides the interface definitions for these services. However, there are implementations of these services in the market, which can be bought off-the-shelf and used directly with a CORBA application. The table below gives a short explanation of each service. (Object Management Group, 1997; Szyperski, 1999)

Name	**Description**
Collection Service	Provides a uniform way to create and manipulate collections, like stacks, queues and lists.
Concurrency Service	Enables multiple clients to coordinate their access to shared resources.
Event Service	Allows definition of event objects that can be sent from event suppliers to event consumers.
Externalization Service	Provides support for externalizing and internalizing objects (recording object state to a stream of data and vice versa).
Licensing Service	Provides a mechanism to control how an application is used, for example by setting start and expiration dates.
Lifecycle Service	Supports creation, copying, moving, and deletion of objects and related groups of objects.
Naming Service	Allows association of object references with symbolic names, and lets clients find objects based on these names.
Notification Service	Provides an extension of the Event Service, with some new capabilities.
Persistent Object Service	Provides interfaces to the mechanisms used for managing persistent storage of objects.
Property Service	Allows arbitrary properties to be associated with objects.
Query Service	Allows clients to invoke queries on collections of objects.
Relationship Service	Allows general relations between objects to be specified and maintained.
Security Service	Provides identification and authentication of users to verify they are who they claim to be, authorization and access control, etc.
Time Service	Handles inaccuracies in a distributed system with multiple asynchronous clocks.
Trader Service	Allows clients to find objects based on their properties, describing the service offered by the object.
Transaction Service	Supports multiple transaction models.

Common Facilities (CORBA Facilities)

The Common Facilities component provides services that many applications may share, but which are not as fundamental as the Object Services. These interfaces are oriented towards end-user applications. There are four areas for standardization so far, though most of the standardization within these areas is not yet finalized and implemented. The areas are *user*

interfaces (for example printing, email, object linking), *information management* (structured storage, universal data transfer, meta-data), *system management* (instrumentation, monitoring, logging) and *task management* (workflow, rules, agents). (Object Management Group, 1997; Szyperski, 1999)

At present, the only current formal specifications that have been adopted are the *Internationalization and Time* specification, which defines a set of interfaces for extracting information from localized representation of dates, times and numbers, and the *Mobile Agent Facility* specification, which provides a standardization of agent management and transfer to achieve a higher interoperability between various agent systems. (Object Management Group, 2000a, January; Object Management Group, 2000b, January)

Domain Interfaces (CORBA Domains)

These are domain-specific interfaces for application domains such as Finance, Healthcare, Manufacturing, Telecom, Electronic Commerce, and Transportation. OMG has a large number of special interest groups, which focus on different application domains. (Object Management Group, 1997)

Application Objects (CORBA Applications)

These are interfaces developed specifically for a given application domain. Basically, any application that anyone writes that uses CORBA counts as a CORBA Application. Because they are application-specific, these interfaces are not standardized, except for a few more broadly useful application objects, like Business Objects. Business Objects are application objects that directly make sense to people in a specific business domain, like customer or stock objects. (Szyperski, 1999; Vinoski, 1997)

Object Request Broker (ORB)

The Object Request Broker that was shortly described above, consists of the following primary components: the ORB Core, an ORB Interface, OMG IDL stubs and skeletons, DII, DSI, and Object Adapters. Clients use the ORB to make requests to objects. Figure 6.13 shows the structure of an ORB.

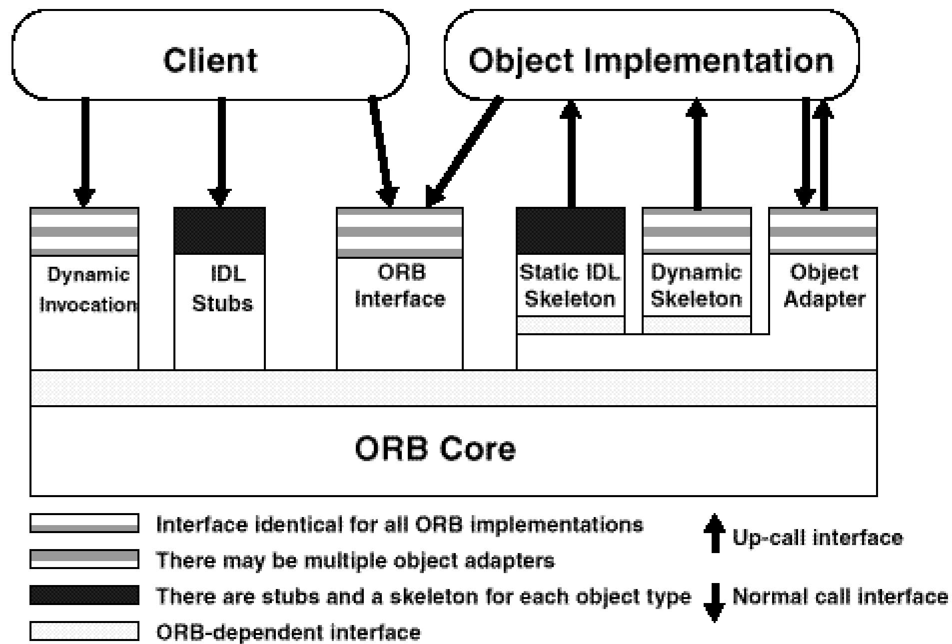


Figure 6.13 Structure of an individual Object Request Broker (Object Management Group, 2001)

Object implementation (Servant)

A CORBA object is an instance of an implementation and an interface. It is identified, located, and addressed by its object reference. An object encapsulates state and operations (that are internally implemented as data and methods) and is capable of being located by an Object Request Broker and having client requests delivered to it. When a client invokes an operation on an object, the object implementation acts as a server for the request. An object and its implementation is not part of the ORB, but it uses the ORB to receive requests from clients and to send responses. (Object Management Group, 2001)

Another name for the object implementation is *servant*. The servant consists of an implementation programming language entity that defines the operations that are supported by the object's interface. Servants can be written in many different programming languages. (Object Management Group, 2001)

Client

A client is the program entity that invokes operations on an object. The client only knows the object through an interface and knows nothing about its implementation. The term client is something that is relative to a particular object. This means that an implementation of one object

may be a client of other objects. The client is not part of the ORB, but it uses the ORB to make requests to objects. (Object Management Group, 2001)

There are three different ways for a client to obtain a reference to a remote object:

- To create a new object, by invoking a creation request, which will return an object reference to the newly created object.

- To invoke a lookup service, like the Naming Service, that allows the client to obtain an object reference by giving the name of the object, or the Trader Service, that allows the client to obtain an object reference by giving the properties of the object.

- By converting a string into an object reference. Objects can be turned to strings and stored into a file or a database, and later be retrieved and turned into an object reference again. (Vinoski, 1997)

Object Request Broker Core (ORB Core)

The ORB Core is the part of the ORB that provides the basic representation of objects and the communication of requests. It supports the minimum functionality to move a request from a client to the appropriate adapter for the target object, and return any response to the client. The ORB hides from the client the object location, the object implementation, whether the object is activated or not when the request arrives and what communication mechanism the ORB uses to send the request. (Object Management Group, 2001; Vinoski, 1997)

ORB Interface

The ORB Interface is an interface to the operations that are implemented by the ORB Core. The purpose of the ORB interface is to decouple applications from ORB implementation details. The operations are available to both clients and server objects. These operations do not depend on any specific *object adapter* (this term is described later), and are the same for all ORBs and all object implementations. Some examples of the operations that the ORB Interface provide are:

- **object_to_string()/string_to_object():** Converts an object reference to a string and vice versa. A stringified object reference can be written down as text somewhere and later be transformed back to a valid object reference.

- **resolve_initial_references():** Helps the client find the object references needed to get started, might for example be used to find the object reference to the Naming Service.

- **BOA_init():** Initializes an object adapter. (Benjamin, Vasudevan, Villalba & Vogel, 1999; Object Management Group, 2001)

OMG IDL stubs and skeletons

The OMG Interface Definition Language is an object-oriented declarative language, that is used to define the objects' interfaces. It is not a complete programming language; it is only used for

describing interfaces, and does not contain any executable statements (like for-loops or if/else-statements). Interfaces declared in IDL are programming language neutral. The interfaces consist of attributes and a set of named operations with the parameters to those operations. Through IDL, an object tells its potential clients what operations are available and how they should be invoked. From these IDL definitions the CORBA objects are mapped to the object implementations, written in different programming languages. (Mahmoud, 2001)

Programming languages with IDL mapping are for example C, C++, Java, Smalltalk and COBOL. This separation of object definition and implementation makes it possible to use any programming language with an IDL mapping to make remote requests to a CORBA object. Although OMG IDL is an object-oriented language, it is possible to use programming languages as C and COBOL with CORBA, as long as there is an IDL mapping for these languages. In these non-object oriented languages an object will be mapped to a collection of functions that manipulate data (e.g. an instance of a struct or record) that represent the state of the CORBA object. (Benjamin et al., 1999)

OMG IDL language compilers generate client-side *stubs* and server-side *skeletons* in a specific programming language. The stubs and skeletons are built directly into the client application and object application. The client makes local invocations to the stub, which transforms these to remote invocations, by marshalling them
and forwarding them through the ORB to the target object. Marshalling means to convert the request from its representation in a programming language to one that is suitable for transmission over the connection to the target object. The skeleton receives invocations, unmarshals them (converting from a transmittable form to a programming language form) and invokes the target method. It also accepts return values, marshals these and sends them back to the client. (Object Management Group, 2001; Szyperski, 1999)

There are two ways for an object implementation to be connected to the skeleton. The first is to derive the implementation from the skeleton, which is usually done by inheritance. This means that the implementation extends the skeleton. The other way is the delegating approach, which means that the skeleton delegates the execution to the implementation, which is kept separate from the skeleton. This second approach is useful when integrating legacy code, since the legacy code will not have to be changed to inherit from the skeleton. (Appelbaum et al., 1999; Schmidt & Vinoski, 1997)

A simple IDL interface definition might look like:

```
interface Meeting
{
attribute string date;
void scheduleMeeting();
};
```

This interface defines a string attribute named “date”. String is a CORBA data type that the IDL compiler must map to an equivalent language-specific type. An attribute keyword corresponds to two operations that the implementation must provide: an operation for retrieving the item and one for modifying it. These operations are sometimes called “getters” and “setters”. The interface in the example also defines an object that has an operation named scheduleMeeting(), which takes no parameters and returns nothing.

IDL interfaces are allowed to inherit from each other, which makes it possible to reuse existing interfaces when defining new services. Multiple inheritance is also allowed. All interfaces implicitly inherit from CORBA::Object. The CORBA::Object interface defines some operations that are applicable to any object. These operations are implemented by the ORB. CORBA::Object provides operations to retrieve interface type information for the object reference, to duplicate and release object references, to create request objects for the DII, etc. (Benjamin et al., 1999)

If running the example file in an IDL compiler with Java mapping, the compiler will create a stub file, a skeleton file, some extra help files and a file that defines the remote interface as a Java interface. This last file will look like:

```
public interface Meeting
extends org.omg.CORBA.Object
{
        String date();
        void date(String arg);
        void scheduleMeeting();
}
```

Now, the implementation of the interface can be written. Some IDL compilers also generate part of the implementation class, so the programmer just can “fill in” the rest.

Dynamic Invocation Interface (DII)

The Dynamic Invocation Interface describes the client’s side of the interface that allows dynamic creation and invocation of requests to objects. Dynamic creation and invocation means that the client does not need to know the type of object it will create/invoke at compilation time in the form of static stubs. (Object Management Group, 2001)

Using DII is more tedious than coding static stubs. When using IDL, the client stub that is generated by the IDL compiler hides the complexity of the low-level communication between the client and server. When a client makes a request to a server object, the client stub performs some steps behind the scenes. When using DII, the programmer must self locate the server object, determine the object’s interface by using the *Interface Repository* (described later), build a request, invoke the request and retrieve the returned values. Some applications, like object browsers or monitors that need to access any object without previous knowledge of their interfaces, are suitable for using DII directly. An advantage of using DII is that it supports easily replacing server objects without affecting the client. (Appelbaum, Claudio, Cline, Girou & Watson, 1999)

Dynamic Skeleton Interface (DSI)

The Dynamic Skeleton Interface (DSI) allows for dynamic handling of object invocations, and is the server side’s analogue to the client side’s DII. The DSI is a way to deliver requests from an Object Request Broker to an object implementation that does not have compile-time knowledge

of the type of the object it is implementing. The client that makes the request does not know whether the implementation is using the IDL skeletons or is using the dynamic skeletons. (Object Management Group, 2001)

Using DSI means that the object implementation is responsible for determining the operation being requested, interpreting the arguments associated with the request and invoking the appropriate internal operation to fulfill the request and returning the appropriate values. To implement an object with DSI requires, as for DII, more manual programming activity than what is required when using static skeletons. DSI can for example be useful when an object needs to offer multiple interfaces. (Appelbaum et al., 1999)

Object Adapters (OA)

An Object Adapter serves as the "glue" between object implementations/skeletons and the ORB. The object adapter adapts the interface of the object implementation to the interface expected by the ORB. Without object adapters, the object implementations would connect themselves directly to the ORB to receive requests. This would mean that the ORB would be forced to handle a large range of different interfaces to different styles of object implementations, which would make the ORB unnecessary complex. The object adapter assists the ORB with registering programming language entities as object implementations, generating object references, activating objects, invoking methods, etc. (Object Management Group, 2001; Vinoski, 1997)

For example, when a client invokes an operation on an object, the object adapter will map the object reference to the corresponding object implementation. If a client tries to invoke an operation on an object that is not in memory, the object adapter activates the object (initializes it into memory) so it can perform the requested operation. The object adapter can also deactivate objects. This means that all objects do not have to reside in memory all the time, but it will appear as if they were, to the client. There may be different adapters provided for different kinds of implementations. A different object adapter is normally necessary for each different programming language. (Object Management Group, 2001; Vinoski, 1997)

Other components

- *Interface Repository:* A service that provides run-time knowledge of object interface types. It can be thought of as a set of objects that encapsulate the IDL definitions of all CORBA types available in a particular domain. Using the IR, a client can locate an object that is unknown at compile-time, find information about its interface, and finally build and send a request to the object. The IR supports dynamic invocation (DII and DSI).

- *IDL Compiler:* A compiler that generates programming language-specific source files for the client-side stubs and server-side skeletons from the interface definitions written in IDL.

- *Implementation Repository:* A repository for information that allows the ORB to locate and activate implementations of objects. It maps server objects to source code files. This repository is assumed to exist in CORBA, but its interface is not specified, and is different

for different ORBs. (Benjamin et al., 1999; Mahmoud, 2001; Object Management Group, 2001)

GIOP and IIOP

Before CORBA 2.0, one problem with commercial ORB products was that they did not interoperate. The CORBA specification did not specify any particular data format or protocols for communication between ORBs. With CORBA 2.0, a general ORB interoperability architecture was introduced. It provides a framework and characterizes some mechanisms that are needed to achieve interoperability between ORBs from different vendors. This architecture is based on the *General Inter-ORB Protocol (GIOP)*, which specifies transfer syntax and a standard set of message formats for the communication between ORBs over any connection-oriented transport protocol that meets a minimal set of assumptions. GIOP is not a complete protocol in itself, but a specification for defining protocols, such as IIOP. IIOP (*Internet Inter-ORB Protocol*) is a specialization of GIOP and specifies how GIOP is built over TCP/IP connections. (Object Management Group, 2001; Vinoski, 1997)

IOR (Interoperable Object Reference)

An object reference is used to uniquely identify an object instance. Object references are not global identifiers that are valid across all machines in a distributed network. They are limited to a local ORB. There is no standard format for an object reference. The reason for this is that it enables implementers freedom to develop more efficient platform-dependent references structures. To pass references to objects across ORBs, an *Interoperable Object Reference* data structure is used. IORs are created from the local object references which are implemented by the ORBs in whatever way they find appropriate. A client application that makes a request to a remote object, does not directly use the IOR. The local ORB creates a local proxy for the remote object represented by the IOR. The client program only sees the local object reference for the proxy. (Appelbaum et al., 1999; Henning, 1998)

The IOR makes the details of how an object reference identifies an object and how a request reaches its destination transparent to the applications. The IOR consists of the following main components (see figure 6.14):

- **Type Name**
  This is an ID for a specific IDL type in the Interface Repository, to allow for retrieval of the object's interface type at run-time.

- **Protocol and Address Details**
  This field specifies a protocol and addressing information appropriate for that protocol. In IIOP, the specified protocol is TCP/IP and the addressing information consists of a host name and a TCP port number.

- **Object key**
  This is a piece of binary data, which looks different depending on the ORB which created it.

It consists of two components: the *object adapter name*, which identifies the particular object adapter to which the object belongs, and the *object name*, which specifies a specific object connected to the object adapter. It is not any problem that the object key does not have a standardized structure, since it is the same ORB that creates the IOR that needs to decode it to find the target object. No other ORBs will need to interpret this part of the interoperable object reference. (Henning, 1998; Object Management Group, 2001)

A single reference may contain several pairs of protocol and object key, which means that one object reference can support multiple protocols or contain different addresses for the same object, for example to provide fault tolerance. (Henning, 1998)

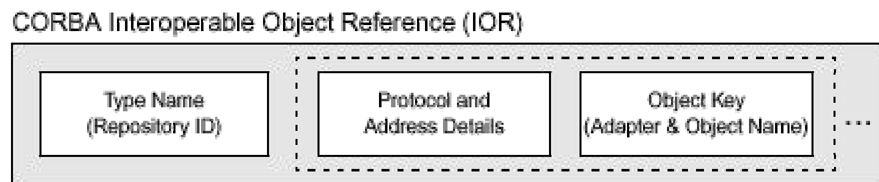


Figure 6.14 Main components of an Interoperable Object Reference (Henning, 1998)

CORBA 3.0

The features described so far are included in CORBA 2.0. Since CORBA 2.0, several new features have and are going to be added, which will all be part of CORBA 3.0. The CORBA 3.0 specification is planned to be released during 2001. Most of the parts of the specification are already available at OMG's website. Here, these new features that have been added since CORBA 2.0 will be described.

Portable Object Adapter (POA)

In CORBA 2.0 the only object adapter specified by OMG was the Basic Object Adapter (BOA). The specification of BOA was partly ambiguous and missing some features, which caused different vendors to develop different extensions to the specification. This resulted in reduced portability between different ORB implementations. The Portable Object Adapter allows object implementations to be portable between different ORB products. The first specification of the Portable Object Adapter was introduced in the CORBA 2.3 specification. The POA also provides some new features like the support of transient objects (short-lived objects), explicit and on-demand object activation and a model for multi-threading. (Mahmoud, 2001; Object Management Group, 2001; Vinoski, 1998)

The request from a client to a servant, through a POA, is shown in figure 6.15. First, the client makes a request, using an object reference to the target object. As described earlier one of the components of an IOR is an object key, that consists of an object adapter name and an object name. When the server ORB receives the request, it uses this object adapter name to find out which POA it is that hosts the target object. The ORB then dispatches the request to that POA.

The POA uses the object name to find out which servant it is that implements the target object and then dispatches the request to that servant. The servant will then carry out the request and deliver the result back to the client through the POA and ORB. (Vinoski, 1998)

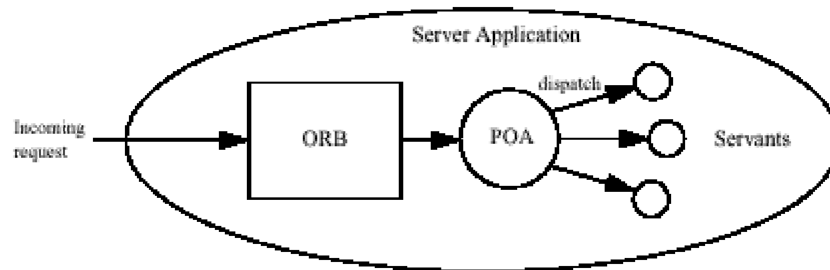


Figure 6.15 Request to a Portable Object Adapter (Vinoski, 1998)

The POA supports two types of CORBA objects: the persistent object that was originally specified by CORBA, and a new short-lived object that is called a *transient* object. Persistent objects have a lifetime that is independent of the lifetimes of any server processes in which they are activated. Transient objects have a lifetime that is bounded by the lifetime of the server process in which they're created. This means, that when their server process die, the transient object dies too. Transient objects involve less overhead than persistent objects, since there is no need to keep any activation information for them. This kind of objects are useful in situations that only require temporary objects. (Schmidt & Vinoski, 1997; Vinoski, 1998).

The activation modes supported by the BOA were centred around server processes. The POA focuses instead on CORBA object activation, and supports some different activation styles:

- *Explicit activation:* The server application programmer explicitly registers servants for CORBA objects by using direct calls on a POA.

- *On-demand activation:* The server application programmer registers a servant manager that the POA calls when it receives a request for an object that is not yet activated. The server manager then decides what to do.

- *Implicit activation:* A request to a servant results in activation without any explicit calls to the POA.

- *Default servant:* The application registers a default servant which will be used whenever a request arrives for a CORBA object that is not yet activated, and there is no servant managers registered. (Schmidt & Vinoski, 1997; Vinoski, 1998)

The BOA specification did not describe how to manage multithreaded applications. The POA can either be single-threaded or let the ORB control its threads. If the POA is single-threaded, the POA will dispatch requests to servants in a serialized manner. The ORB-controlled model allows the underlying ORB to choose an appropriate multi-threading model, so that multiple requests to the POA can be processed concurrently. Applications that use POAs for this model need to be

able to handle multiple client requests simultaneously. Applications that use single-thread model POAs do not need to be thread-aware. (Vinoski, 1998)

The POA also supports separation of servant and CORBA object lifecycles, which means that one CORBA object may be mapped to several servants during its lifetime, and one servant may represent several CORBA objects. This is an important feature to contribute to scalable applications. It means for example, that a database where each entry in the database is treated as a separate CORBA object, could have a single servant that gets the requested database entry when it is invoked, instead of having a separate servant for each database entry. (Schmidt & Vinoski, 1997; Vinoski, 1998)

CORBA Component Model (CCM)

CORBA 3 includes the introduction of a server-side component model (CORBA component model, or CCM). CCM represents an extension and addition to the OMA and CORBA. The CCM standard will allow greater software reuse and flexibility for dynamic configuration of CORBA applications. This section presents some of the most important features of the CCM architecture.

Component

A component is the basic building block in CCM. It encapsulates a design entity and is referenced by a component reference. The component type is an extension and specialization of the object type. Component types can be specified in IDL and represented in the Interface Repository. A component interacts with external entities, such as ORB services or other components, through four different kinds of mechanisms, called *ports*:

- **Facets (Provided interfaces):** An interface contract exposed by a component. Facets allow a component to expose different views to its clients by providing distinct interfaces, that are not related by inheritance. The implementation of the facets are encapsulated by the component and not visible to clients.

- **Receptacles:** Provides a way for a component to connect to other objects, including objects of other components, and invoke operations on those.

- **Event sources/sinks:** Enables components to interact by monitoring asynchronous events. Components can be either event source or event sink for one or more channels. Event sources are used for publishing events and event sinks are used for subscribing to events.

- **Attributes:** Are intended to be used for component configuration. At install time, the component is configured to act in a particular way by setting the value of its configuration attributes. (Levine, Schmidt & Wang, 2000; Object Management Group, 1999, March; O'Ryan, Schmidt & Wang, 2000)

Components also have a single reference that supports an interface called the component's *equivalent interface*, which is an interface that conforms to the component definition. There are two forms of clients that are supported by the CORBA component model: component-aware and component-unaware clients. The component-unaware clients do not know that they are making a

request to a component, and can therefore only invoke functions supported by an ordinary CORBA object. The component-unaware clients can not use the ports, but can invoke operations via an object reference to the component's equivalent interface. The equivalent interface can inherit from other interfaces, called the component's *supported interfaces*. The equivalent interface also provides the possibility for component-aware clients to navigate among the component's facets and to connect to the component's ports. (Object Management Group, 1999, March)

The CCM divides components into four categories:

- **Service components**
  These components have the lifetime of a single operation request. They represent the most effective component form, since they are cheap to create and destroy, and consume resources only when active.

- **Session components**
  Session components can be called more than once, but they will not survive a server restart. These components can be used for functions that require keeping state during a client interaction, but do not require persistent storage, like for example iterators.

- **Process components**
  These components are intended to be used for representing business processes, like applying for a bank account. A process component exists during the whole business process, even if there is a server outage, and maintains its state from one invocation to the next. When the business process finishes the end result from the process becomes a product, like a bank account. So, the process component creates the product account component, and then frees up its resources and vanishes.

- **Entity components**
  The entity components represent truly persistent items in an application, like customers and bank accounts. Usually, they represent data in a database. For entity components, a key is assigned to every component instance, to help in retrieving specific instances. (Brooke, Pharoah & Siegel, 2000; Levine et al., 2000; Object Management Group, 1999, March)

Component Home

A component home manages instances of a specified component type. Its interface provides operations to manage component life cycles (create, destroy) and to find component instances for the type it manages. At execution time, a component instance is managed by a single home object of a particular type. The operations on the home are comparable to static class methods in object-oriented programming languages. All the operations that the developer might want to define on the set of instances of a particular component type, should also be placed here. (Object Management Group, 1999, March)

To use service and session components, a client will invoke the operation for creating a new instance. The component home will return a reference to the client, that will be valid for a single use for the service component, and for repeated use for a limited time for the session component. For using process and entity components, a client may use the create-operation, but will usually want to access already existing components. For accessing an existing entity component, the

client will use the find_by_primary_key()-operation on the component home. Process components do not have primary keys, and therefore the client will have to store their references itself for future use. References may be stored in a client cookie or a Naming or Trader service. (Brooke et al., 2000; Levine et al., 2000)

Container

A container provides the server with a run-time environment for a CORBA component implementation. Each container manages one component implementation and connects it to other components and ORB services. A container is a framework for integrating transactions, security, events and persistence into a component's behaviour at runtime. (Object Management Group, 1999, March)

The *CCM Container Programming Model* (see figure 6.16) defines a set of interfaces, that simplify development and configuration of CORBA applications. There are two categories of interfaces: external interfaces and container interfaces. The external interfaces are interfaces that are available to clients, like supported interfaces, provided interfaces and the component home interface. The container interfaces include internal interfaces and callback interfaces. The internal interfaces can be used by the component to invoke the services provided by the container. The callback interfaces can be used by the container, to invoke services on the component. These interfaces must be implemented by the application developer. They carry out functions necessary to allow the server objects to be activated and deactivated by the container as it manages server resources. (Brooke et al., 2000; Levine et al., 2000; Object Management Group, 1999, March).

The container architecture is a server-side framework, which is built on the ORB, POA (Portable Object Adapter) and a set of CORBA services for transactions, security, events and persistence. A component server is a process which provides an arbitrary number of component containers. The container framework manages the interactions with the ORB, POA and CORBA services, and lets the component developer concentrate on the application logic. The major functionality handled by this framework include creating object references (with the POA), giving access to Home operations, and interacting with the Transaction, Security, Persistence and Event services. (Object Management Group, 1999, March)

There are six container categories in CCM. Four categories correspond to the component categories; service container, session container, process container and entity container. Two categories handle EJB components: the EJBSession container and the EJBEntity container. The last category is called the Empty container and makes all CORBA interfaces available to a component's implementation without restriction. This gives the programmer the possibility to define an own container, and to use any combination of the CORBA services. (Object Management Group, 1999, October)

Transactions

The container provides access to a transaction processing system. The CCM supports both container-managed transactions (which is the simpler form to program), and self-managed transactions. (Brooke et al., 2000)

Security

The CCM provides a secure environment, which may be controlled via a component configuration file. Access permissions can be defined for any of the component's ports as well as the component's home interface. (Brooke et al., 2000)

Persistence

The container provides access to the CORBA Persistent State Service (PSS) for persistence support. There are two modes of PSS operation support; the container-managed persistence model, where the saving and restoring of an object's state over a deactivation/activation cycle is transparent to the programmer, and the component-managed (self-managed) persistence, where the programmer must save and restore state explicitly before deactivation and during activation of the component. (Brooke et al., 2000; Object Management Group, 1999, March)

Events

The container also controls access to event channels, by using a simple subset of the CORBA notification service. Operations for emitting and consuming events are generated from the specifications in component IDL. The container is responsible for mapping these operations to the CORBA notification service. (Brooke et al., 2000)

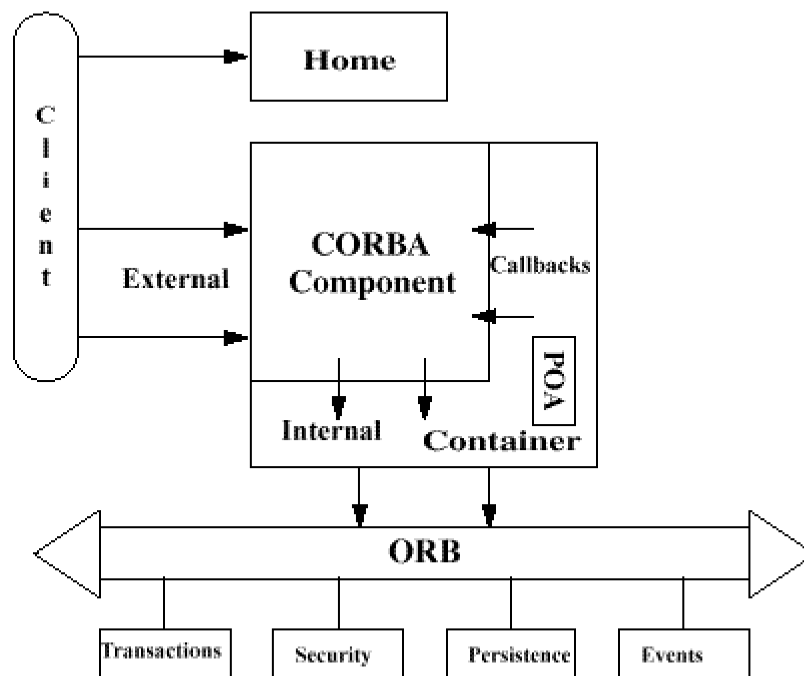


Figure 6.16 Container Programming Model (Object Management Group, 1999, March)

Component Levels

There are two levels of components: basic and extended. The functionality described above applies to extended components. Basic components have a subset of the functionality of extended components. They do not offer facets, receptacles, event sources and sinks (only attributes). They may use transaction, security and partly use persistence, but not the event model. The basic component is defined to be functionally equivalent to the *EJB 1.1. Component Architecture*, to allow easy mapping between CORBA and EJB components. (Object Management Group, 1999, October)

Component Implementation Framework (CIF)

CIF is a framework that defines the programming model for constructing component implementations. It uses a declarative language, *Component Implementation Definition Language (CIDL)*, for describing the structure and state of component implementations. From these CIDL definitions, a component-enabled ORB product should automatically generate implementation skeletons, that automate some basic behaviours of components, such as navigation, activation, lifecycle management, transactions and security, and that can be extended by the developer to create complete implementations. (Levine et al., 2000; Object Management Group, 1999, March)

Open Software Description (OSD)

Component implementations may be packaged and deployed. Open Software Description is an XML vocabulary for describing software packages and their dependencies. CCM uses an extended version of OSD to support component packaging and deployment. The deployment mechanism allows remote installation and activation of a new or modified component. (Levine et al., 2000; Object Management Group, 1999, March)

Resource Management

At runtime, a CCM application will consist of a set of component homes, which create component instances when required. CCM uses a pattern for resource allocation, called *activation-per-invocation*, that can be described by an example. Suppose we have an e-commerce application that consists of customer components (entity components), shopping cart components (session components), and checkout components (service components). For every customer in the database, there is a customer component instance in our application. There will be a shopping cart component instance for every customer who happens to be shopping at the moment. The checkout components are created when the customer presses the "check out"-button, perform their functions and disappear as soon as they are done. Let's suppose we have several millions of customers in our database. Then, it would not be efficient to have every customer and all present shopping cart instances in memory at the same time. Therefore, the CCM runtime manages the instances, by activating an instance when an invocation is made, and deactivating it (and freeing its resources) as soon as the operation is completed. This

activation/deactivation is invisible to the client application. Even if a customer component is not always running, it is always available to the client. (Levine et al., 2000)

CORBA Scripting Language Specification

The Scripting Language Specification is a specification for how to enable the composition and assembly of components using scripts. The specification does not specify a specific syntax, since it should be general enough to encompass several object-oriented scripting languages. (Object Management Group, 1999, August)

CORBA messaging

CORBA 2.0 provided three techniques for invoking operations:

1. Synchronous – The client invokes an operation and blocks waiting for the response.

2. Deferred synchronous – The client invokes an operation, then continues processing. It can later go back and collect the response.

3. One-way – The client invokes an operation and then continues processing. There is no response.

Since synchronous invocation techniques tend to tightly couple clients and servers, CORBA has been criticized for not being suitable for large distributed systems. For this reason, a new specification (the CORBA Messaging specification) has been adopted, that adds two asynchronous request techniques:

1. Callback – With each request, the client supplies an additional object reference parameter. When the response arrives to the ORB, the ORB uses this object reference to deliver the response back to the client.

2. Polling – The client invokes an operation, which returns an object that can be queried at any time to obtain the status of the request. (Vinoski, 1998)

Objects by value

CORBA 2.0 lacks support for passing objects by value, instead of passing a reference between CORBA applications. This has led to the addition of a new construct to the OMG IDL, which is called the *valuetype*. A valuetype supports both data members and operations. When a valuetype is passed as an argument to a remote operation, a copy of it will be created in the receiving address space. The identity of this copy is separate from the original; once the parameter passing operation is complete, there is no relationship between the two instances. Operations invoked on valuetypes are local, and unlike ordinary CORBA object invocations, they do not involve the

transmission of requests and replies over the network. The "Object-by-value"-concept was introduced in CORBA 2.3. (Vinoski, 1998)

CORBA 3.0 Firewall Specification

This is a specification, which provides a standard for controlling IIOP traffic through network firewalls, to allow outside access to CORBA applications. It defines interfaces for a firewall to identify and control the flow of IIOP traffic. It allows CORBA objects managed behind the firewall to have operations invoked on them from the outside world, permitting access to some inside CORBA-based application services and preventing access to IIOP-based services that should not be accessible from the outside. (Object Management Group, 1998)

Implementations

CORBA is an open standard and anyone can implement it without having to obtain a license from the OMG or anyone else. There are several CORBA-compliant products available in the market from different vendors, for example the following three products, which are available for a large number of platforms and support Java and C++ bindings:

IONA's Orbix

Orbix 2000 is fully compliant with CORBA 2.3, including POA and Object by value. It supports parts of the CORBA 2.4 and CORBA 3.0 specifications, for example asynchronous messaging. (IONA, 2001)

Inprises's VisiBroker

VisiBroker 4 is also fully compliant with the CORBA 2.3 specification and partly supports the CORBA 2.4 and CORBA 3.0 specifications. (Inprise, 2001) VisiBroker is also embedded in other products, for example in the Netscape Communicator browser.

BEA Systems' WebLogic

WebLogic Enterprise 5.1 supports CORBA 2.2 and includes support for some CORBA 2.3 extensions. (BEA Systems, 2000)

Mapping to COM and EJB

The CORBA specification includes a chapter, which describes the data type and interface mapping between CORBA and COM (OMG IDL vs. Microsoft IDL). The mapping was designed to be implemented by automated interworking tools. This means that there will be an implementation of a bridge between the CORBA and COM system. Another way to access COM objects from CORBA is to use a wrapper, which contains the required OMG IDL to expose the COM object. But this requires development and maintenance of a wrapper server object for every COM object and should only be used if there are a few COM objects that need to be accessed. (Object Management Group, 2001)

The CORBA component model is based on the Enterprise Java Beans specification, extending the EJB component model with some additional features, such as a tighter integration with the CORBA object model. One of the design goals of the CORBA Components Model was compatibility and interoperability with Enterprise Java Beans, to make it possible to form integrated applications from EJB and CCM components. The CCM specification makes it possible to make a CORBA Component appear as an enterprise bean, and to make an enterprise bean appear as if it were a CORBA component. One chapter in the CCM specification describes the mapping between CORBA and EJB in detail. (Object Management Group, 1999, March)

How to develop a CORBA object, step by step

Here is a short step by step overview of how to develop a CORBA object and a client that can make a request to the object. In this example, the Naming Service is used, but there are other ways to retrieve an object reference, as described earlier.

Server development steps

1. **Write an IDL file that describes the object's interface.**
   Specify the object's interface including its attributes and operations.

2. **Run the IDL compiler.**
   Choose the programming language that you will use to implement the object and run the IDL compiler to generate the language-specific implementation files, client stub files and server skeleton files.

3. **Write the implementation of the object in the chosen language.**
   Many IDL compilers generate a file that contains some of the code required to be a CORBA remote object, which can be extended by the programmer.

4. **Write a server main program that acts as a server for the object.**
   This program shall perform the following steps:
   - *Initialize the ORB and the object adapter*
     This means obtaining a reference to these objects.

- *Create an instance of the object*
- *Tell the object adapter to connect the new object to the ORB*
- *Find the Naming Service and assign the object a name*
  This step is optional. It makes it possible for the client to find the object, by knowing the object's name.
- *Wait for client requests*

5. **Compile the code.**
   Compile all the generated code and the application code with a compiler for the chosen language. (Benjamin et al., 1999)

Client development steps

1. **Write a client application that invokes a request on the remote object.**
   Choose which language to use for the client application. Then write a client application in the chosen language, that performs the following steps:
   - *Initialize the client-side ORB*
   - *Find the Naming Service and look up the object*
     This is one possible way to get an object reference. The client sends the name of the object as a parameter, and gets an object reference in return.
   - *Make a request to the remote object*
     Finally, the client makes an invocation on one of the object's operations.

2. **Compile the code.**
   Compile the client application code with a compiler for the chosen language. (Benjamin et al., 1999)

6.2 Other techniques

Today, the software component market is rapidly developing, and thereby its related integration techniques. There are some techniques under development, which might become key techniques in the future. Here is a brief overview of three such techniques.

6.2.1 Jini

Jini is an object-oriented distributed architecture from Sun, which offers network “plug-n-work”. It provides an infrastructure for delivering services to clients in a network. A Jini system consists of services that can be collected together for the performance of a particular task. Services can use other services, and a client of one service may itself be a service for other clients. Services can be added or removed from the network, and clients can find existing services, without any extra administration needed. Jini is written in Java, but clients and services are not constrained to be in pure Java. They may include other programming languages, and have a wrapper written in Java that communicates with the rest of the system. (Newmarch, 2000; Sun Microsystems, 2000b)

Fundamental parts of a Jini system

Service

A service is an entity that can be used by a client. Services can be devices, like printers or disks, software applications, databases or files. When a new service is plugged into a Jini network, it advertises itself by publishing an object written in Java, that implements the service API, which indicates what functionality the service provides. The protocol used for the communication between this object and the service implementation, can be chosen by the programmer, and can differ between implementations of the same service without changing the client code. (Newmarch, 2000; Sun Microsystems, 2001)

Client

An entity that wants to make use of a service. (Sun Microsystems, 2000b)

Lookup service (Service locator)

The lookup service acts as a locator between services and clients. A lookup service maps service objects to the service implementation. There are also descriptive entries associated with a service to allow the selection of services, based on properties understandable to people. (Sun Microsystems, 2000b)

Network

The network connects services, clients and lookup services. Although the Jini specification is independent of the protocol used for network communication, the only current implementation is based on the TCP/IP-protocol. (Newmarch, 2000)

Architectural Overview

There are three categories of components in a Jini system, that are the building blocks of the Jini architecture: the *infrastructure,* the *programming model*, and *services*.

The **infrastructure** is the core of the Jini technology, and provides the resources that are needed for executing Java programming language objects, for managing communication between these objects, and to make it possible to find and use different services in the network. Jini is based on the Java programming language, and the environment provided for executing objects is the Java Virtual Machine. Java RMI is used for communication between objects across the network. A distributed security system is integrated into RMI and defines how entities are identified and given the right to perform actions in the system. The infrastructure also includes the look-up service, which offers and finds services for clients, and discovery/join-protocols, which define how a service of any kind becomes part of a Jini system. (Sun Microsystems, 2000b)

The **programming model** supports the production of reliable distributed services. It consists of a set of interfaces, for example the *leasing interface*, the *event* and *notification* interfaces and the *transaction* interface. The leasing interface defines a way of allocating and freeing resources. A lease is a contract of guaranteed access to a service over a time period, which is negotiated between the user of the service and the provider of the service. Leases can be exclusive or non-exclusive. An exclusive lease means that only one user has the lease during the agreed upon time. If a lease is not renewed at the expiration date, then the lookup service will remove the entry for it, and the service will no longer be available. The event and notification interfaces enable event-based communication between services. An object may allow other objects to register their interest in events and receive a notification of the occurrence of such an event. The transaction interfaces supply a service protocol, that helps coordinating state changes of objects, in such a way that all or none of the changes to the group of objects are performed. (Sun Microsystems, 2000b)

Entries in the lookup service are leased, which ensures that the lookup provides an accurate view of currently available services. When a service joins or leaves a lookup service, an event is emitted and those objects that have registered interest in such events get notifications. (Sun Microsystems, 2000b)

Services can be made part of a Jini system and offer functionality to clients. Services appear as objects written in Java. A service has an interface, which defines the operations it provides to clients. Some services provide a user interface for direct user interaction; others only provide interfaces to other applications. (Sun Microsystems, 2000b)

Service Registration

A service will be an object (or set of objects), which is living within a server. To make a service available to potential clients, the server must register the service with the lookup service (service locator). This is done in the following steps (see figure 6.17):

1. The server finds a lookup service. If the server knows the location of the lookup service, the server can make a direct request to it, using *unicast* TCP (a specific destination is defined). If it is not known, the server can make a *multicast* UDP request (delivers request to all receivers associated with the multicast group), which will make all available lookup services respond.

2. The request to the lookup service, will cause the lookup service to return an object, called a *registrar*, to the server. The registrar acts as a proxy to the lookup service, and will run in the service's JVM. All further requests that the server needs to make to the lookup service, will be made through this registrar. Usually, the protocol used for this is RMI.

3. The service will then send a copy of the service to the service locator, by making a request to the registrar. (Newmarch, 2000)

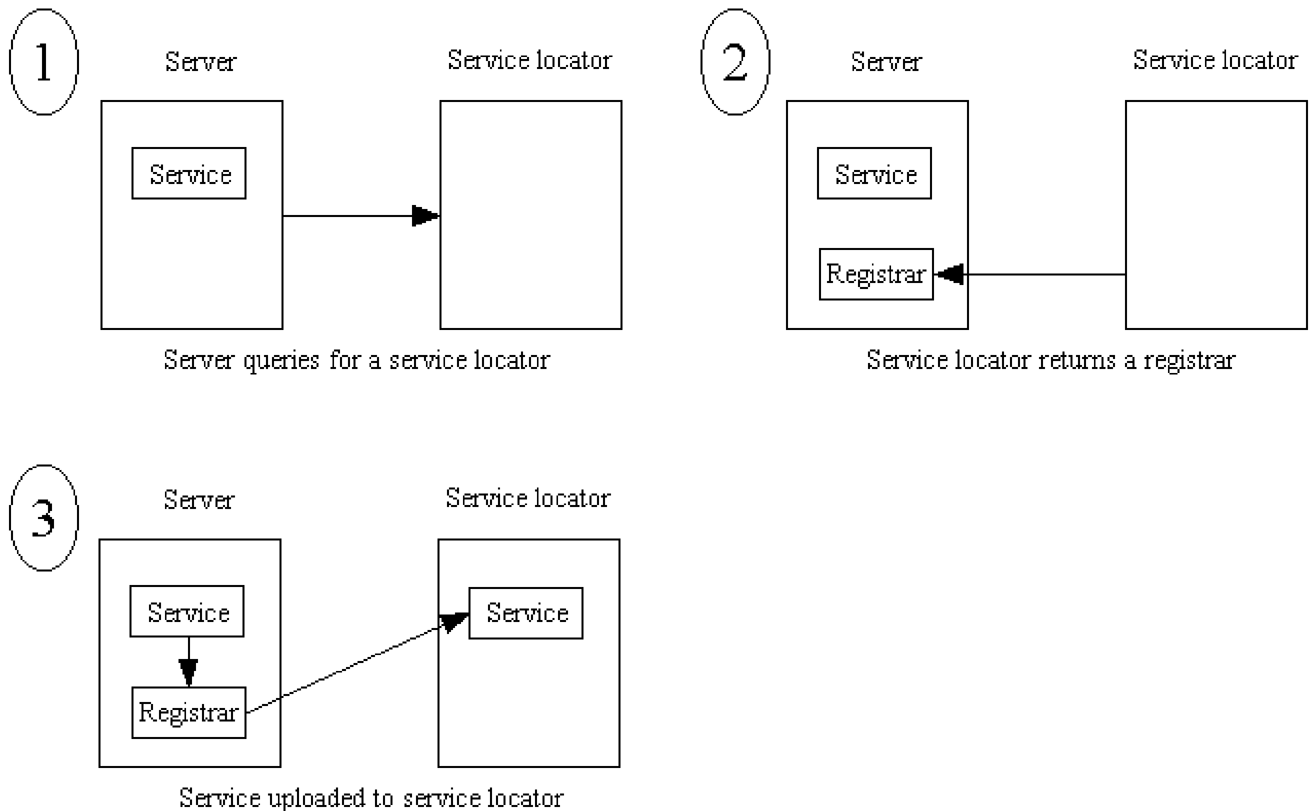


Figure 6.17 Service Registration

Client Lookup

A client who wants to use a service, will have to get a copy of the service into its own JVM. This is done in the following steps (see figure 6.18):

1. The client finds a lookup service, with the same mechanism that the server did.

2. The lookup service returns a registrar, that will be used for the client's further communication with the lookup service.

3. The client requests a copy of the service to be created and distributed to itself. The request is made through the registrar.

4. The service locator returns a copy of the service to the client. (Newmarch, 2000)

When the service locator has returned a copy of the service to the client, the client can make requests of the service object running in its own JVM. If the service provided is a piece of hardware, like a printer, or if the service is controlling some hardware, the copy of the service will not be a real copy of the original service. It will be a *proxy*, which will represent the service, by receiving requests from the client and communicating back to the service, usually using RMI. For the proxy service to find its service, the proxy will be "primed" with its own service's location when it is created. Exactly how this is done is not specified by Jini, and can be done in several ways. (Newmarch, 2000)

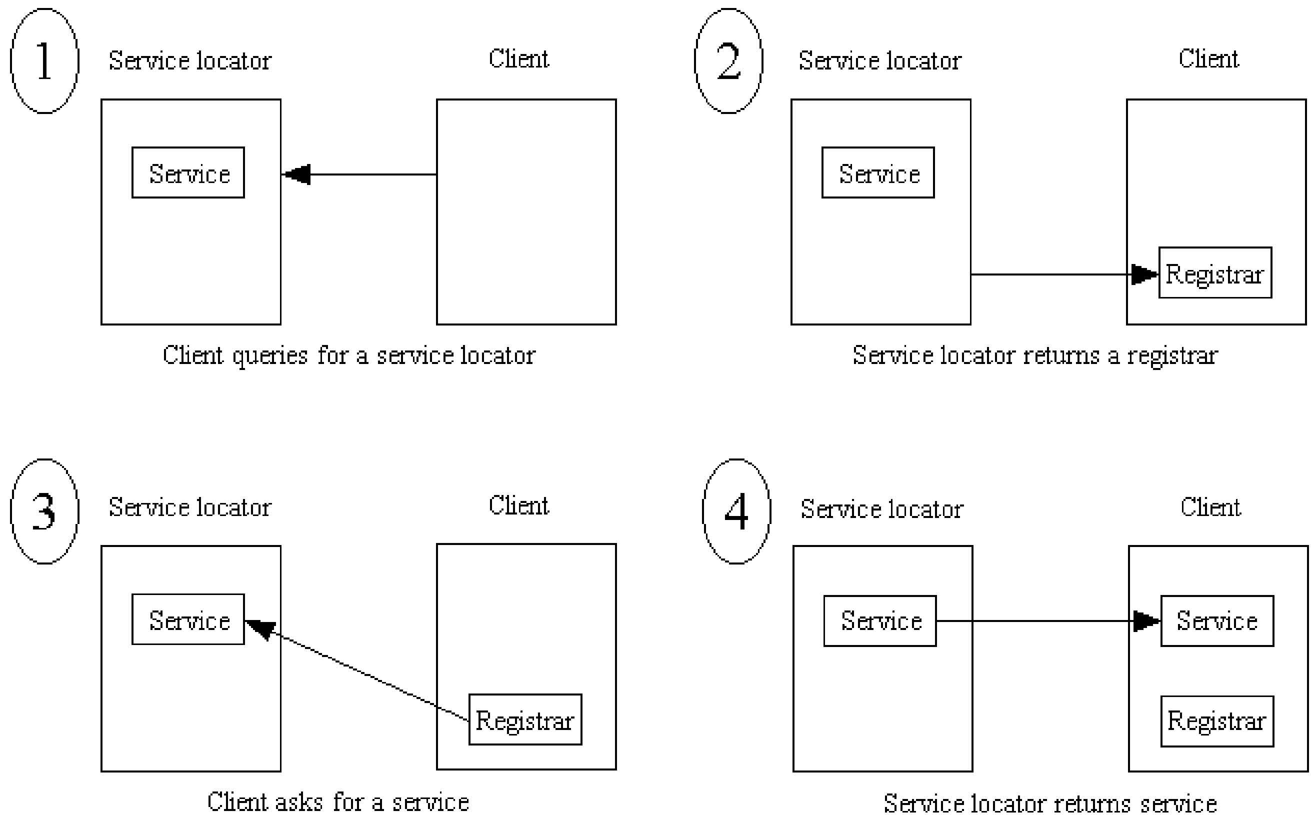


Figure 6.18 Client lookup

If there is a situation, where the client cannot find any service locator, there is one more possibility. It can use *peer lookup*, which means that the client sends out the same identification packet that a lookup service uses to request servers to register their services. Then the servers will attempt to register with the client, like they would do with a lookup service. The client can now choose the services it needs from these requests, and discard the rest. (Sun Microsystems, 2000b)

6.2.2 Message Broker

A message broker is middleware that acts as a broker between one or more entities such as network, middleware, applications and systems, to let them easily share information.

Traditional middleware architecture build on point-to-point links between applications, which means that adding one more application requires a direct connection to all other application with which it will communicate. (Linthicum, 1998)

Message brokers usually use a hub and spoke-type architecture (see figure 6.19). In this architecture the message broker acts as a hub, which connects the applications to each other. To connect a new application, a new connection is only made between it and the message broker. (Linthicum, 1998)

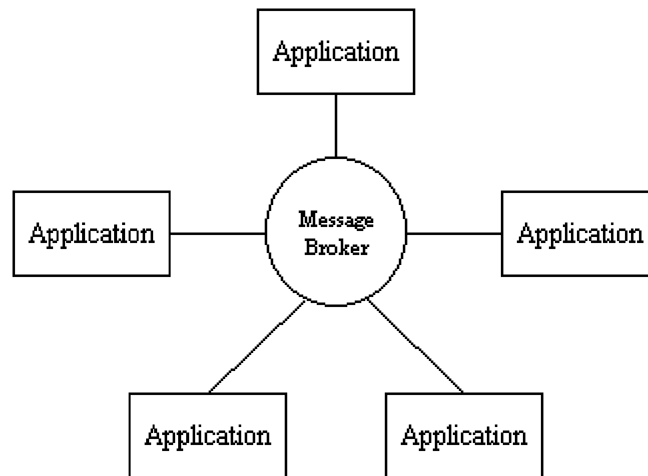


Figure 6.19 Message broker with hub-and-spoke architecture

The message broker connects applications that may be written in different programming languages and reside on different platforms, by routing messages between them, and providing functionality for translating and reformatting messages. Message brokers use asynchronous messaging. Applications put messages to the message broker and other applications get the messages. (Linthicum, 1998)

A message broker's major purpose is to constitute a central integration point and to translate messages and deliver them to the target applications. The message broker could be seen as a processing layer between different applications and platforms, which takes care of differences in application languages, data formats, and operating systems. By making all necessary conversions within the message broker, required changes to the source and target applications are kept at a minimum. (Linthicum, 2001a)

Message brokers can be seen as "middleware for middleware", since it builds on existing middleware technology, most often on messaging middleware. When using traditional middleware, there are components for routing, reformatting and co-ordination of messages that are placed in the applications or in the middleware. These components are here instead placed centrally in the message broker. (Linthicum, 1999)

A message broker should ideally be "any-to-any" and "many-to-many". Any-to-any means that it must be easy to connect different applications and other resources and the developer should be provided with a consistent interface to the different resources. Many-to-many means that when a resource has been connected to the message broker, any other application should be able to use it. (Linthicum, 1999)

Main elements

Message broker are sold by many vendors and have different options and styles. The term message broker is rather an architectural concept, than an off-the-shelf product. The core services that are provided by most message broker products are the following.

Message Queuing

A message broker is built around a message queuing system, which has the ability to accept, process and pass messages. Some message brokers provide proprietary message queuing components, while others support different industry-standard products, like MQSeries from IBM, Java Messaging Service from JavaSoft or the Object Messaging System from OMG. (Beveridge & Perks, 2000; Linthicum, n.d./2001a)

The message queue provides asynchronous, store-and-forward messaging. Store-and-forwarding means that the message broker stores messages until conditions are right to forward them to the recipients. If a connection fails during message transfers, message stores can recognize the failure and retransmit the message when the connection is re-established. (Linthicum, n.d./2001b)

Message transformation

Most message brokers have a message translation layer, which understands all the different formats of the messages that are passed among the applications and changes their formats. It contains a dictionary that keeps information on how each application communicates and which information is relevant to which applications. (Linthicum, 1999)

There are two ways of transforming messages: schema conversion and data transformation. When two different application communicate, the message broker converts the message's data formats so they are interpretable by each application. This is achieved with schema conversion, which means changing the format of a message so that it is interpretable by the target system. Data transformation means simple conversion of data like aggregation or using lookup-tables. (Linthicum, 1999)

Most message brokers break the incoming message into a common format and then translates it into an appropriate message format for the receiving system. This is performed by a rules-based engine. The reformatting of the messages is defined by the programmer by using an interface provided by the message broker. It could be an API or a more user-friendly GUI. The message broker stores application information in a repository that keeps track of the system where the message is coming from, the message's format, the target system, and the message format that the target system expects. (Linthicum, n.d./2001b)

Message brokers are often able to understand a message schema through message identification, so it can automatically convert data to another format. Sometimes there will be a need for the programmer to define a rule for a specific data conversion, for example when converting between numeric and alphanumeric data. (Linthicum, n.d./2001b)

Intelligent routing

Message brokers should have a rules-based engine that implements intelligent routing and transformation. Intelligent routing means using predefined rules to dynamically route messages to different applications. (Linthicum, 1999)

This means that the programmer is allowed to define how a message should be transformed and routed, depending on the format and content of a message. By knowing that a message with a certain format comes from a specific application, the programmer can define a rule, which tells the message broker that these messages shall be transformed according to some specific rules and routed to a specific target application. (Linthicum, n.d./2001b)

Message warehousing

A message warehouse is a database that can store messages that passes through the message broker. By providing this facility, messages will not be lost at a server break-down, since the warehouse acts as a persistent buffer or queue. The database can also be used for message mining, to extract business data for supporting decisions. (Linthicum, 1999)

Repository services

A repository is a database that contains information about the target and source applications, such as data elements, the input, output and relationships between applications. The message broker uses these definitions to parse and process messages and for managing the message flow through the broker. (Beveridge & Perks, 2000; Linthicum, 1999)

Directory services

A mechanism that is used to locate the distributed applications and systems, which are connected to the message broker. The directory service provides the location, identification, use and authorization of network resources, such as source or target application. It supports using a shared set of directory and naming service standards. (Linthicum, 1999)

Adapters

An adapter is a layer between the message broker interface and the source or target application, which adapts the differences between the application and the message broker's native interfaces. It hides the complexity from the user and from the EAI developer. The adapters hide the application's interfaces from the user and lets the user deal with the simplified interfaces instead. For example, a message broker may have adapters for several different major source and target applications (like SAP), or adapters for certain types of databases (like Oracle or Sybase). It could also have adapters for specific brands of middleware. Most message brokers offer software development kits to build new adapters. (Beveridge & Perks, 2000; Linthicum, n.d./2001a)

There are two types of adapters for message brokers, thin and thick adapters. Thin adapters map the interface of the source or target system to a common interface that is supported by the message broker. Thick adapters also contain some functionality, to create an interface against the applications to be integrated that requires almost no programming. Thick adapters use the repository services to get information of source and target applications for interacting with them. (Linthicum, n.d./2001a)

Adapters can also be classified as static or dynamic adapters. Static adapters must be manually coded with the schema for the applications. If the connected application's schema change, static adapters will have to be manually changed by the developer. Dynamic adapters use a discovery process when they are first connected to an application, to find out their message schemas. It might involve looking in the repository and reading application source code. If later, something changes in the message format for an application, a dynamic adapter will automatically understand these changes. (Linthicum, n.d./2001a)

Development environment

Message brokers may provide development tools to aid the creation of adapters. All message brokers provide an API to let the applications access the message broker and to put and get messages from the message queuing component. (Beveridge & Perks, 2000)

Platform services

Many message brokers are not completely self-contained. They require, or provide an option for integrating with an organization's standard platform services, like systems management, security, naming and addressing, communications and component-based services. (Beveridge & Perks, 2000)

6.2.3 Intelligent Agents

There has been some research in the area of using intelligent agents for integrating heterogeneous software systems, and here two examples shortly will be described of how agents can be used for this purpose.

There is no single universally accepted definition of what an intelligent agent is. One of the more commonly used definitions is the Wooldridge and Jennings definition which says that an agent is *"... a hardware or (more usually) software-based computer system that enjoys the following properties:*

- *autonomy: agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state;*
- *social ability: agents interact with other agents (and possibly humans) via some kind of agent-communication language;*
- *reactivity: agents perceive their environment, (which may be the physical world, a user via a graphical user interface, a collection of other agents, the INTERNET, or perhaps all of these combined), and respond in a timely fashion to changes that occur in it;*
- *pro-activeness: agents do not simply act in response to their environment, they are able to exhibit goal-directed behaviour by taking the initiative."* (Jennings & Wooldridge, 1995)

Agent Adapters

The agent adapter architecture consists of a rules-based agent host and adapter components. The agent host is an agent that functions as a host process for the application adapters and mediates between the applications. Each adapter forwards messages from and to an application, and is able to receive notifications from the application. (Yee, 1999)

The agent adapters differ from the message broker adapters in functionality. In the message broker architecture, adapters are responsible for interfacing with the applications. The message broker handles all the validation and transformation of data, and the adapters only pass messages without noting the contents of the messages. In the agent adapter architecture, the adapters are also responsible for applying filters and performing validation of the data. By applying filters at the node level instead of the hub level, less data will have to be sent over the network. By removing much of the complex processing from one central point, scalability can also be improved. (Linthicum, 1999; Yee, 1999)

An agent adapter is also responsible for identifying the type and version of the application it is responsible for and for applying the appropriate adapter component for integration. The agent host is responsible for application session management, resource pooling, and managing state on behalf of the adapters. Managing state is important for adapters that are interfacing to transactional systems. (Yee, 1999)

Agent adapters can also use point-to-point interaction, which means that one agent directly initiates another agent adapter to perform a task. This could be useful when there is a task that is of little or no interest to other applications. This differs from the message broker architecture, where all communication is via the central broker. Facilities such as security, transformation, validation, and routing are available to each agent adapter. The agent adapter registers for the services it is interested in at startup. A central repository manages information on each application/agent adapter and information about the relationship of the information flow between individual agent adapters. (Yee, 1999)

Intelligent Components

Another way to use agents for integrating heterogeneous systems is to take component-based software development one step further: to create intelligent components. Agent-based software development is similar to component-based development in many respects, but could lead to even more flexible componentized systems, according to Martin Griss (2000).

An existing component could be turned into an agent by placing wrapper software around the component, that presents an agent interface to the other software components and which maps outside agent calls to the legacy code and vice versa. (Jennings, 2001)

Griss describes an agent system reference model, that constitutes an architecture for agent components. Just like the component frameworks include support services such as naming and transaction support, the agent framework also supports these kinds of features.

The agent system architecture consists of the following parts (see figure 6.20):

- The *Agency*, which is the local environment (process or server) that provides basic management, security, communication, persistence, naming and transportation of agents.

- The *Agent Management System*, which manages creation, deletion, migration, etc. of agents and provides a service for naming and locating agents by name.

- The *Directory Facilitator*, which supports finding agents by their offered services.

- The Agent Communication Channel, which routes messages between local agents and agents located in other agencies.

- The *Agent Platform*, which provides the communication and naming infrastructure using the Internal Platform Message Transport. (Griss, 2000)

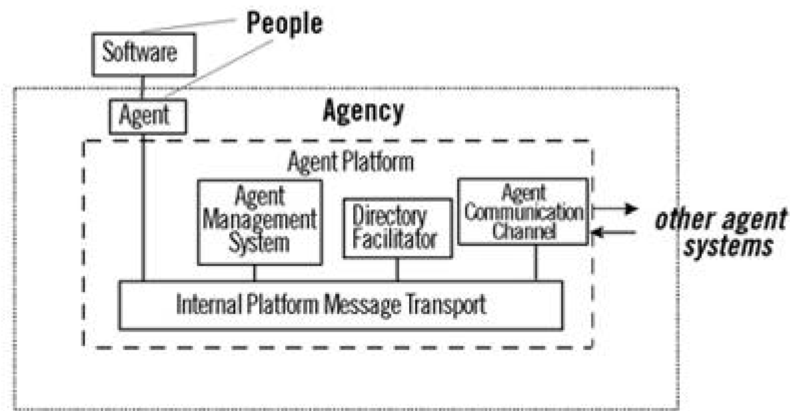


Figure 6.20 Agent System Reference Model (Griss, 2000)

The intelligent components should have a simple interface, that is similar to all components, but allows messages to have a complex structure. This means that instead of having to change the framework and redefine the interfaces as the system evolves, the messages can be dynamically extended. (Griss, 2000)

In agent-based systems, the interaction between the agents usually occurs through a high-level agent communication language. The interactions are conducted at the knowledge level, in terms of which goals should be followed, at what time and by whom, instead of operating at a pure syntactic level, as in the case of method invocations. (Jennings, 2001)

Agents are able to make context-dependent decisions about their interactions. For a complex system it is very difficult to predict at design time, all interactions that will have to take place between its various components. In agent-based systems the components have the ability to make decisions about the nature and scope of their interactions at runtime. Each agent has its own thread of control and has control over its own actions. This leads to a system consisting of autonomous components that can act and interact in flexible ways to achieve their objectives. (Jennings, 2001)

7 Interviews

We have performed four interviews with persons who have had direct or indirect contact with integration questions. One of them was a senior lecturer in Chalmers Technical University in Gothenburg, the second was a manager at Software Lab at Ericsson Microwave Systems in Gothenburg, the third person was a systems developer at Ericsson Microwave Systems in Skellefteå and the last two persons worked at a small company in Gothenburg with Microsoft solutions for B2B (Business To Business). Each interview took about one hour and was semi-structured. Here is an abstract over these interviews. The interview questions, which were formulated as open questions, can be found in the Appendix.

7.1 Interview No. 1

The first person, the senior lecturer in Chalmers had not personally worked with any integration project but is well informed about distributed systems.

7.1.1 Integration Difficulties and Principles

The most difficult problem in an integration project, according to him, is that there is often not enough documentation or knowledge in the organization about the legacy systems. Or the documentation is very difficult to understand. There is often one or a number of such systems that will need to be integrated with the new ones in an integration project. Legacy systems could often be integrated with help of middleware and using standardized interfaces. He meant that well-defined and “clean” interfaces are very important. He thought that design patterns are a good approach for integrating components and mentioned the “facade” and “proxy” as useful patterns for this. He also commented that design patterns do not need to be object-oriented, but they have become popular within the area of object-oriented development. He said that the purpose of design patterns is to put names on general solutions to certain problems, to provide developers with a common language.

He meant that there is nothing such as an optimal integration, since he did not think that there is one solution to all problems. There will always be different systems with different specifications and an optimal integration depends on the specific integration problem.

7.1.2 Component-Based Software Development

He was convinced that component-based software development has a bright future. As an example, he said that twelve years ago it was very difficult and costly to develop a graphical user interface or a simple database. Today it is easy and relatively cheap to do this, because we can buy components from different suppliers for this (database management systems and libraries for GUI-development). It is also relatively cheap to maintain those components, because when we upgrade them we just buy

an upgrade from the supplier. The supplier can keep low prices, since there are many customers sharing the costs for development and upgrading. The same thing applies for "in-house" component development: the components can often be used in different parts of the organization and then the development costs can be shared.

He also said that when buying components, there are some important factors to consider, except for the technical aspects, for example what position in the market the supplier has. It could be dangerous to become dependent on components from a supplier with just a few customers. Then, there is a risk that the supplier will disappear and no other company will be interested in upgrading the components.

He gave the following definition of what a component is: a program part, which is isolated from other program parts and can be accessed through a standardized interface. Usually components are executable, but a more primitive form of components could be non-executable. He meant that a database management system is a good example of a component and that a more primitive form of components are libraries for developing graphical user interfaces, which become harder coupled to other program parts.

He pointed out that this was his interpretation of the word component, and that it is a general problem that there is no commonly accepted definition for this concept.

7.1.3 Integration Techniques

On our question whether we can trust that the different integration techniques can handle what they promise, he meant that in the computer world, you could never trust anything, especially when we are talking about new techniques. He supposed that the best thing is to wait, to see different comments about new techniques and use them first when they are well established on the market.

He did not know which of the techniques (COM, CORBA, EJB) was the most complex to learn. He had had students working with Enterprise Java Beans, and he knew that it took some time for them to become familiar with this technique and to be able to use it.

He also said that CORBA has the advantage of giving a language-neutral standardized way of defining interfaces between servers and clients. COM is a Microsoft standard and not useful outside the Microsoft world.

He had not heard anything about message brokers or JINI but thought that XML was an interesting new technique for integration, since it defines both information and how the information should be structured, which means that the receiver can be unaware of how the information should be structured.

7.2 Interview No. 2

The second interview was with the manager at Software Lab at Ericsson Microwave Systems. This person was working with research about software products that will be introduced to the defence within 10 - 20 years. Software Lab co-operates with other big companies in the world. They work

with middleware technologies such as CORBA. They have had focus in their work to create an independent platform, which makes it easy to use different heterogeneous products on one platform.

7.2.1 Integration Difficulties and Principles

The manager thought that the most important part of an integration project is specifying the interfaces carefully. The trend is that different parts of systems are not all developed simultaneously. There will always be legacy systems.

He meant that middleware is a good pattern for an optimal integration and to use wrapping of legacy systems. A general problem in integration projects is the integration of components in different programming languages. He thought CORBA IDL is a good way to specify interfaces since it is language-independent.

He recommended design patterns for a successful integration project. He meant that design patterns are very useful; because when we talk about design patterns we are talking about actual problems, what kind of structure is valid in a specific problem and about the interplay between elements. Design patterns are very powerful for this, because when we say a name like “proxy”, everyone directly knows what we are talking about. At Software Labs, they have arranged courses, discussing the design patterns described by “the Gang of Four”, i.e. Gamma et al. (1995).

7.2.2 Component-Based Software Development

Regarding component-based software development, he thought it was interesting but did not see it as the future paradigm. He meant that people look at object-orientation in the same way, but after all, object-orientation is just another way to organize things. And even if it is a powerful way of organizing things, it does not help to activate or stimulate human creativity. Besides, the most interesting thing for us is not how components and objects are constructed and how they work. We just want to know what they do. We are interested in their services, and therefore he believed that the future paradigm will be based on loosely coupled services, where the user self can choose how to combine the services (service-oriented software development). At Software Labs for example, they have been working with the development of a system using voice recognition, where the user quickly will be able to request a service by giving a verbal command.

He defined a component as an underlying element that realizes a service.

7.2.3 Integration Techniques

He thought that EJB does not cover all the important areas for distributed components yet. He meant that CORBA 3.0 is a much better alternative, even supporting “real time” transactions. One problem with CORBA has been that many of the CORBA Services in the specifications have been difficult to implement.

He meant that both EJB and COM are very complex. The complexity depends to some degree on the development tool that is used. A disadvantage of COM is that it is owned and specified by Microsoft, as the only key player. The biggest problem with COM is its platform dependence. He mentioned that there had been a German company, trying to implement a COM-solution for Solaris, but it never worked very well. On the other hand, EJB is platform independent but language dependent (Java), which makes it difficult to combine them with each other. He meant that Sun has become a dominant player in the area, having EJB mapped to CORBA and being a key member of OMG.

He did not recognize the concept message broker, but he was familiar with JINI. He meant that JINI builds on the new service-oriented paradigm. For the user, components are not so interesting; it is the services that can be used that are interesting. JINI is based on this philosophy: when printing a paper with your computer it is usually not interesting to know what kind of printer you have. The interesting thing is the printer's service, i.e. the service of printing papers for you.

7.3 Interview No. 3

The third person worked as a systems developer at Ericsson. For the moment, he was working with an integration project where they were using CORBA. He had previously been working with management systems in Windows NT and had experiences of COM.

7.3.1 Integration Difficulties and Principles

When asked about which principles are the most important in an integration work, he said that making "everything work" is not easy. The biggest problem is often that you do not have well-specified interfaces. For example, he said, in his current project they were working with standards for interface specification, but these standards were themselves not clearly specified.

As weaknesses that could be harmful for an integration work, he pointed out the interoperability problems in CORBA. He meant that there were two such problems: the interoperability between different CORBA-implementations from different vendors and between different CORBA-versions. For example, there is a new data type in IDL, named valuetype in CORBA 2.3, which does not exists in previous versions (see the CORBA chapter for details). It means that there might be problems if using this valuetype and wanting interoperability with older CORBA-versions.

7.3.2 Component-Based Software Development

When asked about what he thinks of component-based software development, he said that he had worked with ActiveX-components. He meant that on the Windows platform, CBSD is almost in sole control and he definitely thought it has a future.

He defined a component as a binary, which you can use without having any knowledge about its contents and it can be integrated with help of tools (such as an ActiveX-component). A component has two types of interfaces: an interface for the component's functionality and another for

configuring the component, which gives you the possibility of managing and administering the component with tools. A component does not need to be executable by it self. For example an ActiveX-component does not execute by itself but you can place it in a container, which can make use of it.

7.3.3 Integration Techniques

We asked if the techniques like COM and CORBA keep what they promise. He said that for COM it is true that many programming languages can be used, but it is essential that the compiled code follows a specific binary format, which requires compilers specifically designed for that. Visual Basic, Microsoft's version of Java and C++ supports it.

He meant that it is practically possible to combine different integration techniques, for example creating a bridge between COM and CORBA. Such products already exist in the market, but of course it leads to a higher complexity level in an integration work, and should be avoided whenever possible.

We asked him about his experiences about flexibility and complexity. He thought that CORBA handles flexibility better than COM. In CORBA you can build operations that work as bridges between COM and CORBA. It can count as a benefit for CORBA if you need to integrate CORBA with COM. Besides, CORBA has several services such as naming service, event service, etc. There is not any equivalence for these in COM.

On the other hand many CORBA-implementations such as Orbix and VisiBroker do not have all these services, which are defined in the specification. The most implementations just have a few of these services. He said that in his current project, they used a third-party software from OpenFusion, which has all the CORBA-services.

Also, COM can only be used on the Windows platform, while CORBA can be used on several platforms, even if there are much more CORBA implementations for Windows than for Unix or any other platform.

He thought programming with COM is more complex and that it is easier for a beginner to learn CORBA. He also said that the manuals for both COM and CORBA are not easy to understand.

7.4 Interview No. 4

We had our last interview with two persons, in a company, which works with Microsoft solutions for B2B (Business To Business). They were almost only working with COM solutions.

7.4.1 Integration Difficulties and Principles

When we asked them about the weaknesses that exist in an integration work, they said that the most important and difficult thing in such as projects is describing standards for interfaces in a clear way and to make sure they are followed.

7.4.2 Component-Based Software Development

They both thought that component-based software development is something that will become a future trend. They meant that with the continuing development of the Internet and the increase of distributed computing, the use of components will also increase.

They said that they had never thought of how to define a component, but that it might be defined as something that performs a specific task, and is written in a specific programming language.

7.4.3 Integration Techniques

They mentioned that it was a disadvantage for COM that there is only one key player (Microsoft) that has the total control and sets the standard but they meant that after all, COM really works well in the Microsoft world.

Some times it is enough to divide a system's tasks in a number of servers instead of collecting all of them in one centralized mainframe. For example you can have customer information in one server, order information in another one, etc. They meant that it is not just better to build a system in this way but also with Internet, enterprises will automatically be, more or less, forced to re-design their systems in this way in the future, where enterprises will need to have different servers for different services, not just in different places in the company, but at different geographical locations in the world.

They thought CORBA is a working solution for this but that it is very complex to use and that it does not manage security aspects very well. They thought that CORBA is a good solution for intranets and local networks, but not for the Internet. Here, they thought, it is much easier to work with SOAP, which is a new technique for exchanging structured information in a decentralized distributed environment, using XML. But when large data volumes need to be distributed binary, they still thought that CORBA is a better alternative. In their projects, they use SOAP for integrating their customers' distributed environments with the customers' partners' environments, which might be based on other platforms than Windows.

SOAP sends information in a well-structured way. The sender just sends a packet of information and therefore there is not any heavy loading on the servers. The receiving computer takes care of unpacking the information and making any necessary computations.

8 Prototype

In this section, we will describe the objectives and requirements with our small prototype and the different phases of the development process.

8.1 Objectives

During our literature study we have become familiar with three different techniques for component integration, namely Enterprise JavaBeans, COM and CORBA. To better understand some of the more detailed descriptions of the techniques, we decided to develop a simple prototype with one of these techniques. The ideal approach would have been to develop a prototype with each of the techniques, but because of the time constraints we did not have this possibility. We also hoped that hands-on experience from a prototyping process, would give us a better understanding of the general problems involved in performing an integration task.

We decided to choose EJB, since that would give us a better understanding of the component-model of both EJB and CORBA (as we have mentioned earlier, their component-models are similar). At present, there is no implementation in the market that supports CORBA's component-model. COM is a less interesting alternative to Ericsson, since COM is platform-dependent.

8.2 Requirements

The task we were given was to develop a simple prototype for a portal that would make it possible for a network-operator's customers to log in with user-name and password and view their trouble tickets. A trouble ticket is a record of a problem that has occurred with a customer's network-connection. There are different standard categories of troubles that can occur.

From the beginning, the intent was that we would make a connection to an existing database at Ericsson, but these tables were difficult to interpret since they lacked understandable column names, had complex relationships and we did not have access to any documentation. Therefore, we decided to make a simple local database ourselves and define our own tables.

8.3 Tools/Installation

As development environment we have used Borland's JBuilder 4 on Windows NT. JBuilder is a tool that supports building applications in the Java language, applets, JSP/Servlets, JavaBeans, Enterprise JavaBeans and distributed J2EE applications for the Java 2 Platform. We used Borland AppServer 4.5 to act as a runtime environment for our Enterprise JavaBeans and we

used JDataStore 4, which came with JBuilder to manage our database. JDataStore is an object relational database management system[2].

We had some installation problems with JBuilder, because JBuilder's documentation did not explain all the configurations that needed to be done to make JBuilder find the application server's libraries. When compiling our first beans, we had strange compilation errors, which we did not understand had to do with an incomplete installation. Finally, we could solve this problem, with some help from a person who answered our question in one of Borland's user groups. JBuilder comes with the Java 2 Standard Edition SDK-libraries, but when developing Enterprise JavaBeans, the Enterprise Edition-libraries are required. There was no explanation for this in the documentation and at first we did not realize that we had to download this ourselves. Because of these and some other problems, the installation process took much longer than we had expected.

8.4 Training

We started to learn how JBuilder worked by creating some simple session beans and deploying them in a container on the application server. JBuilder has wizards that simplify the development and deployment of beans. It supports the process of creating an "empty" bean (containing no business logic) with a home and remote interface, and it creates a first version of an ejb-jar file, which will be used for deploying the bean.

We also tested to create some tables in JDataStore and learned how to create a simple entity bean from an existing database table. When we had this basic knowledge of the techniques we would be using, we proceeded to the next step: modelling our prototype.

8.5 Modelling

We decided to have four database tables: Customer, User, Trouble Ticket and Ticket Type. For each database table we would create an entity bean. We decided to use container-managed persistence, since it simplifies the mapping of the data in the beans with the underlying data store.

The client application would contain the functionality to let a customer log in with a user name and password, and to list trouble tickets for the customer. We decided to put the graphical user interfaces as separate classes, to keep them separated from the functionality for a better overview and changeability. To give the user some alternatives, we decided to add choices for listing trouble tickets between a given start and end date and choices for specifying status and type of the trouble tickets of interest. The default end date would be the current date and the default start date would be the date of the first trouble ticket for that customer that existed in the database. We would also have some error messages handling users specifying wrong username or password and typing unexpected date formats. We also decided to list customer information in the upper part of the window.

[2] An object relational DBMS is a hybrid that combines features of both relational and object-oriented database management systems.

The connection between the client and the entity beans was implemented using CORBA. Borland's AppServer builds on a CORBA-based infrastructure (VisiBroker), and provides a VisiBroker java2iiop compiler to enable RMI over IIOP. Using RMI over IIOP we could define the interfaces in Java, letting the compiler create the necessary IDL. This was a chance for us to see how the interoperability worked between EJB and CORBA. An overview of the flow of requests from the client application to the beans are shown in figure 8.1.

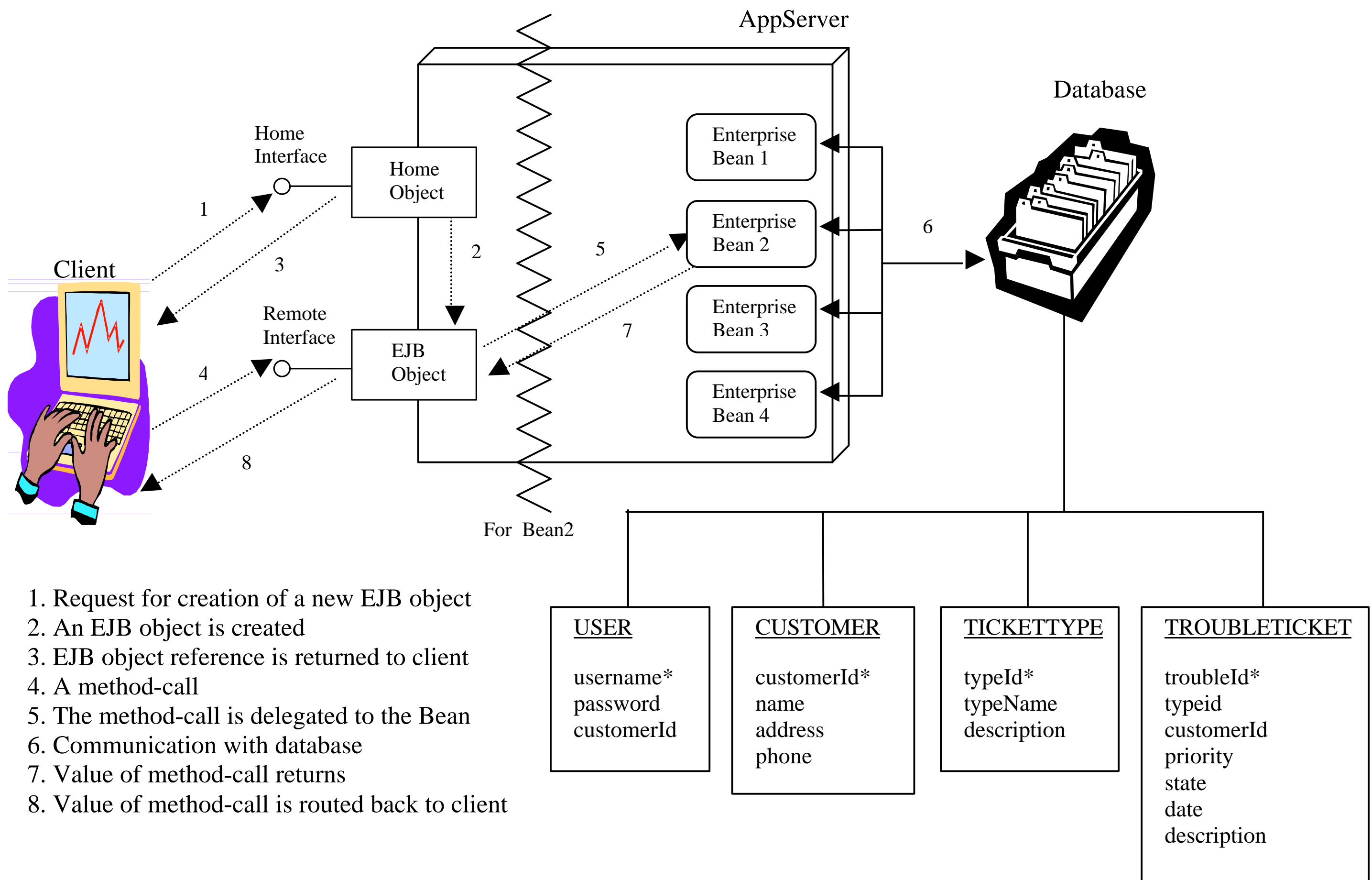


Figure 8.1 An overview of the flow of requests for our prototype

8.6 Programming

Finally, we started to create the database tables, the entity beans and the client application. This step went quite fast and problem-free, since we now had all the knowledge needed to develop our prototype. The main graphical user interfaces for the client application are shown in figure 8.2 and 8.3.

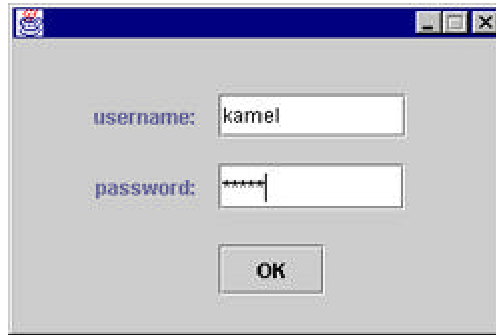


Figure 8.2 Log-in window

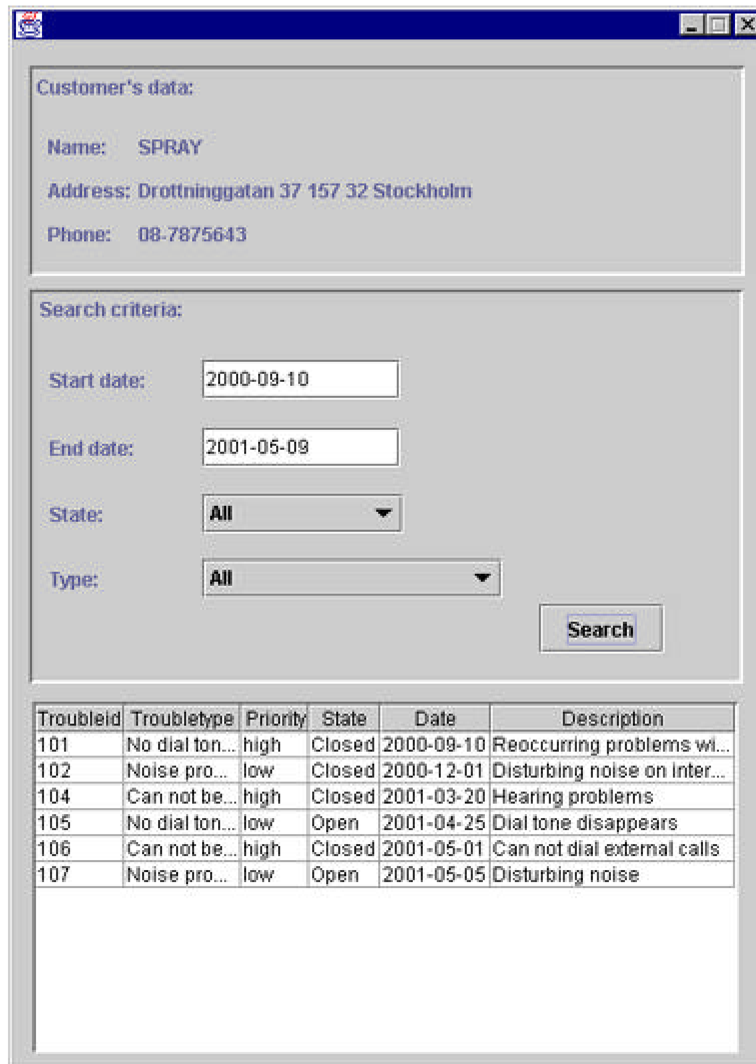


Figure 8.3 Main window letting user specify search criteria and list trouble tickets

9 Results

In this section we will present our results, categorized after each method we have used: the literature study, the interviews and the prototype development.

9.1 Results from literature study

9.1.1 Some points from the literature analysis

- In the last few years a new paradigm in the area of software development has been recognised, namely Component-Based Software Development (CBSD). It focuses on the realization of systems through integration of pre-existing components and shifts the development emphasis from programming software to composing software systems.

- Four major activities characterize the component-based development approach: component qualification, component adaptation, assembling components into systems, and system evolution.

- The selection of a particular architectural style, or the invention of a custom style, is perhaps the most important design decision affecting the integration process. In fact the architecture will drive the integration effort.

- A design pattern is a description of a well-proven solution to a general design problem that may occur in many different situations.

- Design patterns are not only useful for the design of object-oriented systems. Some design patterns could be used at the software component level, for component integration. Those patterns that we have found suitable for application in this area are the adapter, bridge, proxy, mediator and facade patterns described by Gamma et al. (1995), the broker and layer patterns described by Buschmann et al. (1996), and the Component Interaction Patterns described by Eskelin (1999).

- There are three dominating techniques in the area of software component integration, namely Enterprise JavaBeans, COM and CORBA.

- In addition to these techniques, there are several other techniques under development, which might become more widely used in the future. Some examples are JINI, message brokers and intelligent agents.

9.1.2 Comparison of the dominating techniques

Here, we will give an overview of similarities and differences between Enterprise JavaBeans, COM and CORBA. To make a meaningful comparison, we need to compare the parts of the architectures that are on the same level, covering the same area of use. First, we will present a comparison between the component-models, namely MTS, EJB and CCM. Then we will give a comparison of the three distributed architectures DCOM, CORBA and RMI.

MTS vs. EJB

Architecture

The general architecture of MTS and EJB are very similar. As with MTS, EJB supports server application assembly.

A basic architecture of EJB and MTS can be described as follows:

MTS Architecture	EJB Architecture
• The mtx.exe	• An EJB server
• The MTS Executive (mtxex.dll)	• EJB containers that run within the server
• The Factory Wrappers and Context Wrappers for each component	• Home objects and Remote EJBObjects
• The MTS Server component	• Enterprise Beans, which runs within the container
• Support systems like the Service Control Manager (SCM), the Microsoft Message Queue (MSMQ), etc.	• Support systems like Java Naming Directory Interface (JNDI), Java Transaction Service (JTS), etc.

While EJB Components run inside an EJB Container, MTS components run in an MTS Executive. The MTS runtime (mtxex.dll) manages creation, management, and destruction of MTS components like an EJB container, which creates, manages, and destroys EJB components.

Both MTS and EJB work by intercepting method calls and inserting services based on a set of attributes defined at deployment time. MTS uses class factory wrappers and object wrappers to intercept the method calls (the MTS executive is called automatically by the COM runtime). Similarly, EJB requires wrappers for each component type (EJBHome) and each component instance (EJBObject).

Both MTS and EJB application components are location transparent. It means that it is unnecessary for the client of a component to know the physical location of the component in the file system or on the network. A client only needs to know the DNS name of a Server machine from which to get a reference to the component.

Component Types

MTS views components as stateless transient, which will be instantiated for each user, and for each method call. MTS completes or abort a transaction at the end of each method call and destroys information associated with the component's state. At each method call, the client gets a fresh new instance of the component. If the client wants to save the state for later use, the components must store the state manually in a database or other repositories.

EJB supports both transient components (Session Beans) and persistent components (Entity Beans). Stateless Session Beans are almost like MTS Components, while Stateful Session Bean maintains state across different method calls. Across different method calls, the client gets the same old instance of component with the same old data. It requires no code to store the state of this type of Bean. When working with EJB persistent component which represents records stored in a database, you could either write your own code to retrieve and update the data or you could leave it for the container to manage this task.

Interoperability

This feature can be discussed in two matters, protocol support and client support.

Protocol support

MTS components can be invoked from other machines by clients using the DCOM protocol. Local COM can be used if the client is on the same machine.

EJB components can be invoked from other machines by clients using RMI/JRMP or RMI/IIOP.

Client support

MTS supports ActiveX clients on:
Win32
WinCE
Internet Explorer
IIS Active Server Pages
Clients using CORBA/IIOP protocol could be supported in communication with the server through a COM-CORBA bridge.

EJB supports:
Java clients on any platform through RMI
CORBA clients on any platform through IIOP
Web clients through Java Server Pages and servlet
Clients using COM+ protocol could be supported in communication with the server through a COM-CORBA bridge.
Beyond this one EJB server implementation, namely WebLogic supports ActiveX clients.

Transaction Support

MTS supports four types of transaction attributes:

- not supported (the method does not support transactions)
- supported (the method can support the transaction, but it does not require one)
- requires (the method requires a transaction, if the caller does not provide one it will create one)
- requires new (the method always requires a new transaction)

EJB supports the same transaction types, and offers two new types:

- mandatory (the method requires that the caller provides a transaction)
- Bean-managed (the Bean manages its own transactions, specifying its own transaction demarcation)

Portability

Since MTS is a component model for Windows NT and 2000, MTS components are portable only across these platforms.

Portability for EJB components can be discussed in two different levels, namely:
platform level and between different EJB servers/containers.

Since EJB components are developed in the Java language, there is no limit for portability across different platforms.

In theory, EJB components can be ported to any EJB-compliant application server. But since the interface between server and container has been left open in the EJB specification, there may be some difficulties, especially regarding container managed persistent Entity Beans. However, different EJB servers are available from BEA Systems, Bluestone, Gemstone, Borland and Persistence.

Language support

MTS supports C, C++, Cobol, Visual Basic, Delphi, and practically any other development language.

EJB supports Java only.

EJB vs. CCM

The architecture of CORBA's component model is very much similar to the component model of EJB. Therefore, we will not go in to any detailed technical comparison between these two models. CCM provides two component levels, where the basic component provides the same functionality as an EJB component, and the extended component adds some extra features.

Architecture

The extended model of CCM adds facets, receptacles, event sources and sinks as new ways for a component to interact with external entities. In the extended model, components also can utilize the event model of CCM. The session and entity components of CCM correspond to the session and entity beans of EJB. CCM also supports two more component types, service and process components.

Portability

The EJB specification assumes that the components and containers/servers will be run on platforms that already have the Java Virtual Machine installed. The CCM specification assumes that the components and containers/servers can run on any platform.

Language support

The major difference between these two component models is their language support. EJB is suited only for the Java language, while CCM intends to ease the integration of heterogeneous components by supporting components developed in various programming languages.

CORBA, DCOM & RMI

CORBA, DCOM and Java RMI provide the infrastructure needed for clients and objects to communicate in a distributed environment. Their architectures are based on the same basic concepts. They all use an object-oriented RPC-based (Remote Procedure Call) form of communication. To invoke a remote operation, the client makes a call to the client stub (called *proxy* in DCOM), which packs the call parameters into a request message, and sends the message. At the server side, the message is unpacked by the server stub (called *skeleton* in CORBA and RMI), and sent to the target object's operation.

Still, there are some differences between the three architectures. The following table summarizes differences and similarities for the three architectures and some of them are described in more detail below.

	CORBA	DCOM	Java RMI
Support for different programming languages	Can be used with any programming language that has an IDL mapping.	Can be used with different programming languages, since the specification is at the binary level.	Supports Java only.
Platform dependence	Will run on any platform that has a CORBA implementation.	Will run on any platform that has a COM service implementation.	Will run on any platform that has a Java Virtual Machine implementation.
Underlying protocol	IIOP (Internet Inter-ORB Protocol)	ORPC (Object Remote Procedure Call)	JRMP (Java Remote Method Protocol)
Distributed garbage collection	No	Yes. Performs garbage collection by the use of a pinging protocol.	Yes (using the mechanisms in the Java Virtual Machine).
Interface Definition Language	OMG IDL	MIDL (Microsoft IDL)	Java
Multiple interfaces/ Interface inheritance	Supports multiple inheritance for interfaces.	Supports several unrelated interfaces for the same object.	Supports multiple inheritance for interfaces.
Exception handling	Allows throwing CORBA standard exceptions, and user-defined exceptions in IDL, which are	Each method call returns a bit string with return status. Uses Error Objects for richer exception handling.	Allows throwing exceptions, which are serialized and marshaled.

	serialized and marshaled.		
Name mapping to object implementation	Handled by the Implementation Repository.	Handled by the Registry.	Handled by the RMIRegistry.
Type information for operations	Is held in the Interface Repository.	Is held in the Type Library.	Is held by the object itself.
Responsibility for locating an object implementation	Object Request Broker	Service Control Manager	Java Virtual Machine
Pass object by value	Yes	Yes	Yes
Dynamic invocation	Yes	Yes	Yes

CORBA and DCOM both support different programming languages. Java RMI is an extension to the core Java language and depends on many of the features of Java like object serialization and Java interface definitions. Java RMI can only be used in a distributed environment, where all the server and client objects are written in Java. If legacy systems need to interface with RMI systems, adapters have to be written to the older systems.

CORBA and DCOM both use their own variant of IDL for defining interfaces. RMI does not use any separate interface definition language. Instead it uses Java for this purpose. By using IDL, CORBA and DCOM separates interface definition from implementation, which leads to the possibility of using different programming languages for the object implementations.

Reference counting provides a mechanism that allows an object to keep track of its clients and enables it to delete itself when it is no longer needed (that is, when there are no references to the object. DCOM and Java RMI support reference counting for distributed garbage collection. In CORBA, there is no count of object references, and object deletion can only be done by explicit application programming by the developer. CORBA allows a client to convert an object reference to a string, and later converting it back to a usable object reference. Since object references converted to strings not are included in reference counting, it would not work well to both provide object-to-string/string-to-object capabilities and reference counting.

CORBA and Java RMI support interface inheritance. DCOM does not support this feature, but instead allows an object to have several interfaces to the same object, which allows each distinct feature of an object to have a separate interface.

9.2 Results from interviews

9.2.1 Integration Difficulties and Principles

- The major problem in an integration work is the lack of a good understanding and insight over interfaces. It means that interfaces are either not well specified, or there are not enough documents about them (this holds more for legacy systems), or the documentation is written in a way that does not give a good description over the interfaces (this holds more for modern systems).

- Design patterns are useful for software component integration. Design patterns give the developers a common concept base, and represent solutions to general design problems.

9.2.2 Component-Based Software Development

- Component-based software development seems to be an increasing trend, which brings advantages such as quicker development time, less costs, and easier upgrading and replacement of system parts.

- Component-based software development is no silver bullet, though. It will not solve all problems. CBSD could be seen as a powerful way of organizing things, but for example, it does not support human creativity.

- There is not a commonly accepted definition of what a component is. All the interviewees gave their own definition of this concept, which seemed to be affected by the kind of components they personally had come in contact with.

- In future there might be more focus on another paradigm; service oriented software development, where the focus is less on components and how they are constructed, and more on the services that these components offer and their quality.

9.2.3 Integration techniques

- Integration techniques like CORBA, COM and EJB are all complex techniques, which require much training from the developer to learn and to be able to use effectively. A well-designed development-tool can simplify the use of the techniques, and reduce the training needed.

- It is possible to make techniques such as CORBA and COM to interoperate, but it results in a higher complexity in the integration work.

- The CORBA specification covers many useful services, but most vendors have only implemented a limited number of these services.

- It is true that COM works with different programming languages, but it requires compilers, which compile the code according to a specific binary format. Those programming languages, which do not have such compilers, could not be used with COM.

- CORBA could be seen as more flexible than EJB and COM regarding some aspects. It supports many different platforms and different programming languages, and many additional services like trading and event service.

9.3 Prototyping experience

The development of a simple prototype using Enterprise JavaBeans gave us a much better understanding of the theoretical descriptions of EJB's architecture, and of the process of developing and deploying beans in a runtime environment. We could easily relate this to the architecture of CORBA, since CORBA's Component Model is an extension of EJB's.

We were also able to draw some more general conclusions about this kind of integration techniques, which are summarized below.

9.3.1 Development Tool Demanding

Using a development tool such as JBuilder for creating and deploying components means that a lot of work could be automated (for example the creation of the home and remote interfaces and the creation of the ejb-jar file). It also lets the developer deploy the beans in the application server from within this development environment. Other useful features are help with keeping backup copies and version tracking of files.

Even for small projects, there will be a large number of source files when developing enterprise beans. Our simple prototype resulted in 48 java source-files and 71 class files, where a large number consisted of CORBA-classes, like stubs and skeletons. To manage such a large amount of files could be complex without a suitable development tool.

For these reasons we think it is necessary to use a development tool for component development and deployment, to be able to work efficiently and to have control over the process.

9.3.2 Development Tool Dependence

Our biggest problem during our prototyping process was the installation of JBuilder, and to understand the logic of JBuilder and how it acted in different situations. We had some problems with menus sometimes disappearing or getting locked, strange compilation messages pointing at errors in files that JBuilder self had created, etc.

Even if we had many troubles as a direct result of using JBuilder, JBuilder also had many advantages such as the wizards helping to develop and deploy simple enterprise beans. Choosing another development environment would probably have led to other experiences of what was the easy and more difficult steps in the development process, depending on the benefits and drawbacks of the specific development tool. Thus, the development tool is an important factor when evaluating the difficulty and complexity of a specific technique for component development and integration.

9.3.3 Interoperability with CORBA

During our literature study, we read about how to combine EJB and CORBA. The CCM specification contains a chapter describing the mapping between EJB and CORBA. Using CORBA for the communication between the beans and the client gave us an opportunity to see how the described interoperability worked in reality. The development process was not complicated by this choice: all we had to do was to compile the home interfaces with the VisiBroker compiler. The CORBA IDL interfaces were generated automatically in the background from our Java interfaces; we did not have to make any special considerations to handle the CORBA communication. So for this simple prototype the interoperability worked entirely without problems, with JBuilder and the AppServer transparently managing the mapping between EJB and CORBA at development and deployment.

9.3.4 Inherent Technique Complexity

EJB is a complex technique. There are many new concepts to learn for a beginner before being able to use the technique, like home and remote interfaces, deployment descriptor, ejb groups, stubs and skeletons, container, session and entity beans, naming service, narrowing, transaction management, etc. Therefore, a developer not familiar with component-based integration techniques will need some time for learning these underlying concepts. When the developer has this basic knowledge and has succeeded to create and deploy some simple session and entity beans, this process will be easily repeated though, especially if using a tool that automates much of the repetitive work.

9.3.5 Support for basic information management services

EJB's support for transaction management, persistence and security simplifies the work needed to be done by the developer. When developing the entity beans, we only needed to specify a database driver, the location of our database and a few other parameters. The actual calls to the database were automatically generated and hidden from our view, which greatly simplified our work.

9.3.6 Performance

We have worked with Windows NT, 128 MB RAM which is the minimal requirements for the Borland AppServer. The time required for deploying our four entity beans varied between a few seconds and sometimes up to two minutes. What this variation depended on is unclear, but it usually helped to restart the computer when the process became too slow. The communication between the client and the server was also sometimes very slow.

10 Discussion

In today's turbulent and rapidly changing information society, having flexible and updated information systems becomes a critical factor for the survival of many enterprises. To be able to offer such solutions, the IT organizations are increasingly confronted with integration problems. The need for integration appears in various forms, such as integration with legacy systems or building systems consisting of integrated third-party products. Because of its often-vital importance for the enterprise as a whole, IT-organizations, especially at the leadership and management level, must consider these questions at a strategic point of view, and give them a high-level priority.

We think that working with such questions cannot be done with the traditional development methods. It needs a new way of thinking, having a component-based perspective. Just choosing a new technology is not sufficient to overcome the difficulties involved in integration tasks.

10.1 A critical view on our work

To answer the question of how to make Ericsson's third-party products become successfully integrated we did not have the possibility to analyse the different involved systems in this mission because of their complexity, regarding the number of involved systems, their size and interrelations.

It was not easy to find the right path and depth of research for our work, especially during the first weeks. Another problem was the different outlook, which the university and Ericsson had over the same mission.

With respect to these reasons, we decided to have a macro view over the problem area. In the beginning of our work, we tried to get a better understanding over our role in this integration effort using the Zachman table (see the Appendix) placing our selves in the distributed system architecture, taking a design perspective and focusing on distributed architectures.

As a consequence to this choice, we chose to study dominating principles and techniques for software component integration. At the same time we tried to see other thinkable techniques, which could be used in integration projects.

To perform interviews in a way that leads to qualitative results, needs both experience and routine. Our limited earlier experience in this area could be seen as a lack in our research. Another limited resource has been the time aspect. If we would have had more time, we could have developed one prototype for each integration technique, providing us with more detailed material for making a comparison of the techniques regarding aspects such as performance and difficulties of learning.

10.2 Conclusions

Here, we will present the conclusions we have drawn from our results, regarding our three main questions for this thesis work.

10.2.1 Question 1

What dominating general principles exist for software component integration and what are their benefits and drawbacks?

To be able to answer this question, we first tried to find a definition of the concepts component and integration. As we soon realized, these concepts have many different definitions and there is not a single true and all agreed-upon definition. In chapter four we gave a number of direct definitions of component and also gave an indirect definition, by contrasting a component to the definition of an object. In chapter five, we discussed the terms system, integration, architecture and their relationships, and concluded among other things that integration to a high degree is a matter of architecture.

We found during our literature studies, that the dominating principles that are used for integrating software components are design patterns. We have studied some of the more well-known design patterns, and described those patterns we found applicable to the area of component integration, like adapter, bridge, mediator and broker, including their benefits and drawbacks, in chapter five.

That some design patterns are useful for component integration was also later confirmed in our interviews. When studying the different integration techniques, we could also see ourselves that these techniques are built upon some of the design patterns. For example, CORBA is built on the broker pattern, message brokers are built on the mediator pattern and the registrar in JINI is built on the proxy pattern.

10.2.2 Question 2

What different integration techniques exist for this purpose and what are the main differences between them?

Middleware Component Models take a high level approach to building distributed systems. They free the application developer to concentrate on programming only the business logic, while removing the need to write all the "plumbing" code that is required in any enterprise application development scenario. For example, the enterprise application developer no longer needs to write code that handles transactional behaviour, security, database connection pooling or threading, because the architecture delegates this task to the server vendor. There are three dominating Middleware Component Models, namely Microsoft Transaction Server (MTS) Architecture, Sun's Enterprise JavaBeans (EJB), and OMG's CORBA Component Model (CCM).

EJB technology and CORBA are not really competitors. EJB makes it easier to build application on top of a CORBA infrastructure, where CORBA handles the transaction and EJB the business logic with the biggest advantage that CORBA works with all programming languages. Additionally, the recently released CORBA component specification is based on the EJB architecture.

Both Microsoft Transaction Server and Enterprise JavaBeans target the creation of component-based, transaction-oriented applications. Both of them support transactional applications and provide services for scalability and can also be defined as component-based programming model. Because the two technologies attempt to solve many of the same problems, they have much in common.

Almost all of the techniques that are used in an integration work are complex and tool-dependent. But the core of techniques such as EJB is in another level of complexity compared to the complexity that a developer meets in his/her developing work. The last one depends on the tools that a developer uses in his/her work. In other words you can use a complex technique in a relatively easy way and vice versa.

As an example, Microsoft MTS has fewer development options than EJB does. Therefore MTS development might be experienced as easier. But most EJB development environment has tools such as graphic wizards that generate almost all EJB constructs automatically. It means that the application developer experiences the task of development as easy and has no need to be involved in the deep of complexity behind the technique.

MTS's biggest weakness is that it is tied exclusively to Windows NT and 2000, which limits its application in many systems requiring multiple platforms. EJB's and CORBA's biggest strength is that they are available on lots of different operating systems. This diversity means that EJB- and CORBA-based applications can also run on systems that are bigger and more reliable than NT-systems.

EJB supports only a single programming language, namely Java. It is not reasonable to assume that one language will ever be the right choice for every organization and every application, and therefore, by tying itself to a single language, EJB limits it reach.

EJB defines support for two kinds of components (two kinds of enterprise beans). In addition to session beans, which are much like MTS components, EJB's second option, known as entity beans, has no analogue in MTS. Taking this approach, makes it easier for application developers to load and store the components' state and to manage their persistence. It makes it possible to create components that are portable across any datastore.

The EJB specification leaves some details undefined, such as the relation between server and container, which could harm the portability between different vendor implementations.

10.2.3 Question 3

For the specific operating support system at Ericsson Telecom Management, we had the following question:
How can their third-party products be integrated successfully, supporting simple replacement and maintenance?

We have the following suggestions for how Ericsson can continue the work with making their software solutions having a more flexible integration architecture:

- One problem involved with integration questions is that there does not exist *common definitions* over different concepts. Therefore, arranging/creating a common *concept base* could be useful. It is also important to have a common and clear definition of the problem, because the clearer the problem definition is, the more easily we understand what we should do and what we should not do. Having clear definitions of common concepts and of the problem in general is an important starting point, even if it is not enough in itself.

- There exist a number of techniques for software component integration. Every technique sooner or later becomes obsolete and new techniques will continue to arrive on the market. To be able to choose a suitable technique, it is important to specify what is needed in more abstract terms, without defining the needs in terms of a specific technique. A *guidance model* could help to focus on which aspects are important when choosing a specific integration strategy. There is no integration technique, which is the best solution for every integration task; the choice is dependent on how aspects such as flexibility, performance, platform independence, etc. are ranked. There is no integration technique that is better than the others in all respects, and there will never be, since some of these qualities are partly contradictory. The guidance model should also help to put focus on the long-term needs and requirements. Time is an important factor. Sometimes solutions that are good for the present, will be unsuitable for tomorrow. Short-term focus on a specific technique could therefore be very risky.

- We consider component-based software development to be a suitable development form for Ericsson's development of integrated third-party products, since it puts focus on the acquisition of appropriate ready-made components, and the integration of these. Even if CBSD is a rather new paradigm, we believe it is here to stay, which the interviewees also agreed upon. Even if there will be, for example, a service-oriented paradigm in the future, there will still be a need for an underlying component architecture for realizing these services. A development method that views a system at a higher abstraction-level than ordinary system development methods will be required for components-off-the-shelf integration.

- Design patterns could be used to facilitate integration tasks. It facilitates the work in two important ways. First, design patterns can work as a common concept base because when developers are talking about some specific design patterns such as proxy etc., they can understand each other much easier. Second, using design patterns can help in reducing the degree of complexity. For example, design patterns can help in these matters by:
  - Reduction in variety**:** A simple way to ease the integrative problem is to reduce the diversity of elements present in the situation using standardization as a means against "spaghetti" mixture of elements.
  - Reduction in quantity**:** By eliminating a significant number of the elements, the integrative problem may also be eased.

  Therefore, we recommend Ericsson Telecom to educate software integrators about design patterns.

- Giving recommendations about choosing the right technique is difficult. Having a world with the same standards is just a utopia. So we suppose that using COM+ and Microsoft as a common platform is not possible for Ericsson. Enterprise JavaBeans has a well-designed component-architecture and is platform-independent. EJB could be useful when developing new components and when using the Java programming language. For integrating third-party products from different vendors, written in different programming languages, CORBA seems to be the most appropriate alternative, with its new component-model in CORBA 3, which is very similar to the component model of EJB.

- Finally, it is important that management is involved in the integration questions, and increases the degree of motivation within the organization and reserves resources for this work.

We believe that our effort is a pilot study towards a much greater project to bring out a convincing guidance model and studying the semantic aspects of component integration (contracts), which has only been discussed shortly in this thesis.

11 Appendix

11.1 Interview Questions

Background

1. Have you been involved in integration projects? What are your experiences of integration processes?

Integration difficulties and principles

2. What general principles are important when integrating software systems?

3. How would you define an optimal integration?

4. What weaknesses are most damaging to an integration process?

Component-Based Software Development

5. What do you think about Component-Based Software Development? Is it a promising paradigm or something that will soon disappear?

6. What is your definition of a component?

Integration Techniques

7. Do the techniques keep what they promise, when it comes to platform independence for example?

8. What are the benefits and drawbacks of COM, CORBA and EJB? What are their strengths and weaknesses compared to each other?

9. What are your experiences of the techniques when it comes to qualities such as performance, reliability, flexibility, user friendliness and complexity?

10. How is the interoperability between COM, CORBA and EJB? Could they easily be combined?

11. Dou you know message brokers, JINI? Do you think they are good techniques for integration?

11.2 The Zachman Framework

ENTERPRISE ARCHITECTURE - A FRAMEWORK ™

	DATA *What*	FUNCTION *How*	NETWORK *Where*	PEOPLE *Who*	TIME *When*	MOTIVATION *Why*	
SCOPE (CONTEXTUAL) *Planner*	List of Things Important to the Business ENTITY = Class of Business Thing	List of Processes the Business Performs Function = Class of Business Process	List of Locations in which the Business Operates Node = Major Business Location	List of Organizations Important to the Business People = Major Organizations	List of Events Significant to the Business Time = Major Business Event	List of Business Goals/Strat Ends/Means=Major Bus. Goal/ Critical Success Factor	SCOPE (CONTEXTUAL) *Planner*
ENTERPRISE MODEL (CONCEPTUAL) *Owner*	e.g. Semantic Model Ent = Business Entity Reln = Business Relationship	e.g. Business Process Model Proc. = Business Process I/O = Business Resources	e.g. Business Logistics System Node = Business Location Link = Business Linkage	e.g. Work Flow Model People = Organization Unit Work = Work Product	e.g. Master Schedule Time = Business Event Cycle = Business Cycle	e.g. Business Plan End = Business Objective Means = Business Strategy	ENTERPRISE MODEL (CONCEPTUAL) *Owner*
SYSTEM MODEL (LOGICAL) *Designer*	e.g. Logical Data Model Ent = Data Entity Reln = Data Relationship	e.g. Application Architecture Proc. = Application Function I/O = User Views	e.g. Distributed System Architecture Node = I/S Function (Processor, Storage, etc) Link = Line Characteristics	e.g. Human Interface Architecture People = Role Work = Deliverable	e.g. Processing Structure Time = System Event Cycle = Processing Cycle	e.g. Business Rule Model End = Structural Assertion Means = Action Assertion	SYSTEM MODEL (LOGICAL) *Designer*
TECHNOLOGY MODEL (PHYSICAL) *Builder*	e.g. Physical Data Model Ent = Segment/Table/etc. Reln = Pointer/Key/etc.	e.g. System Design Proc. = Computer Function I/O = Data Elements/Sets	e.g. Technology Architecture Node = Hardware/System Software Link = Line Specifications	e.g. Presentation Architecture People = User Work = Screen Format	e.g. Control Structure Time = Execute Cycle = Component Cycle	e.g. Rule Design End = Condition Means = Action	TECHNOLOGY MODEL (PHYSICAL) *Builder*
DETAILED REPRESEN- TATIONS (OUT-OF- CONTEXT) *Sub-Contractor*	e.g. Data Definition Ent = Field Reln = Address	e.g. Program Proc. = Language Stmt I/O = Control Block	e.g. Network Architecture Node = Addresses Link = Protocols	e.g. Security Architecture People = Identity Work = Job	e.g. Timing Definition Time = Interrupt Cycle = Machine Cycle	e.g. Rule Specification End = Sub-condition Means = Step	DETAILED REPRESEN- TATIONS (OUT-OF- CONTEXT) *Sub-Contractor*
FUNCTIONING ENTERPRISE	e.g. DATA	e.g. FUNCTION	e.g. NETWORK	e.g. ORGANIZATION	e.g. SCHEDULE	e.g. STRATEGY	FUNCTIONING ENTERPRISE

John A. Zachman, Zachman International (810) 231-0531

12 References

12.1 Books

Benjamin, M., Vasudevan, B., Villalba, T., & Vogel, A. (1999). *C++ Programming with CORBA*. New York: Wiley Computer Publishing.

Britton, C. (2000). *IT Architectures and Middleware*. New Jersey: Addison-Wesley.

Brown, A. W. (1996). *Component-based Software Engineering : selected papers from the Software Engineering Institute*. Washington : IEEE Computer Society.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Holström, S. (1990). *Konstruktion och verifiering av imperativa program med Dijkstras metod*. Institutionen för Datavetenskap, Göteborg Universitet.

Jia, X. (2000). *Object-Oriented Software Development Using Java*. USA: Addison-Wesley.

Johnson, R., Kast, F., & Rosenzweig, J. (1973). *The Theory and Management of Systems*. McGraw-Hill

Lewis, P. (1994). *Information-Systems Development*. Pitman Publishing.

Lundahl, U. & Skärvad, P-H., 1982. *Utredningsmetodik för samhällsvetare och ekonomer*. Lund: Studentlitteratur.

Magoulas, T., & Pessi, K. (1998). *Strategisk IT-management*. Department of Informatics, Göteborgs University.

Mathiassen, L., Munk-Madsen, A., Nielsen, P., Stage, J. (1998). *Objektorienterad analys och design*. Lund: Studentlitteratur

Norrbom, C. (1973). *Systemteori – en introduktion*. M&B Fackbokförlaget.

Patel, R. & Davidson, B., 1991. *Forskningsmetodikens grunder – Att planera, genomföra och rapportera en undersökning*. Lund: Studentlitteratur.

Roman, E. (1999). *Mastering Enterprise JavaBeans*. New York: John Wiley & Sons.

Ruh, W. A., Maginnis F. X. & Brown, W. J. (2001). *Enterprise Application Integration*. New York: John Wiley & Sons.

Schneider, J. & Arora, R. (1997). *Using Enterprise Java*. Indianapolis, IN: Que Corporation.

Sommerville, I., (1998). *Software Engineering*. Essex: Addison-Wesley.

Szyperski, C. (1999). *Component Software – Beyond Object-Oriented Programming*. New York: ACM Press Books.

Wallén, G. (1993). *Vetenskapsteori och forskningsmetodik*. Lund: Studentlitteratur.

12.2 Web references

Appelbaum, R., Claudio, L., Cline, M., Girou, M., & Watson, J. (1999). *The CORBA FAQ – Frequently Asked Questions* [WWW document]. URL http://www.aurora-tech.com/corba-faq/index.html

BEA Systems. (2000). *BEA WebLogic Enterprise 5.1* [WWW document] URL http://edocs.bea.com/wle/interm/corba.htm

Beveridge, T., & Perks, C. (2000). *Messaging is the Medium* [WWW document]. URL http://www.intelligenteai.com/feature/2000929.shtml

Buschmann, F., Meunier, R. Rohnert, H., Sommerlad, P., & Stal, M. (1996). *The Pattern Abstracts* [WWW document]. URL http://hillside.net//patterns/Siemens/abstracts.html

Brooke, C., Pharoah, A., Siegel, J. (2000). *Creating Commercial Components – CORBA Component Model (CCM)* [WWW document] URL http://componentsource.com/build/corbawhitepaper.asp

Brown, A. W. & Wallnau, K. C. (2000). *An Examination of the Current State of CBSE* [WWW document] URL http://www.sei.cmu.edu/cbs/icse98/summary.html

Carnegie Mellon, Software Engineering Institute (2000). *Component-Based Software Development / COTS Integration* [WWW document]. URL http://www.sei.cmu.edu/str/descriptions/cbsd.html

Carnegie Mellon, Software Engineering Institute (2001). *How Do You Define Software Architecture?* [WWW dokument]. URL http://www.sei.cmu.edu/architecture/definitions.html

Chappell, D. (1997). *The Next Wave, Component Software Enters The Mainstream* [WWW document]. URL http://www.rational.com/products/whitepapers/354.jsp

Christiansson, B. & Jakobsson, L. (2000). *Component-Based Software Development Life Cycles* [WWW document]. URL http://www.idt.mdh.se/kurser/phd/CBSE/

Dr. Dobbs Journal (1997). *The Component Object Model: Technical Overview* [WWW document]. URL http://www.cs.umd.edu/~pugh/com/

Eriksson, D. (2000). *COM* [WWW document]. URL http://www.calpoly.edu/~jeriksso/COM/COM_Paper.htm

Eskelin, P. (1999). *Component Interaction Patterns* [WWW document]. URL http://jerry.cs.uiuc.edu/~plop/plop99/proceedings/Eskelin1/ComponentInteractionPatterns.pdf

Gopalan, S. (1998). *Microsoft Message Queue Server* [WWW document]. URL http://www.execpc.com/~gopalan/mts/msmq.html

Gopalan, S. (1999). *Middleware Component Models - CCM, EJB, MTS* [WWW document]. URL http://www.execpc.com/~gopalan/

Griss, M. (2000). *My Agent Will Call Your Agent* [WWW document]. URL http://www.sdmagazine.com/articles/2000/0002/0002c/0002c.htm

Henning, M. (1998). *Binding, Migration, and Scalability in CORBA* [WWW document] URL http://www.ooc.com.au/staff/michi/cacm.pdf

Higham, T. (2001). *The Reality of EJB* [WWW document]. URL http://advisor.com/Articles.nsf/aid/HIGHT11

Hnich, B., Jonsson, T., & Kiziltan, Z. (1999). *On the Definition of Concepts in Component Based Software Development* [WWW document]. URL http://www.idt.mdh.se/kurser/phd/CBSE/reports/Final_reports/report_03.pdf

Hypertext Webster Dictionary. (2001). [WWW document] URL http://www.fin.gov.nt.ca/webster.htm

Inprise. (2001). *VisiBroker – The Engine for Internet Applications* [WWW document] URL http://www.inprise.com/visibroker

Institute for Telecommunication Science, (1996). *Glossary of Telecommunication Terms* [WWW document]. URL http://www.its.bldrdoc.gov/fs-1037/

IONA. (2001). *Orbix 2000* [WWW document]. URL http://www.iona.ie/products/orbix2000_home.htm

Isazadeh, A., MacEwen, G.H. & Malton (1995). A. *A Review of Post-Factum Software Integration Methods* [WWW document]. URL http://citeseer.nj.nec.com/1567.html

Jennings, N. (2001). *An Agent-Based Approach For Building Complex Software Systems* [WWW document]. URL http://www.ecs.soton.ac.uk/~nrj/download-files/cacm01.pdf

Jennings, N., & Wooldridge, M. (1995). *Intelligent Agents: Theory and Practice* [WWW document]. URL http://www.ecs.soton.ac.uk/~nrj/download-files/KE-REVIEW-95.ps

JGuru (1999). *Enterprise JavaBeans Fundamental* [WWW document]. URL http://developer.java.sun.com/developer/onlineTraining/EJBIntro/EJBIntro.html

Johnson, M. (1997). *A walking tour of JavaBeans* [WWW document]. URL http://www.javaworld.com/javaworld/jw-08-1997/jw-08-beans.html

Kiziltan, Z, Jonsson, T. & Brahim, H. (2000). *On the Definition of Concepts in Component Based Software Development* [WWW document] URL http://www.idt.mdh.se/kurser/phd/CBSE/

Kutsche, R-D., & Sünbül, A. (1999). *A Meta-Data Based Development Strategy for Heterogeneous Distributed Information Systems* [WWW document]. URL http://www.computer.org/proceedings/meta/1999/papers/60/rkutsche.html

Larsson, M. (2000). *The Different Aspects of Component Based Systems* [WWW document]. URL http://www.idt.mdh.se/kurser/phd/CBSE/

Larsson, T. & Sandberg, M. (2000). *Building Flexible Components Based on Design Patterns* [WWW document]. URL http://www.idt.mdh.se/kurser/phd/CBSE/

Levine, D., Schmidt, D., & Wang, N. (2000). *Optimizing the CORBA Component Model for High-performance and Real-time Applications* [WWW-document]. URL http://www.cs.wustl.edu/~schmidt/PDF/middleware2000.pdf

Linthicum, D. (1998). *Message Broker Rising* [WWW document] URL http://www.dbmsmag.com/9809d07.html

Linthicum, D. (1999). *Mastering Message Brokers* [WWW document]. URL http://www.sdmagazine.com/articles/1999/9906/9906c/9906c.htm

Linthicum, D. (n.d./2001a) *How To Free Your Information – Selecting A Message Broker – Part 1* [WWW document]. URL http://eai.ebizq.net/enterprise_integration/linthicum_8.html

Linthicum, D. (n.d./2001b). *How To Free Your Information – Selecting A Message Broker – Part 2* [WWW document]. URL http://eai.ebizq.net/enterprise_integration/linthicum_9.html

Lycos (2001). *The Lycos Tech Glossary* [WWW document]. URL http://webopedia.lycos.com/Communications/

Mahmoud, Q. (2001, January). *Overview of CORBA* [WWW document]. URL http://developer.java.sun.com/developer/Books/corba/ch11.pdf

Merriam-Webster. (2001). *Merriam-Webster Online* [WWW document]. URL http://www.m-w.com/

Microsoft (1999). *Transaction Server—Transactional Component Services* [WWW document]. URL http://www.microsoft.com/com/wpaper/revguide.asp

Newmarch, J. (2000, February). *Jan Newmarch's Guide to JINI Technologies* [WWW document]. URL http://www.javacoffeebreak.com/books/extracts/jini/jini.html

North, K. (1997). *Understanding OLE* [WWW document]. URL http://www.dbmsmag.com/9506d13.html

Object Management Group. (1997, January). *A Discussion of the Object Management Architecture* [WWW document]. URL ftp://ftp.omg.org/pub/docs/formal/00-06-41.pdf

Object Management Group. (1998, May). *Joint Revised Submission - CORBA/Firewall Security* [WWW document]. URL ftp://ftp.omg.org/pub/docs/orbos/98-05-04.pdf

Object Management Group. (1999, March). *CORBA Components – Joint Revised Submission* [WWW document]. URL ftp://ftp.omg.org/pub/docs/orbos/99-02-05.pdf

Object Management Group. (1999, August). *CORBA Component Scripting* [WWW document] URL ftp://ftp.omg.org/pub/docs/orbos/99-08-01.pdf

Object Management Group. (1999, October). *CORBA 3.0 New Components Chapters* [WWW document] URL ftp://ftp.omg.org/pub/docs/ptc/99-10-04.pdf

Object Management Group. (2000a, January). *Internationalization, Time Operations, and Related Facilities* [WWW document] URL ftp://ftp.omg.org/pub/docs/formal/00-01-01.pdf

Object Management Group. (2000b, January). *Mobile Agent Facility Specification* [WWW document] URL ftp://ftp.omg.org/pub/docs/formal/00-01-02.pdf

Object Management Group. (2001, February). *The Common Object Request Broker: Architecture and Specification 2.4.2* [WWW document]. URL ftp://ftp.omg.org/pub/docs/formal/01-02-01.pdf

OpenLoop. (2000a). *Architectural Patterns: Broker* [WWW document]. URL http://www.edatamirror.com/openloop/softwareEngineering/patterns/architecturePattern/arch_Broker.htm

OpenLoop. (2000b). *Architectural Patterns: Layer* [WWW document]. URL http://www.edatamirror.com/openloop/softwareEngineering/patterns/architecturePattern/arch_Layers.htm

Orchard, D. (1998*). Enterprise Java Beans* [WWW document]. URL http://www.pacificspirit.com/Authoring/ObjectMag-Ejb/EJB.html

O'Ryan, C., Scmidt, D., & Wang, N. (2000). *Overview of the CORBA Component Model* [WWW document]. URL http://www.cs.wustl.edu/~schmidt/PDF/CBSE.pdf

Platt, D. (1999). *What The Heck Is COM+, Anyway?* [WWW document]. URL http://www.byte.com/column/BYT19990826S0030

Schmidt, D., & Vinoski, S. (1997). *Object Interconnections – Object Adapters: Concepts and Terminology* [WWW document]. URL http://www.iona.com/hyplan/vinoski/col11.pdf

Sousa, J.P. & Garlan, D. (1999). *Formal Modelling of The Enterprise JavaBeans Component Integration Framework*, [WWW document]. URL http://reports-archive.adm.cs.cmu.edu/anon/2000/CMU-CS-00-162.pdf

Sun Microsystems. (2000a). *Enterprise JavaBeans, Frequently Asked Questions* [WWW document]. URL http://java.sun.com/products/ejb/faq.html

Sun Microsystems. (2000b). *Jini Architecture Specification Version 1.1* [WWW document]. URL http://www.sun.com/jini/specs/jini1.1html/jini-title.html

Sun Microsystems. (2001). *Jini Network Technology FAQs* [WWW document]. URL http://www.sun.com/jini/faqs/

Szyperski, C. (2000). *ObjectiveView- Issue 5 : Components versus Objects* [WWW document] URL http://www.ratio.co.uk/ov5.pdf

Vinoski, S. (1997, February). *CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments* [WWW document]. URL http://www.iona.com/hyplan/vinoski/ieee.pdf

Vinoski, S. (1998, October). *New Features for CORBA 3.0* [WWW document]. URL http://www.iona.com/hyplan/vinoski/cacm.pdf

Yee, A. (1999). *Agent Adapters: Beyond APIs* [WWW document]. URL http://www.sdmagazine.com/articles/1999/9911/9911b/9911b.htm