Component-based Software Development

Abdille Hagi Abdullahi
Component-based Software Development

Abdille Hagi

June 22, 2008
Abstract

Component-based Software development is a promising way to improve quality, time to market and handle the increasing complexity of software management. However, The component-based development is still a process with many problems, it is not well defined either from theoretical or practical point of view. This thesis gives a brief overview of Component-Based Software development and starts with brief historical evolution followed by a general explanation of the method. A detailed discussion of the underlying principles like components, component framework and component system architecture are then presented. Some real world component standards such as .net framework, CORBA CCM and EJB are given in detail. Finally, simple file-sharing-program based on Apache’s Avalon framework and another one based on .net framework are developed as a case study.
Contents

1 Introduction  1
   1.1 Motivation ......................................................... 1
   1.2 Background .......................................................... 2
   1.3 Objectives .............................................................. 3
   1.4 Structure of Report ............................................... 5

2 Component  6
   2.1 Introduction ............................................................ 6
   2.2 Component Characteristics ......................................... 7
   2.3 Component and Object ............................................... 8
   2.4 Component Development ............................................. 9
   2.5 Target Framework .................................................. 10
   2.6 Component Programming Languages ............................... 11

3 Component Frameworks  12
   3.1 Introduction ........................................................... 12
   3.2 Component System Architecture ................................... 13

4 .net Platform  15
   4.1 Introduction ........................................................... 15
   4.2 Common Language Infrastructure ................................. 17
   4.3 Common Language Runtime ......................................... 17
   4.4 COM ................................................................. 17
   4.5 Assemblies - the .net Software Components .................... 18

5 Enterprise JavaBeans, the Sun way  20
   5.1 Introduction ........................................................... 20
   5.2 Types of Beans ........................................................ 21
5.3 Session Beans ........................................ 22
5.4 Entity Beans ........................................ 24
5.5 Message-driven Beans ................................. 26

6 CORBA CCM, the OMG way .......................... 27
   6.1 CORBA Overview .................................. 27
   6.2 CORBA Component Model .......................... 28
   6.3 CCM Framework .................................... 32

7 Case Study Based on Avalon Framework ............. 33
   7.1 Introduction ........................................ 33
   7.2 Problem Statement .................................. 34
   7.3 Design Decisions .................................... 35
   7.4 Server Component Development .................... 36
   7.5 Tests and Result ................................... 39

8 Case Study Based on .net Framework ................ 42
   8.1 Introduction ........................................ 42
   8.2 Problem Statement .................................. 43
       8.2.1 The Server ..................................... 43
       8.2.2 The Client ...................................... 43
   8.3 Design Decisions .................................... 43
   8.4 Server Component Development .................... 44
       8.4.1 Private and Public Assemblies ................ 44
       8.4.2 Viewing the GAC ................................ 46

9 Discussion and Evaluation ............................ 48
   9.1 Assessment of the Three Major Component Worlds . . 48
   9.2 Case Study Assessment ............................. 50
       9.2.1 Using Avalon Framework ........................ 50
       9.2.2 Using .net Framework ........................... 53
   9.3 Assessment of the Two Approaches ................. 54
       9.3.1 Component Security ............................. 54
       9.3.2 Version Control .................................. 56
       9.3.3 Language Independence .......................... 58
   9.4 Designing Component for Reuse ..................... 59
       9.4.1 Designing for Reuse ............................. 60
       9.4.2 Designing with Reuse ............................ 60
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.5 Component Search</td>
<td>61</td>
</tr>
<tr>
<td>9.6 Conclusion</td>
<td>62</td>
</tr>
<tr>
<td>9.7 Achievements</td>
<td>64</td>
</tr>
<tr>
<td>A Case Study Design Specification</td>
<td>67</td>
</tr>
<tr>
<td>B Avalon Framework</td>
<td>68</td>
</tr>
</tbody>
</table>
Acknowledgements

My sincere thanks to my tutor Tim Butler for his support and advice during the thesis. Thanks also to my flatmate Dave Walter who reviewed the report and provided helpful and constructive feedback.
Chapter 1

Introduction

In this chapter, the main objectives of the report are presented. First a general introduction of software development is given assuming the reader is already familiar with basic software development. We then compare that with component-based software development. Next, the historical background of component-based software development is reviewed. Finally, report objectives are outlined.

1.1 Motivation

The world of software development has evolved quickly in the last decade. Over the decades, software has entered all domains that make up the necessary functions of our civilization. Virtually all domains have incorporated software to achieve better functionality that could not be achieved earlier. Some examples are telecommunication, electricity generation and distribution, logistic systems, modern cars and airplanes. Software is becoming more and more an integrated part of the society. For instance, software has allowed new organizational forms, such as business process reengineering and virtual organizations which allow organizations to operate at efficiency levels that are an order of magnitude better than earlier models. In addition all kinds of devices, which increase efficiency, would not be available without software -for example current mobile phones.

In the ever-changing world of software development, organizations are gradually realizing the need for explicit software architecture for their systems. And software engineers are potentially faced with the challenge of
developing multiple, inter-related systems from the scratch while pressured by financial and time constraints. It is therefore essential to establish a well-defined design method in order to build high-quality, reliable, and easily maintainable component or families of systems -the question is now how to approach this?

The last decade has shown that object-oriented technology alone is not enough to cope with the rapidly changing requirements of present-day applications. One of the reasons is that although object-oriented methods encourage one to develop rich models that reflect the objects of the problem domain, this does not necessarily yield software architectures that can be easily adapted to changing requirements. In particular, object-oriented methods do not typically lead to designs that make a clear separation between computational and compositional aspects [Bos00]; this separation is common in component-based systems.

Component-based systems, on the other hand, achieve flexibility by clearly separating the stable parts of the system (i.e. the components) from the specification of their composition. Components are black box entities that encapsulate services behind well-defined interfaces. These interfaces tend to be very restricted in nature, reflecting a particular model of plug-compatibility supported by a component-framework, rather than being very rich and reflecting real-world entities of the application domain.

Building new solutions by combining existing and made components improves and supports rapid development, leading to a shorter time to market. At the same time, quick adaptation to changing requirements can be achieved by investing only in key changes of a component-based solution, rather than undertaking a major release change.

Bringing the levels of reuse in software development into line with that in other engineering disciplines has been a long sought after but satisfactory goal of software engineering was not reached. In the last few years however, new development paradigms have emerged which promise to radically change this situation, and allow software engineering to claim its rightful place in the family of engineering disciplines, that is the component-based software.

1.2 Background

Component-based software development (CBSD) has its roots originating in the late sixties. Software development back then was, and still remains
today, a very young discipline contrasting its counter parts such as civil, mechanical and electrical engineering. As demand for software increased such as usability, robustness, reliability, flexibility and adaptability, major problems in software development were run into: Bad software quality, very low productivity and cost explosion.

As a result of the constant failure of system developers to deliver software systems within budget, on time, and of satisfactory quality for use, NATO called for a conference in 1968 to deal with the so-called “Software Crisis.” A report written by Doug McIlroy in 1969 on “mass-produced software components” first introduced the idea of “software component”. He proposed a software industry that supplied off-the-shelf standard software components. He predicted that developers would build software solutions from existing software components instead of building them from scratch.

In 1986, Brad Cox published a book on object-oriented programming [Cox86] wherein he introduced the idea of “software integrated circuit”, which are pieces of software that can be “plugged” or “unplugged” from a system and are as well replaceable and reconfigurable. Just like an electrical integrated circuit, or a computer hardware component, the software equivalent will provide a fixed function and an interface specification without revealing any of its internal implementation. Cox also stressed the fact that these circuits or components can be sold on the open market.

In 1975, Fred Brooks published his famous book The Mythical Man-Month, wherein he envisioned that there would not be “silver bullet” as software is inherently complicated. Brook’s prediction still remains true at present and is repeatedly mentioned, as a reason why the “software crisis” occurred in the first place and still exists. It is on the other hand believed that CBSD represents the “best practices” in software engineering produced over the last decades.

1.3 Objectives

In this section I will identify the specific things I am trying to achieve by doing this thesis. I would identify core and advanced objectives.

Core objectives are those that I expect to achieve: if I do not achieve most of them in full, and all of them in part, my thesis will not have been a success.
**Advanced** objectives are those that I would like to achieve, but which are dependent on how well I achieve my core objectives. My thesis can still be a success even if I do not achieve them. I will probably have a shopping list of advanced objectives, from which I intend to choose once I have made significant progress with my thesis.

The primary goal of my master’s thesis is to study the current state of component technology. How to use them in software system, design, implementation and maintenance. In details, here are the goals I want to achieve.

**Core**

Find out:

- What are benefits of using components?
- What component technologies are available from component vendors.
- How to design components intended for reuse.
- How to identify components needed for particular purpose, (component search).

**Carry out:**

- Take a component framework specifications and specific component requirements and take these through analysis and design to implementation of components.
- Take application requirements, select appropriate components - and perhaps component frameworks - and assemble (compose) these into a working application.

**Advanced**

**Component framework architect**. Take framework requirements and take this through analysis and design to implementation of framework.
1.4 Structure of Report

The following chapter, chapter two, opens the scene and provides definition of components and component characteristics. I also put component in relation with object. Next, component development approaches are presented. Chapter three explains the important concepts of component framework and component system architecture. Further, chapter three defines and describes component and component characteristics as well as component development. Component system are treated in chapter four.

Based on the above developments and ideas, a number of approaches and technologies are trying to capture their share of the emerging component technology. The next three chapters (chapter 4 to 6) present a detailed account of the three major approaches followed today. These are CORBA-centered CCM standards, which emerged mainly from the world of legacy application integration; Sun’s EJB-centered standards which are now especially strong in the application and web server space; and, finally, .net framework, which evolved out of Microsoft’s strong position in both server side and desktop areas.

Chapter seven and eight presents component development case study based on Avalon framework and .net framework. I explain there how I developed a file-sharing-program based on both of these frameworks. I placed an introduction of Avalon framework as an appendix, so please refer to Appendix B for introduction of Avalon framework. Chapter nine concludes the thesis. You will find there discussion and assessments of my findings as well as achievements.
Chapter 2

Component

This chapter considers component-based system design and outline first component characteristics. Then, we compare component-oriented with object-oriented. We also introduce in this chapter the important concept of component framework. We finally analysis of suitable component programming languages.

2.1 Introduction

Component-based system design, like all modern engineering design, involves reasoning about the behavior of possible system, that is, system that have not been built yet. Similarly, maintenance of this system involves reasoning about the behavioral impacts of possible changes (for example, component substitutions).

Good system designers use variety of general principles to guide system design. One such objective is trying to design systems whose behaviors are easy to analyze and understand.

The importance of this principle comes from the need to compare design alternatives. This requires the ability to analyze quickly the behaviors of many possible systems, most or all of which have not actually been built yet.

One way to design systems with effectively analyzable behavior is to strive for modularizing it, that is, to design systems as assemblies of standard components whose behaviors are already understood in isolation each one of them [Szy98]. This needs be done carefully enough that a system’s behavior is predictable from the individual behaviors of its individual components and
their local interconnections to other components in the system.

Software components are considered, by many researchers [Som99], to be analogous to hardware components in general and to integrated circuits in particular. Other related analogous are Software bus and software backplane. More far fetched are analogies from mechanical engineering - gears, nuts, and bolts, for example.

All the analogies seem to give the impression that the whole world, with the one exception of software technology, is already component oriented! There therefore it supposed to be possible - if not straightforward - to follow the analogies and introduce components to software as well.

2.2 Component Characteristics

Unfortunately, the concepts related to component technology include many aspects. The massive overloading of the term “object” is the best example. The terms component and object are often used interchangeably. Sometimes, the notions of module, class, and component have all become combined by the term “object.” (Sometimes, the same is done to the term “software component”) Combining several terms into one can simplify things apparently, but not to good advantage beyond the simplest thoughts.

The characteristic properties of a component are that it:

- is a unit of independent deployment;

For a component to be independently deployable, it needs to be well separated from its environment and other components. A component, therefore, encapsulates its constituent features. Also, as it is a unit of deployment, a component will never be deployed partially. In this way, a third party is one that cannot be expected to have access to the construction details of all the components involved.

- is unit of third-part composition;

For a component to be composable with other components by such third party, it needs to be sufficiently self-contained. Also, it needs to come with clear specifications of what it requires and what it provides to the host system and other components. In other words, a component needs to encapsulate its implementation and interact with its environment by precise and clear interfaces.
• has no observable state.

A component have not any (externally) observable state - it is required that the component cannot be distinguished from copies of its own. The specific exclusion of observable state allows for acceptable technical uses of state that can be crucial for performance without affecting the observable behavior of a component. In particular, a component can use a state of caching purposes [Szy98].

A component can be loaded into and activated in a particular system. However, due to the stateless characteristic of components, it makes no sense to have multiple copies in the same operating system process.

In many current approaches, components are heavyweight units with exactly one instance in a system [Szy98].

2.3 Component and Object

The concepts of instantiation, identity, and encapsulation lead to the concept of objects. In contrast to the properties characterizing components, the characteristic properties of an object are that it:

• Is a unit of instantiation, it has a unique identity;
• May have state and this can be observable;
• Encapsulates its state and behavior;

Again, a number of object properties directly follow these characteristics. Because an object is a unit of instantiation, it cannot be partially instantiated. Since an object has individual state, it also has a unique identity that is enough to identify the object despite state changes for its entire lifetime.

As objects are instantiated, there needs to be construction plan that describes the state space, initial state, and behavior of a new object. Also, that plan needs to exist before the object can come into existence. Such a plan may be explicitly available and is then called a class. Alternatively, it may be implicitly available in the form of an object that already exist - that is, sufficiently close to the object to be created, and can be clonal. Such a pre-existing object is called a prototype object.

Whether using classes or prototype objects, the newly instantiated object needs to be set to an initial state. The initial state needs to be a valid
state of the constructed object, but it may also depend on parameters specified by the client asking for the new object. The code needed to control object creation and initialization can be a static procedure - usually called a constructor if it is part of the object’s class. Alternatively, it can be an object of its own - usually called a factory object. Methods on objects that return freshly created other objects are another kind - usually called factory methods [Szy02].

Component act through objects and therefore normally consist of one or more classes. 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 classes only, or even to contain classes at all. Instead, a component could contain traditional procedures and even have global (static) variables, or it may be realized in its entirety using a functional programming approach, or using assembly language, or any other programming approach.

2.4 Component Development

As object-oriented programming addresses the fundamental aspects of programming object-oriented solutions, component-oriented programming addresses the aspects of programming components. A definition of component-oriented programming [Szy98] in the style of typical OO programming definition is that component-oriented programming requires support of:

- polymorphism (substitutability);
- modular encapsulation (higher-level information hiding);
- late binding and loading (independent deployability);
- safety (type and module safety).

Component-oriented programming is a young discipline, and a lot of work remains to be done. A proper methodology for component-oriented programming is yet to be found. Most existing methodology work only within a component. The difficulties resulting from the complex interactions with other components are not sufficiently covered. Yet, some significant progress towards component-oriented development methodology has been made recent years. Several researchers address the unique combination of top-down
(starting from requirements) and bottom-up (starting from existing component) approaches that component orientation demands, for instance, \cite{Atk02} and \cite{Che00}.

Some issues can be addressed at the level of programming languages and approaches. Connection-oriented programming in particular. Component may need to call other components to perform certain task. This corresponds to normal programming in which abstractions depend on other abstractions. It is also said that normal programmed calls correspond to a pull model of programming. The information is “pulled” in as needed. Information may also “pushed” out as it arises, rather than being pulled by a procedure call invocation. With such relation in place, it is logic to say that caller and callee are “connected”

2.5 Target Framework

A component need to be placed in a component framework to function, it cannot function outside its defined environment. Component frameworks define such environments. On the other hand, a component object can be designed to operate in multiple such environments at the same time.

Depending on the component system architecture, frameworks are separated according to various roles. For example, each framework may take care of one particular mechanism that operates across components. In this case, a distribution framework may be responsible for distribution of component instances across machines. A separate framework would be responsible for compound document integration. A component may well need to interact with both frameworks to implement objects that can be distributed and that function within a compound document.

While the first component frameworks are just appearing on the marked, there are no proper integrating component system architectures. It is danger to invest heavily in component solutions based on a single framework. Because it is almost impossible to “divorce”.

Although application frameworks are very successful, it is almost impossible to shift existing component from one framework to another. It is already difficult to combine multiple traditional frameworks as most of them have been designed in total isolation. However, it is even harder to shift a solution from one framework to another with similar functionality. This is commonly required as providers of frameworks go out of business, no longer
support all the platforms required by a solution, or “better” frameworks become available.

2.6 Component Programming Languages

Programming languages in principle, component programming could use almost any language - and almost any paradigm. However, minimal requirements exist. Component programming rests firmly on the polymorphic handling of other components [Szy98]. As interactions with other components need to be dynamic, late binding has to be supported. Safety-by-construction arguments additionally ask for support of encapsulation and type-safety, module-safety and garbage collection in most cases. Additionally component programming requires way to explicitly state dependencies and, ideally, keep such dependencies configurable. At the language level, the object-oriented paradigm comes closest to expanding into the area of component-oriented programming.

The number of programming language that support component-oriented programming at a helpful level is still quite small. Many mainstream languages, such as COBOL, Object COBOL, FORTRAN, C, C++, Pascal Object Pascal, Module-2 Eiffel, or Smalltalk lack the support for encapsulation, polymorphism, type safety, module safety or any combination of these [Szy02]. Probably the most important component-oriented languages at this time are Java and C#. Other reasonably component-oriented languages are Ada 95, Module-3, Oberon and Component Pascal.
Chapter 3

Component Frameworks

This chapter enlightens the important concepts of component framework and component system architecture which components need in order to function.

3.1 Introduction

A component framework is a collection of rules and interfaces (contracts) that regulate the interaction of components plugged into the framework. A component framework typically enforces some of the more important rules of interaction by encapsulating the required interaction mechanisms. The concept is applied sometimes hierarchically, such that component frameworks are themselves components plugging into higher-tier component frameworks.

Component frameworks are clearly the most important step for lifting component software off the ground. Most current emphasis has been on the construction of individual components and basic “wiring” support of components. It is unlikely that components developed independently under such conditions are able to cooperate with each other usefully. The primary goal of component technology - independent deployment and assembly of components - is not achieved.

Component framework supports components conforming to certain specifications and allows instances of these components to be “plugged” into the component framework. The component framework establishes environmental conditions for the component instances and regulates the interaction between component instances. What precisely is it that a component framework contributes to a system architecture? If its purpose were just to collect “useful”
facilities, then it would be no more than a traditional “toolbox”-style library. As, by creation, a component framework accepts dynamic insertion of component instances at runtime, it also has little in common with class framework. In fact, implementation inheritance is not normally used between a component framework and the components it supports [Szy98].

The key contribution of component framework is partial enforcement of architectural principles. By forcing component instances to behave and perform certain tasks via mechanisms under control of a component framework, the component framework can enforce security, minimum allowed performance, or some ordering on event multicasts and therefore exclude entire classes of errors caused by bugs or races that could occur.

A good framework can reduce the cost of developing an application a lot because it lets you reuse both design and code. They do not require new technology, because they can be implemented with existing object-oriented programming languages.

Unfortunately, developing a good framework is expensive. A framework must be simple enough to be learned, yet must provide enough features that it can be used quickly and catches for the features that are likely to change. It must explain the theory of the problem domain, and is always the result of domain analysis, whether explicit and formal, or hidden and informal [Syz98]. Therefore, frameworks are most likely developed only when many applications are going to be developed within a specific problem area, allowing the time savings of reuse to get back the time invested to develop them.

Today, there are only few component frameworks on the market, but their number is growing fast.

3.2 Component System Architecture

Component frameworks are focused architectures whereas component system architectures consider interaction across frameworks. By analogy, think the architecture of a single building compared with that of a master-planned city. Today only few component system architectures exist. An older example is Windows DNA (Distributed Network Architecture). More recent ones are J2EE (Java 2 Enterprise Edition) and .net. There is a convincing reason not to stop at the architectural level of frameworks but to move on to entire systems. Independently designed frameworks are very difficult to combine as
they each want to take control. By the very nature of frameworks, they want
at the regulation and possibly enforcement of part interaction. Unless there
is a clear higher-level view of where, when, and how parts of frameworks
overlap or interact, there is no handle for the framework architect to ensure
proper interframework cooperation [Szy02].

![Diagram of a multilayer architecture](image)

Figure 3.1: A multilayer architecture with three order - components, compo-
nent frameworks and a component system.

Most larger, fully functional systems introduce multiple frameworks. How-
ever, interaction across frameworks is a difficult problem that needs to be
addresses on a higher architectural level. If development of good frameworks
is difficult, then beginning of good system architectures is truly hard. Layers
and hierarchical decomposition are very useful in component systems. Each
part of a component system, including the components themselves, can be
layered as components may be located within particular layers of a larger
system [Szy98]. A component system, as introduces above, arranges an open
set of component frameworks. This is a second-order architecture, in which
each of the component frameworks introduces a first-order.

Figure 3.1 shows how component instances communicate with each other
either directly (for example by using CORBA events or JavaBeans events)
or indirectly via a component framework that mediates and regulates com-
ponent interaction. The same choice recurs when component framework
instances interact - the mediator in this case is the component system.
Chapter 4

.net Platform

A number of approaches and technologies are trying to capture their share of the emerging component technology. Next three chapters (chapter four to six) present a detailed account of the three major approaches followed today. This chapter gives an overview of Microsoft’s .net technology. We specially focus on component related parts of .net such as Common Language Infrastructure, Common language Runtime and Assemblies.

4.1 Introduction

Microsoft announced the .net initiative in July 2000. The initiative aims to bring into line a wide range of Microsoft’s products and services under a common vision of interconnected devices of many kinds, from servers to mobile PCs to specialized devices. Incorporated into .net are COM+ component services; the ASP web development framework; a commitment to XML and object-oriented design; support for new web services protocols such as SOAP (Simple Object Access Protocol), WSDL (Web Services Description Language), and UDDI (Universal Description, Discovery, and Integration); and a focus on the Internet.

At a technical level, .net targets three levels:

- Web services;
- Deployment platforms (servers and clients);
- Developer platform.
Microsoft plan is to make available a number of foundational core services. A first such service has been available for a while - .net Passport, a service to authenticate users.

Most interesting for my thesis, there is a new developer platform comprising CLR (Common Language Runtime), frameworks, and tools (see figure 4.1). CLR contributes a new component infrastructure that can shield components from the details of the underlying hardware platform [Thu01]. Like JVM (Java Virtual Machine), CLR defines a virtual instruction set to isolate from particular underlying hardware.

![Visual Studio .NET Diagram](image.png)

Figure 4.1: The Microsoft .NET platform

At the top layer of the .net architecture is a new development tool called Visual Studio .NET (VS.NET), which makes possible the rapid development of component-based solutions, Web Services and other applications. VS.NET is an Integrated Development Environment (IDE) that supports four different languages and features such as cross-language debugging and the XML Schema Editor.

At the center of .net is the Microsoft .net framework, which is a new development and runtime infrastructure that changed the development of software applications on the Windows platform. Microsoft defines .net framework as such:

“The .net framework is the programming model of Microsoft .net-connected software and technologies for building, deploying, and running Web applications, smart client applications, and Extensible Markup Language (XML) Web services applications that expose their functionality programmatically over a network using standard protocols such as SOAP, XML, and HTTP.” [Microsoft]
4.2 Common Language Infrastructure

The common language infrastructure (CLI) specification establishes a language-neutral platform, something like CORBA. Unlike CORBA, though, CLI also defines an intermediate language (IL) and deployment file format (assemblies), such as Java bytecode, class and JAR files. Unlike CORBA and Java, CLI include support for extensible metadata [Thu01]. The common language runtime, part of the Microsoft .net framework, is the Microsoft implementation of the CLI specification. CLR goes beyond CLI compliance and includes support for COM and platform interoperation.

CLI comprises the specification of execution engine services (such as loader, JIT compiler, and garbage-collecting memory manage), the common type system (CTS), and the common language specification (CLS).

CTS and CLS play two balancing roles. The CTS scope is the superset of many languages’ core concepts in the type space [Thu01]. CLIL-compliant code can operate over the entire CTS space. However, no two languages cover the exact same CTS subset. For code implemented in different languages to interoperate, the CLS space is useful. CLS is a strict CTS subset that is constructed in such a way that a wide variety of languages can cover it completely.

4.3 Common Language Runtime

CLR is an implementation of common language infrastructure (CLI) specification, adding COM+ interoperation and Windows platform access services. In particular, CLR offers dynamic loading and unloading, garbage collection, context interception, metadata reflection, remoting, persistence, and other runtime services that are fully language independent [Thu01]. Presently, Microsoft supports four languages on CLR - C#, J#, Managed C++, and Visual Basic.net - that is they are the .net software components.

4.4 COM

COM is Microsoft’s old component object model and it is likely to be of continuing for years to come. Its interoperability with latest release of CLR is particularly strong, despite the fact that COM+ was released in mid 2000. COM is a binary standard for the efficient interoperation across component
boundaries. A COM component can implement several COM classes, each uniquely identified by a class ID (CLSID). Each COM class can implement several COM interfaces. A COM interface provides a set of operations and is uniquely identified by an interface ID (IID). A COM object is an instance of a COM class, but does not necessarily constitute a single object (split object). Clients use COM objects only via the interfaces provided by that object. Each interface has a QueryInterface operation that can be used to ask for any of the other interfaces of the COM object based on IIDs. COM object servers execute in processes that can be partitioned into COM apartments.

COM+, which is an extension of COM, integrates into COM previously separate and somewhat colliding support technology [Szy02], such as transactional processing, asynchronous messaging and load balancing. The most important products are the Microsoft Transaction Server (MTS) and the Microsoft Message Queue server (MSMQ).

4.5 Assemblies - the .net Software Components

Microsoft’s COM documentation inconsistently used the term component to mean a COM class or a COM module (DLLs or EXEs), often forcing readers to consider the context of the term each time they encountered it. In .net, Microsoft has solved this confusion by introducing a new concept, assembly, which is a software component that supports plug-and-play, much like a hardware component. These assemblies are in line with the “component” defined before in this thesis.

Assemblies are units of CLI deployment. It is a set of files in a director hierarchy, roughly equivalent to the contents of a Java JAR file. A compulsory part of an assembly is its manifest, which is a table of the assembly’s contents. In the case of single-file assemblies, the manifest is included in that file; otherwise it is in a separate file. Files in an assembly can be split into module and resource files. Where modules contain code and resources immutable data.

An assembly is either private to a single application or shared among multiple applications. Shared assemblies are dependent on exact naming and name resolution scheme, private assemblies are not.

A shared assembly is uniquely named by a Strong Name (SN, service in .net) and is composed as follows:
- String name = (publisher token, assembly name, version vector, culture)
- Version vector = (major, minor, build, patch)

The publisher token is the SHA-1 hash of the public half of a public/private key pair. .net guarantees uniqueness by using the unique public/private key pairs. Given this, all assemblies that are to be shared (and therefore called shared assemblies) by multiple applications must be built with a public/private key pair [Thu01]. Public/private key pairs are used in public-key cryptography. Since public-key cryptography uses asymmetric encryption, an assembly creator can sign an assembly with a private key, and anyone can verify that digital signature using the assembly creator's public key, as the public key is public.

To sign an assembly digitally, one must use a public/private key pair when building the assembly. At build time, the compiler generates a MD-5 hash over the entire assembly files, signs the hash with the private key, and stores the resulting digital signature in a reserved section of the assembly file. The public key is also stored in the assembly.

To verify the assembly's digital signature, the CLR uses the assembly's public key to decrypt the assembly's digital signature, resulting in the original, calculated MD-5 hash. In addition, the CLR uses the information in the assembly's manifest to dynamically generate a hash value. This hash value is then compared with the original MD-5 hash value. These values must match, or we must assume that someone has tampered with the assembly.
Chapter 5

Enterprise JavaBeans, the Sun way

This chapter presents Sun’s component-technology, Enterprise JavaBeans. We study here the four different types of EJB beans - stateless session, stateful session, entity, and message-driven beans.

5.1 Introduction

The Java 2 Platform, Enterprise Edition (J2EE) is a robust suite of middleware services that make life very easy for server-side application developers. J2EE builds on the existing technologies in the Java standard edition, J2SE.

J2EE is a specification, not a product. It specifies the rules that people must agree on when writing enterprise software. Vendors then implement the J2EE specification with their J2EE-compliant products.

Because J2EE is a specification, it is not tied to one vendor; it also supports cross-platform development.

*Enterprise JavaBeans* (EJB), which is part of J2EE suite, is a platform-neutral architecture that defines a specification for creating distributed applications using reusable, middle-tier components from various vendors. It is an agreement between components and *application servers* that enable any component to run in any application server. EJB components (called enterprise beans) are deployable, and can be imported and loaded into an application server, which hosts those components.

EJB extends the JavaBeans component model for code reuse from GUI development to server-side, middle-tier business application development. It delivers the speed, simplicity and reusability of JavaBeans component devel-
opment, along with the added scalability and security needed for enterprise
server applications. Using the Enterprise JavaBeans technology, component
developers can create rich, flexible, components that reflect complex business
processes and can be extended as an application evolves. One of the main
goals of the Enterprise JavaBeans is to make complex mission-critical issues,
such as persistence, security and transactions transparent to the component
developer.

The EJB architecture maps out the programming and deployment en-
vironment for distributed software components. It defines the entire pro-
gramming and deployment environment for distributed components. This
environment includes EJB server and container as well as other com-
ponents needed to support the distributed bean instance. The architecture also
addresses other enterprise services that support distributed components, in-
cluding transaction, naming, and persistence services.

The EJB server (see figure 5.1) is responsible for efficient resource man-
agement. It pools, recycles, and manages limited system resources such as
processes, network connections, database connections, transactions, security,
threads, and memory. The EJB container houses EJB instances; it's basically
the framework of classes responsible for managing the distributed bean
instance, as well as other objects needed to support the bean. The EJB
container makes method calls on the bean instance to notify the bean of
important events. The bean instance can also make calls into the EJB con-
tainer.

5.2 Types of Beans

There are four kinds of EJB beans - stateless session, stateful session, entity,
and message-driven beans. Message-driven beans are new in EJB 2.0 and a
bit different from session and entity beans. They are all united by a common
top-level contract between beans and containers and their use of deployment
descriptors. Session and entity beans on top of that share the design of EJB
object and EJB home interfaces.

Every EJB home interface has a standard method create to instantiate a
bean. In the case of entity beans it also has a standard method findByPri-
maryKey to locate an existing instance by its primary key. The methods on
an EJB object interface are all bean-specific as specified in the deployment
descriptor. The EJB container cycles beans through defined lifetime stages,
create, setContext, passivate, activate, remove and so on.

5.3 Session Beans

A session bean is created by a client as a session-specific contact point. The client accesses business methods located inside session beans on the server. The session bean is a non-persistent object, so after the client is finished with the session bean, the server will release it. This means the session bean is unavailable to the client.

Clients use the session bean’s remote interface, to call the bean. The remote interface contains the business methods that the client can invoke on the bean.

During their lifetimes, session beans transition between states. From the client’s perspective, the bean is first in the "nonexistent" state. The client then creates the bean and calls its business methods. When the client is finished, it calls the remove() method, and the bean terminates.

From the server’s perspective, a bean instance exists in a free pool on the server until the client activates it. Then, the bean is activated and its
business methods execute. Later, the container may passivate the bean if it is inactive.

The EJB architecture contains two types of session beans - stateful and stateless. The difference between the two is the way in which the bean handles client information. The stateful session bean maintains client conversational state information, while the stateless session bean does not.

The deployment descriptor for a bean marks the session beans as either stateful or stateless.

**Stateless session**

![Stateless Session bean life cycle](image)

Figure 5.2: Stateless Session bean life cycle

A stateless session bean is a bean that holds conversations that span a single method call [Rom02]. Some business processes naturally take a single request conversation, therefore, it does not require state to be maintained across method invocations.
Scaling is a major advantage of stateless beans. Because stateless beans don’t keep "per-client" state information, different clients can share them at different times. When clients call methods on the bean, the container assigns stateless beans from the free pool to them. After the method call returns, the container returns the bean to the free pool.

**Stateful session**

A *stateful session bean* is a bean that is designed to service business processes that span multiple method requests or transactions [Rom02]. To accomplish this, stateful session beans retain state on behalf of an individual client. Some business processes are naturally long conversations over several requests. An example is online web shop; the user can add products to the online shopping cart. The bean must then track the user’s state from request to request.

Stateful beans don’t scale as well as stateless beans do, because stateful beans are assigned to an individual client. However, the container can passivate a stateful bean to disk to reduce the working set of objects running on the server [Rom02]. The container will re-activate the bean when it’s needed, providing some scaling benefits to the system.

When the client creates a stateful session bean, the container takes a bean out of the free pool and assigns it to that particular client. Unlike stateless beans, a stateful bean is dedicated to a client until that client calls the `remove()` method. Once `remove()` is called, the stateful bean returns to the free pool, where it can be assigned to another client.

### 5.4 Entity Beans

One of the key benefits of EJB is the power to create entity beans. Entity beans are persistent, which means they can store and retrieve the fields of the bean from a persistent store. They also have primary keys, which means they can be "looked up" by the client when needed.

The idea behind entity beans is to use objects corresponding to database entities and encapsulate access to actual database records [Rom02]. An entity bean can correspond to a single row in relational table. The mapping of entity beans to and from database rows is called persistence.

Entity beans can manage their own persistence by directly using JDBC to operate over databases. Alternatively, they can use container-managed
persistance driven by object-to-table mappings defined during the deployment process [Rom02]. Entity beans are persistent and live long after the client that created them. For example, a client creates an entity bean, and a record is added to the database. Logically, the entity bean lives for as long as the record is in the database. A month later, if a different client wanted to look up and talk to the bean that was previously created, the new client would call a finder method. At this time, the container would activate a bean instance from the free pool. The data from the database record would be used to initialize the data in the activated bean.

There are two options when designing the persistence strategy for an entity bean: bean-managed persistence or container-managed persistence. With bean-managed persistence, the bean developer must place code in the bean to create, save, restore and delete the database record. With container-managed persistence, the container manages these actions.
5.5 Message-driven Beans.

With the introduction of EJB 2.0, support for data-driven composition was added to the model [Rom02]. This is done by adding an entirely new bean type - message-driven beans (md-beans). Like entity and session beans, md-beans reside inside an EJB container. But unlike session and entity beans, they don’t have remote or local interface or home interface. The only way to instantiate and use an md-bean is to register it for particular message queue or topic.

![Diagram of Remote Method Invocations](image1)

![Diagram of Messaging](image2)

Figure 5.4: RMI vs Message-driven bean

Message-driven bean is an alternative to remote method invocations (see figure 5.4). The idea behind md-bean is that middleman sits between the client and the server [Rom02]. This middleman receives messages from one or more message producers and broadcasts those messages to one or more message consumers. Because of the middleman, the producers can send message and then continue its work without waiting until the message is been received, the middleman takes care of this.
Chapter 6

CORBA CCM, the OMG way

This chapter introduces CORBA component infrastructure, the concepts of CORBA component and its runtime environment. We then discuss different types of CORBA components, their connections, and deployments. We also study here CORBA Component Model (CCM).

6.1 CORBA Overview

CORBA (Common Object Request Broker Architecture) is the glue that can link objects which have been written in different programming languages, reside on different platforms, and reside in different server and client locations. The heart of CORBA is the object request broker (ORB). The ORB provides facilities for object management: it connects objects to other objects, and helps objects communicate with object services. Simply put, the ORB helps objects “talk” to other objects.

CORBA has often been described as an “object bus”, where objects can register to talk to other objects. Perhaps a better analogy is the remote procedure call (RPC). RPC is a facility that enables a program on a computer to make a “function call” on another remote computer in the network. CORBA extends this analogy to objects: Objects can call remote objects’ methods, and these objects, no matter where they are on the network, can then communicate. This is the simple CORBA philosophy.

Figure 6.1 shows the basis for the Object Management Architecture (OMA), CORBA’s base architecture. The ORB is the key component. There are CORBA application objects, CORBA facilities, an ORB, and CORBA ser-
services. CORBA's common facilities provide generic functions that can be used in specific applications (database management, for example). CORBA's many services perform a number of object management functions. Objects register with this service to let the other objects know their existence.

![CORBA Object Management Diagram]

Figure 6.1: CORBA Object Management

In CORBA, there are client objects and server objects. Server objects are those that perform a specific service, and will have their methods remotely invoked. Client objects simply invoke methods on the server objects.

In order to use CORBA, developers use the Interface Definition Language (IDL) to describe the server objects. IDL is a standard programming language-independent definition language that describes objects and is used to generate code that works with the CORBA environment. IDL describes the object's interface, the public methods, and member variables of an object. Because IDL is not programming language-specific, an interface definition for an object written in C++ would look the same as the same object written in Java.

6.2 CORBA Component Model

CORBA 3 is the latest incarnation of the suite of CORBA standards. Although, the final set of specifications was released September 2002, sub-
stantial strides have already been made to improve on almost all aspects of CORBA 2. Besides a patch up of many object services, the single biggest contribution is probably the new \textit{CORBA Component Model} (CCM). Occasionally, CCM is also referred to as CORBAComponents [CCM99].

CCM is an ambitious logical extension of Enterprise JavaBeans. CCM introduces several new features, promises a fully compatible embedding of existing EJB solutions and aims to maintain the original CORBA principle of being both language and platform independent [Sie00].

Strictly speaking, CORBA component is any piece of functionality that runs in the CCM environment, and takes advantage of the services that the environment provides. It can be any size, but the most efficient applications are built of relatively small components. That is because the component is the smallest unit that the CCM runtime can swap in and out to optimize throughput. That is not the whole story, though, because components are also the unit of reuse, and you can lose some of the benefits of reuse if components are too small. One way to come around this is to keep the components small and group them into \textit{assemblies}, which can be reused just as components can.

A CCM application is an assembly of CCM components, each of which may be custom-built or off-the-shelf, in-house or acquired. Enterprise JavaBeans components and CCM components can be combined in a single application. Individual components are shipped in component packages that contain an XML document detailing their contents, which can include binaries for multiple platforms. CCM assemblies contain an XML document describing the set of component packages they refer to and the deployment configuration of these. A CCM component itself can consist of multiple segments (see figure 6.2). CCM runtimes load only those segments which are needed to optimize throughput.

\subsection{6.2.0.1 CCM components}

CCM components are classified in four different categories, depending on how long they and their object references are expected to last, whether or not they have persistent state, and how this state is exposed to the client. These are \textit{service}, \textit{session}, \textit{entity}, and \textit{process} components. (The session and entity categories correspond to stateful session and entity beans in EJB, respectively [Sie00].)

A CCM application provides declarative information on component cat-
CCM assembly

Figure 6.2: Components, assemblies and segments in CCM.

gories and component factories. A portable object adapter (POA) uses this information to create and assign servants to component instances. Service components are instantiated per incoming call and thus cannot maintain state across calls. Instances of session components maintain state for the duration of a transactional session and allow for multiple calls within such a session. Instances of process components have persistent state - their lifetime corresponds to the lifetime of some process they are servicing and is as such arbitrary [Sie00]. Finally, entity components have persistent instances that correspond to entities in some database - they can be accessed by presenting the database entity’s primary key.

A CCM component is programmatical characterized by a number of features. Figure 6.3 summarizes the structure of a component and how it interacts with the outside world. The various stubs and skeletons a component has are referred to as ports. Ports are classified into facets, receptacles, event sources, and event sinks [Sie00]:

Facets are the potentially multiple interfaces that a component provides to its clients.

Receptacles are the client stubs that a component uses to invoke other components.
Figure 6.3: CCM components with their many features.

Event-sources are the named connection points that emit events of specified type to one or more interested consumers, or to an event channel.

Event-sinks are the named connection points into which events of a specified type may be pushed a supplier or an event channel.

A component can also incorporate client stubs used to invoke other (non-component) CORBA objects - for example, the naming or trader service. Other features of the model include:

- Primary keys, which are values that instances of entity components provide to allow client identification of the instances.
- Attributes and configuration, which are named values exposed via accessor and mutators.
- Home interfaces, which provide factory functionality to create new instances.
6.3 CCM Framework

CORBA 3 defines a *Component Implementation Framework (CIF)*, which includes generators that accept CIDL (component implementation description language) input and generate implementation code that completes explicitly provided component code.

In addition, every component instance is placed inside a CCM framework (Figure 6.4). Components interact with POA (portable object adapter) as well as transactions, security, persistence, and notification services via interfaces on their framework. A framework also has receptacles that accept callbacks into that component instance.

![CCM Framework Diagram](image)

Figure 6.4: CCM Framework [CCM99]

A number of options are available for each of the four services that CCM packages. Transaction control can be framework-managed or self-managed. In the framework-managed case, a component configuration states if transactions are supported, required, required new, or not supported. The framework will begin and end transaction to meet these requests. Similarly, persistence can be declared as framework-managed or self-managed. For security, required access permissions can be declared on operations in CIDL and will be checked by the framework.
Chapter 7

Case Study Based on Avalon Framework

The coming two chapters (chapter seven and eight) present a component-based case study. In these two chapters I will explain how I developed a component-based file-sharing program. This chapter, chapter seven is based on Avalon framework and EJB as component technology. We first state requirements and design decisions. Finally, we discuss the development approach.

7.1 Introduction

Avalon framework is a java based open source software which consists of interfaces that define relationships between commonly used application components and several lightweight implementations of the generic components. Avalons focus is server side programming and easing the maintainability and design of server focused projects. In this chapter I will, as a case study, design, develop, deploy and assemble a component based on Apaches Avalon framework.

Avalon framework is an interesting design that serves well for a case study. The software is freely released under the terms of the Apache Public License (APL). After I have got through Avalon’s different part I decided to download the framework and install it to my computer in order to develop a components based on Avalon’s framework. Then I downloaded some of the already existing component and tested how Avalon’s component deployment works. After that I implemented a simple “Hello, world!” component based on Avalon framework.
All it did was to open and listen to specific port on my system and return an HTML file to who ever connects to that port. The HTML-file contains visitor’s host name, IP-address and a short greeting. That was how far I came in my Interim Progress Report.

In my final report I decided to implement and look at more complex system, namely, a file-sharing-program based on Avalon framework. A system which allow its users to share files over the Internet or Intranet. I chose this system because of two reasons; first, Avalon is suitable for server-side applications and a file-sharing-program has a server side. This will hopefully give me an insight about component-oriented programming based on open software framework. Secondly, I have in the past designed and implemented a file-sharing-program based on multicasting technology.

Beyond the knowledge I gained from my previous file-sharing-project I could not reuse neither the code nor the design! First, my previous project was based on multicasting group communication which uses the connectionless UDP for communication, while my current one uses the connection-oriented TCP/IP. Second my current project uses the client-server paradigm while my previous was based on serverless peer-to-peer communication. In addition my current server is just a component hosted by Avalon framework, not a standalone server.

Now back to the thesis, in the following subsection I will outline my requirement specification and the design decisions I have to make for the system. The complete design specification in UML is in Appendix A.

7.2 Problem Statement

Produce a client-server system that allows multiple users to share files over the Internet or Intranet. The server-side of the system should be a component which is “pluggable” to Avalon framework.

The Client

Implement simple client software to interact with the server, preferably with GUI. To be able to use the system, the user need to have a client software. Graphical user interface which users can interact with the system. When the client is executed, it prompts the user for their user name and server’s host name. The system will authenticate the user and retrieve, from the user, a
list of her shared files.

The user will then be able to issue queries and view search results or view a list of all available files. Users will also be able to download files direct from other users to their local machine.

The Server

The server part of the system is the main part and it will be a component hosted by Avalon framework. Users who want to share files will be able to connect to a "well-known" server. At this stage the client must know the server address. Of course we can easily make preconfigure server address if we have dedicated server such us napster server. Each user will eventually send to the server a list of their shared files, if the user has something to share with other users. The server puts together all the lists from the clients and makes it available to the clients to make search on it.

The user will be able to stay connected to the server as long as she want. When a user leaves (disconnects from the server), her list will be removed from the combined list.

7.3 Design Decisions

Graphical User Interface (GUI)

The client application will have GUI for logging in and issuing query. It will also show progress bar while downloading files which visualize the state of the download. When the client application is started the user will get a pop-up window with the server name and user name textboxes that the user will have to fill out to connect to the server.

If the connection fails, for some reason, the same window will pop up again, otherwise the program starts a new larger with menu-bar and textbox for search. The connection could fail if the user supplies invalid server address or user takes a name which is already in use.

Note that the primary design goal of the system is to verify programmability, reusability, performance and robustness of component-based system. Therefore, the GUI issue is intentionally kept simple.
Functional Design

The server is multi-threaded and will allow multiple clients to make search at the same time through the server where all clients are connected to. The server will be a Java application that can be run under any JVM-compliant platform. Users using different platforms will be able to share files, as long as they are connected to the internet (or the same intranet). Two client applications will be able to run from the same physical machine. The shared files can be any sort of media such as text, sound or graphic with any size.

File Downloads

Once a client receives a query result from the server, user can initiate to download one of the files in the query result. Files are downloaded out-of-network i.e. a direct connection between the source and target client is established in order to perform the data transfer. File data will never be transferred over the server.

7.4 Server Component Development

When building Avalon-compliant component (or a component in general), the following is a typical order of operation.

1. Develop the component
   
   (a) Write the component files.
   (b) Build the component using Ant tool.
   (c) Package the component and related resources into a .jar file.

2. Assemble
   
   (a) Write config.xml, assembly.xml and environment.xml files.
   (b) Package the component and related resources into a .sar file.

3. Deploy
   
   (a) In the case of Avalon, drop the .sar file into phoenix/apps/ directory
4. Start or restart the framework `phoenix/bin/run.bat|sh`

5. Create optional standalone test client.

1.a Write the Component

Writing the component is obviously the main task, apart from writing normal java-files the component needs to follow all the rules of writing a standard Avalon component. I initially created two interface files, `FileSharingServer` and `FileSharingServerMBean`, to declare what this component is offering to the host system. Then I created `FileSharingServerImpl` class which extends `AbstractLogEnabled`, a logging class in Avalon framework. The class also implements the two interfaces and some of Avalon’s interfaces which the component needs. These interfaces are `Contextualizable`, `Serviceable`, `Configurable`, `Initializable`, `Disposable` and `ConnectionHandlerFactory`. The `FileSharingServerImpl` class makes the component Avalon-compliant. After that, I created three java classes which do the business logic of the component. These classes are `FileSharingHandler`, `Search` and `Item`:

`FileSharingHandler` takes care of all incoming connections from client, such as in logging, out logging and enquiries.

`Search` carries out the search logic. I kept the search algorithm simple and the only searchable attribute is the file name. The file names are kept in a vector.

`Item` defines the file format as well as the incoming and outgoing package format. It is simply the system’s communication protocol. The client side of the system has similar file.

Steps 1.b, 1.c and 2.b above are done by Ant build file (explained below). Now the component is ready. However, I can not test it on the spot because the component has no main-method. I have to build, package, deploy and run it in a framework. After running it, if it shows error I have to go back to the source file, fix the error, and then follow the same route again. This is one of the difficulties of developing component. It is a lot easier to use component than to develop it!
2 a) Write config.xml, assembly.xml and environment.xml

In Avalon, each component must contain the following three files:

config.xml: The purpose of the config.xml file is to provide configuration data to each of the components that require configuration data. See the Appendix B for the config.xml file.

assembly.xml: The purpose of the assembly.xml file is to define how the Server Application is assembled from its components. This requires naming each component, specifying the implementation class for each component and wiring together components.

environment.xml: The purpose of the environment.xml file is to configure environmental or Server Application wide settings. This means being able to set the security policy and configure logging settings. This file is written in XML and the end-user can view what sort of security policy the component requires. The end-user (assembler) can alter the permissions in the security policy if so is desired.

2. b) Build, jar and sar.

Avalon components use Apache Ant for building the component and packaging them into JAR and SAR archives. Ant is a Java-based build tool. In theory, it is kind of like Make, but without Make’s wrinkles. The configuration files are XML-based and are extended using Java classes instead of extending with shell-based commands. For more information, please refer to http://ant.apache.org.

JAR is zip-like tool used in java for archiving java files. JAR stands for (Java Archive). The SAR (Server Archive) file format is the standard distribution format of Server Applications. It is a standard Jar file with a specific directory layout. The config.xml, environment.xml and assembly.xml file must be stored in SAR-INF/ directory of the archive. All jar files, including both those that contain components and those that contain support classes are stored in the SAR-INF/lib/ directory. Avalon has a template Ant build.xml file. I used this template to create the final build.xml file for my component. The complete build file and other configuration files are in Appendix B.
3 a) Deploy

Currently deploying a server application under Avalon is simply a matter of dropping the .sar file into the directory phoenix/apps/, where the framework looks to load the .sar file, and starting the server. After I restarted the Avalon it confirmed that the component was loaded.

Now the server component is up and running. I created simple client to test the server. Please refer to the Appendix A for client design.

7.5 Tests and Result

Tests must be carried out to verify that a system really conforms to its specification. Even bugs and logical errors in the program can be found under the test. It is best if someone outside the development group carries out the tests. To satisfy the objectives, both static and dynamic techniques of system checking are used. Static techniques are concerned with checking of system representations such as the design diagrams and source code, this has been done. Please refer to Appendix A for design diagrams.

Dynamic tests involve exercising the program using data like the real data processed by the program. The existence of program defects or inadequacies can be found from unexpected system outputs or behavior. I carried out tests during the implementation phase to verify that the software behaves as intended and after the implementation was complete. The errors I encountered during the implementation have been located and repaired.

After completion of the implementation, two tests were particularly interesting:

- To see that the server component, after it is been deployed to a system, has not introduced any problem to the host system, in times of resource consumption and security risks.

- To test that the communication between server and clients works correctly. Such as queries and file transfers.

Unfortunately there is no test tool for Avalon framework. Chances of detecting performance change of a specific component are very limited. The test machine has not shown any performance degrading after the component was started and even when it was serving clients.
In times of security, two port scanning were carried out against the test machine by using port scanning scripts. First one before the server was started and second one while the server was in operation. I then compared the two results to see if the server system opened any new ports to the outside world in the later scanning. For my surprise, it did really opened two new and undocumented ports beyond the server port I defined myself. These ports are 4128 and 4139. It is generally hard to find out what service is running behind an open port. However, some ports do give hints. So, I made further investigation to see what services these ports were offering to the outside world by telnetting to each one of them. Unfortunately, the telnet stuck and has not provided any clue to go further. All I know is that one of the services is using HTTP protocol. “HTTP/ 1.1 400 Bad Request” was dumped to my terminal when I disconnected from the port.

Fortunately, Avalon is an open source and any one can easily download the source code and see what these ports are for. I am sure someone have already done this, just the Avalon’s technical documentation is bit behind.

7.5.0.2 Communication tests

The system was constructed so that two or more clients can run from the same physical machine. Therefore the server and all clients for the tests were on the same machine. They could equally have been from different machine as they were all communicating by the use of TCP/IP. So these tests are equally applicable to physically distributed system.

The machine I used to develop the system was standalone and not connected to a network. To get an overview of how the prototype works in reality, I started four different clients, where each one of them has its own private shared directory and then let them exchange files. The purpose of running four clients at same time was to control that all queries reach the server and responses to the clients were correct and even sent to the right client. Performance test has been carried out through different file transfers between the server and clients with successful result.

The prototype is not slower than Windows’ own file copying. The time to transfer file is almost the same (a file of 18 MB in size was average 6 second quicker for the prototype at ten tests). The test files were Word document, PDF-files, Mp3-files and java files. All these formats were transferred intact and could be used without any problem. Searching with a file name worked correct.
To conclude, the prototype conforms to its specification. Testing can only demonstrate the presence of errors. It cannot show that there are no errors in a program.

Figure 7.1: Centralized server system.
Chapter 8

Case Study Based on .net Framework.

This chapter is similar to chapter seven, however, in this chapter the implementation is based on Microsoft’s .net technology.

8.1 Introduction

The .net architecture strives to support componentization in every way that makes sense. Thanks to metadata, component programming is much easier in .net framework because we no longer have to worry about the registry for component deployment and other twists in COM era in order to support componentization. Microsoft .net provides a simpler way to build and deploy components.

In chapter seven I made a case study about a file-sharing-program based on Apache’s Avalon framework and was implemented in Java. In this chapter I will make comparable case study but this time based on Microsoft’s .net framework and implemented in C#. As in chapter seven, a system which allow it’s users to share files over a network.

Microsoft has introduced a new concept, assembly, as a result of inconsistent use of the term component in Microsoft’s earlier documentation. Assembly is a software component that supports plug-and-play, much like the component defined earlier in this thesis. In practice, an assembly can contain or refer to any number of types and physical files (including picture files, .net PE files, and so forth) that are needed at runtime for successful execution.
8.2 Problem Statement

Produce a client-server system that allows multiple users to share files over the Internet or Intranet. The server-side of the system should be a component which is “pluggable” to Microsoft’s .net framework.

8.2.1 The Server

Users who want to share files will be able to connect to the “well-known” server. Each user will eventually send to the server a list of their shared files, if the user has something to share with other users. The server puts together all the lists from the clients and makes it available to the clients to make search on it. The user will be able to stay connected to the server as long as she wants. When a user leaves (disconnects from the server), user’s list will be removed from the combined list.

8.2.2 The Client

To be able to use the system, the user must have the client software. When this is executed, it prompts the user for their user name and servers host name. The system will authenticate the user and retrieve, from the user, a list of her shared files.

The user will then be able to issue queries and view search results or view a list of all available files. Users will be able to download files direct from other users to their local machine.

8.3 Design Decisions

Graphical User Interface (GUI) The client application will have GUI for logging in and issuing query. It will also show progress bar while downloading files which visualize the state of the download. When the client application is started the user will get a pop-up window with the server name and user name text boxes that the user will have to fill out to connect to the server. If the connection fails, for some reason, the same window will pop up again, otherwise the main client application starts with menu-bar and text box for search. The connection could fail if the user supplies invalid server address or user name is already in use.
Note that the primary design goal of the system is to verify programmability, reusability, performance and robustness of component-based system. Therefore, the GUI issue is intentionally kept simple. I have reused the same client as in chapter seven.

8.3.0.0.1 Functional Design The server will allow multiple clients to make search at the same time through the server where all clients are connected to. The server is a C# application that can be run on .net framework. Users using different platforms will be able to share files, as long as they are connected to the Internet (or the same intranet). Two client applications will be able to run from the same physical machine. The shared files can be any sort of media such as text, sound or graphic with any size.

8.3.0.0.2 File Downloads Once a client receives a query result, user may initiate direct download of one or more of the files in the query result. Files are downloaded out-of-network i.e. a direct connection between the source and target client is established in order to perform the data transfer. File data will never be transferred over the server. The servers function is only to mediate between the clients.

8.4 Server Component Development

A primary goal of the .net framework was to simplify the development and use of binary components and to make component-oriented programming easy and accessible. As a result of that .net does not force some special rules for component development, such as separation of interface from implementation. Every .net class is consumed as a binary component by its clients and is literally a component. I will not go into detail how I developed the classes in the server component as they are the same as in chapter seven. These classes conform to the traditional object-oriented programming. I will emphasize here the componentization part of the development.

8.4.1 Private and Public Assemblies

There are two kind of components in .net, private assembly and shared assembly. Private assemblies are simple and can be copied to the same folder as the calling client. When we want to uninstall a private assembly, all we
need to do is to remove the file from the folder. However, when we want to
share assembly with other applications, we need to do some work.

Shared assemblies are installed to a single location, called GAC, Global
Assembly Cache. Registering assemblies against the GAC makes it act as a
system component, such as system DLL that every process in the machine
can use. A condition for GAC registration is that the component must have
version and originator information as well as other metadata. These two par-
ticular items allow multiple versions of the same component to be registered
and executed on the same machine. Other two metadata are Assembly's face
name and culture information which are optional.

There are three general steps for installing assembly into the
GAC:

1. Use the Strong Name sn.exe tool in .net framework SDK to obtain
   a public/private key pair. This tool generates a random key pair
   and saves the key information in an output file, for example,
   originator.key.

2. Build assembly with an assembly version number and the key
   information from originator.key file.

3. Use the .net GAC tool gacutil.exe to register the assembly in the
   GAC. This assembly can now be used from other assemblies on
   the same machine, regardless of their physical location.

The first step in building an assembly with a strong name is to obtain a
cryptographic key pair. The .net framework SDK includes a Strong Name
utility (SN.exe) that can be used to generate a key pair. The key pair that
is generated by the Strong Name utility can be kept in an optional file. The
following command uses the Strong Name tool to generate a new key pair
and store it in a file called originator.key:

C:\server\sn -k originator.key

The -k option generates a random key pair and saves the key information
into the originator.key file. I will use this file as input when I build my server
component. Let's now examine steps 2 and 3 of registering the component
into the GAC.
In order to add version and key information into the server component (developed using C#), we need to either add the metadata in one of the component files or put it in separate file, manifest (the metadata that describes the assemblies). I chose the later one and named the manifest file the more conventional name AssemblyInfo.cs. The manifest file looks like this:

```csharp
using System.Reflection;
[assembly: AssemblyKeyFile("originator.key")]
[assembly: AssemblyVersion("1.0.0.0")]  
[assembly: AssemblyTitle("File Sharing Server")]  
[assembly: AssemblyDescription("Simple file sharing server")]  
[assembly: AssemblyProduct("Server")]  
[assembly: AssemblyCopyright("Abdille Hagi ©2003")]
```

Having done this, we can build the server assembly with the following commands in the server folder:

```
C:\server\csc /out:Server.exe *.cs
```

Once we have built this component, we can register it into the GAC:

```
C:\server\gacutil /i Server.exe
```

### 8.4.2 Viewing the GAC

.NET framework has Shell Cache Viewer for viewing the GAC. Now that we have installed our server component into the GAC, let us see what the GAC looks like. On my machine, the Shell Cache Viewer appears when I navigate to C:\WINDOWS\Assembly, as shown in figure 8.1.
As we can see, the Shell Cache Viewer shows among others the Server component, its version number and public-key-token value. Well done it is installed as a component in .net framework and ready for use.
Chapter 9

Discussion and Evaluation

In this chapter I will evaluate component-based software and the thesis as whole. I start with comparison of the three major component worlds I presented early in this thesis, CORBA CCM, EJB and .net. The focus of the comparison between these three state of the art technologies will be mainly on how they manage component technology. Finally, I will discuss component-based software development and the possible advantages and disadvantages of adapting to this approach.

9.1 Assessment of the Three Major Component Worlds

Recently, there have been numerous debates about Enterprise JavaBeans (EJB) and Microsoft’s .net technologies. While, technically comparing and contrasting these technologies is beyond the scope of this thesis I do want to provide two interesting pointers. Ed Roman, the author of Mastering Enterprise JavaBeans and CEO of The Middleware Company, wrote an extensive assessment “J2EE vs. Microsoft.NET” whitepaper, available at www.theserverside.com. J2EE wins.


Similarities

Noticeably, the shared characteristics of the approaches discussed cannot help to make decisions as to which approach to follow. However, the rich sharing
of characteristics shift the decision in favor of using component software technology in the first place, whatever the concrete approach is.

All approaches rely on dynamic polymorphism (some books call inclusion polymorphism or subtyping), encapsulation and late binding mechanisms. All approaches support interface inheritance. In other words, all approaches rely on some sort of object model.

Most approaches support:

- A specialized component model for application servers - EJB, COM+, CCM;
- Uniform data transfer, JAR files, CLI assemblies;
- Some form of persistence or serialization. Deployment descriptors or attribute-based programming.

**Difference**

- *Binary standard per platform:* Java avoids an actual binary standard by standardizing on bytecode instead. CORBA still does not define binary standards. .net framework, like Java, stays one level removed from binary standard and standardizes MSIL (Microsoft Intermediate Language) instead.

- *Source-level compatibility and portability:* The .net CLR provides the common language specification (CLS) to guide language bindings that yield a high degree of interoperability without actually prescribing the individual language bindings. CORBA is particularly strong in standardizing language bindings that ensure source code compatibility across ORB implementations. For Java, the agreement on the Java language specification solves the problem as long as no other languages are used to target the Java platform.

- *Garbage collection and memory management:* Java relies totally on garbage collection, Java also defines distributed object model and supports distributed garbage collection. CORBA today does not offer a general solution to the global memory management problem in a distributed object system [Szy98]. CLR performs garbage collection and lease-based invalidation of remote references.

49
• *Implementation and applications:* Here, all approaches have their home fields. CORBA is strongest for traditional legacy integration at the level of enterprise computing. .net in desktop side. J2EE and .net now dominate non-PC and PC-based server solutions. CLR implementations are mainly limited to those provided by Microsoft. CORBA and J2EE implementations are available from many vendors.

• *Communication:* CORBA supports the IIOP as a standard communication protocol for inter-ORB interoperation. .net uses DCOM as its native communication protocol and supports multiple messaging formats.

### 9.2 Case Study Assessment.

I have developed a case study that provides a file-sharing system. This case study, which is implemented in both .net framework and Avalon framework, serves as a prototype that illustrates various component-oriented concepts and technologies. The two frameworks I have used for the implementation took different approaches to solve the same problem. Following is discussion about these two different implementations.

#### 9.2.1 Using Avalon Framework

Avalon framework is a java based open source software which consists of interfaces that define relationships between commonly used application components and several lightweight implementations of the generic components. Avalons focus is server side programming and easing the maintainability and design of server focused projects.

Currently Avalon is a parent of five sub-projects: Framework, Excalibur, LogKit, Phoenix, and Cornerstone. Avalon began as the Java Apache Server Framework that had the framework, utilities, components, and a server's kernel implementation all in one project.

Avalon provides some useful components and utilities that I could incorporate into my own system. After my analysis of Avalon framework was complete, I implemented the component and Services that make up my file-sharing system.
Utilities in Avalon framework is divided into seven main categories: Configuration, Context, Activity, Service, Log, Parameters, Thread, and Miscellany. It is common for a component to implement several interfaces of these utilities to identify all the concern areas that the component is worried about.

Here I will go through the process of implementing and using a server component based on Avalon framework.

The following steps are required before a component is ready for assembling.

1. Write the Component.

2. Associate MetaData with the Component.

3. Package the component and related resources into a SAR file.

Writing the actual component is the main task that takes the most of the time. The component must follow all the rules of writing a standard java classes and a standard Avalon component design such as separation of concern and inversion of control.

Avalon components have associated metadata that declares what resources they are capable of providing to other the components and what resources they need to be provided by other the components or the host framework. The Avalon metadata specification is an extensible mechanism to attach metadata to components.

The final step is packaging up the implementation files, Meta descriptors and other resources into a sar file.

In my main class, FileSharingServer, I chose to implement sex interfaces from Avalon:

```
public class FileSharingServerImpl
    extends AbstractLogEnabled
    implements Contextualizable, Serviceable, Configurable,
    Initializable, Disposable,
    ConnectionHandlerFactory
{
    ...
```

I chose to implement the Initializable and Disposable interfaces. I want my solution to be as secure as possible and explicitly track whether the component
is fully initialized or not. Since specific information about the environment may change, or may need to be customized, I also implemented the Configurable interface. The component makes use of other components, and the method that Avalon provides to get instances of the required component is by using a ServiceManager. I will need to implement the Serviceable interface to get an instance of the ServiceManager.

After implementing these interfaces and other support classes I will have to write the configuration files. Avalon has configuration files, and they allow an administrator to alter vital configuration information. Avalon can read a configuration file in XML format, and build the components in a system from it. This is where I found the most difficulty in Avalon framework. Configuring right is very tricky.

9.2.1.1 Source modules

The file-sharing server comprises of the following main modules, all contained within the src/ directory:

- FileSharingServer Contains the Java source for server component.
- conf/assembly.xml Contains the assembly instructions for the resulting server archive (SAR) file.
- conf/config.xml Contains the configuration for the resulting server archive (SAR) file.
- conf/environment.xml Contains instructions about the environment and logging.

9.2.1.2 Component Deployment

Currently deploying a server application under Avalon is simply a matter of dropping the .sar (Server ARehve) file into the appropriate directory (apps/). Avalon supports hot deployment, applications can be added, removed and updated in the application directory without restarting the framework.

9.2.2 Using .net Framework

.net framework is the latest participant to the field of component-oriented programming, and it addresses the requirements of component-oriented pro-
gramming in a way that is very much easier to use. These improvements are of little surprise, because the .net framework architects were able to learn from both the mistakes and successes of previous framework technologies such as DCOM, CORBA, and JavaBeans.

Throughout the component development it was clear that the primary goal of the .net framework was to simplify the development and use of binary components and to make component-oriented programming more accessible. As a result, .net framework doesn’t enforce some core principles when programming components comparing to Avalon framework.

9.2.2.1 Developing Component

Developing component under .net framework does not differ much about developing normal software.

.net enforces a few of the core concepts and simply enables the rest. Doing so provide help to both ends of the skill field, component-oriented programming and object-oriented programming. All .net programming languages are component-oriented in their very nature, and the primary development environment, Visual Studio, provides tools, views, and wizards that are oriented toward developing components.

9.2.2.2 Composing Assemblies

In .net framework there are many ways to compose a component. The only two requirements are:

1. Every assembly must contain a manifest, either in a separate file or in one of the source-files.

2. Every assembly module that contains IL must embed in it the corresponding metadata for that IL. (this is normally generated by the compiler)

Components can optionally contain resources such as source-files or images. The file-sharing server component contains seven source-files and a manifest-file. The metadata was embedded during the build.
9.2.2.3 Component Deployment Models

As I mentioned before, there are two models of deploying component in .net framework, *private* and *shared*. Deploying a private component is as simple as copying it to the directory of the application using the component. In my case study it was question of shared component. A shared component is an assembly that can be used by multiple client applications. Shared assembly must be installed in a well-known global location called the (GAC) Global Assembly Cache, which can usually be found in the following directory: C:\WINDOWS\ASSEMBLY\GAC.

Installing the file sharing server into the GAC was straight forward. I first created public/private key pair, build the component with this key and version number and finally use GAC tool (gacutil.exe) to register the assembly into the GAC. That was all I needed to do to make the component system-wide component. No plumbing code or scripting was needed, since all the metadata for the component is stored inside the .net assembly.

9.3 Assessment of the Two Approaches

Component-based framework requires both component systems that support the approach and component developers that follow its core principles and its discipline. Nevertheless, it is hard to tell the difference between a true core principle and a mere feature of a component framework.

There is a single common key feature of component technology that seems to cause most technical difficulties: Components come from separate sources and are deployed by third parties. Even if a component fails to function, it must not violate system-wide rules. For that reason, component safety, version control and language independency are important core principles.

In the following subsections I will discuss these three important core principles and compare how Avalon framework and .net framework put them into practice.

9.3.1 Component Security

In component-based programming, components are developed independently from the client applications that will use them. Component developers have no knowledge of how end user or client application will try to use their work. The component could have been used maliciously to corrupt data or to hack
to a system without proper authentication or authorization. Similarly, a 
client application has no knowledge of how the component it is interacting 
with is internally implemented. It could well be a malicious component that 
abuses the credentials the client provides. In addition, even if both the 
component and the client have no bad intent, the end user could still try 
to hack into the host system or do some other damage (even by accident). 
To minimize the danger, a component framework needs to provide a security 
infrastructure to deal with such scenarios.

Avalon Framework Security

Security in Avalon framework is mainly based on Java security model. Avalon’s 
Guardian Component is used to manage permissions based on properties of 
the code and who is running the code. The Guardian loads its permission 
sets from a database or configuration file. It uses the standard Java security 
model to enforce access to the resources which a specific component may 
request.

The Guardian component uses a security policy to decide which individual 
access permissions are granted to running code. These permissions are based 
on the code’s characteristics, such as where it is coming from, who is running 
it, whether the code is digitally signed, and if so by whom. Attempts to access 
protected resources raise security checks that compare the needed permissions 
for the attempted access with the granted permissions. If a security policy is 
not explicitly given, the default policy is used which is the classic “sandbox” 
policy as implemented in JDK 1.0 and JDK 1.1.

The sandbox is customizable Java security model which allows each re-
stricted piece of software to run within it’s own restricted “sandbox”. The 
program must “play “only inside its sandbox. It can do anything within 
the boundaries of its sandbox, but it can not take any action outside these 
boundaries. The sandbox for untrusted Java application, for example, pro-
hibits many activities such as: Writing or reading to local disk, making 
network connection etc.

Java security model is one of the language’s key architectural features 
which makes it a proper technology for networked environments.
.net Framework Security

The .net framework offers a robust security model for dealing with component security. Each component that runs on the CLR must pass the CLR security policy.

.net security model is based on an elegant concept: code identity and using an administration tool. The system administrator grants components certain permissions to access external resources such as the filesystem, the network, the registry, the user interface, and so on. In .net framework a component itself has a code identity, which includes information such as the component’s shared name, version number, public key, culture, and where the code came from (local, Internet, or intranet). This information is also called the component’s evidence, and it facilitates to identify and grant permissions to the code.

At runtime, whenever a component tries to access a resource or perform a privileged operation, .net verifies that the component and its calling components have permission to perform that operation. The CLR examines what evidence the component provides to establish its identity and authenticity.

9.3.2 Version Control

A component technology must provide some sort of version control and allow components to evolve along different paths. It must also allow components and clients to evolve independent of each other. Component deployer should be able to deploy newer version of existing component without disturbing existing client applications. On the other hand client developer should be able to deploy new version of the client and expect it to work with older version of component. Well designed component technology should allow different versions of the same component to be deployed on the same machine side by side.

Avalon’s Version Management

Avalon provides component service that enable component version control. The component version management is not enforced in Avalon framework as it is in .net framework. It is up to the component developer to include the version component and the client component to verify it is dealing with the right version of the component.
The version component in Avalon provides persistence and access to version information. The version of a component is specified by using a dot-delimited string of three unsigned integers, \textit{major.minor.micro} defined as follows:

- **micro** - If the micro version is incremented (i.e. from "2.5.10" to "2.5.11"), then the changes in the component are small forward compatible fixes or documentation modifications etc.

- **minor** - If the minor version is incremented (i.e. from "2.5.10" to "2.6.0"), then a backward compatibility with prior version is granted with the previous clients. Something may have changed in the component implementation. (i.e. new methods could have been added)

- **major** - If the major version number is incremented (i.e. from "2.5.10" to "3.0.0"), then a backward compatibility with prior version of the component is not guaranteed.

The architects of Avalon framework did not set component-version control as one of the main goals. The use of version control is merely optional and up to the component developer to enable it.

### .net Version Management

.net framework seems to learn from the “DLL hell” in earlier versions of Windows OS which was caused by the lack of component version control. One of the major goals set for .net platform is to simplify component version control.

As mentioned before, there are two types of component modes in .net framework, \textit{private} and \textit{shared} component. The component-versioning challenges are closely related to the component mode. Private components are far less affected by versioning issues, because each client application comes with its own private set of compatible components, I have to explicitly interfere to cause version incompatibilities. On the other hand, shared components can cause a lot of versioning problems because they are stored in a well-known global location and are used by multiple applications. Nevertheless, .net framework allows multiple applications to share server components and different client applications to use different versions of the same server components. Hence, in .net, I can have different versions of the same component...
to be deployed on the same machine side by side. During the development of my server component I could increment my version number and deploy newer version of the server component while the previous version is still there.

The .net entity responsible for managing component compatibility and making sure a client application always gets a compatible component is called the assembly resolver.

The assembly resolver always tries to give the client component a version-compatible server component. The resolver first looks in the global location called GAC and loads the compatible component from there if it is found. If no compatible component is found in the GAC, the resolver looks in the client application’s folder. If the client application folder contains a compatible private version of the component the resolver loads and uses that version. .net throws an exception if no matching private component is found. By default, .net does not let a client application interact with an incompatible component.

.net framework has a mechanism for generating and manipulating version numbers. The version number is composed of four numbers:

major.minor.build.revision.

9.3.3 Language Independence

In component oriented programming, the client is developed independently of the server. Because the client component interacts with the server component only at runtime, the only thing that binds the two together is binary compatibility. Consequently the programming languages that implement the client and server components should not affect their ability to interact with each other at runtime. Programming language independence means exactly that: when I develop components, my choice of programming language should be irrelevant. Programming language independence promotes the interaction of components, and their adoption and reuse in general.

.net Programming Language

.net achieves certain level of programming language independence through the implementation and architecture of the Common Language Runtime (CLR). CLR makes available a common context within which all .net components can execute, regardless of the programming language in which they are written.
All .net components are compiled to IL (Intermediate Language) before runtime, regardless of the programming language, the result is language-independent by definition. At runtime, the JIT compiler links the component entry points to the client calls.

To develop .net components, I must use one of the .net programming languages and compilers available with the .net framework. The core programming languages in .net are Managed C++, VB.NET, C#, and J#. Third party compiler vendors are also targeting the CLR, with more than 20 additional languages. Older compilers produce code that does not target the CLR and therefore can not be managed by the CLR.

Avalon programming language

The original proposal of Avalon framework was named Java Apache Server Framework. A framework that had the framework, components, utilities, and a server’s kernel implementation all in one project. It was to be the basis for all Java server code at Apache. Since then Java is the only programming language used in Avalon framework.

As we all know Java is platform independent and is widespread programming language. I think this is the main reason why Avalon architects preferred and targeted Java as the Avalon programming language.

9.4 Designing Component for Reuse

Designing component for reuse is the process of designing and implementing a component once and using it over and over again in different contexts. This will clearly realize higher productivity increases, taking advantage of best-in-class solutions and the consequent improved quality. The idea of designing software systems from semi autonomous components which can be reused in different contexts is intuitively appealing. Nevertheless, just organizing applications in forms of components does not by itself make sure that these components will be reusable, or that meaningful levels of reuse will actually happen. In reality, to achieve significant levels of reuse the generality and scopes of component must be carefully analyzed, designed and developed for reuse.

From the viewpoint of component-based software development, there are two kinds of activities: designing with reuse and designing for reuse. de-
signing with reuse is designing new solutions by combining existing reusable components. Search and retrieval are the main activities here. Designing for reuse is the major task of the two activities and it is about analyzing, designing, implementing and documenting components for future reuse.

9.4.1 Designing for Reuse

The design effort in component design for reuse is normally organized following traditional software design approaches, with the emphasis of component standards. For example, each component provides two types of interfaces:

- Component provided interface, which specifies the public services this component will provide to its clients.
- Component required interface, which defines the services this component requires from the underlaying infrastructure in order to function properly.

Designing for reuse requires both system infrastructure which supports the approach and component designer that follow component-based software development principles. Designing for reuse should also possess the characteristics of sound engineering: loose coupling, tight cohesion, and sufficient capabilities. The one major challenge when designing component for reuse is to predict the future use of the software that you are about to design and implement. The flexibility of software makes for a overload of options and many possible combinations of usage and the component designer have to make many design decisions to make sure both re-usable and re-useful component are achieved. Even though designing component for reuse has been practiced in one form or another over many years it is still a relatively young discipline and it is not yet clear what is feature of the component technology being used or true principle of component reuse.

9.4.2 Designing with Reuse

Systematic component reuse and the reuse of software influence today almost the whole software engineering process. Source code is most commonly reused. But software reuse is not only source code reuse. Many different artifact for reuse range from algorithms and ideas to any document that is created during the software life cycle.
There are different levels of software reuse which need to be addressed when designing component with reuse. *Copying source code*, for example, is the lowest level of software reuse. *Function libraries* are better form of reuse than copying source code, but it is not extensible. *Class libraries* are improved form of reuse, and they are extensible. Nevertheless, it requires deep knowledge before classes can be reused. Additionally, it supports only “white-box” reuse. The client code will be affected if internals of classes changed. For example, in an object-oriented language such as Java or C++, extended classes are coupled to the parent class implementation. Changes in any of the parent classes in the hierarchy of the inheritance would break extended classes [Wan05].

*Reusing component* supports highest level of software reuse because it allows various types of reuse including black-box reuse, gray-box reuse, and white-box reuse. White-box reuse means that the source code of a component is made available and can be studied, modified, adapted and reused. Black-box reuse is the highest level and it is based on the principles of information hiding. The component interface defines the services a client software may request from the component [Wan05]. The client software has no knowledge about the component internals and how the interface is implemented. On the other hand, the component provides the implementation of the interface that the clients rely on. As long as the part of the interface which the client is interacting remain unchanged, components can internally be changed without breaking clients. Gray-box reuse is somewhere in between black-box reuse and white-box reuse.

### 9.5 Component Search

One of the greatest obstacles in component-based software development that a component assembler is faced with is finding suitable components. The situation is made worse by the lack of a method for cataloging and classifying components and a universally applicable method for searching and retrieval from a repository. Developers are faced with a lack of search tools that can effectively assist the finding of qualified components from one or more heterogeneous repositories based on a given query. Searching and subsequently selecting reusable components from component repositories has become a key problem for component-based development and also for achieving the overall usability of the components themselves.
Current component models have made component-based software development practical. Nevertheless, these models are limited in the sense that their support for the specification/characterization of software components mainly deals with structure and syntactic issues. To avoid mismatch and misuse of components, more common specification of software components is required, especially in a situation where components are dynamically discovered and used at run-time over company intranets and the Internet [Han99].

Component matching, a basic aspect of the component search problem, has been well-studied problem [Fra94], resulting in many different matching techniques such as facet, keyword, signature and specification matching techniques. Nevertheless, each matching technique when individually used for component search often produces a large or small number of (sometimes irrelevant) hits. There have been industrial and experimental projects that build systems from existing components, the approaches taken are ad hoc and heavily rely on the specifics of the components and systems concerned. That is, component-based software development is still very much in its childhood.

9.6 Conclusion

Software component technology is an approach which tries to maximize the reuse of existing software components. An obvious advantage of this approach is that overall development time is reduced. Fewer software components need be specified, designed, implemented and validated. However, time reduction is only one potential advantage of reuse. Systematic component reuse in the development process offers further advantages:

1. System reliability is increased, reused components, which have been exercised in working systems, should be more reliable than new components. These components have been tested in operational systems and have therefore been tested to realistic operating conditions.

2. Overall process risk is reduced if a component already exists; there is less uncertainty in the costs of reusing that component than in the costs of development. This is an important factor for project management as it reduces the uncertainties in project cost estimation. This is particularly true when relatively large components such as sub-systems are reused.
3. Effective use can be made of specialist. Instead of application specialists doing the same boring work on different projects, these specialists can develop interesting new reusable components which encapsulate their knowledge.

4. Software development time can be reduced. Bringing a system to market as early as possible when it is mostly wanted is often more important than overall development costs. Reusing components speeds up system production because both development and validation time should be reduced.

On the other hand component-based software development is inherently complex and difficult to master. In addition, plugging into your system a third-party component creates fear, uncertainty, and doubt (FUD) as you have no access to the component’s internals. Therefore, buying components is not an easy task. Components are typically “black-boxes” with their source code not available to the end-user.

Along with the component-based software advantages, there are also some disadvantages:

1. Components have often only a brief description of their functionalities. Mostly only accompanied by a user’s manual, nothing else.

2. It is hard to guarantee critical system-wide properties in the presence of arbitrary third-part components.

3. Components carry no guarantees of adequate testing.

4. There are no or only a limited description of the quality of the component [Crn01].

There is considerable indication that software component technology is emerging as a major factor in how systems are being, and will be, built for the near future and many computer scientists are hoping that it will bring solutions to all the problems that led to software crisis. However, as it is currently, it is still immature for most companies to jump to implementation now. My advice, based on the short research I made, is to either “wait and see” or “proceed with caution”.
9.7 Achievements

Doing this thesis has been very instructive and interesting. I have learned and deepen my knowledge of CORBA, EJB, .net and XML. However, I have to admit that the thesis has had its difficult days. I decided early to use Avalon’s framework for implementing my prototype component, for the simple reason that the framework is written in Java and is freely downloadable. In addition, development of Avalon framework is hosted at Apache, known its successful Apache web server project, among other successful projects.

To start with, creating the component its self was not much problem, but building it with Avalon’s build file was a headache. Avalon uses Ant for building (compiling and packaging) its components. Ant is a build tool, like Make, which uses XML for specifying the order of what need to be done. I used Avalon’s build-file template to create my component’s build-file and after I run Ant to build the component it started printing out long stack of errors. It took me a long time to fix those errors. The main problem was that the Avalon’s development documentation was not developer-friendly (Avalon has recently released new developer whitepaper, among others).

To conclude, I have achieved my core objectives. On other hand, I have not touched my advanced objective which was to develop a component framework. Completing my core objectives was its self advanced.

In the course of this thesis I have initially defined component-based software and related technology, such as component framework, component system architecture. I also explained how components are developed, deployed and used in chapter 2. I then investigated three main component technologies, namely CORBA CCM, EJB and .net. Finally, I designed, implemented, assembled and tested a simple component-based file-sharing-server on Avalon framework and .net framework.
Bibliography


Appendix A

Case Study Design Specification
Design Specification

1 Introduction

1.1. Purpose

This document provides the design for the file sharing program. I tried to keep the requirement, and the program as whole, as simple as possible.

![Client Use Cases Diagram]

Figure 1 Client Use Cases

2. Behavioral Requirements

This section describes a set of scenarios that illustrate, from the user’s point of view, what will be experienced when using the program under various situations. All the actors are assumed to be “User”.

2.1 Use Case: Log on

Precondition:
- User’s system has the client software

Event Flow:
- User starts the client software
- The application pops up the “Log on” window prompting user to enter server’s hostname and user’s desired username.
- User enters hostname and username and clicks on “OK” button
- The client application closes the “Log on” window and starts the client application.
- The server verifies the username so it does not clash with any other current user.
- The client application initially scans User’s shared folder and sends a list of all file names in the shared folder to the server.

Post condition:
- User is verified and allowed to use the server

Exception:
User is prompted to enter username again because the username entered is already in use.
User is prompted to enter host name again because the server hostname entered is unknown.

2.2 Use Case: issue query
Precondition:
- Client software has been started
- User has been successfully logged on
- There is at least one other user logged on to the server

Event Flow:
- User types the search string in the Search text field.
- User clicks on “Search” button or presses “ENTER”
- The search string is sent to the server via TCP/IP connection.
- Server makes search and sends back any match to the client

Post condition:
- Client receives the query result

Exception:
- Client application times out because there was no match on User’s search string

2.3 Use Case: Retrieve file
Precondition:
- User has been successfully logged on
- User issued query
- User received query result from the server.

Event Flow:
- User double clicks the desired match.
- The client application establishes TCP/IP connection to the remote client.

Post condition:
- The client application downloads the file to the local machine

Exception:
- Client application times out because the other user is no longer available.

2.3 Use Case: Disconnect
Precondition:
- User has been successfully logged on

Event Flow:
- User chooses Exit from the File menu or clicks the close X.

Post condition:
- The client application informs the server that it want to disconnect.
- The client application shuts down.

Exception:
- The server maybe unreachable.
2.4 Graphical User Interface (GUI)
The client application will have GUI for logging in and making search. The GUI also has progress bar for downloading files which visualize the state of a download. When the client application is started the user will get a pop-up window with the server name and user name textboxes that the user will have to fill out to connect to the server. If the connection fails the same window will pop up again, otherwise a new larger window will appear with menu-bar and textbox for search.

3 Functional design

3.1 Description
The server allows multiple clients to make search at the same time through the server where all clients are connected to. It is a Java application that can be run under any JVM-compliant platform. It supports users to share files from different platforms, as long as they are connected to the internet (or the same intranet). Two client applications can be running on the same physical machine. The shared files can be in any sort of media such as text, sound or graphic with any size.

3.2 Risks
The main risk in system, as all centralized-server system, is the “one point failure”. If the server goes down, for some reason, the whole system will collapse, because the server is the connecting point of all the clients. Server connection might get lost due to power failure or due to physical problem in the server. The server might crash due to bug in the server application.

3.3 Technical issues
The primary design goal of the system will be to verify programmability, reusability, performance and robustness of component-based system. Therefore the GUI and security issue will be kept as simple as possible.

3.4 File Retrieval
The client initiates a session with a remote client’s daemon by sending the name of the requested file. When retrieving a file the communication between the clients does not go through the server, the retrieving client establishes direct TCP/IP connection to the other client’s file server and downloads the desired file(s). Each client is basically a client and a server at the same time; while they are client to the server they also host their own file-server which other clients can download files from.
Use Case: User Log on

1. Log In

1.1: Login User

1.1.1: Authenticate User

1.1.2: Authenticated/Denied

1.2: Prompt Re-enter Login

Figure 2 Log on Use Case

Use Case: Issue Query

1: Issue Query

1.1: Send Query String

1.1.1: Search Query

1.1.2: Query Hit

Figure 3 Search Use Case
Figure 4: Server class diagram
Server class diagram (C#)

Figure 5: Server class diagram (C#)
Figure 6 Client class diagram
Figure 7 Retrieve File Use Case
Figure 8 Login Dialog

Figure 9 Screen shot of a search result
Figure 10 Screen shot of downloading file
Appendix B

Avalon Framework

The Apache Avalon project is an attempt to create, design, develop and maintain a common component framework and set of ready components for applications written using the Java programming language. Avalon is a framework which allows components of varying size to be created and used in an application, via a specific set of life cycle methods. Avalon is mainly used in server-side oriented applications, but it is quite flexible and can even be used in client-side oriented applications.

The area of usage for Avalon framework is relatively broad. One may only want to create specific components that can be managed in a well defined way, or may want to use the many services and components available within Avalon project, or want to create complete applications, such as E-mail or FTP server, in one of the server oriented containers such as Phoenix. This means that you can use Avalon framework for only what you need to use it, and it is scalable as your application grows.

The framework concept is broad in software industry. A Framework that focus on a single industry like financial systems or medical is called vertical market framework [Ava02]. The reason is that the same framework may not work in other industries. Framework that is generic enough to be used across multiple industries is known as horizontal market framework. Avalon is clearly a horizontal market framework. It is possible to build vertical market framework on top of Avalon’s Framework.

In the beginning Avalon started as the Java Apache Server Framework that had the framework, components, utilities, and a server’s kernel implementation all in one big project.

As all the pieces of Avalon project were of different development stages, and had different release cycles, Avalon project was broken down into smaller and easily managed subprojects. The main subprojects are: Avalon Framework, Phoenix, Excalibur, Cornerstone and LogKit [Ava02]. The change also helped developers new to Avalon to get quick start and learn Avalon in distinct subprojects. This dramatically changed Avalon’s learning curve. Following is a short description of the main subprojects:

Framework

The Framework is the specification and basis for all the sub-projects within Apache Avalon. It defines the contracts, interfaces and default implementations for Avalon sub-projects. The Framework is the biggest sub-project, and consequently is the most mature project.