MIPRO 2001, Opatija, Croatia, 21. - 25.5.2001

Component-Based Development – a New Approach in Software Development

Ivica Crnkovic, Magnus Larsson
Ivica.Crnkovic@mdh.se , http://www.idt.mdh.se/personal/icc
Magnus.Larsson@mdh,se, http://www.idt.mdh.se/personal/icc
Mälardalen University, Department of Computer Engineering
721 23 Västerås, Sweden

TABLE OF CONTENTS

Summary

Component-based development is a new paradigm in software development. The basic principles of this new trend is to re-use components that are developed independently of the final product. New products are developed by inclusions of the already completed components, and in this way the development time and costs can be dramatically decreased. Component-based software engineering has many advantages, but also many disadvantages. For example, it is more difficult to predict the behavior of external components built in the products. The maintenance and service require a new approach to achieve the quality and reliability of the products.

There are many consequences of component-based development. Huge, in-hose built monolith applications are being replaced by, component-based, flexible applications based on standard solutions. Similar (or the same) problems are being solved by using standardized components. Big software companies do not need as many software developers as before, they focus on the development on their "core-business", while the rest is bough, or outsourced. This implies emerge of many smaller specialized small companies. What are the trends in component-based software engineering, what are the consequences and what are the chances for smaller countries like Croatia to participate in this process? These are the topics of the report.

1 Introduction

Software systems are becoming increasingly complex and providing more functionality. To be able to produce such systems cost-effectively, suppliers often use component-based technologies instead of developing all the parts of the system from scratch. The motivation behind the use of components was initially to reduce the cost of development, but it later became more important to reduce the time to market, to meet rapidly emerging consumer demands. At present, the use of components is more often motivated by possible reductions in development costs. By using components it is possible to produce more functionality with the same investment of time and money. When components are introduced in a system, new issues must be dealt with e.g. dynamic configurations, variant explosion and scalability. Some of these issues are addressed with the discipline Component-Based Software Engineering (CBSE).

CBSE provides methods, models and guidelines for the developers of component-based systems. Component-based development (CBD) denotes the development of systems making considerable use of components.

Although very promising, CBSE is a new discipline and there are many associated problems which remain unsolved. Many solutions can be arrived at, by using principles and methods from other engineering disciplines, such as configuration management. This report describes some of these disciplines, presents proposals and analyses possibilities of applying different methods in CBSE.

2 A History of Component-Based Development

The Component-based development is close related to ***reuse***. The idea about reusing pieces of software originates from early sixties when the term Software Crises was mention first time. The basic idea is simple: When developing new systems use components that are already developed. When you develop specific functions you need in your system, develop it in that way that this function can be used by other systems in

the future. Although the principle is simple, it has been shown that the implementation is quite hard. Process of improving reuse has been long and laborious.

One of the earliest cases of successful reuse is the development of different libraries, for example mathematical libraries. These libraries include functions (for example mathematical functions such as sine, cosine, matrix operations, etc.), which are referred to in the source code and then linked together with the proprietary code.

The success of this type of reusable entities lies in several facts:

- There exists well-defined theory about these types of functions.
- The communication between the application and these functions is simple. It is of procedural type. The application invokes the functions sending to it input parameters, and the library responds by the execution of the function and returning the output parameters.
- The inputs and outputs are precisely defined
- Relative good error handling – If the inputs are erroneous, the output will usually return a specific value denoting an error.

A disadvantage and limitation of these types of components is the inflexibility. For a new version of the library, the application must be rebuilt. This problem has partially been solved by introduction of dynamic or shard library which can be loaded separately form the application. Another type of inflexibility is the limitations of types of input and output parameters. When changing types of parameters (for example using text elements instead of numbers in a sort function), a new library function must be used.

Another type of reusable entities we can find in application implementation in a form of client/server separation. The application (client) sends requests to the server, which provides a service to the application. Typical examples are databases or Graphical User Interfaces (GUI), such as X-windows. To make servers reusable a standard protocols in form of API (Application Programmable Interface) of the communication is defined. For relational databases there exists a language SQL. Different database providers use the same standard and in this way make it possible that different databases can be used by the same application without rewriting their code. In the X-Windows case, there exists a standardized API which is highly adaptable. The adaptation feature is very important for components because it enables many variations of its use. However it also introduces additional complexity, since a lot of parameters must be defined. For this reason additional API-s have been defined for different purposes and GUI styles.

From the business point of view, the companies are interested in developing the functions that will give the added value for the customers. They are not interested in developing general-purpose supporting functions or an infrastructure that makes possible execution of the “core-business” functions. A typical example of infrastructure is Operating System, and it is not strange that Operating Systems are typical reusable “components”. The problem with operating systems in the past was that they were very expensive and big, so in many cases the entire system had to include a lot of parts that never have been used but required resources which increased the costs. For these reasons many companies developed their own operating systems which were adjusted to requirements of the system.

In order to enable reusability the first step in the development is to divide the system in well-defined parts/components. These components can be developed internally. The next step in the evolution is to outsource the development, or to buy those parts that are

not of the primary interest for the company and if possible, replaced by standard components developed. Figure 1 shows an example of the system evolution. In eighties the systems were monolith, always developed from the very beginning, including the hardware development, the basic software development, such as operating systems and even development of the development environment itself, including compilers, debuggers, etc. In nineties the hardware part become more standardized and it was possible to buy it (for example PCs, or Unix workstations). The general-purpose operating systems have been used more and more as they became cheaper. Nowadays, typically, standard hardware and infrastructure software is used, as well as standard user interface. Only those parts that are directly related to the customer requirements are internally developed. This evolution has some important consequences. The development time is being decreased significantly, and the development costs have been reduced. However, another factors for the successful business become important: Many components must be bought and in this way the frame for the profit decreases. Also, the quality control on the system becomes more difficult, since the system includes parts from other providers.

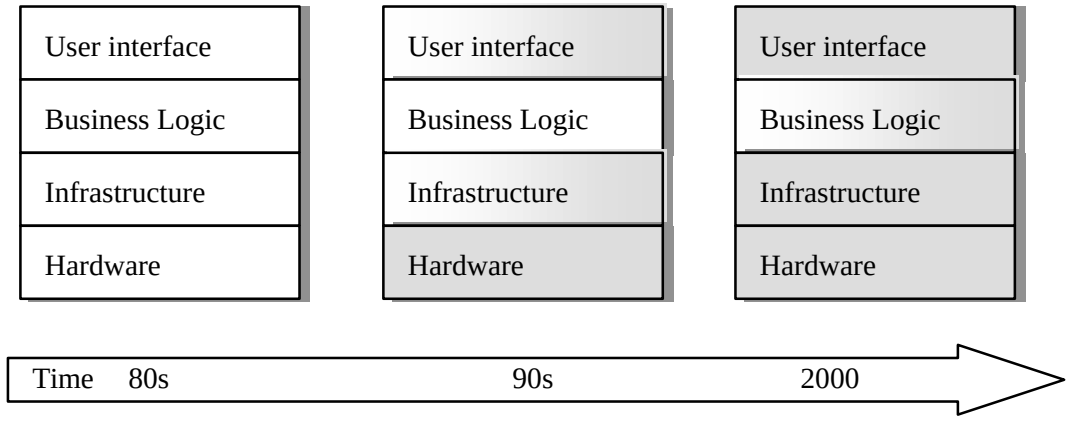

Figure 1. System reusability evolution

By emerge of the Internet and by establishing few operating systems, a requirement for running applications distributed over the network becomes important. Similarly, a new requirement of application compatibility between different operating systems becomes significant. The third requirement that is essential today is the ability of replacement of a component without re-building the application. These requirements, and the demands on the collaboration between applications independent of the operating system requirement lead to new paradigm of the software development: **Component-based Software Development.**

An example of integration of components at run-time can be seen in the Microsoft Office package. An MS Excel document can be a part of an MS Word document, and the opposite. Similarly, we can develop an application and used in it MS Excel “component” as a part of our application. The main advantage of this approach is the possibility of updating MS office, and getting new features in our applications, without rebuilding them.

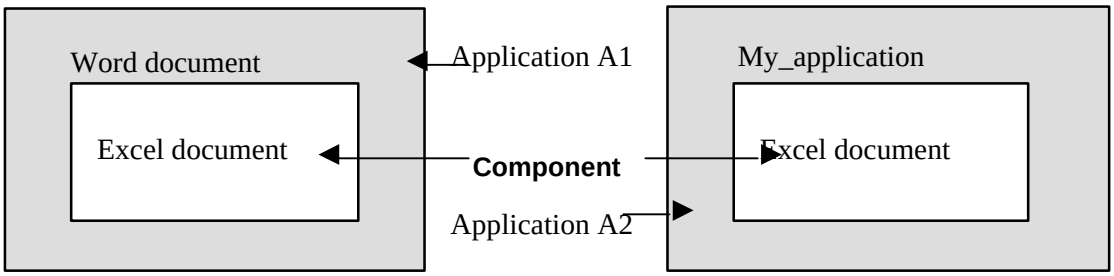

Figure 2. Integration of Microsoft Office applications

3 Software Architecture

The structure of software is traditionally not seen at system run-time. The structure is something that is defined during the design phase and it is used for easier development by dividing complex part in several relatively independent parts.

With the component-based development and recognition of parts of the system even at run-time, the structure design, also called architecture design, becomes one of the most important parts of the development process.

A component-based system is typically defined as n-tier structure, where n can be two, mostly three, or even four, five, etc. A tier, or a layer is a part of the application which provides a specific functionality (also called business logic) and has a well-defined interface to other layers. Figure 3 shows an example of a three-tiers architecture. The lowest level consists of a data repository, for example a relational database. The middle level presents the business logic, i.e. the functional and computational part of the application, where data accessed from data base are manipulated. The top tier presents a user interface, for getting input data and displaying results. Dividing the applications in these levels it is possible to make them independent of each other as much as possible. This in turn enables more flexibility for reusing standard components, or updating parts of the application. For example, the business logic part of the application does not need to be changed when we replace a database, or add a new graphical interface. It enables to use different interfaces applied on the same business logic part.

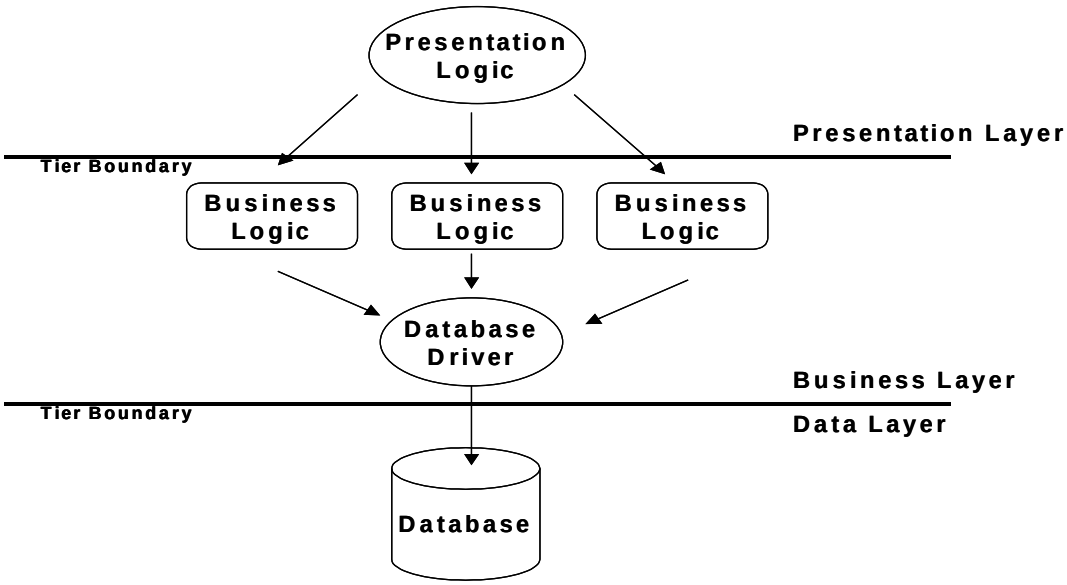

Figure 3. Three-tiers architecture

4 Component Definitions

What are components? Can they be uniquely specified, identified and processed? There have been a lot of discussions what a component actually is. “Components are for composition. *Nomen est omen*”. This is a quote that most people agree upon when discussing about what components are. But to come up with a precise and well-understood definition of a component, which everybody agrees upon, is not an easy task. The mean of the term has been changed during time and often has been related to the technology used in time.

Many have tried, but the result is a flora of different definitions that are slightly different. This phenomenon is very common when many different persons with varied backgrounds have used the word for different problem domains. Following are a variety of definitions specified in literature today:.

1. A component is a non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.

2. A **run-time software** component is a dynamically bindable package of one or more programs managed as a unit and accessed through documented interfaces that can be discovered at run-time.

3. 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 party.

4. A **Business component** represents the software implementation of an "autonomous" business concept or business process. It consists of all the software artifacts necessary to express, implement and deploy the concept as a reusable element of a larger business system.

We present all these different definitions to point out that it is not easy to make a unified definition. Before a component-based system is designed, a definition of component has to be agreed upon to set the context.

The software component definition is widely accepted today, which says that a component is a part of software in a binary form (i.e. it is not necessary to rebuilt it), with contractually specified interfaces (i.e. defined API and all assumptions in which the component can work), A component can be deployed independently (i.e. it can be dynamically loaded into the system, or dynamically replaced). It is a subject to composition by third party (i.e a component must have a mechanism which makes it possible to integrated it in the system without modifying and rebuilding it).

5 Interacting with Components

Components express themselves through interfaces. An interface is the connection to the user that will interact with a component. If an interface is changed the user needs to know that it has changed and how to use the new version of it.

Functions that are exposed to the user are usually called Application Programmable Interface (API). If there is a change to the API, the user has to recompile his code as well. This is not the case in interpretative languages like Smalltalk or Java, but for compiled languages such as C/C++.

In an object-oriented world, an interface is a set of the public methods defined for an object.
Usually the object can be manipulated only through its interface. In C++ the user has to recompile the code only when an interface, referred from the code, is changed. There is also a drawback that the user of the class must use the same programming language throughout the whole development.

Separating the interface from the implementation is a way to avoid this tight coupling. This kind of separation is made with binary interfaces as done in CORBA and COM, the component models described in the next section. Binary interfaces are defined in an interface definition language (IDL) and an IDL compiler, which generates stubs and proxies, makes the applications location transparent.

An example of using the same interface but different implementations is shown in Figure 4: By a separation between the interface and the implementation it is possible to run new clients together with old server components or vice versa. The word processor is called the client and the dictionary is called the server since it provides functionality to the word processor. It is possible to upgrade to new versions of the word processor and dictionary component independent of each other.

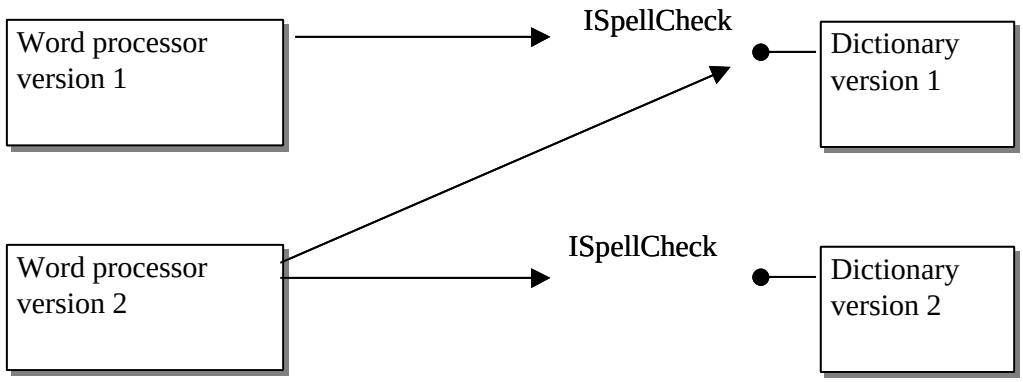

Figure 4. The possible combinations between old and new clients and their components.

Even if an interface has not been changed, its implementation can be changed. This increases flexibility of possible updates, but also introduces a possibility of having uncontrolled effects. For this reason, it is of interest to know if the implementation has been changed.

6 Component Models

The component models define the standards forms and standard interfaces between the components. They make it possible to components to being deployed and to communicate. The communication can be established between components on the same node (computer) or between different nodes. For the later we are talkies about component distribution.

Component models are the most important step to lift the component software off the ground. If components are developed independent of each other then it is highly unlikely that components developed under such conditions are able to cooperate usefully. The primary goal of component technology, independent deployment and assembly of components is not achieved.

A component model supports components by forcing them to conform to certain standards and allows instances of these components to cooperate with other components in this model.

The three major component models are used today with success. These three are COM, JavaBeans, and CORBA and all of them have different levels of service for the application developer. Table. 1 shows the corresponding technologies for each level of service.

	COM	**Java**	**CORBA**
Basic components	COM components	JavaBeans	CORBA objects
Distribution	DCOM	RMI	CORBA IIOP
Enterprise services	COM+	EJB/J2EE	CORBAServices

Table. 1. The different technologies used at different levels of service

Distribution is provided with a communication protocol that has been added to the basic component model. COM uses Distributed COM (DCOM), Java has Remote Method Invocation (RMI) and CORBA uses Internet Inter-ORB Protocol (IIOP). Support for business components can be found in COM+, EJB and CORBAServices.

There is a difference between systems that have their components tightly coupled together and those that have loose references between the components. In case of loose references the components connect to their fellow components when needed and not in the build phase. For these kinds of systems, it is much more a challenge to determine what the system will look like when it is started. To be able to predict the behavior we need to know which components will cooperate. All three models presented in this section are loosely coupled with support for dynamic invocation and lookup.

6.1 Component Object Model (COM)

The Component Object Model provides a model for designing components that have multiple interfaces with dynamic binding to other components. COM is an open standard, which has been implemented on many different platforms, but the main platform is of course Microsoft Windows for which it was first developed. Components expose themselves through interfaces and only interfaces. The interfaces are binary which makes it possible to implement the component in a variety of programming languages such as C++, Visual Basic and Java. A COM component can implement and expose multiple interfaces. A client uses COM to locate the server components and then it queries for the wanted interfaces.

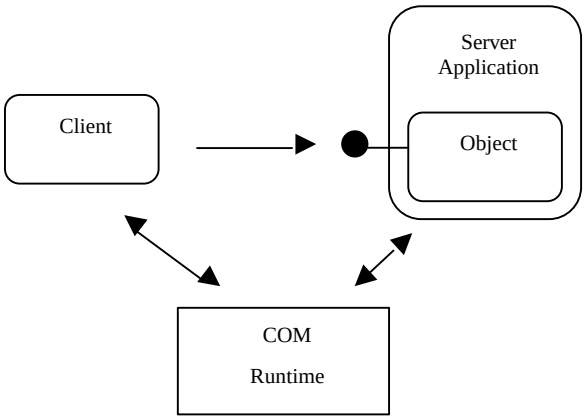

Figure 5. COM establishes the connection between client and server.

By defining interfaces as unchangeable units, COM solves the interface versioning problem. Each time a new version of the interface is created a new interface will be added instead of changing the older version. A basic COM rule is that you cannot change an interface when it has been released. This makes couplings between COM components very loose and it is easy to upgrade parts of the system indifferent from each other.

DCOM is the protocol that is used to make COM location transparent. A client talks to a proxy, which looks like the server and manages the real communication with the server.

COM+ is an extension to COM with technologies that support among others: transactions, directory service, load balancing and message queuing. Figure 6 shows how clients can connect, through an Internet Information Server (IIS) or DCOM, to the business logic, which is implemented with COM+. The business logic uses ActiveX Data Objects (ADOs) to access the data in the databases. Compare this picture with the EJB technologies to se the similarities.

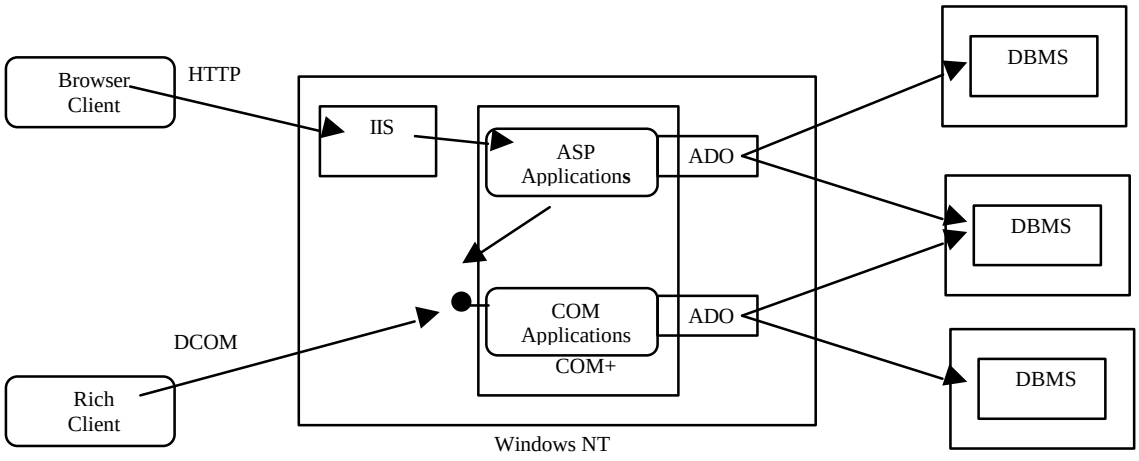

Figure 6. The principal architecture of how COM+ is used in a three-tier architecture.

6.2 Enterprise Java Beans (EJB)

Enterprise Java Beans is a component architecture for server-side components used to build distributed systems with multiple clients and servers. A Java Bean is a reusable component that support persistency and can inter-operate across all platforms supported by Java. EJB uses Java Beans but it is a lot more than a component model. EJB provides support for transactions and security over a neutral object communication protocol, which gives the user the benefit to implement the application on top of a protocol of choice. EJB is part of the Java 2 Platform Enterprise Edition (J2EE) which includes many other technologies remote method invocation (RMI), naming and directory interface (JNDI), database connectivity (JDBC), Server Pages (JSPs) and Messaging services (JMS).

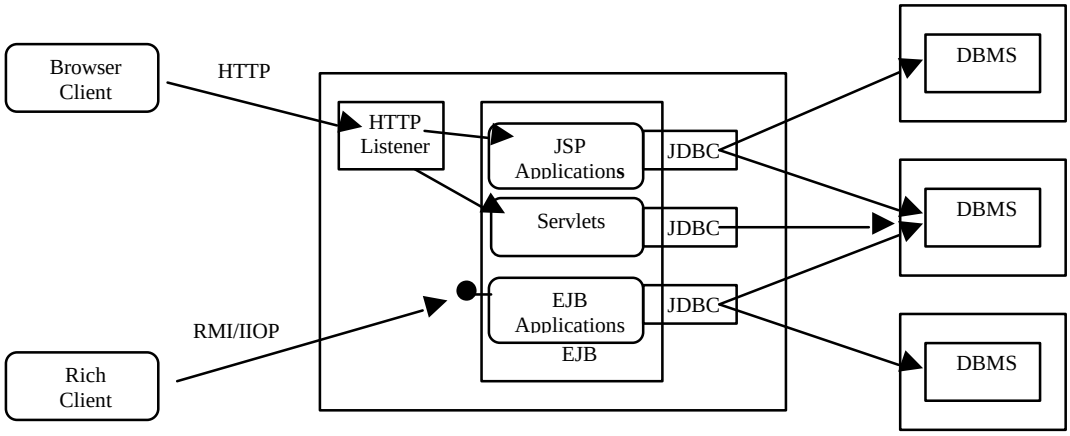

Figure 7. The principal architecture of how EJB is used in a three-tier architecture.

Figure 7 shows the architectural style of EJB used in a three-tier application. The clients connect to the server components through either a web server or directly using remote method invocation (RMI). The server components that implement the business logic reside within an EJB container with the support for transactions and security. The data is stored in databases, which are managed with some database management service (DBMS) and is accessed through the data base connectivity component (JDBC). Java server pages (JSP) or servlets are used when the thin web clients access the system through the Internet. Compare Figure 7 with Figure 6 to see the similar technologies used in the COM+ environment.

To make a JavaBean an Enterprise bean the JavaBean has to conform to the specification of EJB by implementing and expose a few required methods. These methods allow the EJB container to manage beans in a uniform way for creation, transactions etc. A client to an enterprise bean can virtually be anything, for example a servlet, applet or another enterprise bean. Since enterprise beans may call each other then a complex bean task might be divided into smaller tasks and handled by a hierarchy of beans. This is a powerful way of “divide and conquer”.

There are two different kinds of enterprise beans: session and entity beans. Session beans live as long as the client code that calls it. Session beans represent the business process and are used to implement business logic, business rules and workflow.

EJB is designed so it can run together with CORBA and access CORBA objects easily.

6.3 Common Object Request Broker Architecture (CORBA)

The Common Object Request Broker Architecture (CORBA) is a standard that has been developed by the Object Management Group (OMG) in the beginning of the nineties. The OMG provides industry guidelines and object management specifications to supply a common framework for integrating application development. Primary requirements for these specifications are reusability, portability and interoperability of object based software components in a distributed environment. CORBA is part of the Object Management Architecture (OMAwhich covers object services, common facilities and definitions of terms.

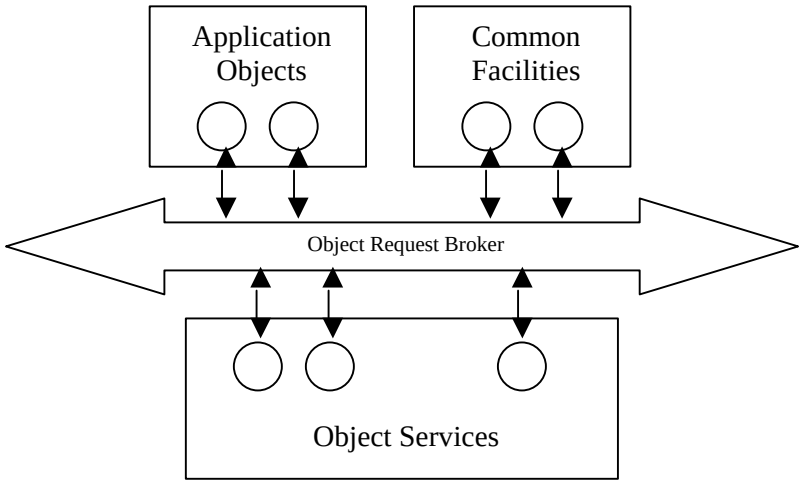

Figure 8. The parts of the Object Management Architecture.

Object services are for instance naming, persistency, events, transactions and relationships. These can be used when implementing applications. Common facilities provide general-purpose services like information, task and system management. All services and facilities are specified in IDL. An object request broker (ORB) provides the basic mechanism for transparently making requests and receiving responses from objects located locally or remotely. Requests can be made through the ORB without regard to the service location or implementation. Objects publish their interfaces using the Interface Definition Language (IDL) as defined in the CORBA specification.

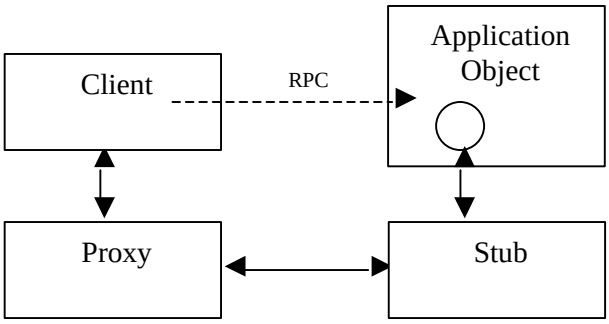

Figure 9. Clients communicate with RPC transparently with the server.

Objects are stored in an interface repository where they can be found and activated on demand from the clients. The stubs and proxies are generated from the IDL specification that each object provides for its interfaces.

7 Commercial Off The Shelf

Commercial Off The Shelf (COTS) is a common way to gain functionality without having to write everything ourselves. Components are sometimes wrongly referenced as COTS, Certainly, components might be COTS but it does not mean that COTS have to be components. A vendor sells COTS products as unmodified units that can be used for development.

When a system is designed with third-party components then it is common to talk about commercial off the shelf (COTS) components. Development with COTS has many advantages:

- Functionality is instantly accessible for the developer.
- The components may be less costly compared to in-house development.
- The component vendor may be an expert in the particular area of the component functionality.

However, along with all the advantages, there are also several disadvantages:

- A COTS component has often only a brief description of its functionality.
- The component carries no guarantee of adequate testing.
- There are no or only a limited description of the quality of the component.
- The developer does not have access to the source code of the component.

Knowing all the disadvantages, buying COTS components is not an easy task. COTS components are typically “black boxes” with their source code not available. Developers have to identify certain properties of COTS components to properly integrate them with a system under development. A property of a component is its characteristic that the developer needs to understand to do the integration. Examples of component properties are functionality, limitations, correctness, preconditions robustness and performance. To get to know what the properties are, extensive testing of the component has to be carried out. There are various approaches to do this kind of testing e.g. Random, “black-box” and “white-box” test generators.

COTS components can be categorized in groups where the functionality is the same. If there are more than one vendor of a component it is beneficial to design the system for component exchangeability. An architecture that supports the exchange of component with the same functionality is more stable if the support for a used component is dropped, since a new one can replace the obsolete one.

8 Outsourcing

One way of getting external components is to buy them as COTS. Another way is place the development into another development organization. This process is designated as outsourcing. There are many similarities between COST and outsourcing – in both cases software is developed somewhere else. The main difference is the possibility to control the development process in the case of outsourcing.

There are many reasons why it can be profitable for a company to outsource a part of its development. The two main factors which motivates the companies to do outsourcing are: Time to market and reduced costs. By outsourcing, the development process time will usually decrease as the development can be done in parallel. Another

strong motive is the cost reduction. If another company can develop software for significantly less costs it is profitable for the first company to outsource that development to that company. Very often the subcontracting companies are placed in developing countries where software developer are paid much less, but have high competence.

The experience has shown that outsourcing is not as simple as it can be expected. There are numbers of problems which can raise, and which can have political, economical and cultural origins. Very often communication between the partners is not sufficiently good, or the expectations from both sides are different. In many cases there are problems in insufficiently defined requirements or specifications, not enough precisely defined deliverables, bad calculated costs, hidden costs, and so on.

For this reason the contract (also called subcontract) is crucial for the successful outsourcing. The contract must clearly specify the interface between the partners, the inputs and outputs from both sides. Even the development process may be a part of a subcontract.

Outsourcing of development only is not enough. The improvement, adoption, and in general maintenance should also be a part of the contract since there exists no software that does not require maintenance.

9 Component Based Development Process

This section describes the differences of how to develop components and how to develop with components. It is important to do this distinction to make it clear how to use different methods. The developer of a component has to think about how to make the component open for integration with other components and not so much about how to integrate other components.

Certain advise for the developers and users of components are given in this section. As a result of studying different aspects of component-based systems, we provide a list of advises for this area. This section is divided into two parts, one for the component developer and one for the component integrator (the user of components).

9.1 Development Cycle

The development cycle of a component-based system is different from the traditional ones. For instance the waterfall, iterative, spiral and prototype based models.

Development with components differs from traditional development. There is a new development process for CBSE and it differs from the traditional waterfall model. Figure 10 shows a comparison between the two different development processes. Gathering requirements and design in the waterfall process corresponds to finding and selecting components. Implementation, test and release correspond to create, adapt, deploy and replace.

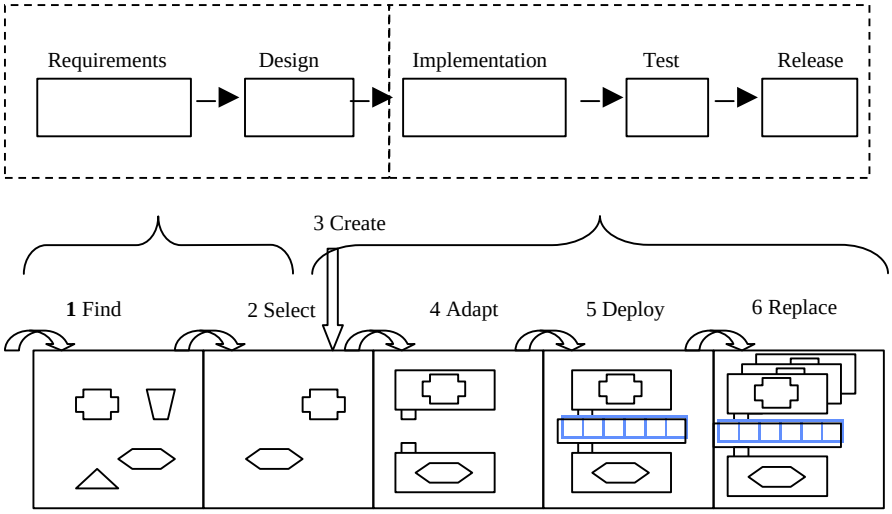

Figure 10. The development cycle compared to the waterfall model.

The different steps in the component development process are:

1. Finding components that may be used in the product. Here all possible components are listed for further investigation.
2. Select the components that fit the requirements of the product.
3. Create a proprietary component that will be used in the product. We do not have to find these types of components since we develop them ourselves.
4. Adapt the selected components so that they suit the existing component model or requirement specification. Some component needs more wrapping than others.
5. Compose or deploy the product. This is done with a framework or infrastructure for components.
6. Replace old versions of the product with new ones. This is also called maintaining the product. There might be bugs that have been fixed or new functionality added.

All the different steps have different technologies to support the developers.

9.2 Developing Components

When developing and designing components, we recommend the following advises:

- ?? Always document all the features of the component. Do not restrict the documentation to functionality and document all other properties as well. E.g. Performance, resource consumption, limitations, robustness, etc.
- ?? Provide test-suites with the component so that the customer can test your component in their environment. It is extremely important to test an imported component in the environment it will operate. Remember the Ariane 5 rocket explosion that was due to a change in the environment requirements and not in the software design.
- ?? Provide source code if possible, it might help the application developer to understand the semantics of your component.
- ?? Make the components so they easily integrate into existing component frameworks. Describe what frameworks the component work with and describe how to make it work with other frameworks as well.

- Components need to be carefully generalized to enable reuse in a variety of contexts. However, solving a general problem rather than a specific one takes more work.
- Make sure that the application developers can adopt the component to their requirements. This can be done with sink interfaces where the user adds its own interface to the component so that the component can use that interface to communicate with the user.

9.3 Developing with Components

Before acting and taking decisions on how to build applications from components, we recommend that the following questions and thoughts be considered:

- The time your product is off the market can be greater than the time saved getting your product to market if your component supplier drops the product. Can you accept this risk?
- The functionality provided by the component may not remain precisely what you need over time, forcing you to create wrappers that get around this. Things are getting even worse if you are not getting support from the vendor.
- The functionality of the component may be more than you actually need, requiring you to write restrictive wrappers for functionality that you do not want to be used. Use of unintended functionality may cause problems.
- If you succeed to get the source code from the component vendor, can you really maintain it if something goes wrong?
- A malfunction in the component may cause an error in your product. Are willing to have a certification strategy for this. Your customer wants your product to work without having to think about your internal design. You have to provide the fix of the problem even though the error is in the third-party component.
- If you ask the component vendor to customize the component for you, are you aware that you now are strongly dependent on the vendor? The vendor can charge you anything they please.

We have the recommendations to the component integrator:

- Make a thorough evaluation of the component suppliers. Are they suitable as a supplier? Do they have good quality products and support? Check their economy so they don’t easily bankrupt.
- Put a lot of effort into the legal agreement with the supplier. This may save you if the supplier goes out of business or if they refuse to support you.
- Create good and long term relations with the supplier for better cooperation.
- Limit the number of partners and suppliers. To many will increase the costs and the dependencies.
- Buy “big” components where the profit is greatest. The management of to many small components can consume the profit.
- Adjust the development process to a component-based process.
- Have key persons that are assigned to supervise the component market. They shall keep track of new components and trends.
- Try to get access to the source code.

- Test the components in your environment.
- All these advices do not give a complete solution to all the problems that have to be dealt with but they state that developing for and with components has to be carried out carefully with a second thought.

10 Business Opportunities for small Countries

The modern component-based technology and the Internet have changed radically the software development process. Twenty and even ten years ago, only large companies could produce large software. The equipment was expensive, the development environment was expensive, and software was strictly connected to the equipment. Large software consisted of monolith large applications, which were difficult to maintain and improve. For all these reasons it was very difficult for smaller companies to compete with large companies. A similar situation was valid for smaller and economically weaker countries. It was impossible to compete with strong countries, simple because the technology was too expensive.

By emerge of PC-technology two factors had crucial impact to changes in the software production. The hardware production was separated from the software development. Hardware became standardized. The barrier monopoly of software/hardware development has been broken. With the standardization of hardware the competition opportunity has dramatically increased, which had direct impact on the prices. The cheaper hardware has opened possibilities for competition within the software development. The result was much cheaper software. Now a private person can afford the same computer configuration as in office.

However, the software development remained complex and it required more and more efforts. The reason for that is that requirements become more complex and that the domain of computer use has dramatically increased. Also the demands on integration of different types of applications and systems became more important. The system have turned from closed and dedicated to open, and dedicated parts integrated with the general-purpose parts. In a way a paradox has happened – hardware is getting cheaper, development tools cheaper, but the software development becomes more expensive. In a system development that exist of both hardware and software, the costs for the software development become dominant (one example, the costs of robots development at ABB, Sweden is 90% for the software development and 10% for the hardware development. This means that the intellectual property becomes dominant. The production becomes less significant and often is placed in countries with cheaper working power.

The extensive exploitation of Internet has consequences such as easy access to information and easy access to software. The competition becomes much harder, but this time it is the knowledge that matters.

The component-based approach, in combination with Internet, will revolutionary change the software development process. Everything can be found on the Internet today. In the component-based approach it is much easier to use a general-purpose component already developed by someone else, than develop it itself. At the same time it is much simpler to advertise the products and place them on the market. The expensive agents and advertising, the direct contacts are still important, but not the only way to succeed on the market. A good and fast web side is a big advantage.

This new paradigm of software development is still under the process of change change, there is a chance for new actors to appear on the scene. Not only new

companies, but also new countries or new regions. To succeed the following prerequisites must exist (condition sine qua non):

- Communication and Internet infrastructure
- Knowledge
- Worldwide cultural and economic integration

This is not easy to achieve. The building up of the infrastructure requires systematic approach and strategic decisions from the state and the government. Today, this infrastructure is as important as the classic infrastructure (such as roads), and those countries which will neglect this, will severe servility in the nearest future.

Building up knowledge is a much longer process. It must happen on two levels – high competence of professionals and a broad general knowledge of “ordinary” citizens. This means that the education strategy must have two tracks – the high level education with the research academy education in focus, and the education systems at primary and secondary schools.

The integration with high development countries is possible achieve by opening the borders, participate in the common projects and any kind of events, and finally the importance of a knowing English is crucial.

Many smaller countries have today a chance to join the high development countries, and it is important not to miss that chance, because it can happen that in the future there will be no more chances.