

Design Patterns for Distributed Programming: Sharing Objects with Java

*A thesis submitted in fulfilment of the
requirements for the award of the degree*

Master of Science (Hons)

at the

UNIVERSITY OF WOLLONGONG

by

Robert Ott

Department of Computer Science

June 1996

UNIVERSITY OF WOLLONGONG

Declaration Relating to Disposition of Thesis

This is to certify that I *Robert Ott* being a candidate for the degree of *Master of Science (Hons)* am fully aware of the policy of the University relating to the retention and use of higher degree theses, namely that the University retains the copies of any thesis submitted for examination and that the University holds that no thesis submitted for a higher degree should be retained in the Library for record purposes only, but within copyright privileges of the author, should be public property and accessible for consultation at the discretion of the Librarian.

In the light of these provisions I declare that I grant the University Librarian permission to publish or to authorise publication of my thesis on whole or in part, or grant access to it, as he deems fit.

Signature of Author _____

Witness _____

Date _____

Disclaimer

I hereby declare that I am the sole author of this thesis. I also declare that the material presented within is my own work, except where duly acknowledged, and that I am not aware of any similar work either prior to this thesis or currently being pursued.

Robert Ott

June 1996

Abstract

This thesis presents results from studies on the design of distributed applications. After studying implementations of schemes for object sharing, and broker based systems, two new design patterns have been abstracted. The first pattern, Simple Shared Object, describes a client/server based system for sharing objects across address spaces. The pattern is implemented in C++ using an experimental class library for distributed programming. The second pattern, Deputy, is a contribution to multi-threaded programming techniques. The pattern describes a general way for delegating commands from one object to another by using multi-threading. The implementation of the pattern is shown using the Java programming language.

The main part of this thesis shows a broker based system for sharing objects across address spaces. The system is an advanced version of the Simple Shared Object pattern and is fully implemented in the Java programming language. The system is called Simple Shared Object Request Broker (abbreviated to SSORB). A sample application using the SSORB system shows how a distributed application can be implemented. Then, a short evaluation of the system touches aspects such as portability, performance and reusability of the SSORB system. The SSORB system fulfills the lack of classes for the facilitation of object sharing across address spaces in the Java development kit.

Dedication

I dedicate this thesis to my parents,
who have been adoring my life from birth
with affection and love.



Ich widme diese Dissertation meinen Eltern,
die mein Leben seit meiner Geburt mit Zuneigung
und Liebe schmücken.

Acknowledgments

An important key point of every research is to have someone to talk about problems throughout the research, in short, to have a smart guardian angel as a supervisor. I was lucky to have Associate Professor Neil A. B. Gray as my supervisor and would like to express my sincere thanks to him. I learned him not only as a smart and correct expert in object-oriented technology, I also experienced Neil as a helpful and concerned friend.

What is a research student doing when he comes home from the University? I was pleased to have my two housemates Hakan Yilmaz and Murat Polat to share most of the time here in Wollongong. I experienced these two guys as very close friends and remember endless nights discussing about our research. I will never forget the joyful time we spent together here in Australia.

A research in Computer Science is also a collaboration between experts all around the world. I would like to thank all the individuals who helped me by e-mail. I especially would like to express my thanks to Grady Booch (Rational Software Corporation), Gerard Meszaros (Bell Northern Research, Ltd), Douglas C. Schmidt (Washington University) and Bob Hathaway (SIGS Publications) for their contribution to this research.

A special thank-you is due to my friend Roman Selm in Switzerland, who has set the course of my career dominantly at those days as he was my boss in Grossenbacher Elektronik AG. I also would like to thank him for his critical comments to my paper "Simple Shared Object".

I believe that every person on earth has a soulmate, a special and close friend. I call Edgar Sattler as my soulmate and want to thank him for his encouragement and the psychological support he gave via phone, fax, e-mail, c-mail, letters and postcards. ; -)

Finally and most importantly I wish to thank my parents, my sister and my brother for their encouragement and help during my stay here in Australia.

Table of Contents

Declaration and Disclaimer	ii
Abstract.....	iii
Dedication	iv
Acknowledgments	v
Table of Contents.....	vi
Table of Figures	xi
1. Introduction	1
1.1 Motivation.....	1
1.2 Objectives.....	2
1.3 Outline	3
2. Problem Context	5
2.1 Overview.....	5
2.2 Object-Oriented Technology - from Rhetoric to Reality	5
2.3 Design Patterns	7
2.3.1 Definition of Design Patterns.....	7
2.3.2 The Idea of Design Patterns	9
2.3.3 An Example of a Pattern.....	10
2.3.4 Antipatterns	11
2.3.5 Design Patterns - A Way to tame C++.....	12
2.4 Application Frameworks.....	13
2.4.1 Smalltalk Framework (MVC-Triad).....	14
2.4.2 The MacApp Framework	16
2.4.3 Object-Studio - A Framework with Database Access and CASE.....	18
2.5 Distributed Programming.....	19
2.5.1 Reasons for Distributed Programming	20

2.5.2 Common Object Request Broker Architecture (CORBA)	21
2.5.3 The Java language and the World Wide Web	23
2.6 Summary	26
3. A Pattern for Sharing Objects - “Simple Shared Object”	28
3.1 Overview	28
3.2 An Experimental Class Library - Distributed Object Communication	28
3.2.1 The I/O-Dispatcher Object	29
3.2.2 Channel Objects	29
3.2.3 Coms Objects	30
3.2.4 The System (Collaborations)	30
3.2.5 A Typical Object Composition in an Example	31
3.3 Extension for Sharing Objects	32
3.4 “Simple Shared Object” - An Architectural Pattern for Simple Object Sharing	33
3.4.1 Intent	33
3.4.2 Motivation	33
3.4.2.1 Needs for Shared Objects	35
3.4.2.2 Behaviour of Shared Objects	35
3.4.3 Applicability	35
3.4.4 Structure	36
3.4.5 Participants	39
3.4.6 Collaborations	42
3.4.6.1 Shared Object Creation	42
3.4.6.2 Shared Object Modification	45
3.4.6.3 Shared Object Deletion	46
3.4.7 Consequences	47
3.4.8 Implementation	49
3.4.9 Sample Code	50
3.4.10 Variations	51
3.4.11 Known Uses	51
3.4.12 Related Patterns	52
3.5 Summary	52
4. Simple Shared Object Request Broker (SSORB) - A Tailored System for Java	53
4.1 Overview	53
4.2 Broker Systems for Collaborative Applications	54
4.2.1 On what systems (architectures) could CSCW applications base in the future?	54
4.2.2 Could another programming language than C++ be more appropriate?	55

4.2.3 Could an already existing model be specialised for simple object sharing?	56
4.3 The Basic Model of the SSORB	57
4.4 Message Objects within the SSORB System	58
4.5 The Broker	60
4.6 Object Servers	62
4.6.1 Referencing of Shared Objects across Address Spaces	63
4.6.2 “Well-known” Proxy Lists	65
4.7 Client Applications	65
4.8 “Deputy” - A Behavioural Pattern for Delegating Commands to a Multi-Threaded Deputy Object	67
4.8.1 Intent	68
4.8.2 Motivation	68
4.8.2.1 Requirements of Deputy Objects	69
4.8.2.2 Behaviours of Deputy Objects	69
4.8.3 Applicability	70
4.8.4 Structure	70
4.8.5 Participants	73
4.8.6 Collaborations	74
4.8.7 Consequences	75
4.8.8 Implementation	76
4.8.8.1 The “DPYCommand” Class	76
4.8.8.2 The “DPYSmartCommand” Class	77
4.8.8.3 The “DPYDumbCommand” Class	77
4.8.8.4 The “DPYReply” Class	78
4.8.8.5 The “DPYDelegator” Interface	79
4.8.8.6 The “DPYDeputy” Class	79
4.8.9 Sample Code	80
4.8.10 Variations	81
4.8.11 Known Uses	82
4.8.12 Related Patterns	82
4.9 Summary	82
5. Implementation of the SSORB System	84
5.1 Overview	84
5.2 The Implementation in General	85
5.2.1 Name Conventions	85
5.2.2 Pseudo Code	86
5.2.3 Complete Class Diagram of the SSORB System	86
5.2.4 Inter-Component Communication with Message Objects	88

5.2.5 Exception Handling within the SSORB System.....	90
5.2.5.1 The “SSOException” Class.....	91
5.2.5.2 The “SSOReplyException” Class.....	91
5.3 The Broker Implementation.....	92
5.3.1 The Java Code for the Broker Classes.....	92
5.3.1.1 The “SSORBCustomerHandler” Class.....	92
5.3.1.2 The “SSORBReceptionist” Class.....	95
5.3.1.3 The “SSORBAdministrator” Class.....	96
5.3.1.4 The “SSORB” Class.....	100
5.4 The Customer Implementation - Object Servers and Client Applications.....	101
5.4.1 The Java Code for the Application Independent Object Server and Client Application Classes.....	101
5.4.1.1 The “SSOMainInterface” Interface.....	101
5.4.1.2 The “SSOReceptionist” Class.....	102
5.4.1.3 The “SSORegistration” Class.....	104
5.4.1.4 The “SSOProxyHandler” Class.....	105
5.4.1.5 The “SSOProxy” Class.....	107
5.4.1.6 The “SSOProxyRef” Class.....	110
5.4.1.7 The “SSOProxyList” Class.....	112
5.5 Summary.....	116
6. A Sample Application and Evaluation.....	118
6.1 Overview.....	118
6.2 “Application Joiner” - A Sample Application using the SSORB System.....	119
6.2.1 Analysis.....	119
6.2.2 Design.....	122
6.2.2.1 Identification of the Sharable Classes.....	122
6.2.2.2 Specialisation of the User Interface Classes.....	124
6.2.3 Evolution.....	127
6.2.3.1 Implementation of a Specialised Sharable Proxy Class.....	127
6.2.3.2 The Application Joiner Program - A Posed Example.....	135
6.2.4 Modification.....	137
6.3 Evaluation.....	137
6.3.1 Portability.....	138
6.3.2 Performance.....	139
6.3.3 Reusability.....	142
6.4 Summary.....	143
7. Conclusions and Future Research.....	144
7.1 Conclusions.....	144
7.1.1 Contributions.....	144

7.1.1.1 Contributions to Design Patterns	144
7.1.1.2 Contributions to Distributed Programming	145
7.1.1.3 Contributions to Applications Using Java.....	145
7.1.2 Reflections	146
7.1.2.1 Keeping a Catalogue of Design Patterns	146
7.1.2.2 Java: An Alternative to C++	146
7.1.2.3 Distributed Software Systems.....	147
7.1.2.4 The World Wide Web: An Important Means for Research	147
7.2 Future Research.....	147
Appendix A: Abbreviations.....	150
Appendix B: Trademarks and Terms	152
Appendix C: Java Code Listings.....	154
Bibliography	157

Table of Figures

Figure 2.1: Class diagram of the Singleton pattern.....	10
Figure 2.2: The components of the Model-View-Controller triad.....	15
Figure 2.3: Traditional programming and MacApp programming.....	16
Figure 2.4: Some of the popular classes within the MacApp class library	17
Figure 2.5: An Object Request Broker (ORB) and its environment.....	22
Figure 2.6: From Java language source code to the Java interpreter.....	24
Figure 3.1: A particular application using the experimental class library	31
Figure 3.2: A typical object composition using the experimental class library	32
Figure 3.3: A distributed application with shared objects	34
Figure 3.4: Class diagram of the Simple Shared Object pattern	37
Figure 3.5: Illustration of the Simple Shared Object pattern by an object diagram.....	40
Figure 3.6: Collaboration between participants in the creation process.....	43
Figure 3.7: Collaboration between participants in the modification process.....	45
Figure 3.8: Collaboration between participants in the deletion process.....	47
Figure 4.1: Traditional and broker based Client/Server systems	56
Figure 4.2: The components of the SSORB system and their responsibilities.....	58
Figure 4.3: The components of the Simple Shared Object Request Broker.....	61
Figure 4.4: A typical object constellation within a SSORB object server.....	63
Figure 4.5: The proxy reference class within the SSORB system	64
Figure 4.6: A typical object constellation within a SSORB client application.....	66
Figure 4.7: A deputy executing delegated commands	68
Figure 4.8: Class diagram of the Deputy pattern.....	71
Figure 4.9: Object diagram of the Deputy pattern.....	73
Figure 4.10: Collaborations between participants of the Deputy pattern.....	75
Figure 5.1: The classes within the SSORB system and their major relationships.....	87
Figure 5.2: Interaction table for message exchanges within the SSORB system.....	89
Figure 5.3: The structure of the SSORBCustomerHandler class	93
Figure 5.4: The structure of the SSORBReceptionist class	95
Figure 5.5: The structure of the SSORBAdministrator class	97

Figure 5.6: The structure of the SSORB class	100
Figure 5.7: The structure of the interface SSOMainInterface	102
Figure 5.8: The structure of the SSORceptionist class	103
Figure 5.9: The structure of the SSORegistration class	105
Figure 5.10: The structure of the SSOProxyHandler class	106
Figure 5.11: The structure of the SSOProxy class	108
Figure 5.12: The structure of the SSOProxyRef class	111
Figure 5.13: The structure of the SSOProxyList class.....	113
Figure 6.1: A virtual collaboration house “Application Joiner”.....	120
Figure 6.2: Scenario of a user entering the virtual collaboration house.....	121
Figure 6.3: The sharable classes of the Application Joiner program.....	123
Figure 6.4: A sketch of the Application Joiner user interface	124
Figure 6.5: The GUI classes of the Application Joiner program	126
Figure 6.6: The structures of the classes User and UserProxy	128
Figure 6.7: A screen dump from the sample SSORB application (on Windows 95)....	136
Figure 6.8: A screen dump from the sample SSORB application (on Unix).....	136
Figure 6.9: Number of messages exchanged, classified by message tasks	140
Figure 6.10: Direct connection between an object server and a client application	141

Chapter One

1. Introduction

1.1 Motivation

Currently, computers are used primarily as stand-alone systems, this is despite the fact that most computers are nowadays networked. The degree of collaboration via these networks is limited. Most collaborations across networks are limited to the exchange of electronic mail messages between people and the down-loading of data files. Collaborative applications require “shared objects” that can be used by users at different locations concurrently. An example for a collaborative application would be a common white-board (for brainstorming or discussion) used concurrently by a number of collaborators.

Over the past ten years, powerful programming frameworks such as MacApp, ET++ and Smalltalk derivatives have been facilitating the implementation of application programs that are essentially “editors” that can be used by a single user. Classes in these frameworks handle the creation of graphical user interfaces, interactions with the user, and the handling of files (persistence). Support for networks, such as classes facilitating the implementation of CSCW applications (Computer Support for Co-operative Work), is usually missing or very primitive.

Neil Gray [Gray95] provided an experimental class library for distributed applications. The library includes network handling classes that allow programs in different address spaces to communicate. In addition, the library provides classes for handling X-Windows user interface components such as action buttons and radio clusters. The library is implemented in C++ and it can easily be adapted to different operating systems. Distributed applications using the experimental class library communicate via I/O-Dispatcher objects.

This research uses Gray's experimental class library as starting point for finding ways for sharing objects across address spaces.

1.2 Objectives

The first part of this research extends the existing experimental class library with additional classes. These additional classes allow the creation of objects that can be shared across address spaces. The basic idea of this extension was to have primary copies of shared objects on a server program in one address space, and mirrored copies of the shared objects on client programs in other address spaces.

From this implementation a more general design pattern has been abstracted. The pattern is called Simple Shared Object and describes a way for sharing objects across address spaces. The pattern guarantees that the mirrored copies of shared objects on clients are automatically updated to the state of the primary copies on the server.

Evaluation of the Simple Shared Object pattern suggested that a more advanced system for sharing objects should be broker based. At this stage, Sun Microsystems Inc. released its new programming language "Java". A short exploration of the promised features of Java led to the decision to implement the new broker based system for sharing objects in Java (Simple Shared Object Request Broker - "SSORB").

The SSORB system combines the Simple Shared Object pattern with the basic structure of CORBA, without getting too complex. The design and the implementation of the SSORB system uses multi-threading. In some parts of the broker based system, it was necessary to delegate commands from one object/thread to another. This led to the identification of a second design pattern, one that allows the delegation of commands from one object/thread to another. The commands are delegated to a "deputy object" executing commands in parallel in different threads. Thus, the design patterns has been named "Deputy".

A further step was to evaluate the SSORB system. This has been done by implementing a sample distributed application using the SSORB system for sharing objects. The sample application helps to understand how the SSORB system can be applied for implementing CSCW applications.

1.3 Outline

Generally, all chapters (except chapters 1 and 7) begin with an “Overview” section and finish with a “Summary” section.

Chapter 2 surveys the literature on “design patterns”, “frameworks” and “distributed programming”. It explains the major aims of object-oriented programming, introduced with a short history. Also discussed are the ways in which “design patterns” can provide greater reusability of software systems. Finally the chapter touches on the importance of computer networks and the resulting opportunities for collaborations across these networks so leading into a discussion of distributed programming techniques.

Chapter 3 describes briefly Neil Gray’s experimental class library for distributed programming on which this research is based. The description shows how the class library enables objects in different address spaces to communicate. Then, the extensions for the experimental class library to provide sharable objects across address spaces are discussed. Finally, the chapter shows a design pattern called “Simple Shared Object” for sharing object across networks. The “Simple Shared Object” pattern has been published by SIGS Publications in the online-journal “Object Currents” [Ott96].

Chapter 4 presents the design of an advanced system for sharing objects. The system is based on the Simple Shared Object pattern. The system is called “Simple Shared Object Request Broker (SSORB)” and uses an application independent broker between object servers and client applications. The central broker can handle multiple object servers and their client applications concurrently. Even though the system is meant to be implemented in the Java programming language, the design is mostly programming language independent. At the end of the chapter, a design pattern called “Deputy” is described. The pattern has been extracted from common needs within the SSORB system and has been extended for more general use. The “Deputy” pattern has been submitted for publication in the journal “Java Report” (SIGS Publications).

Chapter 5 shows the implementation of the SSORB system in the Java programming language. All application independent classes of the SSORB system are explained showing class structures in diagrams and explaining the major methods in detail. The implementation of some major components is shown using pseudo code. The broker of

the SSORB system does not require any application dependent code. Thus, it is directly executable using a Java interpreter. On the other hand, object servers and client applications must specialise classes provided by the SSORB system.

Chapter 6 shows a small application program using the SSORB system and evaluates this system. The sample application is shown in four phases: analysis, design, evolution and modification. It illustrates how a distributed application could be implemented using the SSORB system for sharing objects. The evaluation considers three factors: portability, performance and reusability.

Chapter 7 concludes this thesis. It describes the contributions to design patterns, to distributed programming and to applications using the Java programming language. Furthermore, it explains my reflections after conducting this research. Finally, the last section identifies directions for future research on an advanced SSORB system.

Chapter Two

2. Problem Context

2.1 Overview

This chapter introduces the notions of **Design Patterns**, **Frameworks** and **Distributed Programming**. A short history of the object-oriented approach in software development summarises developments of the last decade. The contribution of design patterns and frameworks to today's software industry illustrates the importance of these techniques.

The discussion shows how design patterns apply in practice and why they are becoming increasingly important for software designers. Frameworks are described briefly and it is shown that the existing frameworks are mostly inappropriate for distributed programming.

The increasing availability of computer networks around the world and within companies opens new ways of collaboration between people at different locations. Distributed applications can allow users to work together across networks. The increasing demand of distributed programming in modern applications is discussed in the section 2.5 (Distributed Programming).

The last part of this chapter summarises the conclusions resulting from this literature survey. It will show why this research has focused on design patterns for distributed programming.

2.2 Object-Oriented Technology - from Rhetoric to Reality

With the inauguration of object-oriented approaches in computer programming in the 1980's, scientists and programmers thought they had found "the solution" to create reus-

able software. The reality showed that it was just one of the first steps, however a very important one, forward to reusability of software components.

During the last decade, dozens of object-oriented programming languages were born and many of them have already died. As a result, so-called “reusable software components” could not be reused because the language, in which the components were written in, were not available long enough or the programming language had experienced a “major” release update. In the late 1980’s, some object-oriented programming languages such as C++ and Smalltalk became widely known and they ran through approaches of standardisation. Today, ironic joke messages, such as that below, come across the Internet and show a touch of reality.

Question: “How many C++ programmers does it take to change a light bulb?”

Answer: “You’re still thinking procedurally. A properly designed light bulb object would inherit a change method from a generic light bulb class, so all you’d have to do is send a light bulb change message!”

After the establishment of some object-oriented programming languages, the foundation for reuse of software components was set. Class libraries with “semifinished” software pieces such as network handling classes and graphical user interface classes have been built and facilitate today’s software development process. These class libraries became well known with the term *framework*. Many software houses distribute frameworks with so-called ‘unique’ features. In fact, although every implementation of a framework might be different, the features that are supported by most of these frameworks are quite similar. A more detailed explanation of some frameworks will be shown in section 2.4 (Application Frameworks).

Another approach to provide software reusability is to describe small software components in a way that they can easily be implemented without focusing on specific frameworks or programming languages. Such descriptions explain reusable patterns of software components that occur over and over again. The name *Design Patterns* has been established for such descriptions over the last few years. A definition of Design Patterns is given below in section 2.3 (Design Patterns).

2.3 Design Patterns

Human beings have almost always been searching new ways to do things or to have things done. These new ways should reach aims faster, better or with more quality than before. Shipwrights manufactured ships to cross oceans, architects created steel constructions to build skyscrapers and space scientists found ways to transport people to moon and back to earth. Most of these ways have been based on other people's research and experiences. People used successful patterns of how things were done before. From this point of view, it is not a new technology to reach aims by using patterns, rather it is that we use patterns more consciously.

Reusability has been an important issue in software development processes during the changes of processor architectures. Hardware components could easily be thrown away while software components had to survive over years. Pieces of code have been reused by copying these pieces to new solutions for new computer generations. Some pieces of code could not directly be copied from one system to another. However the patterns that were implemented in such pieces could be extracted and reused for new software solutions on different computer architectures. There are still some different views about the definition of design patterns. The definition of Erich Gamma *et al.* [Gam+95] will be applied in this thesis and is described as follows.

2.3.1 Definition of Design Patterns

Erich Gamma *et al.* [Gam+95] used the remark of Christopher Alexander *et al.* [Alex+77] which says, “*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice*”. According to Gamma, a pattern should consist of four essential elements, which are a **pattern name**, a **problem**, a **solution** and the **consequences**.

Just as each person has a name, each pattern needs a **pattern name**. That name should be short and to the point. Each well-known pattern will increase the vocabulary of software designers and programmers. It gives a foundation for detailed design discussions without spending time for explaining each used design pattern over and over again. Just as programs use indexes to reach rows in databases, design patterns can easily be re-

called information in people's mind by using unique pattern names. However, it is mostly one of the harder jobs to find pattern names that really are short and to the point.

The **problem** element of a design pattern describes when the pattern can be applied. In terms of sections within a description of a design pattern, the problem is described in the sections *Motivation* and *Applicability*. The motivation section focuses on the problem and its context. A small example where the described problem can occur, might help a reader to understand the context of the problem. Sometimes a design pattern can only be applied when a list of conditions are met. The applicability section gives detailed information for when a pattern should be used and when it should not be used.

The description of how a problem is solved is shown in the **solution** part of a design pattern. This part is usually divided into the sections *Structure*, *Participants* and *Collaborations*. The section *Structure* shows with help of a diagram the important elements of a pattern. The diagram normally presents a class diagram in a well-known notation such as Booch [Booch94], Yourdon [Your94] and Rumbaugh [Rumb+91, Rumb95]. The section *Participants* defines relationships and responsibilities of objects within a design pattern. Participants (important objects) are named and should be consistent throughout the document. The use of common notation in this section is also very helpful for a reader. The third section, which belongs to the solution part of a design pattern, is the *Collaborations*. Focussing on the participants, the collaborations between objects are shown in various interaction diagrams. Sometimes there is a need to show scenarios of participant interactions in this section. To summarise, the solution part of a design pattern shows the *Structure* (e.g., a class diagram), the *Participants* (e.g., an object diagram) and the *Collaborations* (e.g., an interaction diagram) with which the described problem can be solved.

A description of the benefits and the drawbacks of a design belongs to the **consequences** of a pattern. A critical evaluation of 'features' and 'non-features' of a pattern helps readers understand when a pattern is appropriate for a specific problem and when it is not. If a design pattern is intended to solve a problem in a specific programming language or for specific operating system, appropriate limits should be specified in the consequences of a pattern. Sometimes, there is a need to describe the consequences of

adapting a design pattern to application frameworks. Such descriptions of adaptations to frameworks such as MacApp can help a reader apply a design pattern.

2.3.2 The Idea of Design Patterns

Reusability in object-oriented programming can be seen as building houses out of bricks. Houses can be built by simply setting layers of bricks over each other. However, houses are expected to remain over decades or even centuries. Such life times are only possible if the bricks are well aligned and if the right amount of mortar is used to hold the bricks together. A replacement of houses with programs, bricks with objects and mortar with design shows that one of the important parts of a software development process is the design. However, how can the adequate design time be found? Many software projects died before the designs were finished because of an underestimation of the planned time for the designs. Other projects died because of wrongly designed interfaces. The use of design patterns could be a solution to some problems that occur in daily object-oriented software development.

The idea to use a framework for object-oriented software development has its root in the early 1980's when Smalltalk-80 was growing. Smalltalk-80 uses an object constellation that is called Model-View-Controller triad (see also section 2.4.1) for object representation on graphical user interfaces. Variations on the MVC triad [Kras+88] can be found in most of today's popular application frameworks for graphical user interfaces. The MVC triad can be seen as a design pattern and may be the most reused pattern in object-oriented technology.

In the early 1990's, design patterns became an important issue in object-oriented design approaches. Important milestones set Erich Gamma with his doctoral thesis [Gam92] and the later published book [Gam+95] describes many reusable elements for object-oriented software. Currently, at conferences such as Object-Oriented Programming Systems, Languages and Applications (OOPSLA) and European Conference on Object-Oriented Programming (ECOOP), researchers and program designers from the industry discuss design patterns more than ever before. Wolfgang Pree [Pree94] expresses design patterns as: *"In a nutshell, design patterns became a hot topic in the object-oriented community"*.

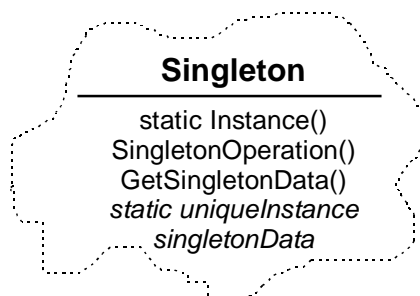


Figure 2.1: Class diagram of the Singleton pattern

2.3.3 An Example of a Pattern

A simple, however very useful, design pattern which Gamma's book [Gam+95] describes is called *Singleton*. Many programmers dream of applications without 'global' objects. Unfortunately, programs often need a few instances that can be accessed globally. An easy solution is to declare these instances as static objects. However, how can we guarantee that there is not a 'hidden' corner in a program creating a second instance of a global object? The *Singleton* pattern can give this guarantee and it provides furthermore that sole instances can be extended by subclassing.

Figure 2.1 show that the Singleton pattern consists of just one class. The constructor of a singleton class is declared as protected or private; this guarantees that the single instance can only be accessed via the static member function `Instance()`. The rest of the member function can be implemented like a normal class. The following C++ code shows a singleton class with just one 'pseudo' member function `DoSomething()`.

```

// -----
// Declaration of class Singleton
// -----

class Singleton {
public:
    static Singleton *Instance(void);
    void DoSomething(void);
protected:
    Singleton(void);
private:
    static Singleton *sInstance;
};
  
```

```

// -----
// Implementation of class Singleton
  
```

```
// -----  
Singleton *Singleton::sInstance = 0;  
Singleton *Singleton::Instance(void) {  
    if (!sInstance)  
        sInstance = new Singleton;  
    return sInstance;  
}  
Singleton::Singleton(void) {  
    // ...Construction of the singleton  
}  
void Singleton::DoSomething(void) {  
    // ...Implementation of DoSomething()  
}
```

This short description of the Singleton pattern can not be as detailed as the full description in Gamma's book [Gam+95]. However, it gives an idea that such a pattern can be reused over and over again. It can also be understood that a design discussion in a software project can be simplified by using terms such as *Singleton*, *Proxy*, *Observer* and *Abstract Factory*.

It should be noticed that Booch [Booch94] uses the term *mechanism* for object groups with a defined functionality. Therefore, it can be said that design patterns are also mechanism in Booch's terminology.

2.3.4 Antipatterns

Andrew Koenig introduced with the article [Koen95] the term "Antipatterns". He describes antipatterns as design patterns that look superficially like solutions for a specific problem, which, however, do not work in practice. This is an interesting perspective on patterns. Why should design patterns, which are ineffective in practice, be worth to be described? Koenig explains an answer for that question with an example how Thomas Edison tried to build the first electric light. Another example why it can be worth to describe "wrong" patterns can be seen as follows.

Human beings learn probably most effectively by experiences. The rule "Once bitten, twice shy!" can also be applied in software development processes. Probably every analyst, designer and programmer remembers bugs that they made during their software evolutions. It is probably these "bugs" that changes these scientists into experienced, smart experts during years of development. When designers can understand the learning

processes that others went through, they can save time and therefore money within their own projects. Descriptions of antipatterns can spread unintentional experiences, caused by wrong designs, to many other designers.

Antipatterns have another important role, when they are used in conjunction with well-designed patterns. Such “pattern-antipattern” pairs give designers, trying to understand a pattern, some kind of experiences by reading the antipatterns. Antipatterns can also warn designers not to do the same mistakes which other designers already made.

In short, antipatterns can enrich analysts and designers with valuable information for well grown patterns. Using “pattern-antipattern” pairs might contribute to make design patterns more understandable.

2.3.5 Design Patterns - A Way to tame C++

Soukup [Souk94] gives in his book “Taming C++” some examples of large software projects which died, because they became too complex or too complicated. Programmers themselves could not understand their own written code after projects became bigger. A major reason of that grade of complexity was that relationships between objects looked like big spider nets, however these nets were not organised like the nets of real spiders.

When programmers shift from structured, modular programming techniques to the object-oriented technique, they often try to apply their old style into an object-oriented approach. The fact, that the step towards to object-oriented programming requires changes of programming styles causes often that these approaches fail. When designers learn to divide large software projects in small and coherent parts, the first step to well-designed software is done. This rule supports the idea that the software quality and reusability can be improved by using well-organised design patterns in software projects.

Edward Yourdon [Your94, pp. 310-312] used the term “patterns” in a slightly different meaning than design patterns are used in object-oriented technology. He used in his book the term patterns for rules, originally published by Tom Love [Love91], which should be applied for well designed object-oriented programs. These rules or guidelines for developing well designed object-oriented programs are, however, very important and can be recommended to be applied in most large and small software projects.

Another guide that supports well-mannered object-oriented design in C++ is “Taligent’s Guide to Design Programs” [Gold94]. It is a useful reference with design guidelines such as C++ programming conventions and programming techniques. The guide also touches portability issues and describes how to use class templates in a way that they can easily be maintained. Companies creating object-oriented application libraries for internal use can apply such guidelines and profit from the experience of professional software companies such as Taligent.

All rules, guidelines and especially design patterns can support designers to create “tame software”. “Tame software” means programs that are divided into small independent components with well-defined interfaces to each other. Each component within a software should know its boundary and should not influence other unrelated components.

2.4 Application Frameworks

A framework is a collection of cooperating classes that are put together as a reusable library. A framework is not a finished software product, rather it is a solid base to build applications. Frameworks focus mostly on a specific problem in software development. One framework may provide a base on which programmers build user interfaces, while another framework provides a basic structure for network operations. A framework for object-oriented programming can be customised for application specific purposes by subclassing the framework classes.

Most frameworks dictate a specific architecture to an application. Some frameworks even expect to be the only framework for an application. Sometimes there is a need to use two or more frameworks for a specific application. A framework can be inappropriate for solving a specific problem, if the dictated architecture is too restricting for an application. A close look into frameworks shows that they are full of patterns. These patterns are implemented as “ready to use” elements. A comparison of frameworks for similar environments (e.g. frameworks facilitating graphical user interfaces) shows that such frameworks mostly consist of similar patterns. The detailed implementation of a pattern, however, can differ from one implementation to another.

Like design patterns, frameworks are becoming more and more important. Especially large software projects use layers of frameworks. The code, written by using frameworks, is more “standardised” than independently implemented code. This is probably a result that most application code is influenced by the frameworks it uses. Using frameworks to create applications, however, also requires that the programmers must be able to trust the frameworks they use. In other words, it is extremely important that framework implementations are robust. Nothing can be worse than changing a framework when an application is about to get finished. Therefore, a careful evaluation of a framework for a specific problem can be one of the most important decisions within a software project.

The following subsections introduce some common used frameworks, beginning with the old, though still popular Smalltalk-80 environment.

2.4.1 Smalltalk Framework (MVC-Triad)

Smalltalk-80, as an early framework, became popular with the description of the language by Goldberg and Robson [Gold+83]. As described previously, Smalltalk-80 uses a class grouping that is commonly known as the “Model-View-Controller” concept or triad. This triad became the title of one of the first, commonly reused design patterns in programming history. Erich Gamma *et al.* say [Gam+95, pp. 4] “*Looking at the design pattern inside MVC should help you see what we mean by the term pattern*” and Wolfgang Pree [Pree94, pp. 71] describes MVC as “*Despite MVC’s deficiencies, Smalltalk’s MVC classes together with about 25 subclasses can be considered as the first application framework for GUI (Graphical User Interface) programming*”.

Figure 2.2 shows the components of the Model-View-Controller triad and illustrates the communication between them. The three components **model**, **view** and **controller** can be defined as follows.

A **model** is an object that holds all the data that can be displayed. A model in Smalltalk-80 has usually no references to views, except if the model has to send update messages to views. This can be the case, when a model has a buffer function for information between the memory data and the persistent data. The Figure 2.2 shows this re-

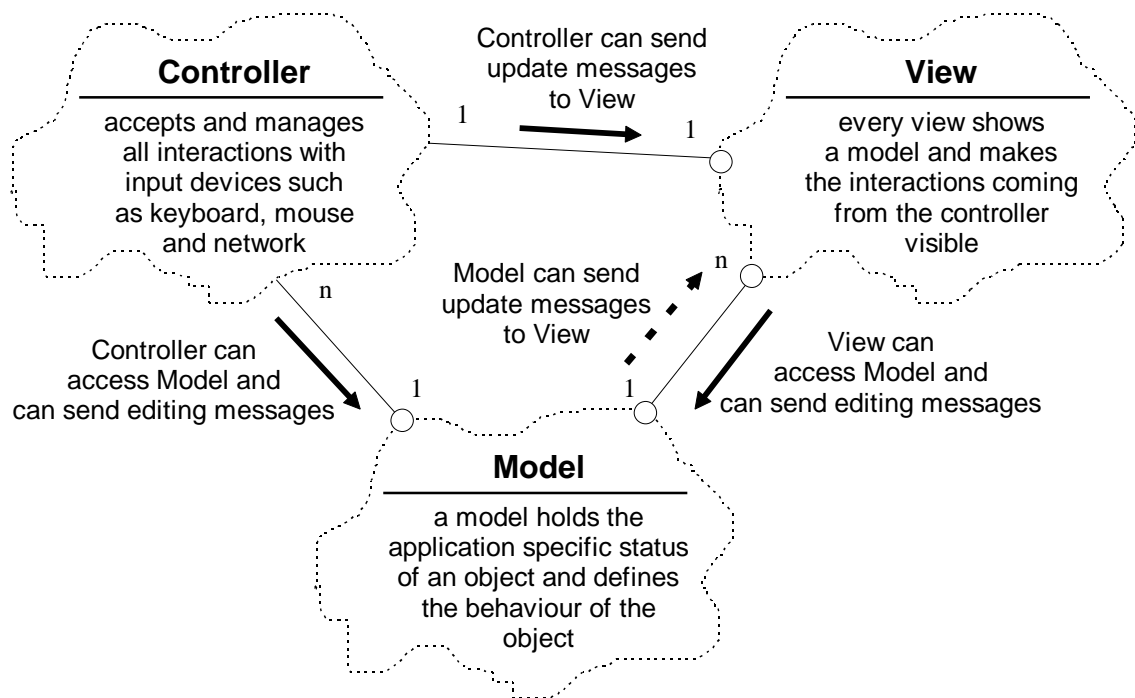


Figure 2.2: The components of the Model-View-Controller triad

relationship with a dotted arrow from model to view. There is always only one model for a specific object.

A **view** is the visual representation of a model object. A model can have many views while a view belongs exactly to one model. The class View is purely abstract. The subclasses of class view know how an object (model) has to be displayed. A number of pure virtual methods have to be redefined in every of these subclasses. A view can have a super view and zero or more subviews. The pure abstract classes View and Model predefine most interactions between a model and its views. A view has references to its model and its controller (if a controller is needed).

A **controller** provides the protocol for all user interactions. The only way a user can access a model and its views is through a controller. There is only one controller active at a time. For instance, an active controller can be seen as a field with an active cursor in it. When a user clicks with a mouse to a specific view, the controller for this view gets active. Controllers and views occur mostly in pairs, meaning that a controller must be involved if a view or model needs to be accessed by a user interaction. A controller has references to the corresponding view and its model.

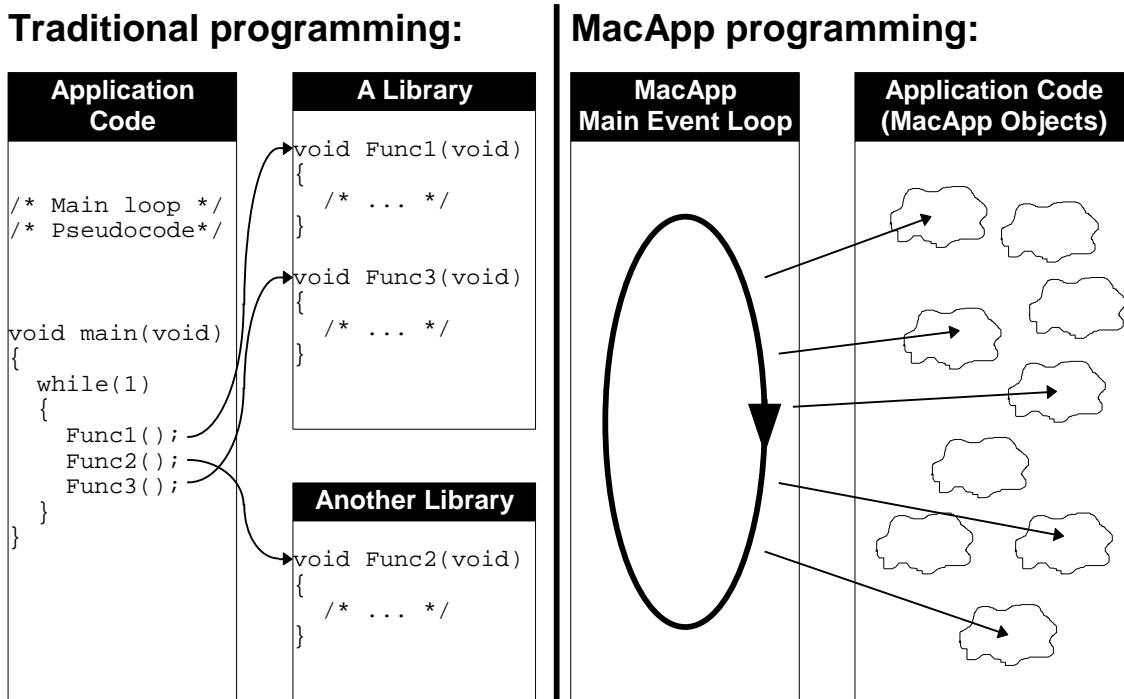


Figure 2.3: Traditional programming and MacApp programming

Beside these three main classes within the Smalltalk-80 framework, many other classes allow programmers to create window based, object-oriented application in an easy way. Some commentators claim that Smalltalk-80 is one of the best languages to learn what object-oriented programming means. This claim is probably a result of the fact, that in Smalltalk-80 everything, even a small integer number, is an object. The support for creating GUI applications has been a part of Smalltalk-80 since it was developed at Xerox Palo Alto Research Center.

2.4.2 The MacApp Framework

MacApp is a framework that simplifies writing applications for the Macintosh computer family. The framework is developed by Apple Computer Incorporated and tends to be the most popular framework for writing object-oriented Macintosh applications. The framework concentrates on writing GUI applications. Therefore most of the classes within the MacApp framework are TDocument, TView and TCommand classes. The TDocument and TView classes can be seen as the synonym to the Model and View classes of the Smalltalk-80 framework. This two classes (and their subclasses) also cover the functionality of the Controller class from the Smalltalk-80 framework.

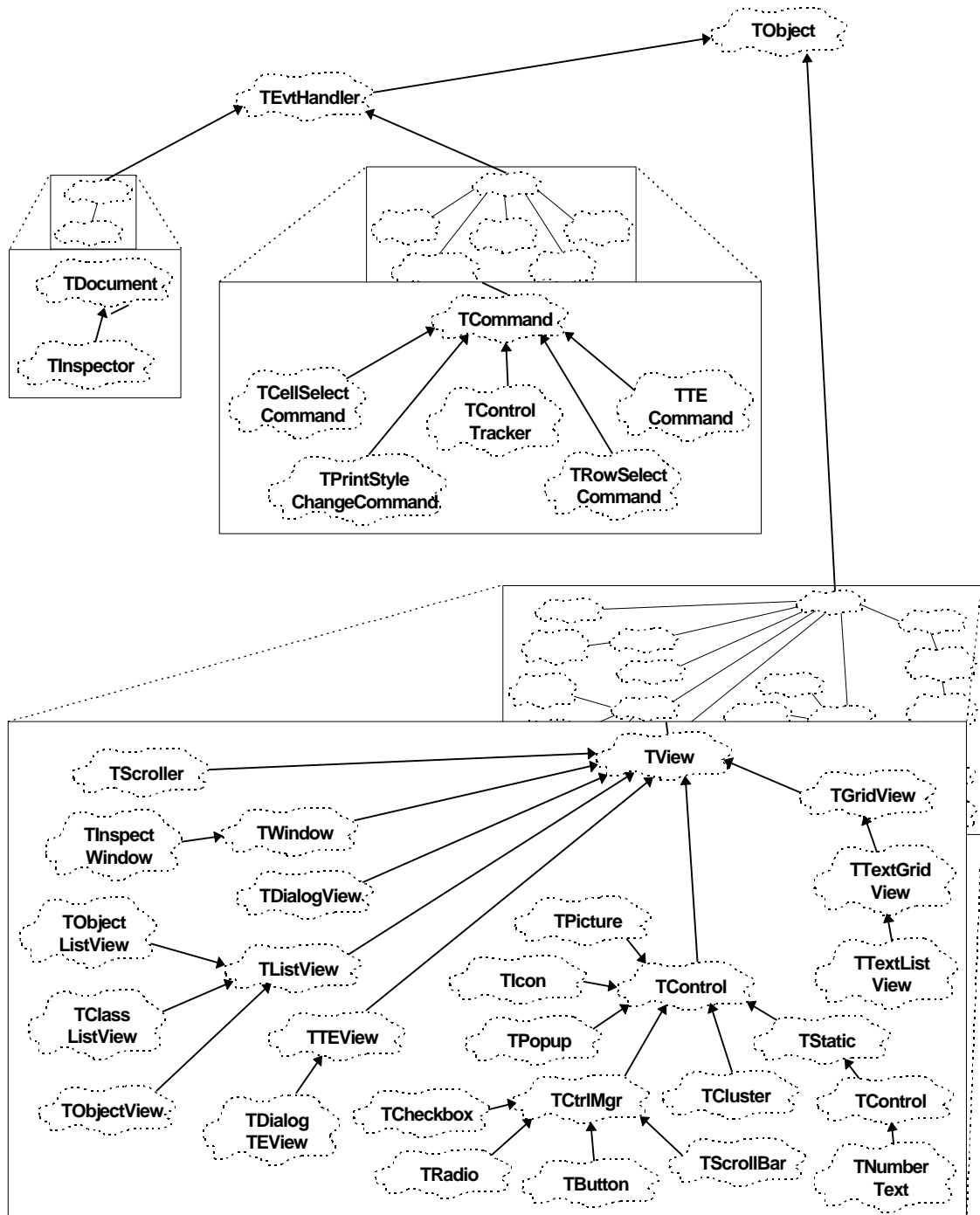


Figure 2.4: Some of the popular classes within the MacApp class library

Where is the difference between programming in a traditional way and programming with frameworks such as MacApp? Wilson *et al.* [Wils+90] describe that programming with MacApp is like using an “Upside-Down” library. Figure 2.3 shows a comparison between traditional programming and MacApp programming. Traditionally, programs used to call library functions one by one from a main program and its sub-modules. The control was almost always up to the programmer. With MacApp, on the

contrary, the control is always up to MacApp. MacApp defines when code written by the programmer is called. As a result, the structures of MacApp applications look more standardised than traditionally implemented programs. Programmers can reuse already written code more easily.

Figure 2.4 shows the organisation and the hierarchy of some of the popular MacApp classes. Most interactions between classes such as TDocument, TView and TCommand are predefined and programmers have to follow strict guidelines when these classes are used. The well-organised class hierarchy has been applied by thousands of programmers and can be seen as a good example for “reusability” in today’s software industry.

In short, MacApp can be seen as exemplary for a GUI framework. It facilitates object-oriented programming with design guidelines and simplifies GUI handling in a high degree. However, it should be considered, that this framework is proprietary for creating application for the Macintosh computer family and it is specialised for GUI applications. If a project needs to be platform independent (operating system and GUI), then more portable frameworks have to be considered.

2.4.3 Object-Studio - A Framework with Database Access and CASE

Object Studio is a framework, a CASE tool, a database access interface and a Smalltalk interpreter, all in one product. Object Studio is a product of Easel Corporation and has been developed over the past few years. The first version was called “ENFIN” and was just a Smalltalk programming language with database access classes for most of the common relational database systems such as SYBASE, ORACLE, INGRESS and INFORMIX.

Programming with Object Studio is again different to programming with frameworks such as MacApp. Object Studio tries to facilitate programmers first with the design of an application, then as an editor for GUI windows and finally on to the generation of finished Smalltalk application code. An integrated design tool allows the programmer to draw diagrams such as class diagrams, object diagrams and entity relationship dia-

grams. After an application has been designed, the programmer can extend semifinished Smalltalk classes, which are generated from the drawn diagrams.

An interesting point in this implementation of a CASE with a framework is that the system keeps drawn diagrams consistent with the code as changes and extensions are made by the programmer. Another helpful feature is that the written application can be interpreted on different operating systems such as Windows 95/NT, OS/2 and UNIX, and on different GUIs such as Windows 95/NT, Presentation Manager and OSF/Motif [Koba91].

Object Studio includes a class library with interfaces to most of the popular relational database management systems (RDBMS). These classes allow programmers to manipulate database tables on a server without dealing with SQL statements.

Even though the previously described features seem to cover most of the needs for creating GUI client server database applications, Object Studio can not be seen as a framework for real distributed programming. Rather it is more a “clever” interface to relational database management systems, operating systems and graphical user interfaces.

2.5 Distributed Programming

First of all, it might be appropriate to define what distributed programming means. Brown used in his book “UNIX Distributed Programming” [Brown94] the following definition:

“Distributed programming is the spreading of a computational task across several programs, processes or processors.”

The increasing availability of multi-processor computers, local area networks (LAN) and wide area networks (WAN) results in new requirements for application programs and their development frameworks. Houses, cities, countries and continents are connected through networks more than ever before. These networks increase the transfer rate to a degree, that an application can be spread to computers all around the world. The World Wide Web (WWW) is an example of a distributed application that connects the whole world to an information base for researchers and companies and even for every household.

Spurr *et al.* show in their book [Spur+94] some examples of distributed applications that allow users to collaborate across computer networks. Spurr *et al.* describe in detail the problematic nature of book and script corrections between publishers and authors, occurring before publications. Some approaches that allow efficient communication between publishers and authors are shown. The approaches are based on electronic mail communication between the collaborators. According to Spurr, electronic mail can be seen as a successful and commonly used CSCW application.

In this thesis, the focus within distributed programming is set on frameworks and design patterns that facilitate the creation of CSCW (Computer Support for Cooperative Work) applications.

2.5.1 Reasons for Distributed Programming

There are many reasons for applying distributed programming. The following paragraphs will show three major reasons why programmers use distributed programming. These reasons are improving performance of software through **concurrency**, **resource sharing** and **collaboration** of users by using computers and their networks.

Firstly, computer performance can be increased through **concurrency**. Concurrency in computing means that two and more processors share their jobs in a single computer. Trew and Wilson [Trew+91] say that multi-processor systems and concurrent programming is the only way to meet the market's increasing demand for high-speed, low-cost computing. There are multi-processor systems, also called supercomputers, which consist of thousands of processors. Supercomputers are used for many purposes such as virtual reality animation, weather simulation and calculation of properties of complex biological molecules. Even personal computers in households are using computer power in an increasing demand. An XT computer used to be enough powerful to satisfy word processing requirements for private use. Today a 486-DX2/66 personal computer with 4MB RAM is hardly powerful enough to run a word processing program such as MS-Word and Word Perfect.

Secondly, **resource sharing** is another reason for distributed programming. Computer components such as printers, file servers, scanners, CD-ROM drives and modems are often shared across networks. Some systems provide a high transparency, meaning

that users do not need to know where their files are and how a printout finds the way through several networks to a chosen printer. Chris Brown [Brown94, pp. 9] describes transparency as: “*In networks terms, transparency means that you can access the remote resources in just the same way as you access the local resources (ie. the ones that are attached to the machine you are sitting at)*”. There is a great deal of distributed programming necessary to provide users and system managers with real transparent computer systems.

Lastly, **collaboration** between people across networks is today a necessary means for some companies to compete with others. Selm *et al.* show in their book [Selm+96] that information exchange is one of the key characteristics of Quality Capability for companies that are implementing Total Quality Management. Many things such as arranging appointments, reviewing documents and simply chatting can be done by employees without even leaving their office. Many businesses are flattening their organisations and are lowering their centralised overhead to departments acting in different regions and locations. However, the employees within these companies still need to collaborate with each other for certain tasks. That is the point where distributed programming can allow users to collaborate by using their computers and the networks between. Programs, that enable users to collaborate efficiently are, unfortunately, very rare. Therefore, many software houses concentrate on new ways to let people collaborate across computer networks.

In short, concurrency, resource sharing and collaboration are three main reasons for distributed programming. There is an increasing demand for applications, allowing users to collaborate across networks.

2.5.2 Common Object Request Broker Architecture (CORBA)

This section will briefly describe the framework CORBA [CORB92, Rich95, Sole92] that allows object to be distributed across networks. An Object Request Broker (ORB) is a system providing exchange of requests and responses between objects across networks. The Object Management Group (OMG) selected in September 1991 a standard interface for ORB. This standard is called CORBA and has been defined by Digital Equipment

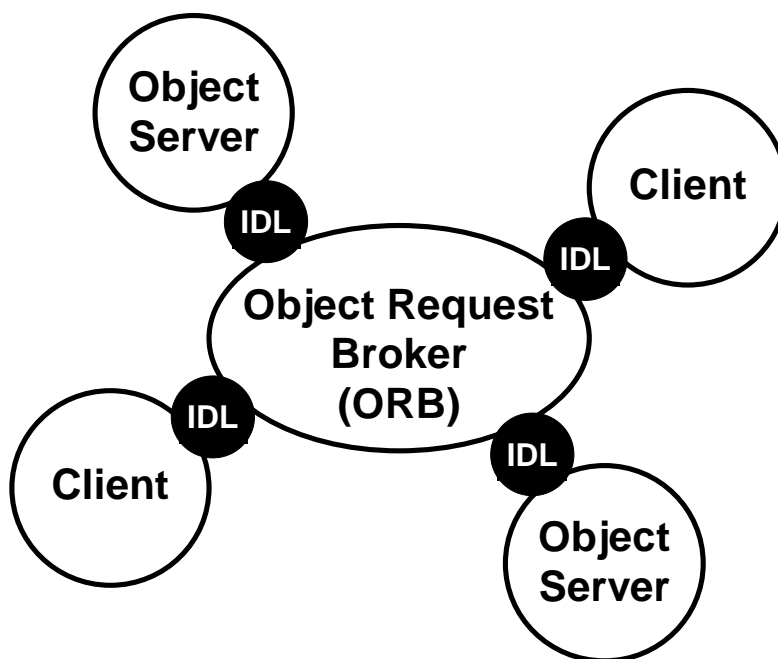


Figure 2.5: An Object Request Broker (ORB) and its environment

Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Corporation, Object Design Incorporated and SunSoft Incorporated.

When request and response messages have to be exchanged between objects at different locations, a well-defined protocol is needed. An Interface Definition Language (IDL) is part of CORBA and is probably its most important feature. Applications access local and remote object services by using this IDL. The IDL of CORBA includes three kinds of types that can be part of request and response messages. These types are basic types (e.g., signed and unsigned integer, floating point variable, boolean, string), constructed types (e.g., union, array, sequence) and an object reference type. The CORBA system provides a repository that holds definitions made for applications.

Figure 2.5 shows an ORB with two object servers and two clients connected to it. It can be seen that clients can only access an object server through the ORB. There is an IDL interface between the broker and every component connected to it. Request messages can be sent from a client through the broker to an object server. It can also be seen that clients do not need to know where the object servers are located. It is the broker that knows where messages have to be sent.

In short, CORBA is a well-defined system for object sharing across networks. The joint proposal of the companies involved in the definition of CORBA can result in a stan-

dard for tomorrow's distributed programming systems. However, it should be considered that CORBA is a full-blown system allowing distributed programming. Sometimes there is a need for simpler mechanisms that still allow objects to be shared by the various parts of a distributed application.

2.5.3 The Java language and the World Wide Web

The Java language is currently relatively unknown and it is a new object-oriented programming language facilitating distributed programming based on the World Wide Web. Sun Microsystems Incorporated is working on a large project that should allow programmers to develop advanced software for consumer electronics. The Java language is a part of this larger project and defines a C++ based object-oriented programming language with some enhancements. Sun Microsystems Incorporated defines the Java language (by using a lot of buzzwords) in the document "The Java Language: A White Paper" [Gosl+95] as follows:

"Java: A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language."

This definition sounds very promising. The following paragraphs show some important features of the Java language that are related to distributed programming.

The Java language was intended to use the C++ syntax, because it is the most common used object-oriented language. During the development of the Java language, however, it has been realised that C++ was not fully suitable for solving a number of problems. Therefore, the language was changed. Despite this, the Java language is fairly close to C++.

Distributed programming within the Java language is supported by an extensive library for inter-process communication based on TCP/IP protocols such as HTTP (HyperText Transfer Protocol) and FTP (File Transfer Protocol). Java applications can access objects across the Internet via URLs (Uniform Resource Locators) as easily as applications access objects on the local computers.

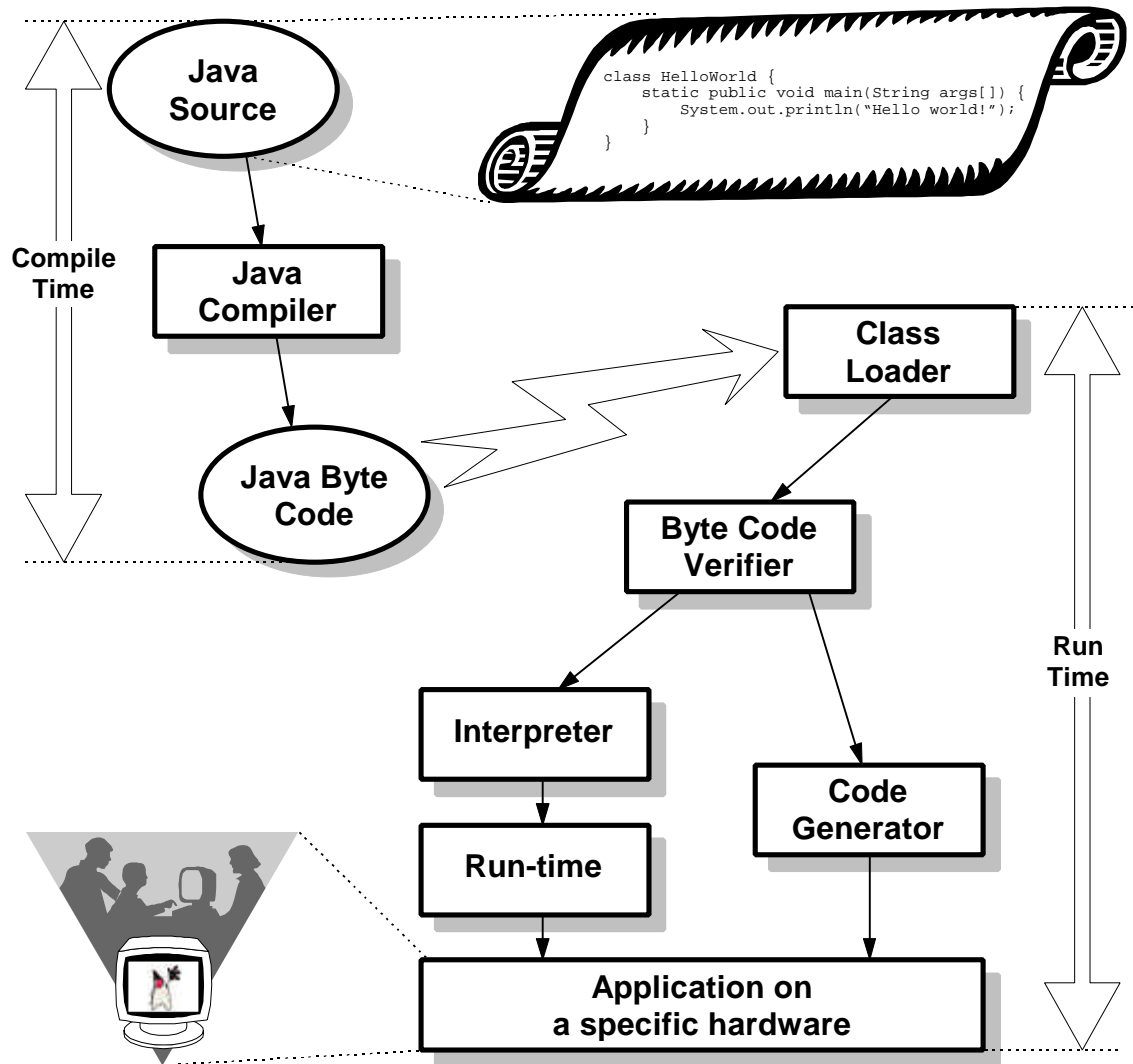


Figure 2.6: From Java language source code to the Java interpreter

A problem in daily C++ programming is faced when a program part corrupts data by writing on addresses where it should not. In the Java language, referencing of objects is done by a so-called “true array”. The “true array” is a system that allows controlled access to objects and it is able to free memory by garbage collection. In addition, casting of arbitrary integers into pointers is in the Java language not possible.

One of the most impressive features of the Java language can be found in the architecture, how Java is loading and binding application parts, that can be spreaded across the Internet. Figure 2.6 shows an adapted diagram taken from the document “The Java Language Environment: A White Paper” [Gosl+95, pp. 42]. It can be seen that a Java source code has to be compiled into a Java byte code (compile time). This code is plat-

form independent and is the base file that will be copied through networks (see flash in Figure 2.6) to clients.

Then, at run time, a class loader acts as a sort of gatekeeper and is responsible for receiving a Java byte code and it verifies the received code (e.g., code consistency, type checking). After the code has been verified, it is ensured that either the code has been rejected or it is in the following “clean” state [Gosl+95, pp. 43]:

- There are no operand stack overflows or underflows
- The types of the parameters of all byte code instructions are known to always be correct
- No illegal data conversion are done, like converting integers to pointers
- Object fields accesses are known to be legal -- private, public or protected

Then, the code will be interpreted by a Java run-time interpreter or it will be converted into fast, platform specific code machine code. The definition which program part should be interpreted and which one should be converted into machine code is defined in the source code. This technology makes it possible that even time critical application parts such as animations and multi-media presentations can be handled in an adequate performance. According to James Gosling and Henry McGilton [Gosl+95, pp. 48], the performance of a byte code converted to machine code is roughly the same as native C or C++.

Even though just a few features of the Java language could be explained in the previous paragraphs, it can be understood that the technology of distributed programming with the Java language is innovative and outstanding. After publication of the specifications of the Java language and its corresponding World Wide Web browser HotJava, most of the leading software houses showed an extreme interest. The following news is probably only a drop in the bucket:

- One of the first responses of the software market came from Netscape Communications Corporation. At May 23rd 1995, Netscape Communications Corporation announced [NSPress95] the press release “*Netscape to License Sun's Java Programming Language.*”

- IBM announced [IBMPress95] at December 6th 1995 the press release *"IBM today announced that it has licensed Sun Microsystems, Inc.'s Java programming language and intends to use it to enhance the way customers view and interact with content on the Internet World Wide Web."*
- Microsoft Corporation announced [Green95] at December 7th 1995 the press release *"MOUNTAIN VIEW, Calif. - Sun Microsystems Inc. said Thursday it signed a letter of intent to license the Java Internet programming language to Microsoft Corp."*. Heather Green [Green95], a reporter working for Bloomberg Business News, stated the response of the Wall Street to Microsoft news with *"Sun shares were up 3 3/8 at 93 1/8. Microsoft shares are down 1 1/8 at 89 1/2"*.

Besides Netscape Communications Corporation, IBM and Microsoft Corporation, many other important software houses such as Oracle and Borland licence the Java language for various purposes. All these reactions show the importance of distributed programming in today's software industry. It can also be said that design patterns for distributed programming should suit to programming languages such as the Java language.

2.6 Summary

This chapter has surveyed the importance of design patterns in software technology and has explained some frameworks supporting reusability of software components, and introduced the term distributed programming and its nearly infinitely increasing demand.

Design patterns are becoming more common and describe well-designed program parts in a reusable and mostly language independent way. A definition of design patterns has been described and it will be applied in this thesis. Currently, there are many conferences discussing design patterns in all fields and these patterns will give a foundation for tomorrow's software industry.

Application frameworks have implemented many patterns of reusable software components and represent a well example for software reusability. The problem of most of these available frameworks is that they are usually proprietary and can not easily be connected with other frameworks. It can also be realised that frameworks facilitating

distributed programming are relatively rare and the demand for such frameworks is becoming increasingly high.

Network systems such as the Internet and the World Wide Web are becoming very popular and form an information base for almost everything. There is an increasing demand for application programs allowing users to collaborate via networks. Design patterns for distributed programming will contribute to the reusability of program components facilitating the creation of applications for cooperative work. Companies join to organisations such as the Object Management Group to develop platform independent and reusable systems supporting distributed programming (e.g., CORBA, Java). There has not been an established standard for distributed programming yet. Therefore, it is important to describe components facilitating distributed programming in a language independent way by using the strategy of design patterns.

Chapter Three

3. A Pattern for Sharing Objects - “Simple Shared Object”

3.1 Overview

This chapter starts with a brief description of an experimental class library for distributed programming on which this research is based. This experimental class library is part of previous research by Neil Gray [Gray95] and facilitates communication between objects in different address spaces. The library also offers some GUI classes to create primitive X-Window components such as action buttons and radio clusters.

After the implementation of some simple applications using the experimental class library, it was realised that additional classes could facilitate object synchronisation between applications running on different machines.

The second half of this chapter explains a design pattern called “Simple Shared Object” that uses proxies to synchronise objects at different locations. The pattern has been implemented in C++ on a Unix system. A server controls access to primary copies of shared objects and it forwards update messages to clients holding mirrored copies of shared objects they use.

3.2 An Experimental Class Library - Distributed Object Communication

This experimental class library has been implemented to facilitate the construction of applications that cooperate across networks. Such application programs are meant to run on different clients connected via communication “channels”. Objects within ordinary single-user application programs are generally manipulated by a user sitting in front of a terminal. Objects within applications using the experimental class library are manipulated

through exchange of messages between event sources. An event source can simply be a mouse click initiated by the user sitting in front of the computer on that the application is running. Another event source can be a key, pressed on a keyboard 10'000 kilometres away from the actual object source.

A particular program using the experimental class library consists of various objects that work together. The three primary object types are an **I/O-Dispatcher** object, **Channel** objects and **Coms** objects.

3.2.1 The I/O-Dispatcher Object

In distributed applications, inputs arrive from many sources, not just from the local user. Many requests for acting arrive via network connections and these may have to be handled much as if they were generated by the local user. Inputs should be handled “immediately”. Input handling should involve only limited processing (access or update of data, or change to graphics displays). Once an input is handled, the program should be ready to handle the next input from whatever source. Normally it is impractical to poll for input.

Such requirements on input handling imposes constraints on the structure of a program. Either separate “threads” must run for each input or the program must have some mechanism for “waiting” for the next input from multiple active devices. When an input does arrive, it must be dispatched to the appropriate handling object.

An I/O-Dispatcher is the actual handler of all incoming (and outgoing) events. It can be seen as the heart of an application. The I/O-Dispatcher is able to register and de-register objects that handle input events. After some “handlers” have been registered, it is able to “run”, meaning it calls registered object one by one to handle eventually arrived input events.

3.2.2 Channel Objects

Channels manage the input events or messages associated with particular “file-descriptors” (communication links). A class called “Channel” within the experimental class library is a purely abstract class and defines an interface that lets channels (with as-

signed “file-descriptors”) handle input events. The library also offers a few specialised channel classes that are able to handle predefined type of file-descriptors.

These specialisations are an `XChannel` class, a `ConsoleChannel` class, a `ReceptionistChannel` class and a `NetChannel` class. These specialisations are offering predefined functionality as follows:

- **XChannel** objects are able to handle `XTerminal` events.
- **ConsoleChannel** objects handle input coming from a controlling character terminal (e.g. a VT100 or an `XTerm` window).
- **ReceptionistChannel** class is usually used on the server side of a particular program. A concrete instance would listen at a predefined port for client requests.
- **NetChannel** objects are handling network connections and are therefore responsible for inter-process communications between objects at different locations.

3.2.3 Coms Objects

Object communication between programs running in different address spaces is an important requirement for distributed applications. The experimental class library offers a partially abstract base class “Coms” that defines the capabilities of object communication across address spaces. For instance, within a collaborative multi-user editor, Coms objects can be data records that are able to be manipulated by the collaborators.

Application specific, specialised Coms classes have to implement some abstract member functions to support communication tasks required for information exchange. Therefore, every specialised Coms class implements methods for handling requests, handling messages and handling replies. There are also a few other methods that have to be implemented (e.g. class identification `getClassName()`).

3.2.4 The System (Collaborations)

Distributed applications using the experimental class library may look different from one implementation to another. However, a common pattern will probably always influence application programs. Figure 3.1 shows the various objects and the connections between these objects in a particular program.

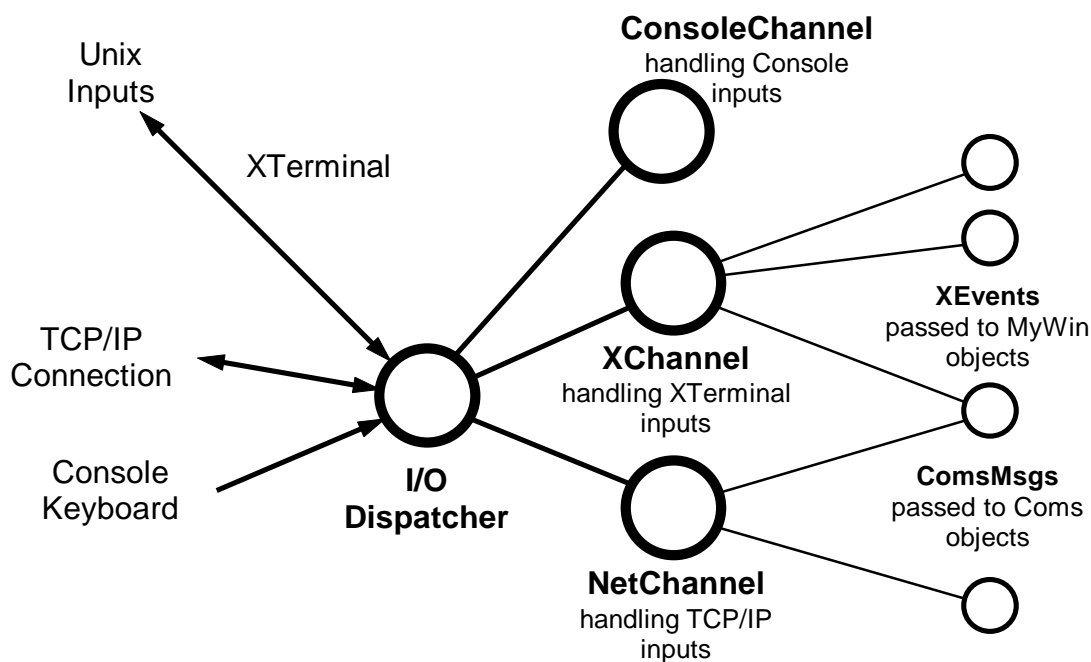


Figure 3.1: A particular application using the experimental class library
 Source: Unpublished “experimental class library” description

The Figure 3.1 shows that all input events are received by an I/O-Dispatcher. It is this I/O-Dispatcher that knows to which object (Channel) an event has to be passed to. The figure also shows three specialised Channel objects (ConsoleChannel, XChannel and NetChannel) that can stand either alone (like a ConsoleChannel object) or that are connected to Coms objects (XChannel and NetChannel).

3.2.5 A Typical Object Composition in an Example

Derived from the general view of object collaboration in Figure 3.1, an example might help to understand in what way a client and a server program could communicate through the experimental class library.

Suppose, a client object has to communicate with a server object. An easy solution would be to connect these two objects direct through a TCP/IP port that would allow them to communicate. Using the experimental class library involves some other objects with predefined responsibilities.

Figure 3.2 shows a typical object composition within a client/server application program using the experimental class library. An I/O-Dispatcher is listening on a server for clients that also run I/O-Dispatcher’s connected to the server. The object composi-

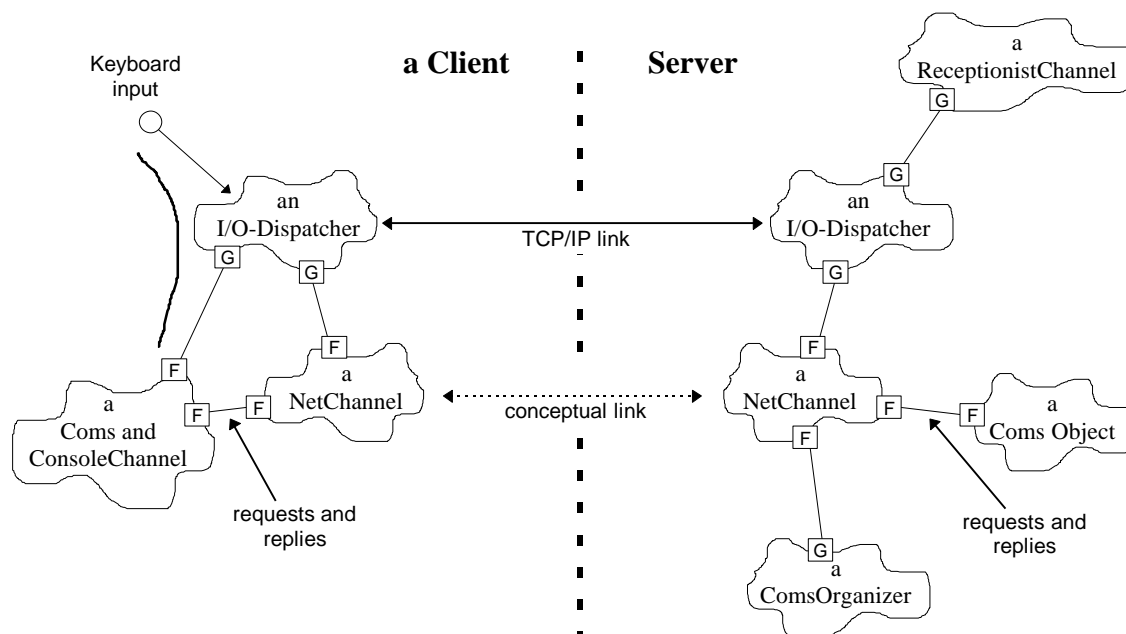


Figure 3.2: A typical object composition using the experimental class library
Source: Unpublished “experimental class library” description

tion looks mostly symmetric on client and server side, except that the server includes a ReceptionistChannel object and a ComsOrganizer object. A client Coms object may also be a subclass of ConsoleChannel, meaning it can handle keyboard input messages as well as NetChannel messages. Mostly, a NetChannel object on server side exchanges messages with a NetChannel object on a client. These messages are transferred through the I/O-Dispatcher objects on both sides. As a result, a conceptual link can be drawn between the NetChannel objects on client and server.

This brief description of the experimental class library gives an idea how object can communicate across address spaces by I/O-Dispatcher objects on client and server side. Neil Gray’s description [Gray95] explains deeply the features of this architecture and shows further a concrete example of a CSCW application.

3.3 Extension for Sharing Objects

During implementation of some simple application using the experimental class library, it was realised that additional classes could support transparent object sharing across address spaces. CORBA is a system for sharing objects across address spaces and is commonly known to support this issue. However, CORBA is not appropriate for small, dis-

tributed applications, because it is simply a too big system for application with a small amount of shared objects.

Distributed CSCW applications are usually expected to react rapidly to user interactions. Therefore, it could be advantageous to have primary copies of objects on a server and mirrored copies in the address space of client applications. However, in such ‘mirrored’ systems it is necessary to synchronise the shared objects across the address spaces.

The following pattern “Simple Shared Object” shows a system in which objects can be shared in a simple way.

3.4 “Simple Shared Object” - An Architectural Pattern for Simple Object Sharing

This pattern has been published [Ott96] by SIGS Publications in the Journal “Object Currents”.

3.4.1 Intent

The Simple Shared Object pattern allows object sharing across networks. The primary instance of a shared object will exist in the address space of a server program; client programs have “mirrored” copies of shared objects that they use. All accesses to shared objects work via proxy objects and the processes of creating primary and proxy objects are automated. The pattern is specifically designed for user interactive, low traffic, distributed applications where clients work closely together.

3.4.2 Motivation

There is an increasing demand for collaborative applications that allow several users to work together across networks. Proposed “Computer Support for Co-operative Work” (CSCW) applications include various forms of document editing and review, conferencing, scheduling and so forth [Spur+94]. A variety of “distributed frameworks” are under development. Such frameworks facilitate the creation of collaborative distributed applications. These “distributed frameworks” include CORBA (Common Object Request Broker Architecture) and OLE (Object Linking and Embedding).

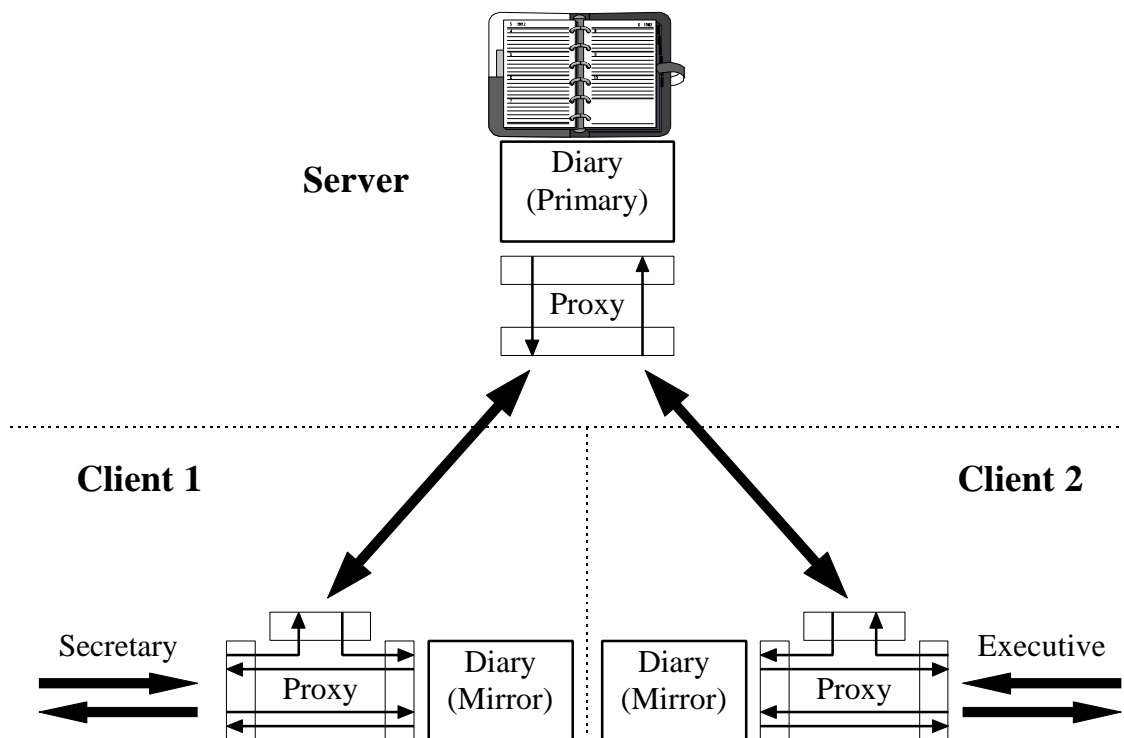


Figure 3.3: A distributed application with shared objects

Although not yet fully mature, such frameworks will bring substantial benefits to those implementing distributed applications. However, there is sometimes a need for simpler mechanisms that still allow objects to be shared by the various parts of a distributed application. CORBA style solutions may incur overheads, or may simply be unavailable to the developers. The Simple Shared Object pattern is specially designed for user interactive, low traffic CSCW applications.

Figure 3.3 shows an example of a distributed application where a secretary and an executive are using a common diary. The secretary may receive phone calls and arranges appointments for the executive. Then, the executive can confirm, shift and cancel appointments to suit personal requirements. Both client programs should allow their user to browse the diary without delays caused by remote shared objects. Therefore, each client has synchronised “mirror” copies of the primary diary in its address space allowing fast read access to the diary. Modifications on the shared diary are done via the server. Clients do not need to check with the server for updates because they are automatically notified by the server whenever changes are made to any primary object that they mirror.

The pattern supports these requirements while focusing on simplicity.

3.4.2.1 Needs for Shared Objects

The pattern is intended to satisfy the following common needs for shared objects:

- **Need for shared data:** A need for shared data is generally met by “database” servers that interface to Relational Database Management Systems (RDBMS). However, many distributed applications need to share data in ways that are incompatible with constant interconversion to/from flat database tables.
- **Need for synchronisation:** Distributed application programs need shared objects that are synchronised across different address spaces. This is actually one of the reasons why programmers use shared objects instead of databases. Shared objects are not always passive, and active objects require smart synchronisation mechanisms.

3.4.2.2 Behaviour of Shared Objects

The pattern is intended to give shared objects the following common behaviour:

- **Notification:** All copies of shared objects should be consistent; consequently, mirror copies must be notified whenever there are changes to the primary. The notification acts as a “software interrupt” that gets handled using an event handler mechanism.

3.4.3 Applicability

Use the Simple Shared Object pattern when

- a system has to share objects across different address spaces.
- the server part and the client parts are working closely together within a distributed system.
- server or clients have to be notified whenever shared objects are created, modified or deleted.
- no other framework for distributed programming such as CORBA or OLE is available, or they are inappropriate for the specific problem.
- the objects which should be shared support serialisation and de-serialisation of their state.

- the programmer wants to be aware of how and when shared objects are synchronised.
- the communications traffic needed to synchronise the shared objects is low.

3.4.4 Structure

The system uses proxy objects to control access to objects that must be shared. Client proxies belong to classes that inherit from class *Client Proxy* and server proxies belong to classes that inherit from class *Server Proxy* (see Figure 3.4). In addition, the client proxy classes must duplicate the public interfaces of the classes whose instances have to be shared. Thus, the example diary proxy objects will combine the synchronisation behaviour of client and server proxy objects with the application specific behaviour of a diary. Relations among the classes are shown, using Booch [Booch94] notation, in Figure 3.4. “Concrete Proxies” inherit either from class *Client Proxy* or from class *Server Proxy* and each proxy holds a reference to an instance of a concrete object (e.g. the shared diary of Figure 3.3).

- **“Shared Object Registration” class:** The server process needs to maintain a list of all shared objects. This is the responsibility of an object that is an instance of the (Singleton [Gam+95]) class *Shared Object Registration*.
- **“Client Communicator” class:** Each instance of a specialised *Client Proxy* class creates a *Client Communicator* object and holds a reference to it. The *Client Communicator* object has a pointer back to its corresponding *Client Proxy* object. These *Client Communicator* objects are responsible for all the low-level details involved with communications to the server.
- **“Server Communicator” class:** Each *Client Communicator* object attaching to the server automatically creates a *Server Communicator* object. Like *Client Communicator* objects, these instances of class *Server Communicator* are responsible for all low-level details of communications with clients.

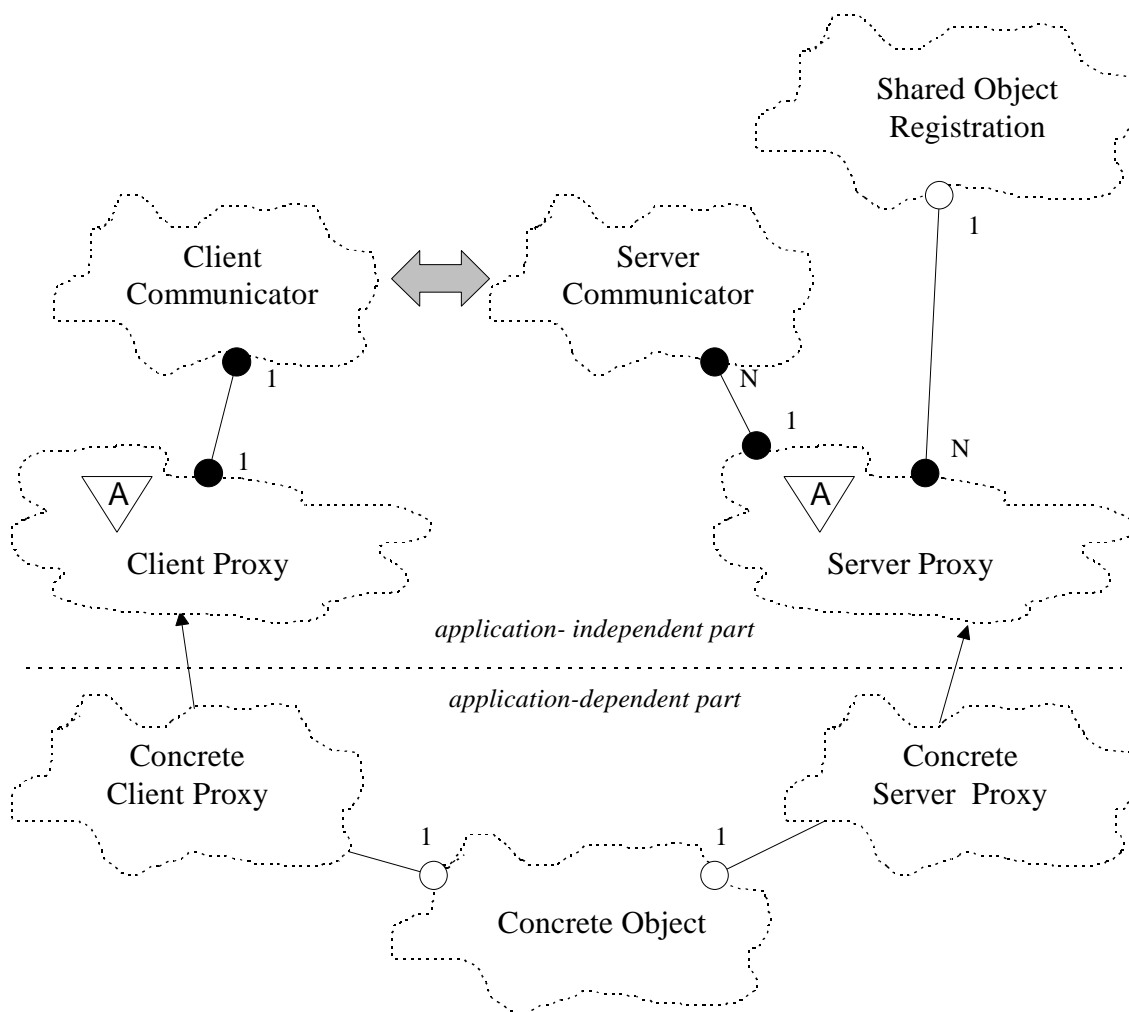


Figure 3.4: Class diagram of the Simple Shared Object pattern

Each *Server Communicator* object holds a reference to a specialised *Server Proxy* object; access to the actual primary copy of the shared object takes place via this *Server Proxy* as an intermediary.

- **“Server Proxy” and “Client Proxy” classes:** The classes *Server Proxy* and *Client Proxy* are partially implemented abstract classes that act as the base classes for client or server proxies. Their members specify the data and mechanisms required for synchronisation.

Server Proxy objects will each have a pointer to an object that represents the actual shared data; construction (and destruction) of the shared data is the responsibility of a specialised subclass (see below). Class *Server Proxy* provides the mechanisms needed for automatic registration and deregistration with the *Shared Object Registration*.

The *Server Proxy* object maintains a list of *Server Communicator* objects that are using it to access a specific shared object. This list implicitly identifies the client processes that have mirror copies of that shared object. When a *Server Proxy* is asked to modify its primary copy of a shared object, it “knows” which clients have to be informed and can use the *Server Communicator* objects to forward change messages to these clients.

The class *Client Proxy* has only to handle a single *Client Communicator* object. Therefore, each specialised *Client Proxy* object holds only one reference to a communicator object. There is no need for *Shared Object Registration* on clients.

Every proxy object (on both clients and server) holds a “Shared Object Identifier”. This identifier is common across the system and is also unique. It is these identifier numbers that are used to maintain consistency across the system. Generally the server allocates these identifiers automatically. An application may predefine some numbers corresponding to “well-known” objects.

- **“Concrete Server Proxy” and “Concrete Client Proxy” classes:** Specialised *Concrete Server Proxy* and *Concrete Client Proxy* classes have to be declared for each class of sharable objects. These classes use inheritance from class *Server Proxy* or class *Client Proxy*.

A proxy object (client and server) has a reference to an object existing in its own address space. So, in the example for Figure 3.3, there are three diary objects (concrete objects), one “Server Diary Proxy” object and two “Client Diary Proxy” objects. The diary proxy on the server holds the address for the diary object in server’s memory. The client proxies hold the addresses of their own local copies.

Every *Concrete Server Proxy* class defines two main member functions that are specialised for the particular class of concrete object with which they work. One member function serialises the specific concrete object; this is used when it is necessary to send the status of the specific concrete object to a client. Another member function is responsible for invoking modification functions of the specific concrete object in accord with update commands received from clients.

A *Concrete Client Proxy* class normally duplicates the public interface of the class for which it is a proxy. It is possible to define that interface in a pure abstract class which helps maintain consistency between proxies and concrete objects (see Proxy pattern in [Gam+95]).

The client proxy's role is to control access to the shared data object. The application seeking to use the object actually accesses the proxy. If a request does not modify the object (e.g. it is invoking a constant function), the proxy simply forwards the request to the local object. Requests that modify objects are implemented using the capability inherited from *Client Proxy*.

Modification of a shared object (e.g. secretary enters an appointment in the diary) is initially handled through the exchange of messages between a proxy object and the master concrete object (diary) on the server. Once the modification has been done to the master concrete object, updates are sent to the mirrored copies on all clients. This exchange of messages is handled by the client and server communicator objects.

- **“Concrete Object” class:** This can be any class whose instances are to be shared and is purely application specific. There is no code in this class showing that instances of this class are shared objects. However, an instance of class *Concrete Object* must be able to serialise and de-serialise itself to synchronise copies across a system.

3.4.5 Participants

Figure 3.5 provides an illustration of the participants (by using Booch [Booch94] notation) in this pattern, using once again the diary example from Figure 3.3. The participants are:

- **Concrete Object Primary (Server):** The concrete object primary represents the primary copy of a shared data object. All modifications are made first to this object, and then they are propagated to the copies that exist in the client processes.
- **Concrete Server Proxy:** A newly created proxy object registers itself with the *Shared Object Registration*. A server proxy creates a concrete object primary during the construction procedure. There is only one server proxy for each concrete object primary.

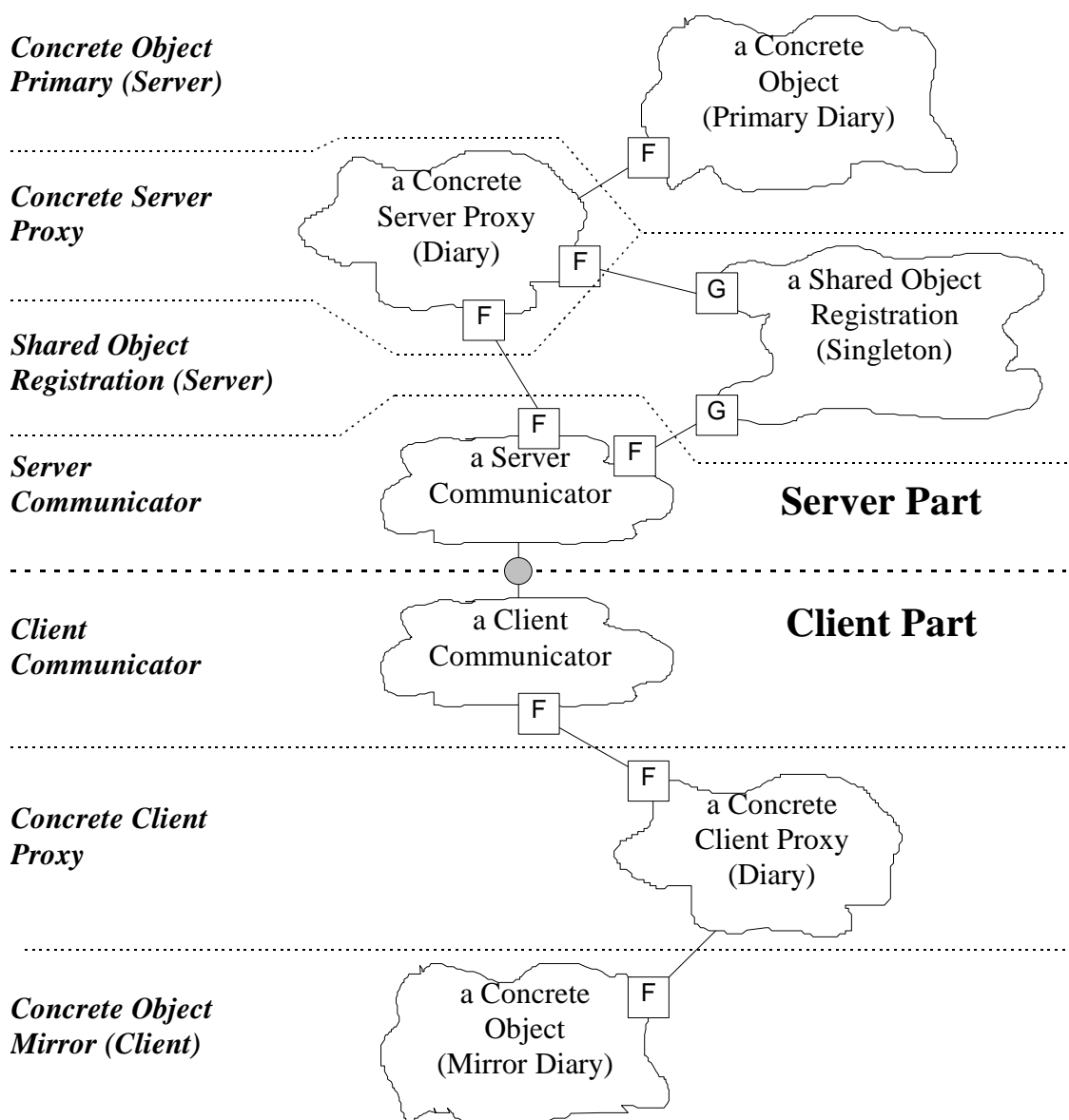


Figure 3.5: Illustration of the Simple Shared Object pattern by an object diagram

When a server proxy is deleted, it deregisters itself from the *Shared Object Registration*. It also deletes the concrete object primary for which it is acting as a proxy. A *Server Proxy* and its associated concrete object only get deleted when there are no remaining *Server Communicator* objects associated with it. Such objects can be made persistent in which case they remain in existence in case a subsequent reconnection is made.

On the server, proxy objects are created in response to requests from objects in a client process. The server is also able to create proxy objects before they are requested by clients. Thus, general shared data objects can be available independent of clients connecting and disconnecting to a server.

- **Shared Object Registration (Server):** The *Shared Object Registration* maintains a structure, indexed by object identifiers, that contains links to all server proxies. The *Shared Object Registration* is also responsible for generation of the identification numbers used for system wide identification of concrete objects.

The *Shared Object Registration* is the only object which knows all shared objects. It is called whenever a client requests a mirrored copy of a concrete object. If the requested concrete object does not yet exist, a new server proxy and the new concrete object are created. The creation process involves interactions between the *Shared Object Registration* and automatically created *Server Communicator* objects (see below). If the shared object already exists, the *Shared Object Registration* puts the *Server Communicator* in contact with the existing *Server Proxy*.

- **Server Communicator and Client Communicator:** Actual communications between server and clients are handled using additional objects that package the low-level applications programmer interfaces to systems such as TCP/IP. The lowest level is handled using the Reactor (I/O Dispatcher) model [Schm95, Schm+94, Wein+88] with additional “wrappers” for Event Handler classes. The additional classes [Gray95] augment Schmidt's Event Handlers with additional behaviours such as those needed to create objects in another address space, and to multiplex a single communication link to serve the needs of many pairs of communicating objects.

Communicator objects occur always in pairs. Such a pair is built out of a *Server Communicator* object and a *Client Communicator* object. All communications between clients and server are handled by these pairs. There can be more than one *Server Communicator* object for a *Server Proxy* object while there is only one *Client Communicator* object for a *Client Proxy* object.

- **Concrete Client Proxy:** Client proxy objects initiate the creation of the corresponding proxies on the server and the communicator pairs between.

As described in section 3.4.4 (Structure), each *Concrete Client Proxy* class must duplicate the public interface and provide a modified implementation of all methods of the corresponding concrete class. All methods changing the concrete objects (non constant member functions in C++) involve interactions with the server. Details identifying

the modifying member function, and its arguments, are serialised and packaged in a message that is forwarded to the corresponding proxy on the server. The server proxy invokes the update functions of the primary copy of the shared structure. Once the primary object has been updated, the server proxy organises transmissions of updates via communicator pairs to all mirror objects. The proxy on the server that modified the primary object provides the data that must be used when notifying all clients.

The messages transmitted to client processes appear as input events. These get forwarded via various intermediaries to the appropriate proxy objects in client processes. In a few special cases (e.g. creation of a new mirror), the message defines the complete state of the new concrete object. In most cases, a message will identify a particular modifier function that must be called and provides the arguments needed for that call.

- **Concrete Object Mirror (Client):** The concrete object mirrors are exact copies of concrete object primaries. Clients only mirror those sharable objects that they need to access. Mirror objects are only updated in response to messages from the server.

3.4.6 Collaborations

The collaborations between participants of the Simple Shared Object pattern can be seen by concentrating on three main aspects. These are: object creation, object modification, and object deletion.

3.4.6.1 Shared Object Creation

Figure 3.6 shows the collaborations needed to create shared objects. It should be realised that even though a client proxy is going to be created, the concrete object and the server proxy object could already exist at the server. The figure shows this with a dotted bar at the top of the concrete object and the server proxy on the server side.

- **Creating the Client's components:** The process begins with the creation of an instance of a specialised *Concrete Client Proxy* class (e.g. `new SharedDiary(objectId)`) by an application on the client side. The constructor needs an object identifier number as an argument. This may be the known identifier for an existing shared object or 0 for a new object. (Existing objects may be “well-known” with fixed identifier numbers. In addition, many applications allow clients to view lists of existing

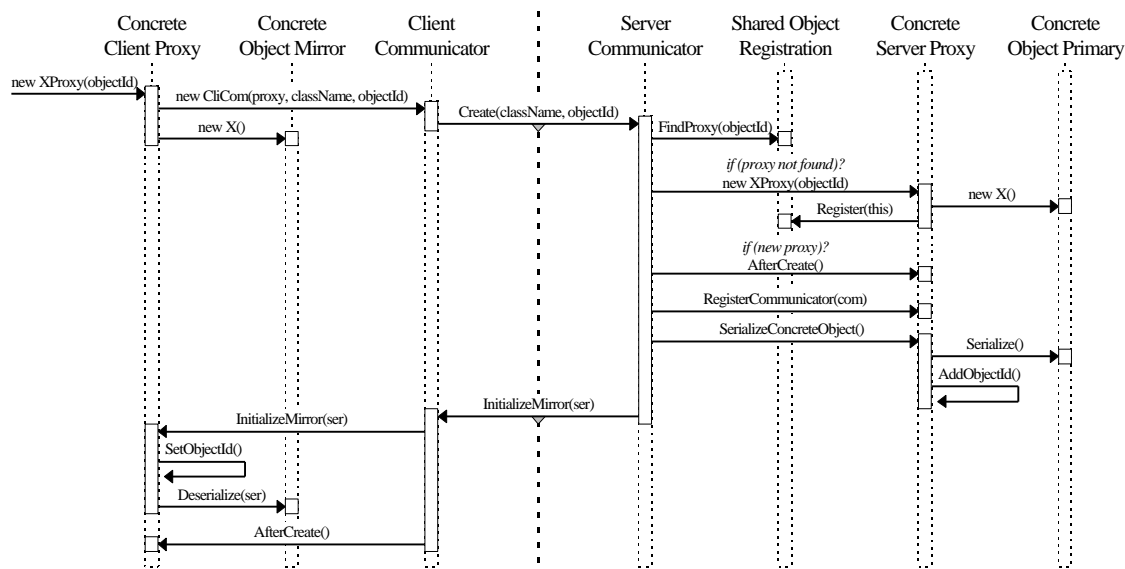


Figure 3.6: Collaboration between participants in the creation process

shared objects and to select objects from these lists; the selection process provides the identifier of the existing object.)

The process of constructing the client's object involves a call to the constructor for the base *Client Proxy* class. This constructor creates a *Client Communicator* object to handle low-level communications and forwards, via this *Client Communicator*, a request to the server asking for access to the shared object. The object identifier is included in this request.

Next, the actual concrete object is created (in the example, this would be an instance of class *Diary*).

The objects created in the client's address space are not yet ready for use. The concrete object (the *Diary*) is in some default empty state. The *Concrete Client Proxy* may not yet know the correct “universal unique identifier” for the shared object. Applications can check the status of objects and “disable” any that are not yet ready for use; even if shown in a display to the user, these objects can not be manipulated.

Final initialisation steps on the client are completed when a reply is received from the server. In the meantime, control returns to the application.

- **Handling Create Request (Server):** A “creation request” from a client results in the server process creating a *Server Communicator* object. As noted briefly above, this in-

teracts with the *Shared Object Registration* to either locate a *Concrete Server Proxy* for an existing object or create both proxy and actual object.

The *Server Communicator* asks the *Shared Object Registration* to return a pointer to the proxy for the object with the given object identifier. This request will either return a pointer to the proxy for an existing object or NULL if the object does not yet exist.

If a NULL was returned, the *Server Communicator* creates an instance of the appropriate specialised *Concrete Server Proxy* (e.g. an instance of *ServerDiary*); this step also creates the actual primary copy of the shared object (the *Diary*). The constructor for *Server Proxy* performs registration with the *Shared Object Registration*. The *Shared Object Registration* will report the actual object identifier assigned if the `objectId` specified was 0. After creating the *Concrete Server Proxy*, the *Server Communicator* will ask it to perform its `AfterCreate()` member function; this is simply a hook for any special class specific behaviours (e.g. loading the contents of a persistent object from an identifiable file).

Once the *Server Communicator* has obtained a pointer to the proxy with which it is to work, it asks for a serialised copy of the concrete object primary (`SerializeConcreteObject()`). The server proxy returns a serialised form as a temporary character buffer that also holds a serialised version of server proxy's object identifier. The communicator then sends the contents of this buffer to the client (`InitializeMirror()`). With this action the server proxy finishes its creation task.

- **Finishing the Creation process on the client:** When the client communicator receives the serialised version of the concrete object, it is able to complete the shared object creation process. The client communicator forwards the received information to the (already existing) client proxy (`InitializeMirror()`).

The client proxy sets its object identifier (`SetObjectId()`) and initialises its concrete object mirror with the received data. Then, the control returns to the client communicator which finishes the creation task with a call to the function `AfterCreate()` of client's proxy.

3.4.6.2 Shared Object Modification

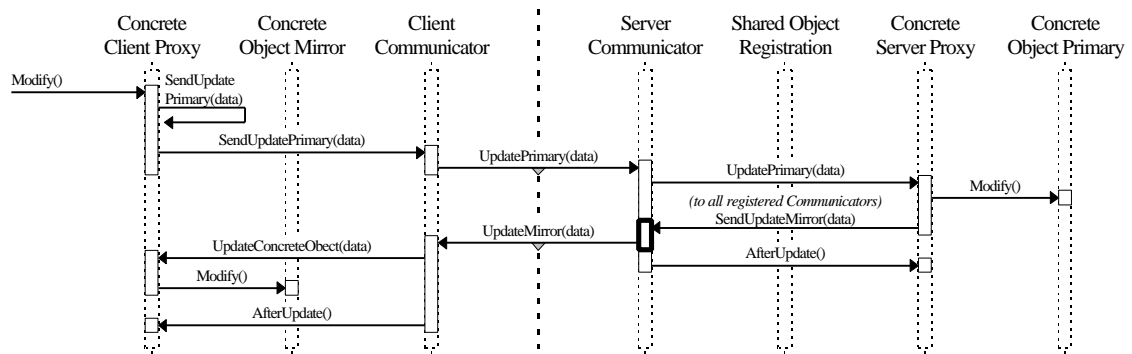


Figure 3.7: Collaboration between participants in the modification process

Figure 3.7 shows the collaborations needed to modify shared objects.

- Sending Modify Message to Server (Client):** The modification process begins with a call of a non constant member function of an existing *Concrete Object Proxy* by an application on the client side. The function `Modify()` shown in Figure 3.7 typifies such a non constant member function. The application specific implementation of the “modify” function has to serialise the modification to a temporary character buffer. The contents of this buffer are the “modify” parameters and a token that identifies the called function (usually a different constant integer for each implemented “modify” function). Then the proxy calls the `SendUpdatePrimary()` function which is implemented in the partially abstract class *Client Proxy*.

The `SendUpdatePrimary()` function forwards the serialised modification to its *Client Communicator* object by calling communicator’s `SendUpdatePrimary()` function. The communicator then sends the modification as a message to the server. After the message has been sent to the server, the control returns to the application. The modification procedures are completed later when the server transmits the modification to all clients.

- Handling Modify Message (Server):** The arrival of a “modify” message from a client results in a communicator on the server side calling an `UpdatePrimary()` function of server’s proxy.

This proxy de-serialises the received modification parameters and, by calling the function specified with the other parameters, modifies the primary copy of the concrete object. After the concrete object has been modified, the server proxy calls the

`SendUpdateMirror()` function of every server communicator in its communicator list. These communicators send the modification as a message to all their corresponding client communicators.

The Server Communicator object finishes the server's modification task with a call to the function `AfterUpdate()` of the specific server proxy. This function deals with any application specific "after update" mechanisms needed in a specific subclass.

- **Receiving Modify Message from Server (Clients):** When a client communicator receives a "modify" message from the server, it forwards the received modification to its client proxy (`UpdateConcreteObject()`). This proxy de-serialises the received modification and modifies the mirrored copy of the concrete object. After the concrete object has been modified, the client communicator calls the notification function `AfterUpdate()` of client's proxy. The proxy can deal in this notification function with any application specific "after update" mechanisms needed in a specific subclass. With this action the client finishes its modification procedure.

3.4.6.3 Shared Object Deletion

Figure 3.8 shows the collaborations needed to delete shared objects.

- **Sending a Delete Message to the Server (Client):** The deletion process begins with a request to delete a *Concrete Object Proxy* by an application on client side (`delete xProxy`). The `delete` operator invokes the destructor for the *Concrete Client Proxy*.

The destructor starts by deleting the concrete object for which it was a proxy and its client communicator. The destructor for the client communicator sends a message to the server notifying it that the client does not use the specific concrete object any more (there are no replies to such notifications). The client proxy is then able to finish its own deletion process.

- **Handling Delete Message (Server):** A arrival of a "delete" message from a client causes the receiving server communicator object to deregister itself from the corresponding server proxy (`DeregisterCommunicator()`).

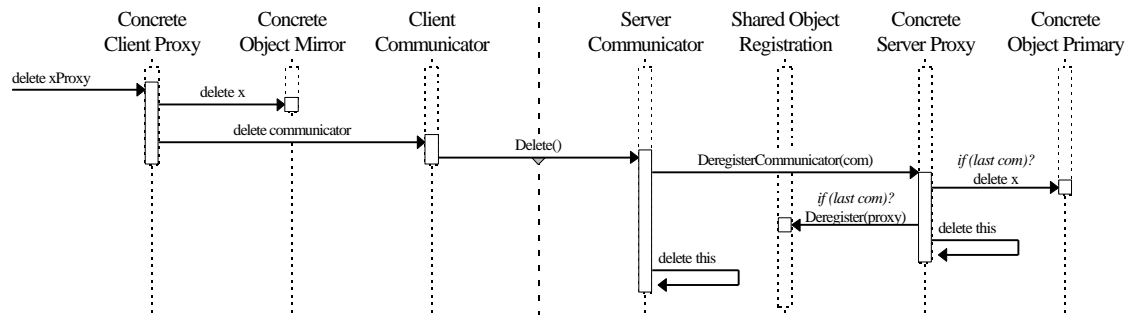


Figure 3.8: Collaboration between participants in the deletion process

If it was the last communicator registered to the specific proxy, then the server proxy usually deregisters itself from the *Shared Object Registration*, deletes the concrete object for which it was a proxy and then deletes itself (some applications have classes whose instances stick around until program termination). The server proxy can deal in its destructor with any application specific “delete” mechanisms needed in a specific subclass. Now the client communicator is able to finish its deletion process by deleting itself.

3.4.7 Consequences

The Simple Shared Object pattern yields the following benefits:

- Low amount of code for application specific object sharing:** Most of the member functions needed to support existence in different address spaces can be implemented in the partially abstract classes *Server Proxy* and *Client Proxy* (see Figure 3.4). As a result, only a few methods have to be implemented in the specific proxy classes. Most of the work that must be done to use shared objects relates to the serialisation and de-serialisation of the concrete object and the implementation of functions in the proxy that correspond to the non constant member functions of the class of the shared object.
- Clear identification of shared objects across different address spaces:** Objects that have to be shared within distributed applications can not be referenced by their memory address. Therefore, each shared object needs some kind of identifier by which it can be referenced by other objects. The Simple Shared Object pattern solves this problem by assigning a unique identifier to the primary

copy of the shared object (the one on the server) and supplying this identifier to each proxy that uses that shared object.

- **Clear separation between concrete objects and their concrete proxies:** The concrete objects do not know that they are shared. Therefore, when instances of already existing classes have to be shared, only new proxy classes need to be implemented to allow these concrete objects to be distributed across different address spaces. The Simple Shared Object pattern encapsulates concrete objects from communication methods needed for synchronisation in a high degree. However, it should be noted that existing classes can only easily be changed into shared classes if they allow serialisation and de-serialisation of their state.

On the other hand, the Simple Shared Object pattern has the following drawbacks:

- **Synchronisation overhead for “write only” shared objects:** If a client uses a shared object only for write access, there may be some unnecessary synchronisation traffic. If a client only modifies attributes of a shared object, it does not need a copy of the primary concrete object in its own address space. If it does have a copy, there is message traffic keeping this copy consistent with the primary. This overhead can be avoided by telling the concrete object proxy in the creation process, that a shared object is only needed for write access. As a result, the client proxy does not need to create a own copy of a concrete object. This behaviour should be considered when using the Simple Shared Object pattern. In practice, however, shared objects are mostly used for read and write access.
- **No transaction manager and locking mechanisms:** Sometimes, modifications on two or more objects depend on each other. In such cases a transaction manager would be desirable. Therefore, the Simple Shared Object pattern should not be used in cases where many transaction based modifications (with commit/rollback mechanism) on shared objects are expected. The incorporation of a transaction manager would give this pattern a grade of complexity that would not deserve the name “Simple”. The Simple Shared Object pattern is designed for low traffic, user interactive applications where changes to shared objects are relatively infrequent.

- **Applications must know about object sharing:** Although the concrete classes do not need to know that they are shared, the applications have to know where they deal with shared objects. The proxy objects do not belong to the same class hierarchies as the concrete objects for which they act; this necessitates changes where an application wants to use heterogeneous collections and polymorphism. In addition, the partially “asynchronous” behaviour of this pattern has ramifications that have to be considered in the design process of applications in which shared objects are used.

3.4.8 Implementation

Patterns can be implemented in various ways. However, all implementations of the Simple Shared Object pattern will encounter similar problems.

- **Caution when clients run on different operating systems (architectures):** Distributed applications frequently involve processes running on machines with different architectures. This has to be considered when implementing the serialisation and deserialisation methods for concrete objects. Different implementations of data handling such as alignment, floating point representation and integer resolution have to be considered. For instance, in some cases it is necessary to convert floating point variables to a specific format which can be handled by all involved operating systems.
- **Proxies should run in a single thread:** Multi-threaded models of execution are becoming increasingly common, but their correct operation depends on locking to prevent interference between threads. The Simple Shared Object pattern has no locking mechanism for object modification and consequently provides no support for developers of multi-threaded applications.
- **Never wait, because waiting blocks shared object synchronisation:** Event driven systems, such as the Simple Shared Object pattern, can never have “wait loops”. A code construct that involved sending a request and “waiting” for a reply over a specific network link will normally break the application. During the wait, other inputs will not be handled.

Instead of “waiting” for an action to be completed by a collaborator before completing a task of its own, the proxy for a shared object has to note (in its state data) that a request to a collaborator is outstanding. Then, when a reply is eventually received, it can use its state data to interpret that reply and complete the processing required.

- **Refer to shared objects only via the shared object identifier:** Whenever shared objects have to be referenced across different address spaces, it is compulsory to use their cross system unique object identifier instead of a concrete object address. The penalty for indirect referencing via object identifiers has to be put up with the cross system unique identification.
- **Persistence:** The Simple Shared Object pattern does not provide persistence, however it could be supplemented by a separate persistence mechanism. Rather than being deleted when all clients depart, the primary object on the server could be saved to persistent storage. The creation mechanisms implemented on the server would have to try to initialise a newly created shared object with appropriate data from the persistent store.

3.4.9 Sample Code

A Simple Shared Object pattern has been implemented at the University of Wollongong on a UNIX operating system. An I/O Dispatcher and Receptionist are managing the communication on each node. Schmidt’s and Stephenson’s published implementation on Windows NT [Schm+94] provides the well known I/O Dispatcher and Receptionist. Another description of an I/O Dispatcher can be seen in ET++ [Wein+88].

For the client and server communicators a derivative of the Reactor pattern [Schm95] is applied. This derivative uses the patterns “NetChannel” and “ComsOrganizer” which are described in Neil Gray’s paper “Pattern for a Framework for Distributed Programs” [Gray95].

Even though the Simple Shared Object pattern is not complicated, the code is necessarily fairly lengthy and consequently has not been included in this thesis. Code for the current implementation is available through the WWW network at <http://www.cs.uow.edu.au/people/ro04/>.

3.4.10 Variations

This description of the Simple Shared Object pattern focuses on a conventional form of simple object sharing. However, every implementation may use variations of this pattern with extended or reduced functionality. Variations could include:

- **Additional notification for more dynamic applications:** Some applications may need more than the two notification mechanisms (`AfterCreate()` and `AfterModify()`) explained in this pattern; for example, some might need a `BeforeModify()` notification. Such additional notification functions can be implemented in one of the main application independent classes *Server Proxy*, *Client Proxy*, *Server Communicator* or *Client Communicator* (see Figure 3.4).
- **Additional handling for “write only” shared objects:** As described in the drawbacks in section 3.4.7 (Consequences), the Simple Shared Object pattern has a synchronisation overhead when shared objects need only be accessed for write operations. Therefore, variations of the Simple Shared Object pattern may implement this functionality for specific “write” intensive distributed applications.
- **Transaction manager for shared object modification:** Another drawback described in section 3.4.7 (Consequences) is that this pattern is not designed for transaction based shared object handling. However, manipulations on shared objects within distributed applications can depend on each other, so a transaction manager might be required. Applications requiring such more complex facilities are probably more suitable for the use of development frameworks such as CORBA or OLE.

3.4.11 Known Uses

Derivatives of the Simple Shared Object pattern have been used in various small distributed applications. In terms of client/server database applications, a form of this pattern can be seen whenever database row objects are cached in the client’s address spaces to allow fast browsing through database tables. However, most such systems fail to provide automatic notification to clients of changes to the database. Consequently, such clients frequently use inefficient and costly mechanisms such as regular polling of the

database. More complex and powerful implementations can be seen in bigger systems such as CORBA and OLE.

3.4.12 Related Patterns

- The **Half-Object + Protocol** pattern [Mesz95] shows object distribution across different address spaces in a more general view.
- The communication part of the Simple Shared Object pattern can be implemented by using the **Reactor** pattern [Schm95].
- For systems where client and server are not working closely together, patterns such as **Warden** [Neve+95] or **Broker** [Stal95] should be considered.

3.5 Summary

This chapter has briefly introduced the experimental class library on that this research has been based. Classes used to extend the library to facilitate transparent object sharing across address spaces have been briefly described. A typical object constellation for client server communication between objects is shown in Figure 3.1 and Figure 3.2.

A pattern called “Simple Shared Object”, that uses the existing classes of the experimental class library, has been described in detail. The pattern offers proxy classes that allow client applications to have mirrored copies of shared objects in their address spaces. It explains further the architecture and the collaborations needed for object synchronisation of the mirrored copies. The pattern also shows an example of a CSCW application and refers to related patterns and shared object systems such as CORBA.

During the implementation of the pattern in C++, it became known that the model could be improved by using a more appropriate programming language for distributed applications. The model can also be extended with an application independent request broker, listening on a server port. Such a broker could handle messages for object synchronisation between connected object servers and client applications.

Chapter Four

4. Simple Shared Object Request Broker (SSORB) - A Tailored System for Java

4.1 Overview

Chapter 3 focused on the Simple Shared Object pattern that describes a system for object sharing across address spaces. This chapter describes an advanced model for object sharing based on the previously described pattern. The advanced model differs from the Simple Shared Object pattern mainly in that an application independent broker component has been put between object servers and client applications. This model for object sharing is named Simple Shared Object Request Broker, or with an abbreviation SSORB.

The description starts with an analysis that justifies the use of a broker system and the implementation mechanisms adopted (section 4.2). Criteria such as architectures (operating systems) for CSCW applications, programming languages and existing models for object sharing are discussed briefly. Then, the basic model of the SSORB system and its three main components are introduced in section 4.3 (The Basic Model of the SSORB). Section 4.4 provides a brief summary of the message objects that are exchanged between the three main components of the SSORB system.

The actual design of the SSORB system is presented in the following sections:

- Section 4.5 presents the request broker component.
- Section 4.6 introduces the structure of object servers and discusses the problems of object referencing across address spaces.
- Section 4.7 describes the basic structure of client applications and touches on the issue how the shared object are connected to a user interface.

- Section 4.8 describes a design pattern called “Deputy” that has been applied in all three main components of the SSORB system. The pattern characterises a general approach for delegating commands to a multi-threaded deputy object. This general pattern has been abstracted from similar components that existed in several parts of the SSORB system. The sample code of the pattern is shown in the Java programming language.

4.2 Broker Systems for Collaborative Applications

After the implementation and the evaluation of the Simple Shared Object pattern in C++, it was realised that the system for object sharing could be improved and extended in various ways. The following issues were prominent:

- *On what systems (architectures) could CSCW applications base in the future?*
- *Could another programming language than C++ be more appropriate?*
- *Could an already existing model be specialised for simple object sharing?*

4.2.1 On what systems (architectures) could CSCW applications base in the future?

The success of a CSCW application probably depends on the architectures that are supported by the application. Electronic mail has often been used as the most successful example for a CSCW application. What made electronic mail so successful? Beside being a cheap means for information exchange between people, electronic mail systems are available on almost all operating systems and user interfaces. Most providers of electronic mail software support a common standard for exchanging messages.

Today, the World Wide Web represents a virtual place where people all around the world can search for information. Tools such as Netscape Navigator and Mosaic allow users to browse the Internet without dealing with networks and operating systems directly. Therefore, the World Wide Web could be an ideal place for running CSCW applications. However, even though the World Wide Web is an important place for distributed applications, there is still a need that CSCW applications also run directly on operating systems such as Unix, Macintosh and Windows 95/NT.

In short, there is no specific operating system or architecture that should be supported by CSCW applications. Distributed applications need to collaborate across operating systems, network architectures and user interfaces. To use a buzzword, *interoperability* is a main requirement for modern CSCW applications.

4.2.2 Could another programming language than C++ be more appropriate?

C++ is probably the most commonly used object-oriented programming language. Its fast execution code and the high degree of portability gave this language the break through in the late 1980's (after its predecessor C). However, there are unfortunately some missing elements in C++ that makes it difficult to create real portable CSCW applications. These missing elements are:

- **No standard graphical user interface for C++:** The ANSI standard for C++ does not include a standard graphical user interface. Therefore, most C++ implementations use their own GUI libraries to create state-of-the-art applications.
- **No platform independent compiled code:** C++ code is compiled into architecture specific code to achieve good execution performance. One of this feature's drawback is that primitive data types are not standardised in terms of resolution (integers) and floating point representation (different formats depending on operating systems). There are some additional C libraries that support data conversion for network handling. However, these libraries are limited terms of data types that are supported.
- **No standard library for network handling:** Network handling is again not included in the ANSI standard of C++. Therefore, distributed C and C++ application programs, that need to collaborate via networks, have to include architecture specific network libraries.

Even though C++ has some missing elements to provide portable code across operating systems (architectures), the great range of available C and C+ libraries allows programmers to create powerful distributed applications.

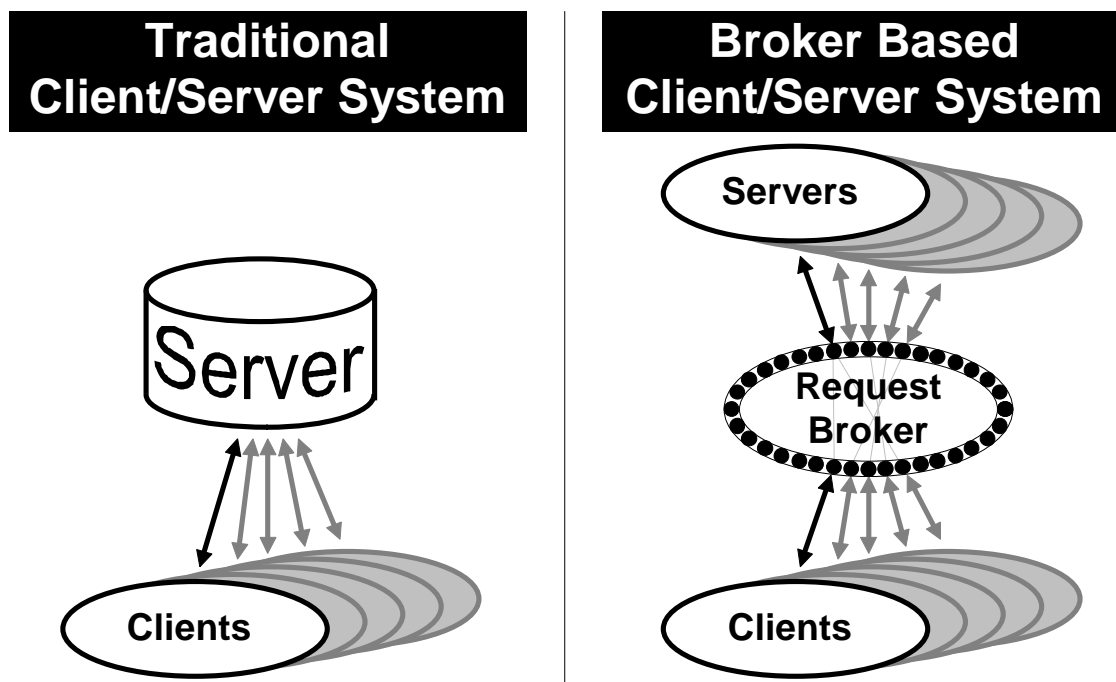


Figure 4.1: Traditional and broker based Client/Server systems

On the other hand, these “missing features” of C++ are supported in the Java programming language. The Java development kit (V1.0) was released in mid January 1996. According to the description of the language [Gosl+95], Java seems to be more appropriate and architecture independent than C++ for the implementation of distributed applications. In order to test whether or not Sun’s promises work in practice, the SSORB system has been implemented in the Java programming language.

4.2.3 Could an already existing model be specialised for simple object sharing?

The terms *distributed applications* and *CSCW applications* are mostly used in conjunction with the already overused buzzword *Client/Server systems*. Client/Server systems have been replacing host based systems over the past ten years. In most host based systems, everything (even the applications programs) ran on a single host and users interacted with this host by using “dumb” terminals. In the late 1980’s, personal computers became cheaper and replaced the terminals of the host based systems. The power of the clients could now be used to run applications and a central server provides these clients with data. In such traditional Client/Server systems, clients are usually direct connected to a server.

Today, there seems to be again a change in the structure of some Client/Server systems. Systems such as CORBA and OLE use a “well-known” request broker that handles clients and servers. Servers connect to such a request broker and offer services. Then, clients are able to use the offered services by communicating via the request broker. Figure 4.1 shows a traditional Client/Server system and a Client/Server system with a request broker. This architecture means that clients do not need to know where servers are actually running. When clients need to use services that are offered by different servers, the clients need only to connect to one central request broker.

In short, the basic model of systems such as CORBA and other broker based systems are like tailored for distributed applications and could be applied to broke replication messages between shared object in different address spaces.

4.3 The Basic Model of the SSORB

The SSORB system is divided into three main components. These components are the Request Broker, the Object Servers and Client Applications. Each of these components has its own responsibilities and functionality.

Figure 4.2 shows the components of the SSORB system and their main responsibilities. It can be seen that the broker is totally application independent. It simply keeps track of connected object servers and client applications, maintaining records of which clients are connected to which server and vice versa. The main responsibilities of the broker are the registration of connecting object servers and client applications and the forwarding messages.

An *object server* is the central point of a distributed application using the SSORB system. All the modifications to shared objects are done first at this place and are then broadcast to clients. The object server is also able to garbage collect shared objects that are not longer in use.

A *client application* is that component of a distributed application with which a user actually interacts. It is responsible for the representation of shared objects and handles interactions with the user. Whenever a user modifies a shared object, the modifications are first sent to the object server that holds the primary copy of the shared object.

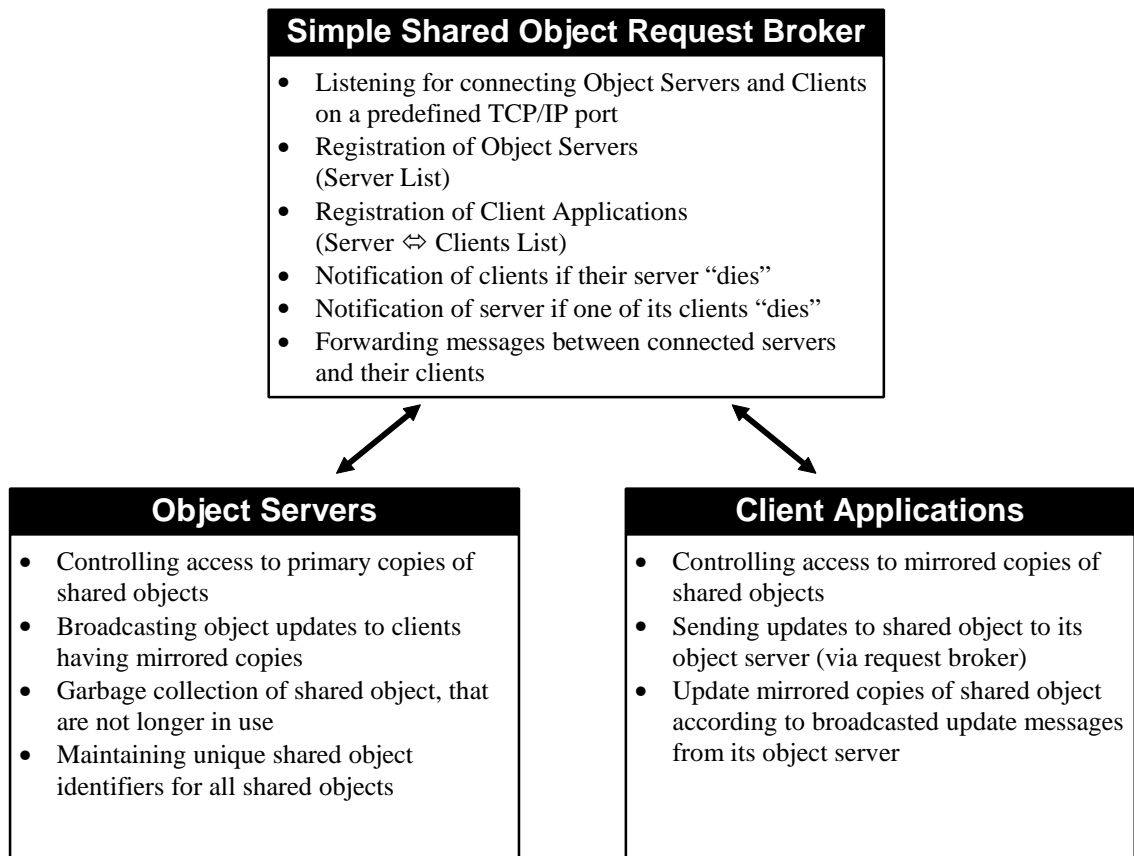


Figure 4.2: The components of the SSORB system and their responsibilities

Then the mirrored copies are replicated. Read operations to shared objects are done directly using the mirrored copies.

Object servers and client applications are both “clients” from broker’s point of view. In order to avoid confusion between “client applications” and “clients”, **object servers** and **client applications** are named as **customers** within the SSORB system.

4.4 Message Objects within the SSORB System

Message objects that are exchanged between the three main components of the SSORB system (the broker, object servers and client applications) have a predefined structure. A message object has the following attributes:

- **Sender Identifier:** The unique identifier of the customer that sends the message (object server or client application). If the sender identifier is 0, the broker itself is the sender of the message.

- **Receiver Identifier:** The unique identifier of the customer that should receive the message (object server or client application). If the receiver identifier is 0, the broker itself should receive the message.
- **Object Identifier:** Every shared object has its own, unique identifier (see Simple Shared Object pattern in section 3.4). If a message is used in connection with a shared object, this attribute holds the shared object identifier.
- **Request Identifier:** The request identifier is defined by the sender when a message is sent across the network. The request identifier is unique within each component (broker, object server and client application).
- **Registration Identifier:** The registration identifier is only used by receptionist objects on the customer side when a request message is put on a list for outstanding requests.
- **Message Tag Identifier:** The tag identifier defines the message type. All used message types are predefined as constants within the message class.
- **Acknowledge Flag:** A message can specify that a request requires a reply. If a request message needs to reply, the acknowledge flag is set. If a message has just to be sent to another component (without reply), the acknowledge flag is unset. By default, the acknowledge flag is set.
- **Message Reply Identifier:** The handler of a message can set a reply identifier (if acknowledge flag is set) to define a request. Positive reply identifiers (including 0) are used for requests that are handled successfully. Negative reply identifiers are used to signal that the request could not be handled successfully.
- **Message Input Stream:** The input stream can hold a stream of primitive data types, string objects and shared object references. This stream is used for deserialisation of shared objects and update parameters.
- **Message Output Stream:** The output stream can hold a stream of primitive data types, string objects and shared object references. This stream is used for serialisation of shared objects and update parameters.

Message objects are exchanged between different operating systems (architectures). In order to provide a full architecture independent message transfer, every element of the input stream and output stream is stored in an architecture neutral format. The serialisation and de-serialisation of shared objects is limited to the following primitive data types and object types (for this implementation in the Java language):

- **Primitive Data Types:** boolean, byte, byte[], char, float, double, short, int, long
- **String Objects:** String objects are serialised and de-serialised in Unicode (see Appendix B, Trademarks and Terms).
- **Shared Object References:** Shared object references can not be exchanged by memory addresses. The SSORB system provides a class (SSOProxyRef) that instances can be serialised and de-serialised in a portable way.

4.5 The Broker

The broker of the SSORB system is itself further divided into three main components. These are a receptionist object, an administrator object, and customer handler objects. When a broker starts, there is a singleton receptionist object listening on a “well-known” TCP/IP port for connecting object servers and client applications.

Another singleton object, the administrator, is waiting to manage message exchanges between connected customers. In broker’s terms, customers are the object servers and the client applications that have connected. Every connected customer has its own customer handler object on the broker. A customer handler registers itself with the administrator object and waits for the arrival of messages from its customer. Every customer handler object runs using its own thread.

The administrator object is able to register and deregister customer handlers and its main responsibility is to forward messages between connected customers. The administrator object is also able to interpret and execute messages that are sent to the broker itself. For instance, an object server is able to send a message to the broker with the request that the broker forward the message to all of its own clients. Such “multiple-forward” requests are handled by the administrator object; a customer handler object,

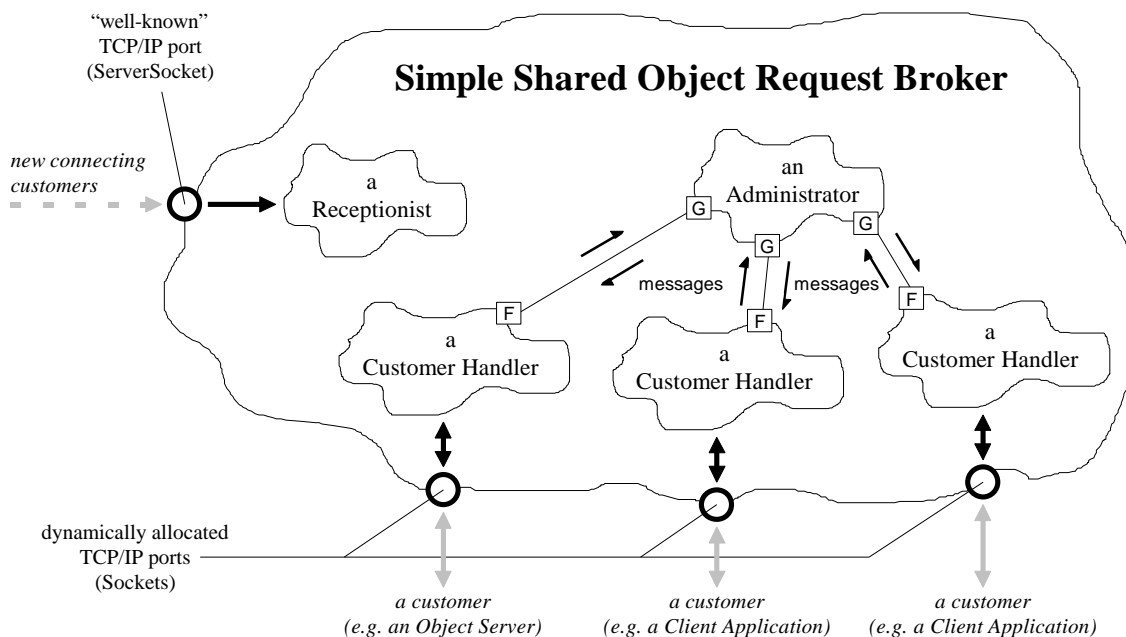


Figure 4.3: The components of the Simple Shared Object Request Broker

that receives a “multiple-forward” request delegates the “multiple-forward” messages to the administrator and is then able to continue immediately its own customer handling.

Figure 4.3 shows the components within the broker in an object diagram. The example shows three customers that are connected to the broker (one object server and two client applications).

A single **receptionist** object is listening at a “well-known” TCP/IP port (ServerSocket). When a customer connects to this “well-known” port, the receptionist object instantiates a new customer handler object and specifies the “dynamically allocated” TCP/IP port as parameter for the newly created customer handler object. Then, the receptionist object returns immediately to its own job of accepting new customers.

A newly created **customer handler** object creates its own thread at construction time and uses the given “virtual” TCP/IP port for communications. The internal thread of control allows each customer handler object to act independently of the receptionist object. Every customer handler gets a unique identifier that is also given to the connected customer (object server or client applications). This identifier is used to define the sender and the receiver of messages that will be exchanged. The broker itself has a well know identifier (which is always 0).

The first message that a customer handler receives from its customer defines whether it is to represent an object server or a client application. With this first message the handler is able to register itself with the administrator object.

The **administrator** object has the important job of exchanging messages. It holds a few hash tables for fast correspondent location. The hash tables hold the relationships between object servers and client applications. One hash table is indexed by “Client Application” customer handlers and refers to their “Object Server” customer handlers. Another hash table is indexed by “Object Server” customer handlers and refers to their “Client Application” customer handlers. All messages are handled in parallel by separate threads for each message. The administrator object is also responsible for informing customers when one of their correspondents disappears (by disconnecting or by a broken connection). When a customer disappears, the administrator object deletes all relationships involving the lost customer by deregistration of the disappeared customer handler object from its internal hash tables.

The administrator class is a subclass of a “so-called” Deputy class. The Deputy class is described in the Deputy pattern at the end of this chapter (see section 4.8 (“Deputy” - A Behavioural Pattern for Delegating Commands to a Multi-Threaded Deputy Object)).

4.6 Object Servers

Object Servers provide primary copies of shared objects that can be used by client applications. All client applications are connected to object servers via the broker. Even though the SSORB system is mainly based on the Simple Shared Object pattern, the object constellation has been changed.

Figure 4.4 shows the object constellation of a typical SSORB object server. A receptionist object (a singleton) is connected to the broker. On the broker side, there is one customer handler object that communicates with this receptionist object. The receptionist object has to be created by the object server program. The receptionist object creates its collaborators objects *proxy handler* and *shared object registration*. After creating the proxy handler object and the shared object registration object, the receptionist

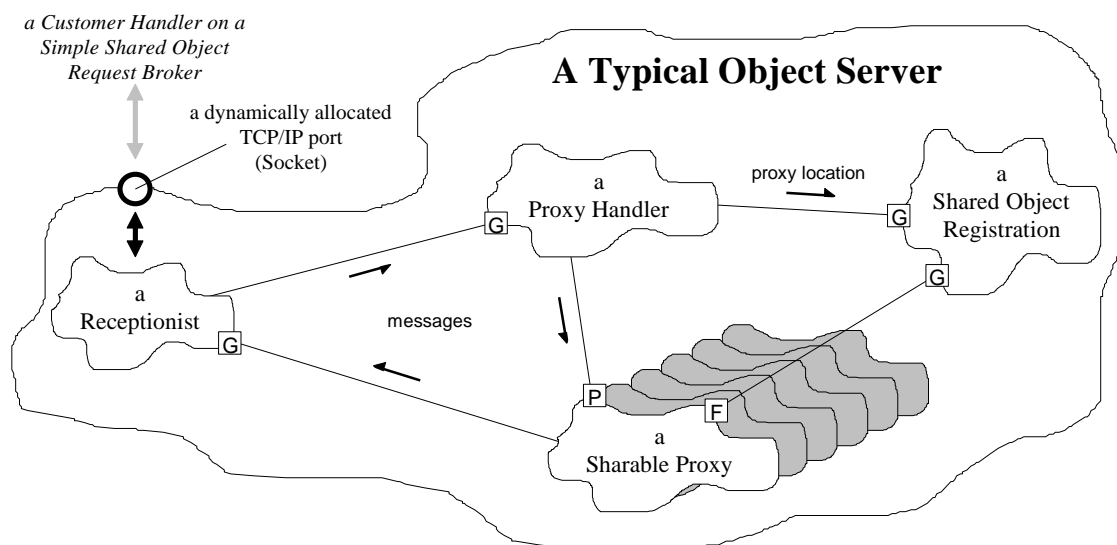


Figure 4.4: A typical object constellation within a SSORB object server

object only communicates with the proxy handler object and with the broker (via its TCP/IP port).

The receptionist object has its own thread, it is responsible for sending and receiving messages from the broker. All messages received from the broker are sent to the proxy handler object. The proxy handler object handles all messages independently of the receptionist; processing messages using its own threads. The proxy handler object works directly with the proxy objects and uses the shared object registration object to locate proxy objects.

The shared object registration object is purely passive. The responsibilities are the same as described in the Simple Shared Object pattern. The registration knows all available proxy objects and is used to locate proxies. It is also used to generate unique proxy identifiers. All proxies are indexed by these identifier numbers for a fast proxy location.

4.6.1 Referencing of Shared Objects across Address Spaces

The Simple Shared Object pattern provides referencing to shared object only via shared object identifiers. It was realised, that this system for referencing is troublesome and complicates application development. As a result, the SSORB system provides a special class (ProxyRef) for handling proxy references. An instance of the proxy reference class hold all information to instantiate a shared object.

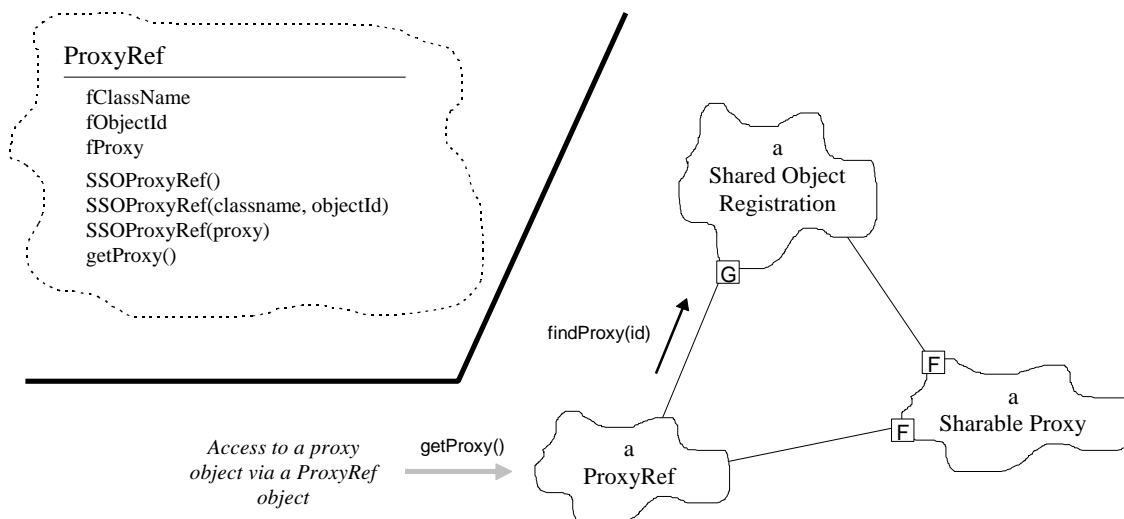


Figure 4.5: The proxy reference class within the SSORB system

A ProxyRef instance holds three attributes. These are the actual proxy class (in Java the class name), the shared object identifier and a reference (real address) to the proxy. The class name and the shared object identifier are either set to a correct value or are NULL. When a proxy needs to be accessed via the ProxyRef instance, a method “getProxy()” is called. This method first searches for the required instance of the proxy (lookup in the Shared Object Registration), if the proxy already exists, a reference to that proxy is stored for subsequent use as an attribute within the ProxyRef object. If a proxy does not already exist, the ProxyRef object creates the required proxy object.

When a ProxyRef instance has to be given to a client application (running in another address space), only the class name and the shared object identifier are transferred to the client application. The client application fills the proxy reference (real address) attribute of its copy of the ProxyRef instance when the proxy object is needed. The message class (see section 4.4) of the SSORB system provides methods to put the necessary attributes into a message output stream and to extract a ProxyRef object from a message input stream.

Figure 4.5 shows the ProxyRef class with the main methods and attributes used for a Java implementation. The figure also shows an ProxyRef object and its relationships to other objects.

4.6.2 “Well-known” Proxy Lists

When a client application connects via broker to its object server, it needs to know some application specific shared objects. For instance, an object server might hold a list of existing users, a newly connected client application might need to know the names of these users. Therefore, the client needs to create an instance of a shared object that represents this list of users. Such a list would be an instance of a “well-known” object in the SSORB system. A “well-known” object list is simply a sharable proxy whose object identifier is predefined and therefore “well-known” by its proxy class.

An application may provide a few “well-known” proxy lists that refer to fundamental shared proxy objects that are required for the application. However, the requirements of “well-known” proxy objects is totally application dependent and is not predefined by the SSORB system.

4.7 Client Applications

Within a client application, the SSORB system provides mirrored copies of shared objects that are used. These mirrored copies are automatically updated to the state of the primary copies on an object server by the SSORB system. The structure of the SSORB components within client applications is basically the same as the structure within object servers. In addition to the object servers, on clients there are additional classes for architecture independent presentation of shared object to users (relying on Java AWT classes to provide this independence).

The Simple Shared Object pattern uses different proxies for clients and servers. It was realised the implementation of proxy classes would be simplified, if the client and server proxies could use the same base class. In the SSORB system, object servers and client application use the same partially abstract base classes. When connecting to a broker, the receptionist object has to be told whether to connect as an “object server” or as a “client application”. Even though the classes receptionist, proxy handler, proxy and shared object registration are the same for object servers and client applications, these object act somewhat differently in servers and in clients (it should be noticed that the term “receptionist” within the SSORB system is used in a different meaning than it is used in the pattern presented by Neil Gray [Gray95]).

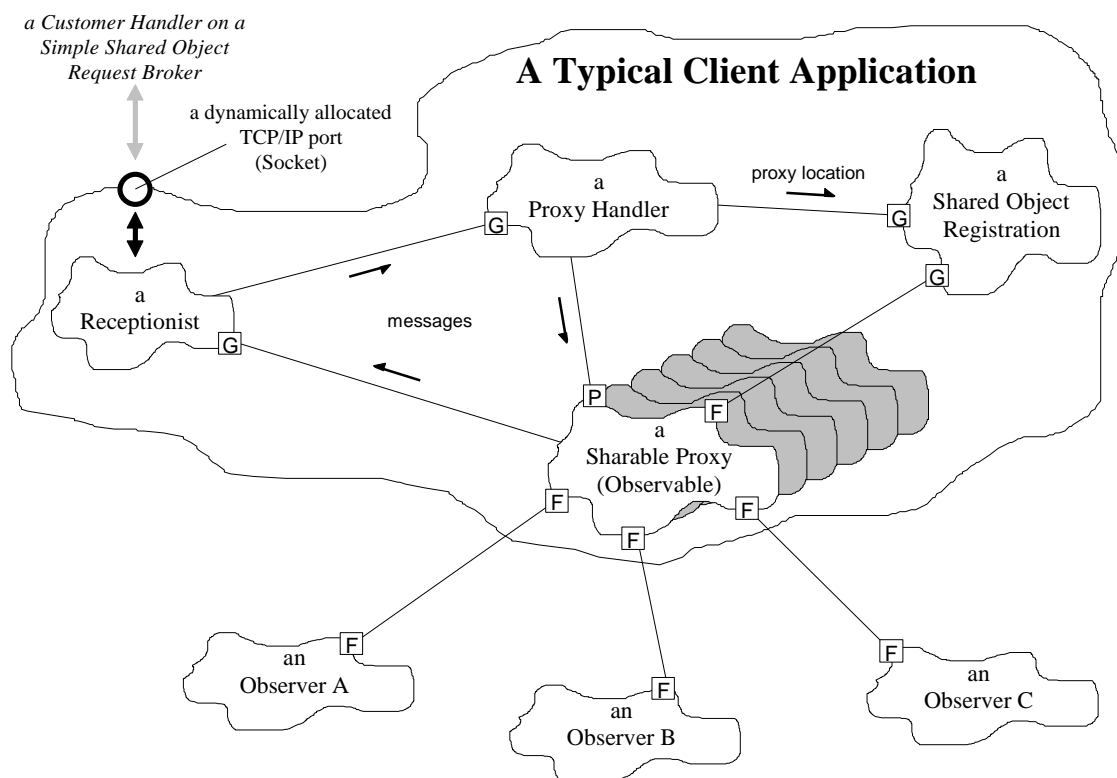


Figure 4.6: A typical object constellation within a SSORB client application

In the SSORB system, a client application uses a shared object registration to locate sharable proxy object that are in client's address space (in contrast to the Simple Shared Object pattern where only servers use shared object registrations). In return, there is only one receptionist object (client communicator of the Simple Shared Object pattern) needed to handle all sharable proxy objects within a client application.

Sharable proxy object usually represent data object that are shared across different address spaces. Therefore, sharable proxy objects need to be handled as model objects (in Smalltalk's language), TDocument objects (in MacApp's language) or observable objects (in the Java language). This thesis shows the implementation of the SSORB system in the Java language. As a result, the Java class names *Observable* and *Observer* are used in this description. Gamma *et al.* use in the Observer pattern [Gam+95, pp293] the term *Subject* for *Observable*. In the SSORB system, the sharable proxy class (base class of all proxies) is implemented as a subclass of *Observable*. That allows to add and remove *Observer* objects (*Views*) to/from all proxy objects.

Figure 4.6 shows a typical object constellation in a client application using the SSORB system. It can be seen that the basic participants are the same as those of the typical object server (see section 4.6). Even though the sharable proxy objects on an object server are actually also “observable” objects, the object server does normally not present data to a user. Most client application will present shared object to a user in one or another way. The figure also shows where the observer objects are connected to the SSORB system. The shown sharable proxy has three sample observers (A, B and C) that are representing the proxy to a user.

4.8 “Deputy” - A Behavioural Pattern for Delegating Commands to a Multi-Threaded Deputy Object

During the implementation of the SSORB system, it was realised that there is often a need to delegate commands or method calls from one object to another. In the SSORB system the customer handler objects (on broker side) and the receptionist objects (on object servers and client applications) delegate the handling of messages to other objects (e.g. a customer handler object on the broker side delegates the handling of messages to its administrator object). Further the receptionist (within object servers or and client applications) delegates the handling of messages to its proxy handler object.

Through generalisation of such examples of delegation, it was possible to find a common pattern for delegating method calls (commands or handling of messages) from one object to another. Furthermore, the generalised pattern is able to act as a resource controller (thread limitation, command queuing). The fact that in this pattern one object does the jobs of another object gave the pattern the name Deputy. The *Collins Cobuild Essential English Dictionary* defines a deputy as follows:

A deputy often acts on behalf of the boss when the boss cannot be present.

The following pattern describes the generalised Deputy pattern and shows sample code in the Java language.

4.8.1 Intent

The Deputy pattern allows objects to delegate commands to other objects. Delegated commands are executed by a *Deputy* object and every delegated command is running in

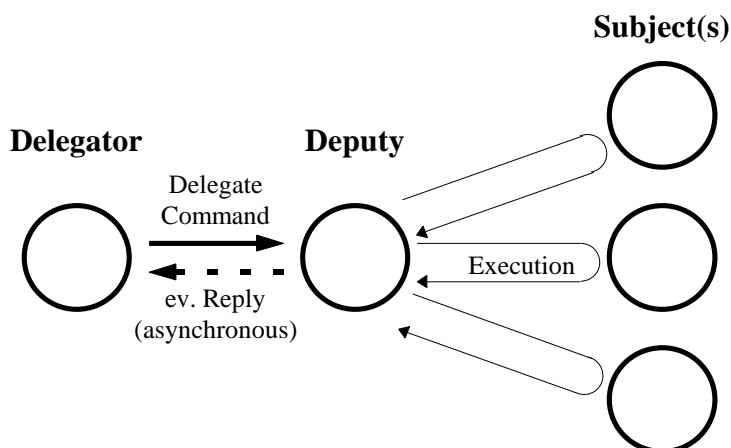


Figure 4.7: A deputy executing delegated commands

its own, single thread. After a command has been delegated to a *Deputy* object, the control returns immediately to the delegating object. A delegated method call is executed asynchronously and is able to reply to the delegating object. The Deputy object has the job of controlling the execution of the delegated commands. Another responsibility of the Deputy object is to control the number of concurrent running threads.

4.8.2 Motivation

Information exchange between objects within an application program always involves method calls from one object to another. In some systems, objects can be “very busy” and are therefore not able to wait until every initiated method call is completed.

Suppose a receptionist object within an application program is responsible for information exchange through a TCP/IP port with another application program. Incoming commands need to be handled and some of these commands may take a few minutes to be processed. In such a system, the receptionist object must be able to delegate incoming commands to another object to be free for further command arrivals.

Figure 4.7 shows the basic elements of the Deputy pattern. A *Delegator* object delegates commands to a *Deputy* object and specifies whether the commands need to be executed in parallel or in sequence to other delegated commands. The *Deputy* object executes the delegated commands and replies to the *Delegator* object if required.

The pattern works actually like an executive and a competent secretary working together. The executive delegates jobs to the secretary. Then, the secretary handles the jobs independently and gives feedback to the executive if necessary.

4.8.2.1 Requirements of Deputy Objects

The pattern is intended to satisfy the following common requirements for delegating method calls to *Deputy* objects:

- **Parameter delivery:** Method calls deliver parameters (arguments) to the called object. A *Delegator* object must be able to specify any number of parameters in any order. Further, the *Delegator* object should also be able to pass any kind of application specific objects as parameters to the *Deputy* object.
- **Sequential execution:** Sometimes, a particular method call may depend on another. Therefore, a *Deputy* object must be able to execute delegated commands in the order in which they are delegated.
- **Parallel execution:** Delegated method calls that are independent should be able run in parallel.
- **Reply after execution:** Sometimes, *Delegator* objects need to get back a reply after a command has been executed. Such replies should also be able to deliver parameters back to the *Delegator* object.
- **Resource controlling:** The *Deputy* object should be able to control the number or concurrent running threads (in parallel execution of commands). Therefore, the *Deputy* object needs to control delegated commands for parallel execution and it should be able to queue parallel commands if the resource limit is reached (number of concurrent executing commands).

4.8.2.2 Behaviours of Deputy Objects

The pattern is intended to give *Deputy* objects the following common behaviours:

- **Never “busy”:** A *Deputy* object must never be busy, meaning it must always be able to accept new commands. If commands need to be executed in sequence, the *Deputy* object must be able to queue these commands until they can be executed.

- **Independence of parallel executing commands:** Parallel executed commands object must never influence other independent commands. If commands influence others, they should be delegated to run in sequence.

4.8.3 Applicability

Use the Deputy pattern when

- a system is implemented in a programming language that supports multi-threaded execution.
- an object is too busy to handle every method call by itself.
- performance needs to be improved through parallel execution of method calls.
- method calls have to be queued in a command list to be handled in sequence.
- methods have to be executed asynchronous to the thread that initiates (delegates) the method calls.
- the number of concurrent executing commands has to be limited.

4.8.4 Structure

The Deputy pattern is divided into four kind of application independent classes. These classes are *Command* classes, a *Reply* class, a *Delegator* class and a *Deputy* class.

The *Command* class has two specialisations, that are a *Dumb Command* class and a *Smart Command* class. The *Dumb Command* class is totally application independent and holds only attributes for the command execution. The *Smart Command* is the base class of application specific “smart” commands classes that implement an “execute()” method. These classes are called “smart” command classes, because they know by themselves how to execute a command. Instances of the class *Dumb Command* are interpreted by a specialised *Deputy* class.

The *Reply* class can be implemented fully application independent. The *Delegator* class is purely abstract (an interface in Java [Gosl+95] terminology) and provides the interface for application dependent classes that instances act as *Delegator* objects.

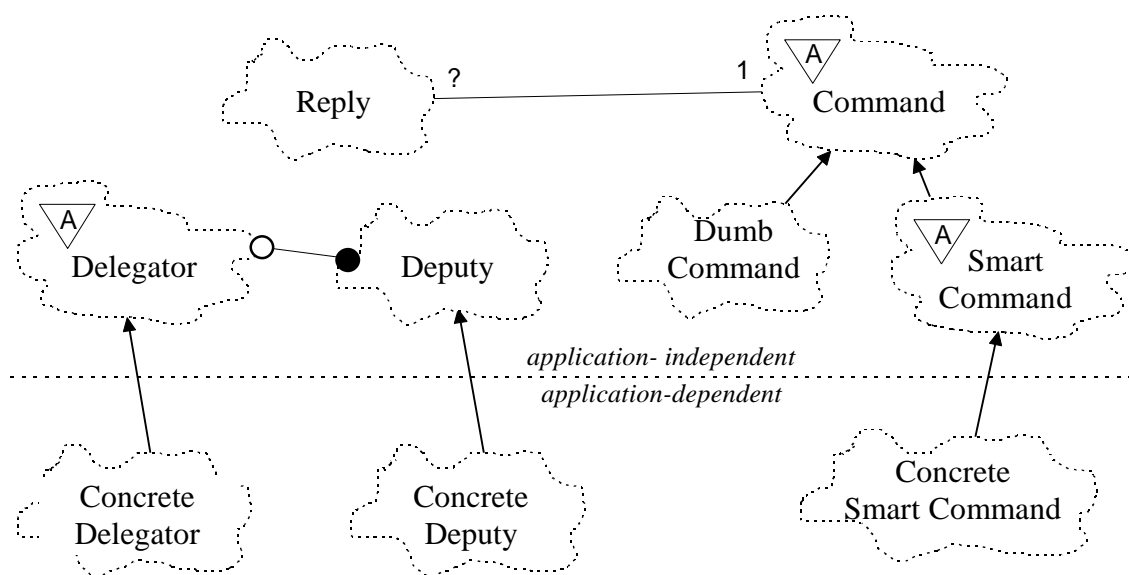


Figure 4.8: Class diagram of the Deputy pattern

The *Deputy* class is fully application independent if it handles only *Smart Command* instances. If *Dumb Command* instances have to be handled, the *Deputy* class needs to be specialised.

The classes of the Deputy pattern are shown, using Booch [Booch94] notation, in Figure 4.8. The classes *Concrete Delegator*, *Concrete Deputy* and *Concrete Smart Command* stand for application dependent classes. In a real implementation, a *Concrete Delegator* object would delegate “dumb” commands to a *Concrete Deputy* object, that interprets the delegated commands. “Smart” commands can be handled directly by an instance of the *Deputy* class (without implementation of a *Concrete Deputy* class).

- **“Command” classes:** As described previously, commands can be handled as “dumb” and as “smart” commands. An instances of a specialised *Command* class is able to hold all information needed for a postponed command execution. At command delegation time, it is specified whether or not a command needs to be executed in parallel or in sequence to other delegated commands.
- **“Reply” class:** An instances of class *Reply* is able to hold a reply identifier and an array of reply parameter objects. *Reply* objects are, if necessary, passed back to the object that has delegated the command and that needs to be informed after the command has been executed.

- **“Delegator” class (interface):** The *Delegator* class is purely abstract. It defines actually only one method that has to be implemented by subclasses. This method is a reply handler that is used whenever replies are needed after a delegated command has been executed. In the Java programming language [Gosl+95], the *Delegator* is a typical example for an interface.
- **“Deputy” class:** The *Deputy* class implements most of the functionality of the Deputy pattern. It implements a method that registers *Command* objects to two major internal lists. One list holds a queue for sequential commands and another list holds a queue for parallel commands. It is also responsible for creating threads that handle delegated commands.

If a deputy object only needs to execute “smart” commands, it does not need to be specialised, because all information needed to execute the command are known by the “smart” commands.

However, if a deputy object need to handle “dumb” commands. It must be able to interpret the context of the delegated commands. In this case, the *Deputy* class needs to be specialised and a method for handling the commands must be implemented (mostly a big switch statement). The handling of dumb commands is needed to decrease the number of “smart” command classes. Dumb command handling seems to be unnecessary in modern, well designed object oriented programs. However, it should be considered that in a Java implementation every class is loaded separately by a class loader. Thus, a limitation of classes can improve the class loading performance, if classes are loaded across the World Wide Web.

- **“Concrete Delegator” class:** Specialised *Delegator* classes are only needed if a delegated command needs to reply after execution. If no reply is needed, any objects can delegate commands to a *Deputy* object.
- **“Concrete Deputy” class:** Specialised *Deputy* classes need to implement if “dumb” commands need to be handled. An implemented “command handler” method of a specialised *Deputy* class is called for each delegated command and every command runs in its own, single thread. The calls are initiated from the base class *Deputy*.

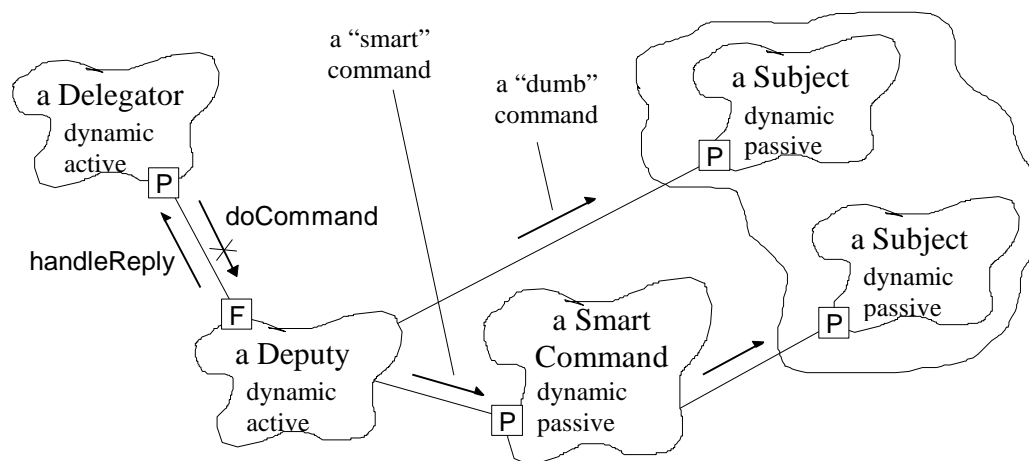


Figure 4.9: Object diagram of the Deputy pattern

4.8.5 Participants

Figure 4.9 provides an illustration of the participants (by using Booch [Booch94] notation) in this pattern. The object diagram looks mainly like the illustration in Figure 4.7. The participants are:

- Command:** As described in section 4.8.4 (Structure), *Command* objects are able to be executed as “smart” and as “dumb” commands. The figure shows a “dumb” command that is handled directly by the deputy and a “smart” command that is handled through the command object itself.
- Reply:** As described in section 4.8.4 (Structure), a *Reply* object is able to hold reply parameters and it also holds a the *Command* object, that has been passed at command delegation time. The whole contents of a *Reply* object is specified at construction time.
- Delegator:** A *Delegator* object creates and delegates command objects. The *Delegator* object specifies at command delegation whether or not the command needs to reply (then it specifies itself as a parameter) and a boolean to specify if the command needs to be executed in sequence or in parallel to other commands.
- Deputy:** A *Deputy* object has initially no relationship to other objects. All objects, on that a *Deputy* object has to apply commands, are passed as parameters (covered in a *Command* object) at delegation time (`doCommand()`). A *Deputy* object maintains one list of commands that are queued for sequential execution and another list of commands

for parallel execution. The delegated commands are proceeded totally asynchronous. If a *Delegator* object needs to get a *Reply* after completing a command, a *Reply* object is asynchronous sent to the initiating *Delegator* object.

The *Deputy* object also controls the number of concurrent running threads for parallel command execution. The thread limit can be specified by the application and is changeable at run time. It can also be defined that no thread limit is applied by the *Deputy* object.

- **Subject(s):** Subjects are objects on that delegated commands are actually applied. A subject can be any kind of application specific objects. It should be considered, that delegated commands are executed asynchronous. Therefore, implementation of application specific *Subject* classes must be thread safe.

4.8.6 Collaborations

Figure 4.10 shows the collaborations within the Deputy pattern. In the example, two “dumb” commands are delegated to a *Deputy* object. One command (Command A) has been delegated not to reply, the other command (Command B) has been delegated to reply after execution. The commands have both been delegated to be executed in parallel.

The threads, that are involved to execute the delegated commands are shown with different shadings. A thread created for executing a command is only alive until the command has been proceeded. The figure shows that the reply for command B occurs asynchronous to the initiating *Delegator* object.

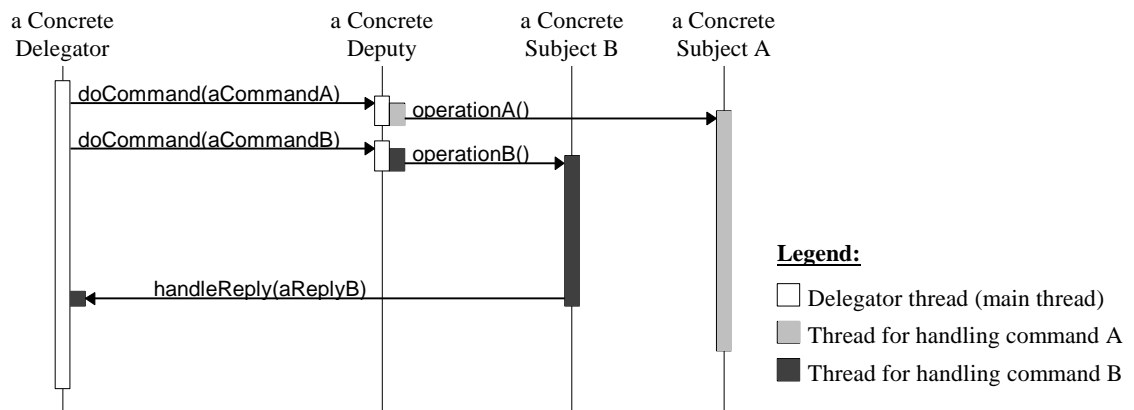


Figure 4.10: Collaborations between participants of the Deputy pattern

4.8.7 Consequences

The Deputy pattern yields the following benefits:

- **Delegator objects are not blocked through time intensive jobs:** Sometimes, an object is not able to handle time intensive commands by itself. Through delegating commands to other threads, an object can become more free to deal with other jobs. Therefore, a *Delegator* object does not block a system if time intensive commands need to be computed.
- **Distribution of command initiation and command execution:** The pattern covers delegated commands into *Command* objects. As a result, applications using the Deputy pattern can be divided into small parts through distribution of command initiation and command execution.
- **Sequential command execution if necessary:** Sometimes, commands have to be executed in a specified sequence. Even though every delegated command is executed in its own, single thread, the Deputy pattern supports sequential and parallel command execution.
- **Resource control (number of parallel threads):** The number of concurrent executed commands can be specified and modified at run time. Therefore, thread limitation can be applied easily, if required.

On the other hand, the Deputy pattern has the following drawbacks:

- **Subjects must be thread safe:** As described previously, every command is executed in its own, single thread. That means, commands are applied to the destination objects asynchronously. Therefore, objects, on that delegated commands execute, need to be implemented as thread safe.
- **Death-locks are possible:** Trough splitting command initiation and command execution into separate threads, it has to be considered in the application design. For instance, a *Delegator* object delegates a command to a *Deputy* object within a synchronised method (monitor method). The thread, handling the delegated command is not able to access the *Delegator* object, because it is blocked by its monitor. Such a behaviour has to be considered in a application using the Deputy pattern.

4.8.8 Implementation

This section describes the application independent classes of the Deputy pattern. The implementation shows code in the Java [Gosl+95] programming language (Java Development Kit V1.0).

The application independent classes are implemented as a Java package with the name `deputy`. All class names of the Deputy pattern begin with the initials DPY to avoid class name mismatches in applications that use this implementation. In order to minimise the length of the shown code in this section, the comment lines of the original implementation have been removed.

The original implementation (with documentation) is available through the World Wide Web at the address <http://www.cs.uow.edu.au/people/ro04/>.

4.8.8.1 The “DPYCommand” Class

The following code shows the abstract class `DPYCommand`, that implements the common methods for specialised command classes such as `DPYDumbCommand` and `DPYSmartCommand`. The only common attribute for command objects is the `DPYDelegator` object. The implementation gives local access to set the `DPYDelegator` object (used by `DPYDeputy` objects) and public access to read the `DPYDelegator` object. The

attribute is only set by a DPYDeputy object if a command needs to send a reply to the delegating object. Otherwise the attribute has its default value null.

```
package deputy;

import java.lang.*;
import java.util.*;
import java.io.*;

public abstract class DPYCommand {
    DPYDelegator fDPYDelegator = null;

    void setDPYDelegator(DPYDelegator delegator) {
        fDPYDelegator = delegator;
    }

    public DPYDelegator getDPYDelegator() {
        return fDPYDelegator;
    }
}
```

4.8.8.2 The “DPYSmartCommand” Class

The abstract class DPYSmartCommand is a specialised DPYCommand class. It only defines the interface for executing a command. Instances of a specialised DPYSmartCommand class are “smart”, meaning they are able to execute the commands themselves. Therefore, there is no need to specialise the DPYDeputy class for command interpretation if only “smart” commands are used. The use of smart commands pretends long “switch” statements in a specialised DPYDeputy class for command interpretation.

```
package deputy;

public abstract class DPYSmartCommand extends DPYCommand {
    public abstract DPYReply execute();
}
```

4.8.8.3 The “DPYDumbCommand” Class

Instances of the DPYDumbCommand class are able to hold a command identifier and command parameters (as an array of objects). Dumb commands are, as the name says, not intelligent, meaning there is no implementation of a command handler in the DPYDumbCommand class. All instances of this class are interpreted by a specialised DPYDeputy class.

```
package deputy;

import java.lang.*;
import java.util.*;
```

```
import java.io.*;

public class DPYDumbCommand extends DPYCommand {
    int      fCommandId;
    Object[] fParameters;

    public DPYDumbCommand(int commandId, Object[] parameters) {
        fCommandId = commandId;
        fParameters = (parameters != null) ? parameters : new Object[0];
    }

    public DPYDumbCommand(int commandId) {
        this(commandId, null);
    }

    public int getCommandId() {
        return fCommandId;
    }

    public int getNumberOfParameters() {
        return fParameters.length;
    }

    public Object[] getParameters() {
        return fParameters;
    }
}
```

4.8.8.4 The “DPYReply” Class

The DPYReply class is very similar to the DPYDumbCommand class. Instead of commands, instances of the DPYReply class hold replies. The attributes of a reply object also include the command object, that has caused the reply. DPYReply objects are exclusively used for commands that have to reply to their delegating object (DPYDelegator instances).

```
package deputy;

import java.lang.*;
import java.util.*;
import java.io.*;

public class DPYReply {
    int      fReplyId;
    Object[] fParameters;
    DPYCommand fDPYCommand = null;

    public DPYReply(int replyId) {
        this(replyId, (Object[]) null);
    }

    public DPYReply(int replyId, Object[] parameters) {
        fReplyId = replyId;
        fParameters = parameters;
    }

    public int getReplyId() {
```

```
        return fReplyId;
    }

    public int getNumberOfParameters() {
        return fParameters.length;
    }

    public Object[] getParameters() {
        return fParameters;
    }

    void setDPYCommand(DPYCommand command) {
        fDPYCommand = command;
    }

    public DPYCommand getDPYCommand() {
        return fDPYCommand;
    }
}
```

4.8.8.5 The “DPYDelegator” Interface

The DPYDelegator interface has to be implemented in Delegator classes that need to get a reply after a delegated command has been executed. If no replies are required, Delegator classes have not to implement the DPYDelegator interface.

```
package deputy;

public interface DPYDelegator {
    public abstract void handleDPYReply(DPYReply reply);
}
```

4.8.8.6 The “DPYDeputy” Class

Due to the length of the Java code for the DPYDeputy class, the listing of this class is shown in Appendix C, page 154.

The DPYDeputy class is the actual handler of commands. It contains two command queues (Vector tables) that handle commands for parallel and sequential execution. The number of threads for concurrent executing “parallel” commands can be limited at construction time and modified at run time. The class offers three delegation methods (doCommand (...)) with different parameters.

For handling “dumb” commands, the DPYDeputy class has to be specialised and a DPYDumbCommand handler has to be implemented for interpretation of the used commands (handleDPYDumbCommand(...) method). The final implementation of the run() method and a few private handler methods are responsible for queuing, de-

queuing and executing delegated commands. The original source code describes all methods and parameters in detail (available at <http://www.cs.uow.edu.au/people/ro04/>).

4.8.9 Sample Code

This section shows a very simple application program using the Deputy pattern. The Subject (see Figure 4.7 and Figure 4.9) is in this example simply the standard output stream. The example creates two commands that are proceeded parallel. The commands handlers are implemented with some sleep cycles to illustrate the immediate return after delegation and to show the parallel execution. The following code shows the two example classes (a Delegator and a Deputy).

```
import java.lang.*;
import java.util.*;
import java.io.*;

import deputy.*;

class DPYDumbExampleDeputy extends DPYDeputy {
    private void sleep(int time) {
        try {
            Thread.currentThread().sleep(time);
        } catch (Exception e) {
        }
    }

    protected DPYReply handleDPYDumbCommand(DPYDumbCommand command) {
        int time, count;
        if (command.getCommandId() == 1) {
            time = 666;
            count = 5;
        } else {
            time = 1000;
            count = 3;
        }
        for(int i = 1; i <= count; i++) {
            sleep(time);
            System.out.println("Handler Command #" +
                               command.getCommandId() + " Count=" + i);
        }
        return new DPYReply(command.getCommandId());
    }
}

public class DPYDumbExample implements DPYDelegator {

    public static void main(String[] args) {
        DPYDumbExample delegator = new DPYDumbExample();
        DPYDumbExampleDeputy deputy = new DPYDumbExampleDeputy();
        System.out.println("Point A");
        deputy.doDPYCommand(new DPYDumbCommand(1), false, delegator);
        System.out.println("Point B");
        deputy.doDPYCommand(new DPYDumbCommand(2));
        System.out.println("Point C");
    }
}
```

```
public void handleDPYReply(DPYReply reply) {
    System.out.println("Reply with ReplyId=#" + reply.getReplyId());
}
}
```

The following output shows that the main function runs straight through the code, without stopping when it delegates the two commands. Then, the two commands (one with reply, the other without reply) are executed parallel. The program at the end, because the queue handlers of the deputy are still running after the commands have been executed.

```
$ java DPYDumbExample
Point A
Point B
Point C
Handler Command #1 Count=1
Handler Command #2 Count=1
Handler Command #1 Count=2
Handler Command #2 Count=2
Handler Command #1 Count=3
Handler Command #1 Count=4
Handler Command #2 Count=3
Handler Command #1 Count=5
Reply with ReplyId=#1
```

4.8.10 Variations

This description of the Deputy pattern focuses on a conventional form of delegating commands (method calls) to a *Deputy* object. However, every implementation may use variations of this pattern with extended or reduced functionality. Variations could include:

- **Cancelling of delegated commands:** Applications may need to cancel delegated commands. Therefore, a variation may offer a member function to cancel a registered command. Commands in process would be killed by deleting their threads. Commands that are not started would simply be deregistered from the command list.
- **Passing of primitive data types as command parameters (for “dumb” commands):** The shown implementation offers parameter transfer of delegated commands by an array of objects. Therefore, primitive data types need to be converted into objects and transferred as an array of objects. The Java programming language supports such conversions between objects and primitive

data types. In other programming languages it could be helpful, to transfer a second parameter array that is able to deliver primitive data types.

4.8.11 Known Uses

Derivatives of the Deputy pattern have been used in various applications. The pattern occurs often, if jobs are passed from one thread to another. However, the generalised model of handling command delegation in a general base class (Deputy) is faced more rarely. The base class *Deputy* provides that the application specific code has not to deal with thread generation and queuing of delegated commands.

4.8.12 Related Patterns

The Deputy pattern can be used with most other patterns that allow thread-safe implementation. Here are some examples how the Deputy pattern may be used with other patterns.

- An **I/O-Dispatcher** [Schm+94, Wein+88] object shown in the patterns presented by Neil Gray [Gray95] may use the Deputy pattern to proceed events in separate threads.
- The **Proxy** pattern [Gam+95] may be used with the Deputy pattern to update remote objects asynchronously.
- The **Command** pattern [Gam+95] may be used with the Deputy pattern to let commands be proceeded by other threads than the initiator thread.

4.9 Summary

This chapter has described an advanced model for object sharing across address spaces, that has been based on the previously explained Simple Shared Object pattern. A brief discussion shows why the advanced model has been designed for an implementation in the Java programming language. The advanced model has been named as the “Simple Shared Object Request Broker” (SSORB).

The design of the three main components of the SSORB system has been described in detail. These components are a request broker, object servers and client applications.

The major object within the SSORB system and their responsibilities have been explained by using object diagrams. Moreover, a class for handing object references across address spaces and the usage of “well-known” object lists have been explained for the SSORB system.

The last part of this chapter describes a pattern for delegating commands to a multi-threaded deputy object. This “Deputy” pattern has been used within the SSORB system in all three main components. The pattern shows an implementation and a sample application in the Java programming language.

This chapter has been focused on the design of the SSORB system. Implementation code is only shown for the Deputy pattern. The next chapter will explain the implementation of the SSORB system and an application using the SSORB system in the Java programming language.

Chapter Five

5. Implementation of the SSORB System

5.1 Overview

The design of the SSORB system has been described in detail in chapter 4. The design of the system is relatively programming language independent. This chapter presents a prototype implementation using the Java programming language. The chapter is divided into three major parts.

The first part, section 5.2, describes general issues of the implementation such as naming conventions and the use of pseudo code. Then, the classes that are used by all components of the system are presented using Booch style diagrams [Booch94].

The second part, section 5.3, describes those classes used in the broker of the SSORB system. The broker classes include a main class that can be executed by a Java interpreter. Therefore, the broker is able to run on any system on which a Java interpreter has been implemented.

The third part, section 5.4, describes the classes that are provided by the SSORB system to create object servers and client applications. These classes are used to implement object servers and client applications.

This chapter does not describe a specific application using the SSORB system. Rather, it focuses on the description of the application independent classes that can be reused in various types of CSCW applications. A sample application using the described classes is described in chapter 6.

5.2 The Implementation in General

The SSORB system has been implemented in the Java programming language. The Java development kit version 1.0 offers a number of standard packages (libraries) such as “`java.net`” (network handling classes) and “`java.awt`” (Abstract Windows Toolkit). Java offers the possibility for extending its functionality with additional packages. One such additional package has already been described in the implementation part of the Deputy pattern (see section 4.8.8). The SSORB system is also implemented as a Java package named “`sso`” and it uses the “`deputy`” package for command delegation.

One class supporting the implementation of a SSORB application as an Applet into a Web page, is implemented in a sub-package named “`sso.SSOApplet`”. A few other classes that support specific issues for presentation of SSORB components as graphical elements are implemented in a sub-packages named “`sso.awt`”.

5.2.1 Name Conventions

In order to avoid class name mismatches in applications using the SSORB system, all class names begin either with the initials “SSO” or “SSORB”. The initials “SSO” are used for all classes that belong to object servers and client applications. The other initials “SSORB” are used for all classes that belong to the broker of the SSORB system. The classes that are used for both, the broker and the customers (object servers and client applications) begin also the initials “SSO”.

Furthermore, the implementation uses name conventions for data members within classes. All static data members (class variables) begin with a lower case ‘s’. All non static data members (instance variables) begin with a lower case ‘f’. Moreover, if a data member name consists of more than one word, the first letter of every word is capital. For instance, a static data member “receptionist list” would be named as “`sReceptionistList`” and a non static data member “object identifier” would be named as “`fObjectIdentifier`”.

All data members that are declared as constant, begin with a lower case ‘k’. Therefore, a constant message identifier for an update message would become the name “`kUpdate`”.

5.2.2 Pseudo Code

Due to the size of the code for the SSORB system, it is unfortunately not possible for this thesis and appendices to include all code from the SSORB system. However, all classes are available via the World Wide Web at address <http://www.cs.uow.edu.au/people/ro04/>.

This chapter does present some of the code from more important classes and methods. In some parts, the Java code has been replaced with pseudo code for brevity. All pseudo code shown in the listings starts with a '@' character. As a result, the pseudo code lines can easily be distinguished from the normal Java code lines.

5.2.3 Complete Class Diagram of the SSORB System

The SSORB system consists of a number of classes that are provided in an additional Java package. The whole SSORB system is based on the exchange of message objects between components. Therefore, a class called SSOMessage is one of the key classes and its instances are used by most of the other classes within the system. This class is discussed in detail later (section 5.2.4).

The Java system provides a special class for exception handling called Exception. The SSORB system provides a specialised Exception class that is named as SSOException. All exceptions within the SSORB system are handled by either this class or by its specialised subclass SSOREplyException.

As described in chapter 4, the SSORB system uses the Deputy pattern for command delegation. Only the classes DPYDeputy and DPYDumbCommand (and therefore also its base class DPYCommand) are used in the implementation of the SSORB system.

The classes SSOMessage, SSOException, DPYDeputy and DPYDumbCommand are the classes that are used on both the broker side and the customer side (object servers and client applications).

All classes of the SSORB system are shown in Figure 5.1. The figure shows the relatively flat class hierarchy of the system. Some classes such as SSOException and SSOMessage have relationships with most other classes within the system. Besides of the inheritance relationships, the diagram shows only a few of the more important "use"

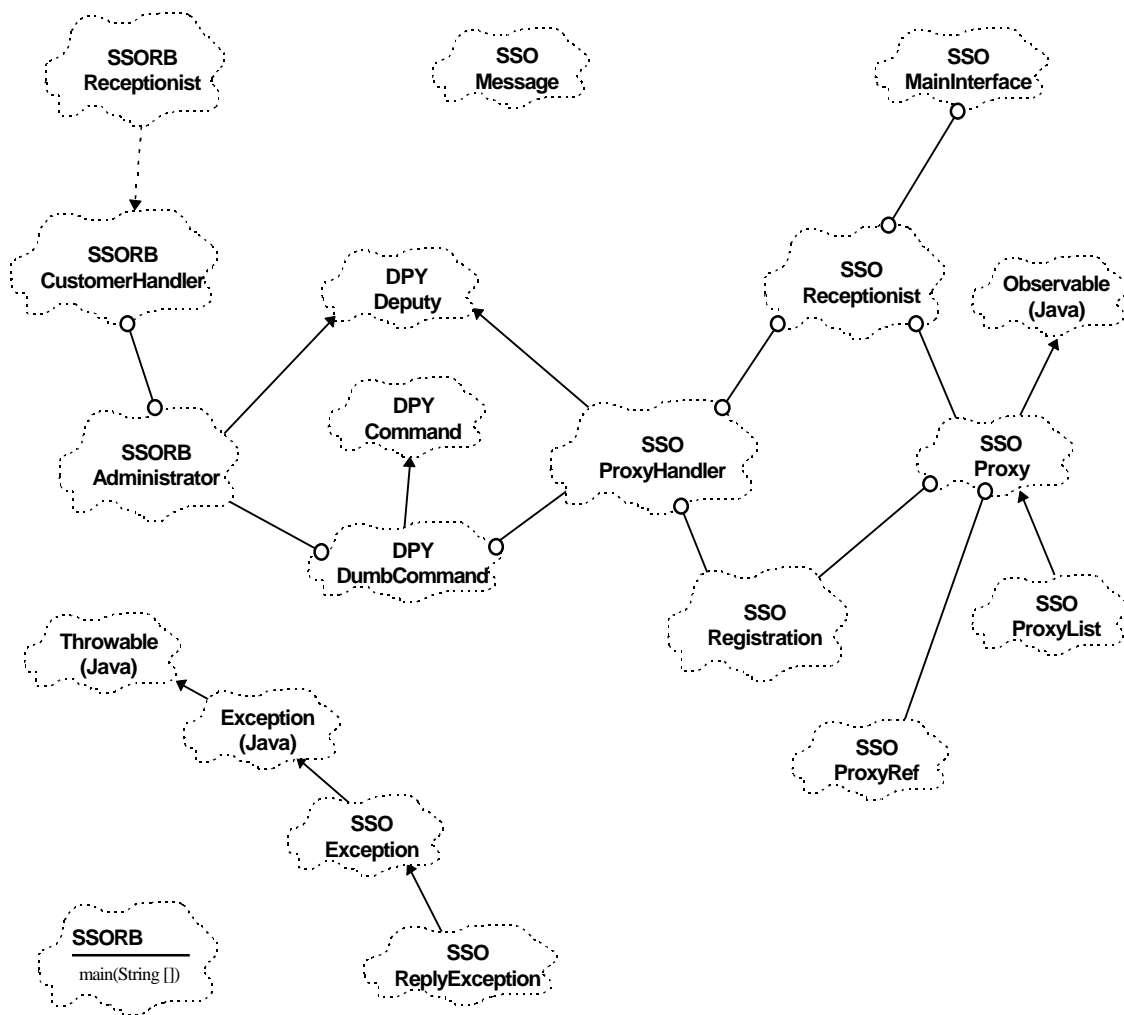


Figure 5.1: The classes within the SSORB system and their major relationships

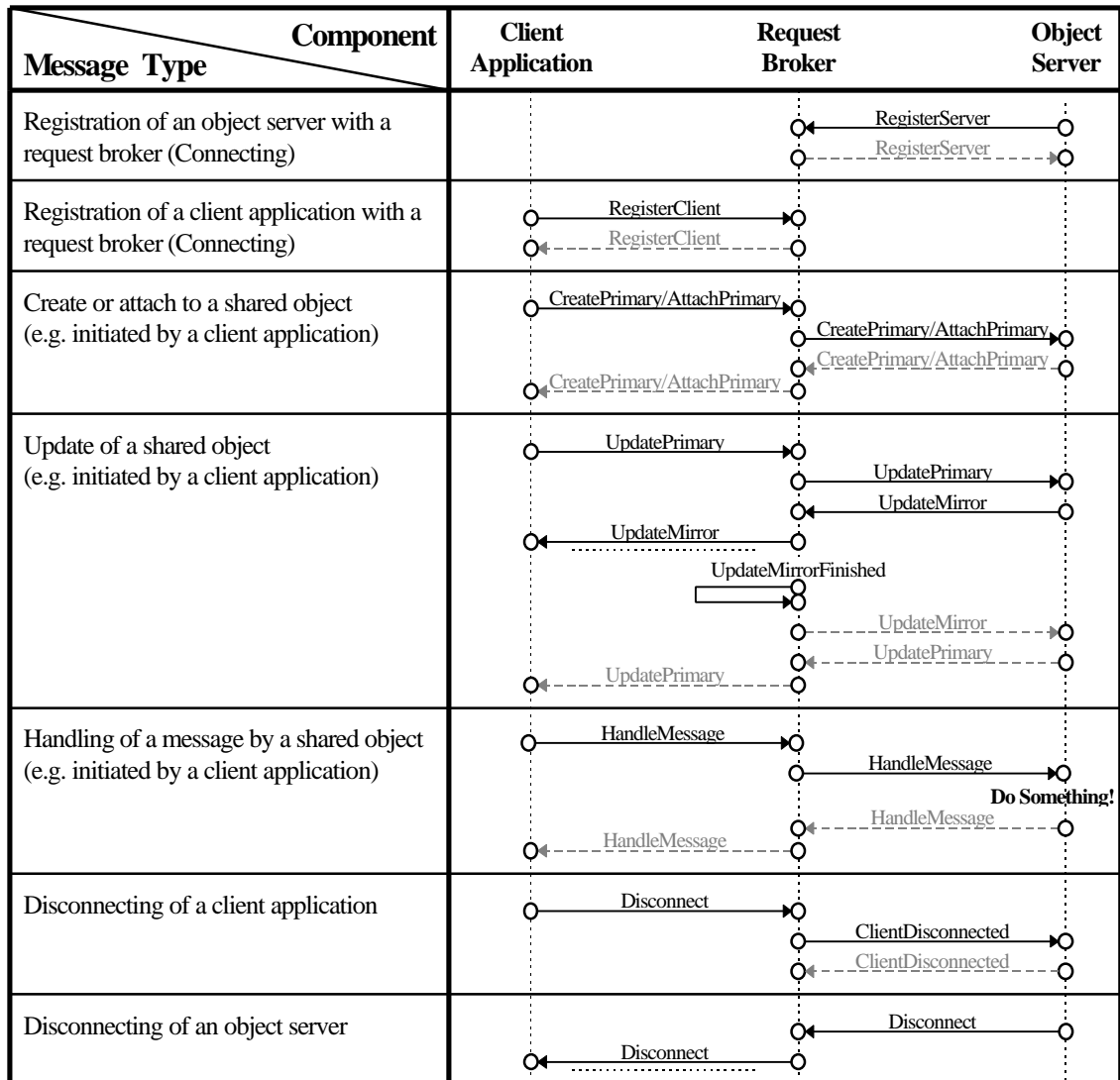
and “instantiation” relationships between classes (showing all relationships would make the diagram overly complex).

The Java programming language does not support multiple inheritance. Therefore, the implementation of the SSORB system is strictly hierarchical. However, some parts of the system use interface classes that allow a specialised class to implement interfaces to other classes. This technique is explained in detail in the description of the Java programming language [Gosl+95].

5.2.4 Inter-Component Communication with Message Objects

Messages are exchanged as instances of class `SSOMessage`. All possible message types within the SSORB system are predefined by constant identifier numbers (`kTag...` declarations in the class `SSOMessage`). The following list shows all used message types:

- **kTagRegisterServer:** Registers a new object server at a request broker, this is the first message that a object server sends to its broker. The message includes a name that identifies the connecting object server (server names must be unique). This message type is only sent from an object server to its request broker.
- **kTagRegisterClient:** Registers a new client application at a request broker, this is the first message that a client application sends to its broker. The message includes a name that identifies the object server that the connecting client application wishes to establish a connection. This message type is only sent from a client application to its request broker.
- **kTagCreatePrimary:** Creates a primary copy of a shared object. Both, object servers and client applications are able to create new shared objects. The content of the message contains the class name (subclass of `SSOProxy` class) and optionally an initialisation stream for the new shared object. This message is usually sent from a client application to its object server. If an object server creates a shared object, then the object server sends this message to itself.
- **kTagAttachPrimary:** Attaches to a primary copy of a shared object. Both, object servers and client applications are able to attach to already existing shared objects. The content of the message contains the class name (subclass of `SSOProxy` class) and the shared object identifier for the already existing shared object. This message is usually sent from a client application to its object server. If an object server attaches to a shared object, then the object server sends this message to itself.
- **kTagDetachPrimary:** Detaches from a primary copy of a shared object. When a client application does not longer need a copy of a shared object, it sends this message to its object server.



Legend:

- Request A request message
- Multiple Request A request message to all client applications for a specific shared object
- Reply A reply message

Figure 5.2: Interaction table for message exchanges within the SSORB system

- **kTagUpdatePrimary:** Updates a primary copy of a shared object. Whenever an update to a shared object is due, a update primary message is sent to the object server. The message can be initiated from both, object servers and client applications.
- **kTagUpdateMirror:** Updates a mirrored copy of a shared object. After an update has been made to the primary copy of a shared object, an update mirror

message is sent to the request broker. Then, the request broker forwards the replication message to all clients having a copy of the shared object.

- **kTagUpdateMirrorFinished:** Update of mirrored copies (in clients) completed. This message is only used internally by the request broker. When the broker queues update mirror messages for different clients, it also queues as last message an “update mirror finished” message. After all client applications have been updated, this message is the trigger to commit the initial sent update mirror request to the object server.
- **kTagHandleMessage:** Handles a message (by the primary copy of a shared object). Sometimes, there is a need to send a message with application specific content from a mirrored copy of a shared object (client application) to its primary copy (object server). This message is used for such message exchanges.
- **kTagDisconnect:** Disconnects a customer. This message is sent when a customer (object server or client application) no longer needs the service of its request broker. The broker deregisters the customer from its internal hash tables when receiving this message.
- **kTagClientDisconnected:** When a client application disconnects from its request broker, the broker informs the responsible object server by sending this message. Thus, the object server is able to delete all relationships to the disappeared client application and its shared object.

Figure 5.2 illustrates the collaboration between object servers, client applications and a broker. It should be noted that messages such as create, attach and update can also be initiated by an object server. The figure, however, shows only examples where a client application initiate such messages.

5.2.5 Exception Handling within the SSORB System

The Java programming language is strict in the manner of exception handling. An implemented method has either to catch all exceptions or it has to declare that it throws uncaught exceptions. This strict exception handling can be seen as disturbing in small implementations. On the other hand, it is probably the only way to tame programmers

from ignoring “some never occurring” exceptions, that often have been the reason for unstable software systems. The SSORB system tries to catches all exceptions that can occur at run time. Most exceptions are caught and handled, some major run time exceptions are converted into SSOException objects that have to be handled the an application program that uses the SSORB system.

The two specialised Exception classes (SSOException and SSORReplyException) are implemented as follows:

5.2.5.1 The “SSOException” Class

```
package sso;

import java.lang.*;

public class SSOException extends Exception {
    public SSOException() {
    }
    public SSOException(String detail) {
        super(detail);
    }
}
```

5.2.5.2 The “SSORReplyException” Class

The class SSORReplyException includes an attribute that identifies an application specific reply identifier. An object server is able to reply to a “kTagHandleMessage” message with a positive reply code for a successfully handled message and negative reply codes for a unsuccessfully handled message. Then, the reply code can be checked by the method “checkReply()” of a specialised SSOProxy class that throws a SSORReplyException if necessary (see description of the method “checkReply()” in section 5.4.1.5 and 6.2.3.1).

```
package sso;

import java.lang.*;

public class SSORReplyException extends SSOException {
    int fReplyCode = 0;
    public SSORReplyException() {
    }
    public SSORReplyException(int replyCode, String detail) {
        super(detail);
        fReplyCode = replyCode;
    }
    public void setReplyCode(int replyCode) {
        fReplyCode = replyCode;
    }
}
```

```
}  
public int getReplyCode() {  
    return fReplyCode;  
}  
}
```

5.3 The Broker Implementation

The broker of the SSORB system consists of four major classes in addition to the DPY-Deputy and DPYDumbCommand classes of the Deputy pattern, and the SSOException and SSOMessage classes.

Two of the four major classes are implemented as singleton's (see Singleton pattern in [Gam+95]). These singleton classes are the **SSORBReceptionist** and the **SSORBAdministrator** classes. The **SSORBCustomerHandler** class is instantiated for every connected customer (object servers and client applications) by the SSORBReceptionist object. The **SSORB** class provides an implementation of the "main()" method. The broker is totally application independent and can be executed by using the Java interpreter.

5.3.1 The Java Code for the Broker Classes

The classes for the request broker of the SSORB system (SSORBCustomerHandler, SSORBReceptionist, SSORBAdministrator and SSORB) are described in the following subsections:

5.3.1.1 The "SSORBCustomerHandler" Class

Figure 5.3 shows the structure of the SSORBCustomerHandler class. The class defines some data shared by all instances. The common data include the administrator object ("sSSORBAdministrator") and the hash tables "sWaitingForReplyThreadList" and "sWaitingForReplyMessageList" that are used to record requests sent to customers. Another shared data element, "sRequestIdCounter", provides a unique integer for each request and that is only accessed by the method "getNextRequestId()".

The class has one constructor that needs a socket object as parameter. The class is exclusively instantiated by the SSORBReceptionist object, listening for connecting cus-

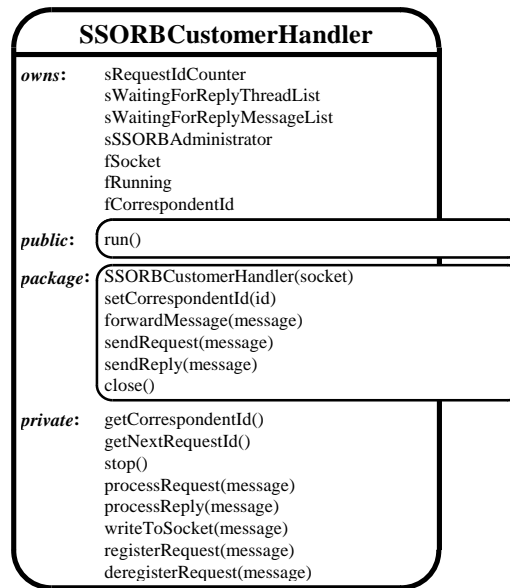


Figure 5.3: The structure of the SSORBCustomerHandler class

tomers. The constructor sets the data member “fSocket” and creates its own thread (“fRunning”) for handling the connecting customer. The code for the constructor is implemented as follows:

```

SSORBCustomerHandler(Socket socket) {
    fSocket = socket;
    fRunning = new Thread(this, "SSORBCustomerHandler");
    fRunning.start();
}
  
```

Every customer handler has its unique correspondent identifier. The methods “setCorrespondentId()” and “getCorrespondentId()” are responsible to access the data member “fCorrespondentId”. This identifier is set by the administrator when the “register” message is received.

The private method “writeToSocket()” sends the context of a message to a socket connected to a customer. This method is called by the methods “sendRequest()” and “sendReply()”. The method “sendReply()” simply sets the message tag for reply (<0) and calls the method “writeToSocket()”. The method “sendRequest()” is implemented as follows:

```

void sendRequest(sso.SSOMessage msg) {
    @ Set SenderId of message to 0 (0 represents the broker)
    @ Set ReceiverId of message to fCorrespondentId
    @ IF customer needs to reply THEN
    @ Register the request "registerRequest()"
  
```

```

@      Send request message "writeToSocket()"
@      'Sleep' (suspend the thread)
@      (after wake up) Deregister the request "DeregisterRequest()"
@      ELSE
@      Send message "writeToSocket()"
@      ENDIF
@
}

```

The method "run()" loops, waiting for and processing messages from a customer. When a message has been received, either the method "processRequest()" or the method "processReply()" is called (depending on the type of message). If a communication error occurs, the customer handler stops by calling the method "stop()". The code for the "run()" method is implemented as follows:

```

public void run() {
@   LOOP until error occurs
@       Read a message
@       Call "processRequest()" or "processReply()"
@   END LOOP
@   Deregister this handler from the Administrator
}

```

An outline for the method "processRequest()" is:

```

private void processRequest(sso.SSOMessage msg) {
@   IF request is for another customer THEN
@       Forward the request via Administrator - "forwardMessageTo()"
@   ELSE
@       Delegate the request to the Administrator "doDPYCommand()"
@   ENDIF
}

```

The method "processReply()" wakes up the suspended threads waiting for the received reply. The method is implemented as follows:

```

private void processReply(sso.SSOMessage msg) {
@   IF reply is for another customer THEN
@       Forward the reply via Administrator - "forwardMessageTo()"
@   ELSE
@       Search the thread that is waiting for the reply
@       Prepare the reply for the waiting thread by
@           putting it into the "sWaitingForReplyMessageList"
@       Resume the waiting thread (WAKE UP!)
@   ENDIF
}

```

The method "forwardMessage()" is called by the administrator. It simply calls the method private method "writeToSocket()". The method "close()" is used by the administrator to close customer handlers for client applications when their

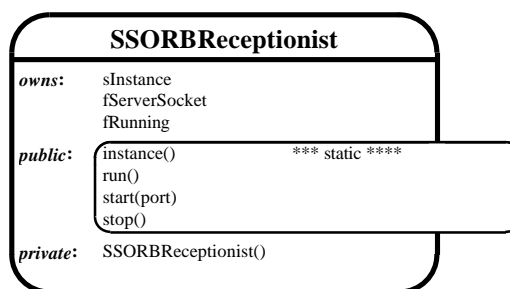


Figure 5.4: The structure of the SSORBReceptionist class

object server disappears. Lastly, the method “stop()” simply stops a customer handler and it is called if a communication error occurs or if a customer disconnects.

5.3.1.2 The “SSORBReceptionist” Class

Figure 5.4 shows the structure of the SSORBReceptionist class. This class is implemented as a singleton; its instance can only be accessed via the static method “instance()” (see Singleton pattern in section 2.3.3). The static data member “sInstance” holds the singleton object. The constructor is declared as private and is therefore not accessible for instantiation.

The receptionist is responsible for accepting customers; it instantiates a SSORBCustomerHandler for each customer. The method “start()” is used to start the accepting process, the argument defines its TCP/IP port. The method “start()” is:

```

public void start(int port) throws sso.SSOException {
    if (fRunning != null)
        throw new sso.SSOException("receptionist already started");
    try {
        fServerSocket = new ServerSocket(port);
        fRunning = new Thread(this, "SSORBReceptionist");
        fRunning.start();
    } catch (IOException e) {
        fRunning = null;
        throw new sso.SSOException(
            "connection to receptionist socket failed");
    }
}
  
```

The method “stop()” stops the receptionist thread:

```

public void stop() throws sso.SSOException {
    if (fRunning == null)
        return;
  
```

```

fRunning = null;
if (fServerSocket != null) {
    try {
        fServerSocket.close();
    } catch (IOException e) {
        throw new sso.SSOException(
            "could not close SSORBReceptionist socket");
    }
}
}
}

```

The method “run()” is called after the receptionist thread has been started. This method contains the actual accept loop and creates the instances of class SSORBCustomerHandler. The method is:

```

public void run() {
    if (fServerSocket == null)
        return;
    while(fRunning != null) {
        try {
            Socket s = fServerSocket.accept();
            SSORBCustomerHandler cHandler = new SSORBCustomerHandler(s);
        } catch (IOException e) {
        }
    }
}
}

```

5.3.1.3 The “SSORBAdministrator” Class

Figure 5.5 shows the structure of the SSORBAdministrator class. Like the SSORBReceptionist class, this class is implemented as a singleton. Therefore it has a private constructor, a static access method “instance()” and it holds its own singleton object in the static data member “sInstance”.

The class SSORBAdministrator creates a unique identifier number (by using its data member “fCorrespondentIdCounter”) for every customer handler. The unique identifiers are created by the private method “getNextCorrespondentId()”. The class owns a number of hash tables for fast location of correspondents (customer handlers). These hash tables include relationship tables such as client applications to object server, object server to client applications, object server name to object server identifier.

The class is implemented as a specialised DPYDeputy class (see Deputy pattern in section 4.8). The singleton instance is able to handle “dumb” commands, that are actually messages delegated by customer handler objects. Therefore, the SSORBAdminis-

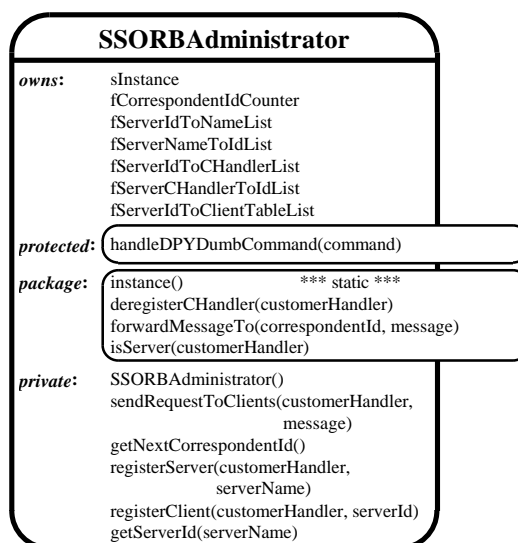


Figure 5.5: The structure of the SSORBAAdministrator class

trator class implements the handler for “dumb” commands (`handleDPYDumbCommand()`). Every delegated command has two parameters that are the received message and the command handler object (provided as an array of objects). The command handler method “`handleDPYDumbCommand()`” is implemented as follows:

```

protected DPYReply handleDPYDumbCommand(DPYDumbCommand cmd) {
    try {
        Object[] objArr= cmd.getParameters();
        sso.SSOMessage msg = (sso.SSOMessage) objArr[0];
        SSORBCustomerHandler cHandler = (SSORBCustomerHandler) objArr[1];
        switch(msg.getTag()) {
            case sso.SSOMessage.kTagRegisterServer:
                @ Register a new object server by calling "registerServer()"
                @ Set correspondent identifier of new connected object server
                @ by calling its "setCorrespondentId()" method
                @ Set message for replying "registered with server-id 'x'"
            case sso.SSOMessage.kTagRegisterClient:
                @ Register a new client application
                @ by calling "registerClient()"
                @ Set correspondent identifier of new connected client appl.
                @ by calling its "setCorrespondentId()" method
                @ Set message for replying "registered with server-id 'x'
                @ and client-id 'y'"
            case sso.SSOMessage.kTagUpdateMirror:
                @ Send update to all clients having a copy of the specified
                @ shared object by calling "sendRequestToClients()"
            case sso.SSOMessage.kTagUpdateMirrorFinished:
                @ Set message for replying "update mirror finished"
            default:
                @ Set message for replying "unknown command"
        }
        if (cHandler != null)
            cHandler.sendReply(msg);
    } catch (Exception e) {
  
```

```

        System.out.println("Administrator error occurred\n" +
                           "Error: " + e.getMessage());
    }
    return null;
}

```

The private method “sendRequestToClients()” is only used to send an “update mirror” message from an object server to all clients that hold a mirrored copy. The method creates a separate “dumb” command for every client application connected to the initiating object server. Then it creates a “last command” used to notify the object server when an “update mirror” process has finished. The method is implemented as follows:

```

private void sendRequestToClients(SSORBCustomerHandler cHandler,
                                 sso.SSOMessage msg)
    throws sso.SSOException {
    @ Create a "dumb" command for every client application that
    @ needs to get the message
    @ Queue all commands to this administrator object itself
    @ by calling "doDPYCommand()"
    @ Queue as last command a message "update mirror finished"
    @ by calling "doDPYCommand()"
}

```

The private method ”registerServer()” registers a new object server recording required information in the internal hash tables. The method is:

```

private synchronized long registerServer(
    SSORBCustomerHandler cHandler,
    String serverName) {
    @ Check if the server name already exist. If so, return an negative
    @ identifier number to indicate an error
    @ Create a new identifier for the new object server
    @ by calling "getNextCorrespondentId()"
    @ Register the new object server with necessary hash tables
    @ Return the new identifier for the new object server (positive)
}

```

The private method ”registerClient()” registers a new client application with the its object server. The internal hash tables are used to register the client-server relationships. The method is implemented as follows:

```

private synchronized long registerClient(
    SSORBCustomerHandler cHandler,
    long serverId) {
    @ Check if the given serverId exists as object servers
    @ If not, return a negative value to indicate an error
    @ Create a new identifier for the new client application
    @ by calling "getNextCorrespondentId()"
    @ Register the new client application with necessary hash tables
    @ Return the new identifier for the new client application
}

```

```
@
}
        (positive)
```

Every registered object server has a server name and a unique identifier. The method “`getServerId()`” searches the unique identifier for a specified server name. If the specified server name does not exist, a negative value is returned (“`kReplyServerDoesNotExists`”). The private method is implemented as follows:

```
private synchronized long getServerId(String serverName) {
    if (!fServerNameToIdList.containsKey(serverName))
        return sso.SSOMessage.kReplyServerDoesNotExist;
    return ((Long) fServerNameToIdList.get(serverName)).longValue();
}
```

Customer handler objects frequently need to forward messages to other customer handlers. For instance client applications exchange messages with the object server to which they are connected. The customer handler objects themselves have, however, no direct connection to their correspondents. Therefore, they use the administration to forward messages. The method “`forwardMessage()`” is used for forwarding messages from one customer handler to another and it is implemented as follows:

```
int forwardMessageTo(long id, sso.SSOMessage msg) {
@   Search corresponding customer handler by using the internal
@       hash tables
@   Exchange the input stream of the message with the output stream
@   Forward the message by calling the method “forwardMessage()”
@       of the receiving customer handler
}
```

The method “`isServer()`” is currently used solely for debugging purposes. The method returns a boolean showing whether the customer handler object is an object server handler or a client application handler. The method “`isServer()`” is implemented as follows:

```
boolean isServer(SSORBCustomerHandler cHandler) {
    return fServerCHandlerToIdList.containsKey(cHandler);
}
```

Finally, the method “`deregisterCHandler()`” is responsible for deregistration of disappearing customer handlers. The method deletes all relationships involving the disappearing customer handler that previously were recorded in the internal hash tables. If the disappearing customer handler was responsible for an object server, it closes all client application customer handlers that were connected to that server. If the disap-

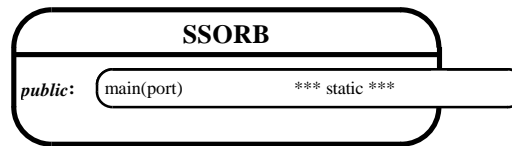


Figure 5.6: The structure of the SSORB class

peating customer handler was responsible for a client application, it informs the responsible object server that a client application has disconnected. The method “deregisterCHandler()” is implemented as follows:

```

synchronized void deregisterCHandler(SSORBCustomerHandler cHandler) {
@   IF cHandler is a handler for an object server THEN
@       Close all client application handlers that are connected to
@       the disappearing object server by calling their
@       method "close()"
@       Remove the disappearing customer handler from the internal
@       hash tables
@   ENDIF
@   IF cHandler is a handler for a client application THEN
@       Send a message "kTagClientDisconnected" to the object server
@       of the disappearing client application
@       Remove the disappearing customer handler from the internal
@       hash tables
@   ENDIF
@
}
  
```

5.3.1.4 The “SSORB” Class

The SSORB class is the main class of the request broker of the SSORB system. It only implements the static “main()” method (see Figure 5.6). The method evaluates one optional parameter, that is the TCP/IP port number on which the broker should listen for customers. If no parameter is specified, the default TCP/IP port number 270266 is used¹. The class is implemented as follows:

```

public class SSORB {
    static int    sOrbPort = 270266;
    static String sTitle   = "Simple Shared Object Request Broker " +
        "(SSORB)\nCopyright (c) 1996 by " +
        "Robert Ott";

    static public void main(String args[]) {
        try {
            System.out.println(sTitle);
            switch(args.length) {
                case 1:
                    sOrbPort    = new Integer(args[0]).intValue();
            }
        }
    }
}
  
```

¹ 270266 is not a magic socket number, it is just the birthday of the author of the SSORB system.

```
        default:
            SSORBReceptionist.instance().start(sOrbPort);
            System.out.println("Broker started at port " + sOrbPort +
                               "...");
            break;
    }
} catch(Exception e) {
    System.out.println("Oyi, something bad happend, " +
                       "I hope you can fix it! ;-)" + "\n" +
                       "==>" + e.toString());
}
}
```

5.4 The Customer Implementation - Object Servers and Client Applications

Object servers and client application use the same classes for sharing objects. Three classes are implemented as singleton's. The first singleton is the SSORReceptionist class which is responsible for message exchange with the broker. The second singleton is the SSOProxyHandler class (a specialised DPYDeputy class). It is responsible for handling commands (messages) delegated by the receptionist. The third singleton is the SSORRegistration class that is responsible for maintaining a list of sharable proxies in client applications and object servers.

An interface called SSOMainInterface defines two abstract methods that have to be implemented in the main program of each object server and each client application. The SSOProxy class is partially abstract; it has to be specialised for every application specific sharable proxy. One such specialisation is the SSOProxyList class that handles lists of sharable proxies. Finally, the SSOProxyRef class is able to handle proxy references in a way that the references can be exchanged over address spaces.

5.4.1 The Java Code for the Application Independent Object Server and Client Application Classes

The classes that are used by applications using the SSORB system are described in the following subsections:

5.4.1.1 The "SSOMainInterface" Interface

Figure 5.7 shows the interface SSOMainInterface. The two defined methods need to be implemented in a specialised class for each application specific object server and client

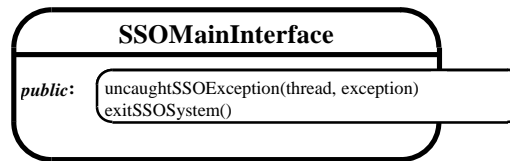


Figure 5.7: The structure of the interface SSOMainInterface

application. The first method “`uncaughtSSOException()`” is called if a major exception occurs during run time. A specialised and therefore application specific SSOMainInterface class is needed as a parameter to connect a SSORceptionist to the request broker. The second method “`exitSSOSystem()`” is defined a central point in a client application or an object server and it is called when the program finished. The specialised implementation of this method usually disconnects the receptionist from the broker. The interface is implemented as follows:

```

public interface SSOMainInterface {
    public abstract void uncaughtSSOException(Thread t,
                                             sso.SSOException e);
    public abstract void exitSSOSystem();
}
  
```

5.4.1.2 The “SSORceptionist” Class

The SSORceptionist class is responsible for communications with the broker. The implementation of this class is very similar to its class SSORBCustomerHandler on the broker side (see section 5.3.1.1). The singleton instance of the class SSORceptionist is responsible for the following activities:

- Connecting to the broker
- Disconnecting from the broker
- Sending requests to the broker.
- Receiving requests from the broker. After the arrival of a request, the receptionist delegates the request as a “DPYDumbCommand” object to the singleton instance of the class SSOProxyHandler.

Figure 5.8 shows the complete structure of the `SSORceptionist` class. Most methods are similar in implementation to those in the class `SSORBCustomerHandler`. Therefore, this section describes only a few of the methods of the class.

The class provides two methods for connecting to the broker. The method `connectAsServer()` is used by object servers and the method `connectAsClient()` is used by client applications. Both methods have the same four parameters. The first parameter `mainInterface` defines the a class that handles uncaught exceptions (by implementing the `SSOMainInterface`). The second parameter `applicationName` defines the application name (unique for each object server). The other two parameters `orbAddress` and `orbPort` define the TCP/IP address and port number where a SSORB broker is listening. The methods `connectAsServer()`, `connectAsClient()` and `connect()` are implemented as follows:

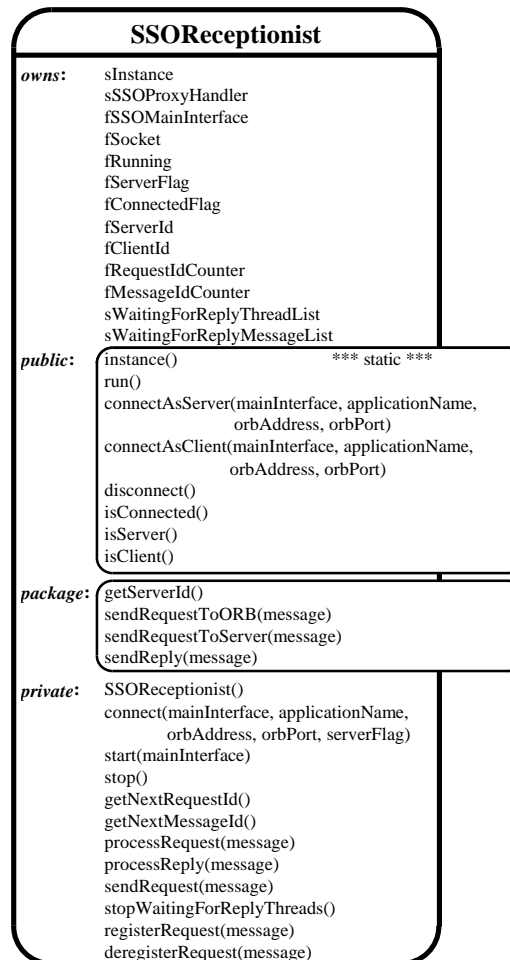


Figure 5.8: The structure of the SSORceptionist class

```

public void connectAsServer(SSOMainInterface excInt, String appName,
                           String orbAddress, int orbPort)
    throws sso.SSOException {
    connect(excInt, appName, orbAddress, orbPort, true);
}

public void connectAsClient(SSOMainInterface excInt, String appName,
                            String orbAddress, int orbPort)
    throws sso.SSOException {
    connect(excInt, appName, orbAddress, orbPort, false);
}

private synchronized void connect(SSOMainInterface excInt,
                                   String appName, String orbAddress,
                                   int orbPort, boolean serverFlag)
    throws sso.SSOException {
    @    Checking the parameters
    @    Create a new Socket object by using the given address and port-nr
    @    Start the own receptionist thrad by calling "start(...)"
    @    IF serverFlag is set THEN
    @        Send a "kTagRegisterServer" message to the broker
    @    ELSE
    @        Send a "kTagRegisterClient" message to the broker
    @    ENDIF
  
```

```

@ Set the identifiers fServerId and fClientId to the values
@   that have been received as a reply (fServerId and fClientId
@   are equal if the receptionist has been connected as server!)
}

```

Messages can be sent to the broker and to the object server by using the methods “sendRequestToORB()” and “sendRequestToServer()”. Messages from an object server to its connected client applications are first sent to the broker. Then, the broker forwards such “server to client” messages to the connected client (e.g. see “update mirror” message in section 5.2.4).

5.4.1.3 The “SSORegistration” Class

The SSORegistration class is implemented as singleton and is only accessible from the classes in the “sso” package. The class holds three data members whereby the static data member “sInstance” just holds the single instance of the registration. The data member “fObjectIdCounter” is used internally for the creation of shared object identifiers for new sharable proxy objects. The identifiers are created by the method “getNextObjectId()”. The third data member of the SSORegistration class is a hash table (instance of class “Hashtable”) that holds all sharable proxy object in the current address space.

Figure 5.9 shows the structure of the SSORegistration class. The methods “register()” and “deregister()” are used to register sharable proxies with the registration. These methods are implemented as follows:

```

void register(sso.SSOProxy proxy) {
    Long id = new Long(proxy.getObjectId());
    fAvailableProxyList.put(id, proxy);
}

void deregister(long objectId) {
    fAvailableProxyList.remove(new Long(objectId));
}

```

The method “deregisterProxiesOf()” is only used within object servers. The method is called by the SSOProxyHandler class when a client application disconnects. An object server initiates with this method the removal of all relationships to the disappearing client application. The method is implemented as follows:

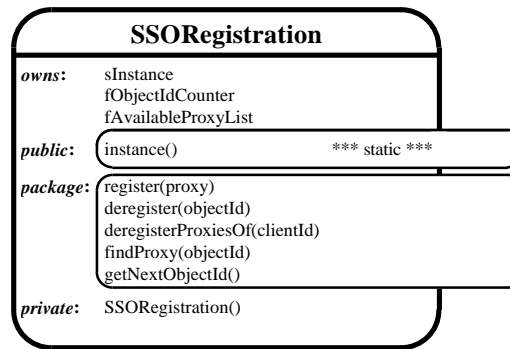


Figure 5.9: The structure of the SSORegistration class

```

void deregisterProxiesOf(long clientId) {
    for(Enumeration e = fAvailableProxyList.elements();
        e.hasMoreElements();) {
        sso.SSOProxy proxy = (sso.SSOProxy) e.nextElement();
        proxy.deregisterAttacher(clientId);
    }
}
  
```

The method “findProxy()” is used to locate already existing sharable proxy object. For instance, when a client application attaches a sharable proxy it first checks whether the required proxy is already in client’s address space. If so, it uses the already existing sharable proxy; if not, it initiates the creation of the sharable proxy via the object server. The method “findProxy()” is implemented as follows:

```

sso.SSOProxy findProxy(long objectId) {
    Long id = new Long(objectId);
    return (sso.SSOProxy) fAvailableProxyList.get(id);
}
  
```

5.4.1.4 The “SSOProxyHandler” Class

Figure 5.10 shows the structure of the “SSOProxyHandler” class. It is implemented as a specialised DPYDeputy class and is again a singleton. The SSORceptionist object delegates commands (messages) to the proxy handler object. Then, the proxy handler object evaluates the content of the message object and does the necessary operations. Most commands have to be applied to a specific sharable proxy object.

When the SSORceptionist object connects to the broker, it calls the method “setSSOMainInterface()” of the SSOProxyHandler instance. This allows it to forward uncaught exceptions to an application specific object (e.g. if the connection to the broker breaks down).

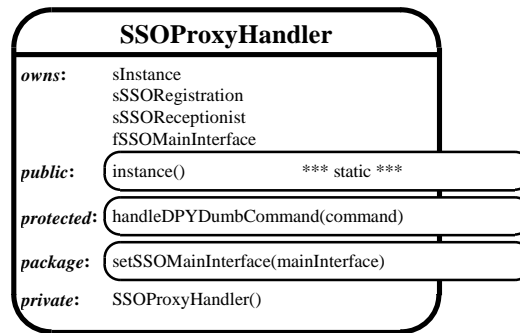


Figure 5.10: The structure of the SSOProxyHandler class

The SSOProxyHandler class implements the method “handleDPYDumbCommand()” for handing commands (messages), delegated by the SSOReceptionist instance. The method is implemented as follows:

```

protected DPYReply handleDPYDumbCommand(DPYDumbCommand cmd) {
    try {
        SSOMessage msg = (SSOMessage) cmd.getParameters()[0];
        switch(msg.getTag()) {
            case sso.SSOMessage.kTagCreatePrimary:
            @ Extract class name from message
            @ Create an instance of the required class
            @ Serialise the new instance by calling the method
            @ "serializeConcreteObject()" of the proxy
            @ Put the serialised version of the proxy into the output
            @ stream of the reply message
            case sso.SSOMessage.kTagAttachPrimary:
            @ Extract class name and object identifier from message
            @ Check with the SSORegistration wheater or not the proxy
            @ does already exist
            @ If not, create an instance of the required class. Otherwise
            @ use the already existing proxy
            @ Serialise the new instance by calling the method
            @ "serializeConcreteObject()" of the proxy
            @ Put the serialised version of the proxy into the output
            @ stream of the reply message
            case sso.SSOMessage.kTagHandleMessage:
            @ Find the required proxy by using the SSORegistration
            @ Call the method "handleRequest()" of the proxy
            @ Put the return value of the executed method as the reply
            @ identifier of the reply message
            case sso.SSOMessage.kTagUpdatePrimary:
            @ Find the required proxy by using the SSORegistration
            @ Call the method "proveUpdateConcreteObject()" of the proxy
            @ IF prove was successful THEN
            @ Call the method "updateConcreteObject()" of the proxy
            @ Call the method "afterUpdate()" of the proxy
            @ Put all client identifiers (attacher list) into a new
            @ message
            @ Put also the actual update information as an
            @ "updateMirror" message into the new message
            @ Send the new request (message) to the broker (that
            @ forwards the request to all clients holding
            @ a copy of the proxy
        }
    }
}
  
```

```

@         ENDIF
    case sso.SSOMessage.kTagUpdateMirror:
@         Find the required proxy by using the SSORegistration
@         Call the method "updateConcreteObject()" of the proxy
@         Call the method "afterUpdate()" of the proxy
    case sso.SSOMessage.kTagClientDisconnected:
@         Call the method "deregisterProxiesOf()" of the
@         SSORegistration and give the client identifier
@         as parameter
    case sso.SSOMessage.kTagDetachPrimary:
@         Find the required proxy by using the SSORegistration
@         Call the method "deregisterAttacher()" of the proxy
@         and give the sender identifier as parameter
    default:
        msg.setReply(sso.SSOMessage.kReplyCommandUnknown);
        break;
    }
    sSSORegistration.sendReply(msg);
} catch (sso.SSOException uce) {
    fSSOMainInterface.uncaughtSSOException(Thread.currentThread(),
        new sso.SSOException("ERROR occured while handling sharable" +
            " proxies\n==>" + uce.getMessage()));
}
return null;
}

```

5.4.1.5 The “SSOProxy” Class

The SSOProxy class is the base class of all sharable proxy classes that instances can be spread over address spaces. Figure 5.11 shows the complete structure of the SSOProxy class. This description of the class concentrates on the methods that are used for the implementation of specialised SSOProxy classes. When reading this description, it might be helpful to check out the sample specialised SSOProxy class in section 6.2. The described methods are grouped into the following types of methods:

- **Construction methods (create and attach):** The class offers two static methods for creating of proxies (`generalCreate()`) and one method for attaching already existing proxies (`generalAttach()`). All three methods need the specialised class name as parameter.

The method “`generalCreate()`” with the “message” parameter is used when a proxy needs to be initialised at creation time (on the object server side). A specified “initialisation-message” results in a call of the method “`initializeConcreteObject()`” when the proxy has been created on the object server side.

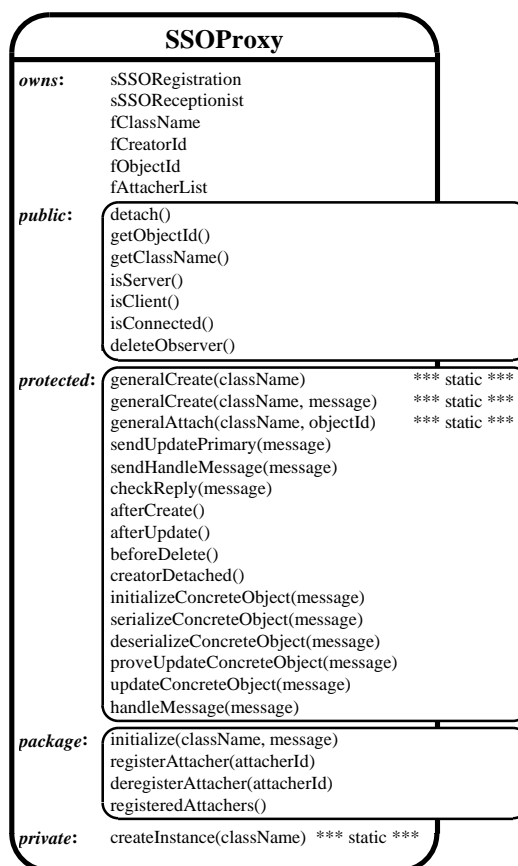


Figure 5.11: The structure of the SSOProxy class

The method “`generalAttach()`” needs, as a second parameter, the object identifier of the proxy that needs to be attached. The attach method creates the required proxy if it does not already exist.

When a SSOProxy instance is requested by a client application, the instance needs to be serialised on the object server side and de-serialised on the client application side. The two methods “`serializeConcreteObject()`” and “`deserializeConcreteObject()`” are used serialisation and de-serialisation of whole proxy instances.

All three “construction” methods are declared as protected. Consequently, a specialised SSOProxy class must implement its own static “`create()`” and “`attach()`” methods that call the “`generalCreate()`” and “`generalAttach()`” methods.

- **Destruction methods (detach):** The method “`detach()`” needs to be called when a proxy is not longer required. The following implementation shows that method behaves different on object servers and client applications.

```
public void detach() {
@   Return immediately, if there are still observers registered.
@   IF isServer() THEN
@       Call the method "deregisterAttacher()" with the identifier
@           of the object server
@   ELSE
@       Call the notification method "beforeDelete()"
@       Deregister the proxy from the SSORRegistration instance
@       Send message "DetachPrimary" to the object server
@   ENDIF
}
```

As shown in Figure 5.1, the `SSOProxy` class is a specialised “Observable” class. The method “`deleteObserver()`” has been overridden. The specialised version calls first the method “`deleteObserver()`” of the base class “Observable” and then it also calls the method “`detach()`”. As a result, a proxy detaches from the SSORB system when the last observer deregisters from the proxy.

- **Modification methods:** Like in the Simple Shared Object pattern, methods that update a sharable object, need to serialise the update information into a message object. Then, update message is then sent to the primary copy of the shared object by using the method “`sendUpdatePrimary()`”. The proxy handler on the object server side receives the message and calls first the method “`proveUpdateConcreteObject()`” of the primary copy. With this method, a specialised proxy is able to prove whether or not the update information is valid. If valid, the method returns a positive value or 0, otherwise it returns a negative value.

If the update information is not valid, the object server simply returns a reply code to the initiator of the update message. Otherwise, the method “`updateConcreteObject()`” is called for the primary copy and for all mirrored copies of the shared object. The method “`updateConcreteObject()`” must be implemented by all subclasses of the `SSOProxy` class.

- **Message handling methods:** Sometimes it can be necessary to send a message to the primary copy of a shared object, without actually modifying the shared object. The `SSOProxy` provides the method “`sendHandleMessage()`” for this purpose. The parameter (a message object) is simply sent to the object server and results in a call of the method “`handleMessage()`” of the primary copy of the shared object.

- **Reply handling:** The method “`checkReply()`” is meant to check a reply message after a call of a method such as “`sendUpdatePrimary()`” or “`sendMessage()`”. This method is not used within the SSORB system, it exists as a “hook” for any specialised `SSOProxy` class that requires reply checking. The method is usually overridden in subclasses. The default implementation of the method simply throws an “unknown” `SSOException` if the reply identifier is negative. The implementation looks as follows:

```
protected void checkReply(int reply) throws sso.SSOReplyException {
    if (reply < 0) {
        throw new sso.SSOReplyException(0, "unknown reply exception");
    }
}
```

- **Notification methods:** Notification is a major feature of the SSORB system. The methods “`afterCreate()`”, “`afterUpdate()`” and “`beforeDelete()`” are called by the SSORB system whenever such an event occurs. The default implementation if the methods “`afterCreate()`” and “`afterUpdate()`” call the method “`notifyObservers()`” of the base class “`Observable`”.

Every shared object is either created by a client application of the object server itself. Sometimes there is a need to initiate some actions if the creator of a shared object detaches the object. The method “`creatorDetached()`” is thought for such actions and is called on the object server side when the creator of a shared object detaches the shared object (`detach()`).

5.4.1.6 The “`SSOProxyRef`” Class

The SSORB system handles shared object references by using instances of the `SSOProxyRef` class. The class holds all information needed to instantiate a specific sharable proxy object. Having a `SSOProxyRef` object in the address space of a client application does not mean that the proxy itself is also in the current address space. The mirrored copy of the referenced proxy gets created in the address space of a client application only if it is needed (by calling the method “`getProxy()`”). A `SSOProxyRef` object can also be serialised into a message object by calling the method “`writeSSOProxyRef()`” of the `SSOMessage` class (or de-serialised by the method “`readSSOProxyRef()`”).

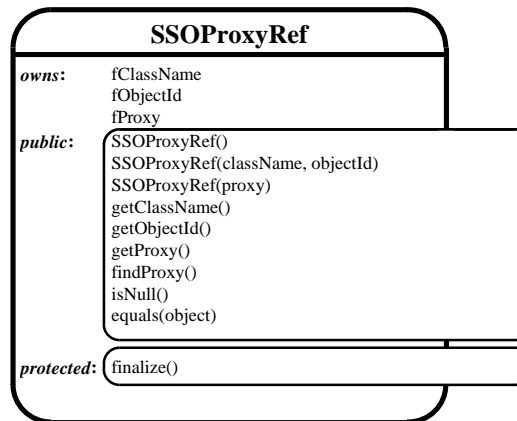


Figure 5.12: The structure of the SSOProxyRef class

Figure 5.12 shows the structure of the SSOProxyRef class. The method “getProxy()” provides the access to the referenced proxy object and it is implemented as follows:

```

public sso.SSOProxy getProxy() throws sso.SSOException {
    if (isNull())
        return null;
    if (fProxy != null)
        return fProxy;
    synchronized (this) {
        if (fProxy != null) // did someone try to foul us?
            return fProxy;
        fProxy = sso.SSOProxy.generalAttach(fClassName, fObjectId);
    }
    return fProxy;
}
  
```

The method “findProxy()” returns a reference to a required proxy if this proxy already exists in the current address space, otherwise the methods return a null value. The method is implemented as follows:

```

public sso.SSOProxy findProxy() {
    if (isNull())
        return null;
    return SSORegistration.instance().findProxy(fObjectId);
}
  
```

The class also provides a specialised “equals()” method for comparison of SSOProxyRef instances:

```

public boolean equals(Object obj) {
    if (obj == null)
        return false;
    if (!(obj instanceof SSOProxyRef))
        return false;
  
```

```

SSOProxyRef ref = (SSOProxyRef) obj;
if (!getClassName().equals(ref.getClassName()))
    return false;
if (getObjectId() != ref.getObjectId())
    return false;
return true;
}

```

The Java programming language uses a garbage collector for removing objects that are not referenced by other objects. Before the garbage collector frees the memory for an object, it calls the method “finalize()”. The SSOProxyRef class uses this method for detaching the referenced proxy object. The method “finalize()” is implemented as follows:

```

protected void finalize() {
    if (fProxy != null)
        fProxy.detach();
}

```

5.4.1.7 The “SSOProxyList” Class

The class SSOProxyList is a specialised SSOProxy class that is able to maintain a list of SSOProxyRef objects. Figure 5.13 shows the structure of the class SSOProxyList. The class has three static data members: a name for the proxy list (fListName), a limit for the capacity of the list (fListLimit) and the actual list of SSOProxyRef object (fSSOProxyRefList). The data member “fSSOProxyRefList” is a normal Vector object of the standard Java package “java.util”. The class also defines and uses some constant integer values that are used for field identification in update messages (e.g. kSetListName, kSetListLimit etc.).

The method “generalCreate()” is used to instantiate a sharable proxy reference list. It is usually called by a specialised subclass. The method has three arguments which are the class name of the subclass, the list name and the list limit. The method serialises the given parameters and calls the method “generalCreate()” of the superclass SSOProxy. The method “generalCreate()” of the class SSOProxyList is implemented as follows:

```

static public SSOProxyList generalCreate(String className,
                                        String listName, int listLimit)
                                        throws sso.SSOException {
    sso.SSOMessage msg = new sso.SSOMessage();
    msg.writeInt(kSetListName);
    msg.writeUTF(listName);
}

```

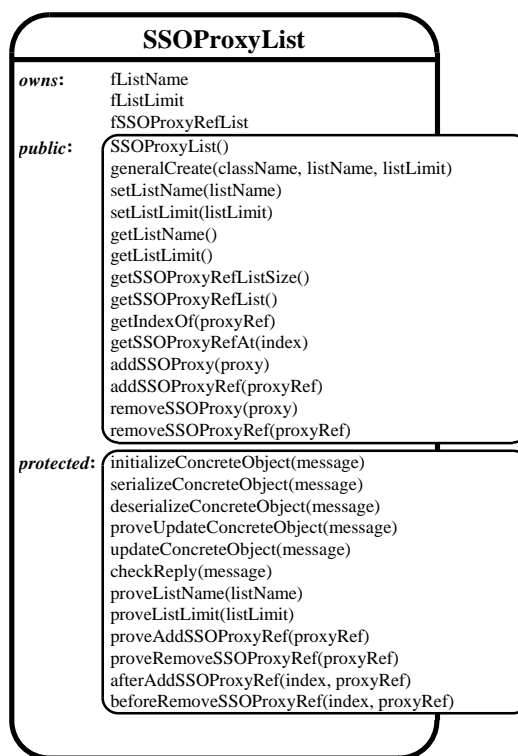


Figure 5.13: The structure of the SSOProxyList class

```

msg.writeInt(kSetListLimit);
msg.writeInt(listLimit);
return (SSOProxyList) sso.SSOProxy.generalCreate(className, msg);
}
  
```

The methods “setListName()”, “setListLimit()”, “addSSOProxy()”, “addSSOProxyRef()”, “removeSSOProxy()” and “removeSSOProxyRef()” are modifying a sharable proxy list. Therefore, these methods serialise their the given parameters into update messages. The following code shows one of these methods (setListName ()) as an example:

```

public void setListName(String name) throws sso.SSOException,
                                           sso.SSOReplyException {
    sso.SSOMessage msg = new sso.SSOMessage();
    msg.writeInt(kSetListName);
    msg.writeUTF(name);
    checkReply(sendUpdatePrimary(msg));
}
  
```

All the “get...()” methods are simple returning the local data members in different variations. The following code shows one of these methods (setListName ()) as an example:

```
public Enumeration getSSOProxyRefList() {
    return fSSOProxyRefList.elements();
}
```

The methods “proveUpdateConcreteObject()” and “updateConcreteObject()” are implemented very similar. There is a switch statement in both methods that is evaluating a message object stream. The only difference between the methods is that the “prove” method proves whether or not the update message is valid and the “update” message does the actual updates. The following implementation shows the method “proveUpdateConcreteObject()”:

```
protected int proveUpdateConcreteObject(sso.SSOMessage msg)
    throws sso.SSOException {
    int ret = 0;
    while(msg.available() != 0) {
        int field = msg.readInt();
        switch(field) {
            case kSetListName:
                {
                    String name = msg.readUTF();
                    ret = proveListName(name); // call application specific
                                                // prove method
                    if (ret < 0)
                        return ret;
                }
                break;
            case kSetListLimit:
                {
                    int limit = msg.readInt();
                    ret = proveListLimit(limit); // call application specific
                                                // prove method
                    if (ret < 0)
                        return ret;
                    if (limit != 0) {
                        if (limit < fSSOProxyRefList.size())
                            return kReplyLimitTooLow;
                    }
                }
                break;
            case kAddSSOProxyRef:
                @ Prove if limit allows another proxy ref elements (limit?)
                @ Prove the proxy ref for null (no null references allowed!)
                @ Prove if proxy ref already exists in the list
                @ Call application specific provings by calling the method
                @ "proveAddSSOProxyRef()"
                break;
            case kRemoveSSOProxyRef:
                @ Prove if the proxy ref element really exists in the list
                @ Call application specific provings by calling the method
                @ "proveRemoveSSOProxyRef()"
                break;
            default:
                break;
        }
    }
}
```

```

return 0;
}

```

Application specific prove requirements can be handled by overriding the methods “proveListName()”, “proveListLimit()”, “proveAddSSOProxyRef()” and “proveRemoveSSOProxyRef()”.

The method “updateConcreteObject()” calls also the notification methods “afterAddSSOProxyRef()” and “beforeRemoveSSOProxyRef()” if a proxy reference is added or removed from the list. The default implementation of these notification methods simply notify connected observers.

The method “initializeConcreteObject()” is called on object server side when a new created sharable object needs to be initialised. The method receives a initialisation message as a parameter. In the SSOProxyList class, the structure of an initialisation message is the same as the structure of an update message. Therefore, the methods “proveUpdateConcreteObject()” and “updateConcreteObject()” can be used in the following implementation of the method “initializeConcreteObject()”:

```

protected boolean initializeConcreteObject(sso.SSOMessage msg)
                                           throws sso.SSOException {
    sso.SSOMessage proveMsg = new sso.SSOMessage();
    proveMsg.copy(msg);
    if (proveUpdateConcreteObject(proveMsg) < 0)
        return false;
    updateConcreteObject(msg);
    return true;
}

```

The method “serializeConcreteObject()” is responsible for serialisation of a whole proxy reference list. The method is implemented as follows:

```

protected void serializeConcreteObject(sso.SSOMessage msg) {
    msg.writeInt(kSetListName);
    msg.writeUTF(fListName);
    msg.writeInt(kSetListLimit);
    msg.writeInt(fListLimit);
    synchronized (fSSOProxyRefList) {
        for(Enumeration e = fSSOProxyRefList.elements();
            e.hasMoreElements();) {
            msg.writeInt(kAddSSOProxyRef);
            msg.writeSSOProxyRef((sso.SSOProxyRef) e.nextElement());
        }
    }
}

```

The de-serialisation method “deserializeConcreteObject()” simply calls the method “updateConcreteObject()” and is implemented as follows:

```
protected void deserializeConcreteObject(sso.SSOMessage msg)
                                         throws sso.SSOException {
    updateConcreteObject(msg);
}
```

Finally, the method “checkReply()” is responsible for throwing exception if an proving of a update parameter is not valid. The method is simply implemented as follows:

```
protected void checkReply(int reply) throws sso.SSOReplyException {
    switch(reply) {
        case kReplyLimitTooLow:
            throw new sso.SSOReplyException(reply,
                "The specified limit is lower then the current entries");
        case kReplyRefExistsAlready:
            throw new sso.SSOReplyException(reply,
                "The specified reference is already in the list");
        case kReplyRefDoesNotExist:
            throw new sso.SSOReplyException(reply,
                "The specified reference is not in the list");
        case kReplyListLimitReached:
            throw new sso.SSOReplyException(reply,
                "The limit of the list is reached, not added");
        case kReplyRefNotValid:
            throw new sso.SSOReplyException(reply,
                "The specified reference is not valid");
        default:
            break;
    }
    super.checkReply(reply);
}
```

5.5 Summary

This chapter described a prototype implementation of the SSORB system. The Java development kit version 1.0 has been used and the system is implemented as an additional Java package named “sso”. The “sso” package uses the Java code of the Deputy pattern that has already been described in section 4.8.

Due to the substantial size of the prototype implementation, only major parts have been described in detail. All classes of the SSORB system have been shown as complete class diagrams, and furthermore, each class is introduced by a separate figure showing the structure (interface) of the class.

The broker part of the system has been implemented application independent, and it can be executed by a Java interpreter. On the other hand, the application independent customer classes for object servers and client applications are not directly executable. Rather, the described classes can be used for application specific and customised object servers and client applications.

An sample application that uses the SSORB system is described in the following chapter 6.

Chapter Six

6. A Sample Application and Evaluation

6.1 Overview

This chapter is divided into two main parts. The first part (section 6.2) describes a sample application that uses the SSORB system for object sharing. The second part (section 6.3) evaluates the SSORB system.

The *sample application* is a program that is meant to allow users to meet other users in a “virtual house” (The “house” or “conference centre” is used as a metaphor to help make the system more easily understood. It captures the idea of a place where people can meet, form workgroups, and undertake tasks). The application uses shared objects for the virtual house, rooms within the house and users in the house. The description of the sample application is shown in four phases: analysis, design, evolution and modification (see also Booch [Booch94]).

The **analysis** part describes the problem context for the sample application. The requirements for the program are specified and limited to keep the application small. The **design** part identifies the classes of the application. The design is shown in two steps, the first step shows the sharable classes and the second step describes the classes needed for the implementation of the graphical user interface. The **evolution** part describes one of the sharable classes in detail. The class is explained in detail to show how sharable SSORProxy classes can be specialised. Screen dumps of the sample application are shown at the end of the evolution part. The **modification** part describes how the functionality of the sample application could be extended.

The *evaluation* of the SSORB system covers three aspects. The **portability** part describes how easily the SSORB system can be used on different operating systems (architectures). The **performance** part is a simple first order analysis of the number of

messages exchanged within an SSORB application. Finally, the **reusability** part lists the key features of the SSORB system that provides the reuse of application specific classes.

6.2 “Application Joiner” - A Sample Application using the SSORB System

After all the theory about the application independent parts of the SSORB system, it is time to show a concrete sample application using the SSORB components. This section describes a very small CSCW-application that, due to its size, is easy to grasp. The description is divided into four phases, corresponding to the typical stages of software development.

Firstly (section 6.2.1), the “analysis” explains the problem context. Secondly (section 6.2.2), the design refines the problem into an object-oriented structure. Thirdly (section 6.2.3), the evolution shows the actual implementation of the sample application. And lastly (section 6.2.4), the modification phase describes possible extensions of the sample application.

The four phases of this “mini-project” are small. The aim of this application is to understand the SSORB system and the problems of a distributed application.

6.2.1 Analysis

Almost all CSCW applications need some sort of an “Application Joiner” program, an initial task that allows users to meet at a virtual place in a computer network. This sample application should allow users to meet others in a “virtual house”. The “virtual house” should work like a big conference building that allows collaborators to meet at a common meeting room. Then, the collaborators form groups and go to different rooms to undertake cooperative work.

A major requirement of such a virtual house is that the collaborators must be able to communicate. They should be able to talk to all other individuals in the same room. Furthermore, a collaborator should be able to contact any other collaborator within the house, even if the desired person is in another room (i.e. there is something equivalent to an internal telephone net of a conference building). A group of collaborators should also be able to limit the number of people within their room.

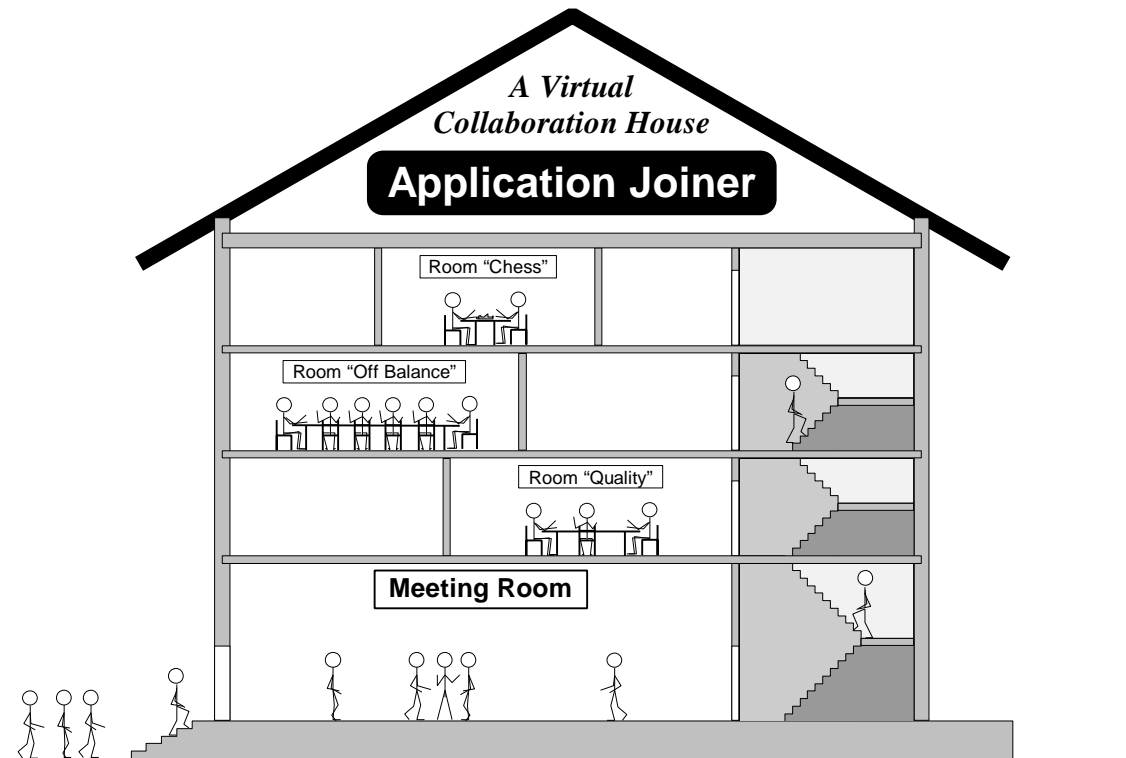


Figure 6.1: A virtual collaboration house “Application Joiner”

The type of collaboration can differ from one room to another. While a group of managers might discuss a quality issue in one room, a group of accountants might work out an off balance problem in another room. In a third room, two individuals might even relax playing a game of chess.

Figure 6.1 illustrates such a virtual house where individuals are collaborating in different rooms. The individuals meet in a meeting room, form groups and define rooms for further collaborations. Then, the individuals enter a chosen rooms and are able to start a collaboration.

The sample SSORB application concentrates on the joining process of individuals. Therefore, the program becomes the name “Application Joiner”. In order to keep the application simple, the requirements for the program are limited as follows:

- Every user has three name attributes that are a *nickname*, a *name* and a field for *other names*. The nickname is unique within the house and it is not changeable once a user entered the house. The other two attributes *name* and *other names* are always changeable.

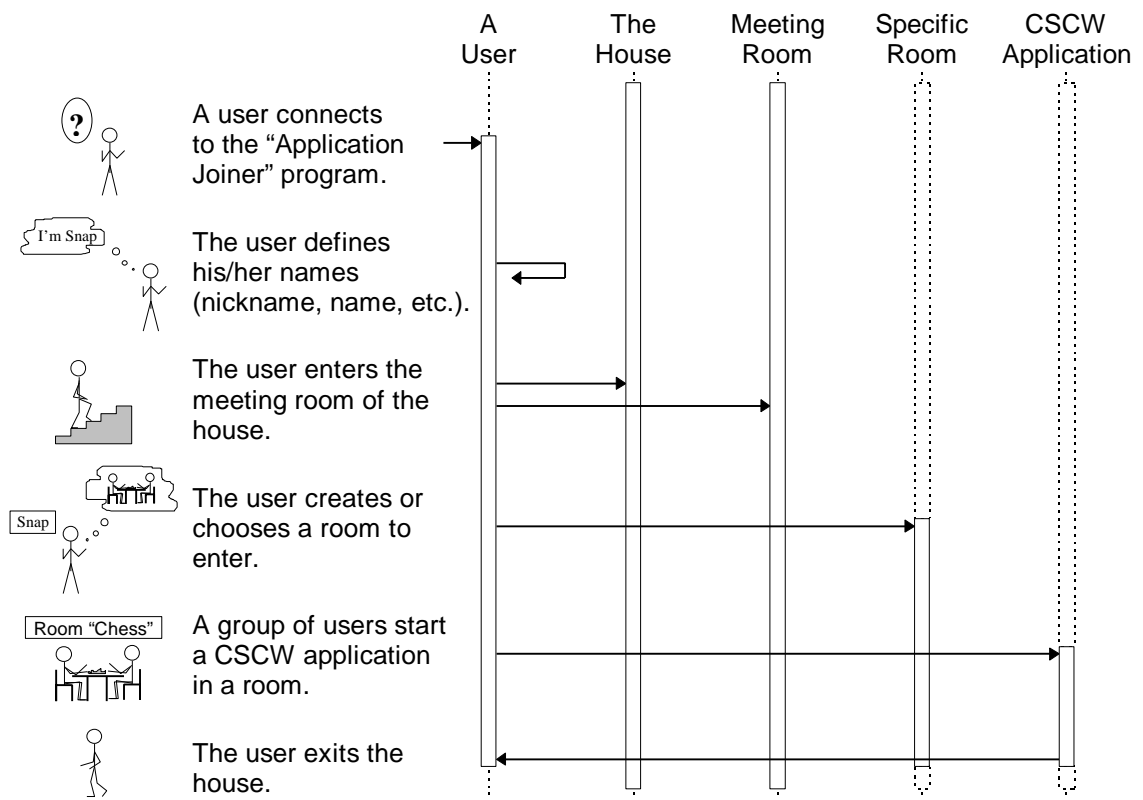


Figure 6.2: Scenario of a user entering the virtual collaboration house

- Every user is able to create new rooms, delete empty rooms and change the limit for the number of users in that room in which the user currently is located.
- A user is able to move from one room to another at any time. However, a user is not allowed to enter a room where the user limit has been reached. Furthermore, a user is not able to change a room limit of a room that is not the current room.
- A user is able to talk to a specific user in the house, no matter in which room the other user currently is. A user is also able to talk to a group of users such as all users in the house, all users in a specific room and all users in the current room. This sample application does not include the refusal of talk messages by a user.

Figure 6.2 shows a scenario of a user visiting the virtual collaboration house. Only the major steps of the joining process are shown in the scenario. In reality, a user might wander through different rooms and talk to a couple of people before he or she starts a CSCW application.

The start of a specific CSCW application once users have met in a collaboration room is not part of this sample application.

6.2.2 Design

This section shows the transfer of the analysed requirements into an object-oriented design. The two subsections of this section focus on different aspects. The first subsection concentrates on the identification of the sharable classes of the application. The sharable classes will be used on both the object server and the client application side. The second subsection is dedicated to the user interface of the client application side. It shows the connection of the sharable data classes to specialised user interface classes. The user interface uses the AWT (Abstract Window Toolkit) classes of the Java development kit (version 1.0).

6.2.2.1 Identification of the Sharable Classes

First, it is necessary to define the classes that need to be shared across address spaces in the Application Joiner program. The following questions are asked to identify the sharable classes.

- *What objects does a user need to know when (s)he enters the Application Joiner program?*
- *What are the relationships between these objects?*
- *What class hierarchy can be extracted from these objects?*

When a user enters the Application Joiner program, (s)he wants probably to know who is already in the house, who is in the meeting room and what rooms are available. Therefore, there are three “well-known” objects that are a list of “users in the house”, a list of “users in the meeting room” and a list of “available rooms”. The user lists “house” and “meeting room” simply refer to user objects that are currently in the house and in the meeting room. Therefore, the list “meeting room” is a subset of the list “house”. The list of “available rooms” refers to “room” objects that again refer to “user” objects that are currently in these rooms.

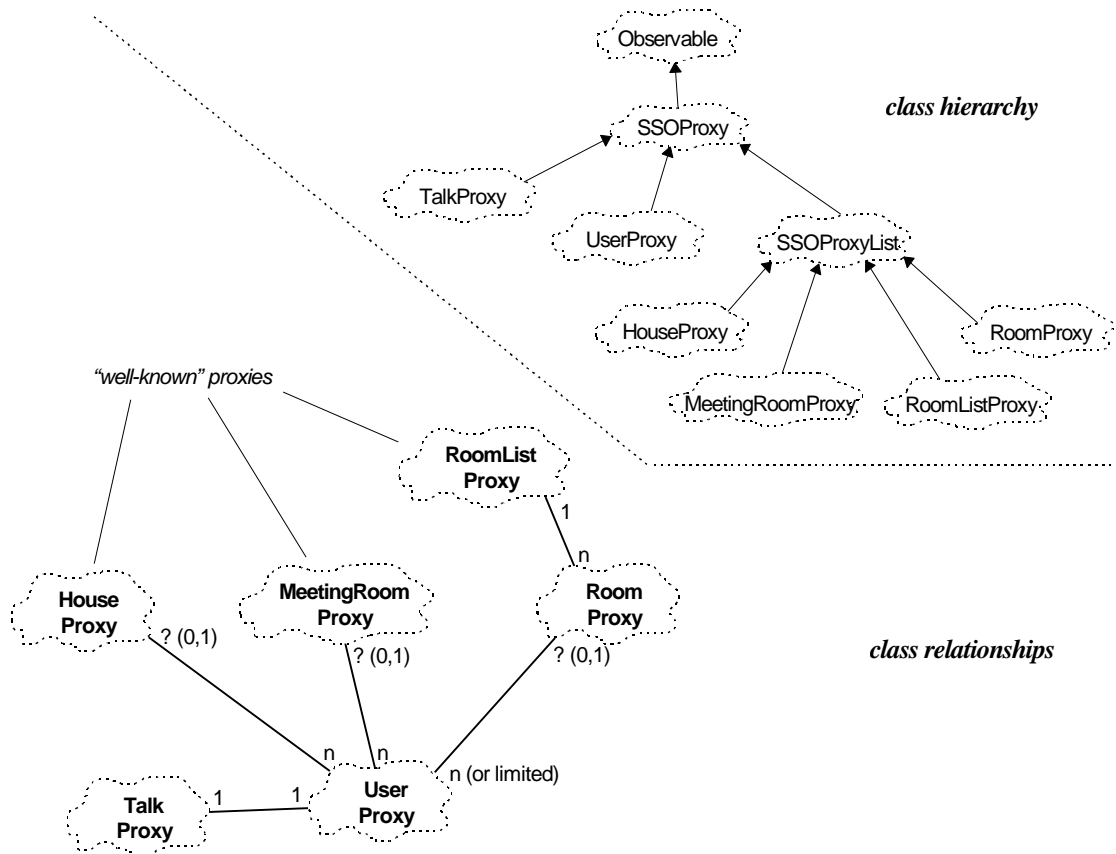


Figure 6.3: The sharable classes of the Application Joiner program

These groups of classes and relations are all somewhat similar. The lists “house”, “meeting room” and “room” can contain any number of users and a user can only be in one room at a time. The list “available rooms” contains any number of rooms and every room is registered with the “available rooms” list.

Figure 6.3 illustrates the previously defined sharable classes, their relationships and the hierarchy that connects the sharable classes to the SSORB system. The figure shows that each user proxy object has its own talk proxy object. The class “UserProxy” is responsible for holding the data members of the user such as name and nickname. The class “TalkProxy” covers all functionality that gives the user the opportunity to talk to other users. The relationship between the classes “UserProxy” and “TalkProxy” is simply “one-to-one”.

All relationships shown in Figure 6.3 are references across address spaces. Therefore, each of these references needs to be implemented as a SSOProxyRef reference object.

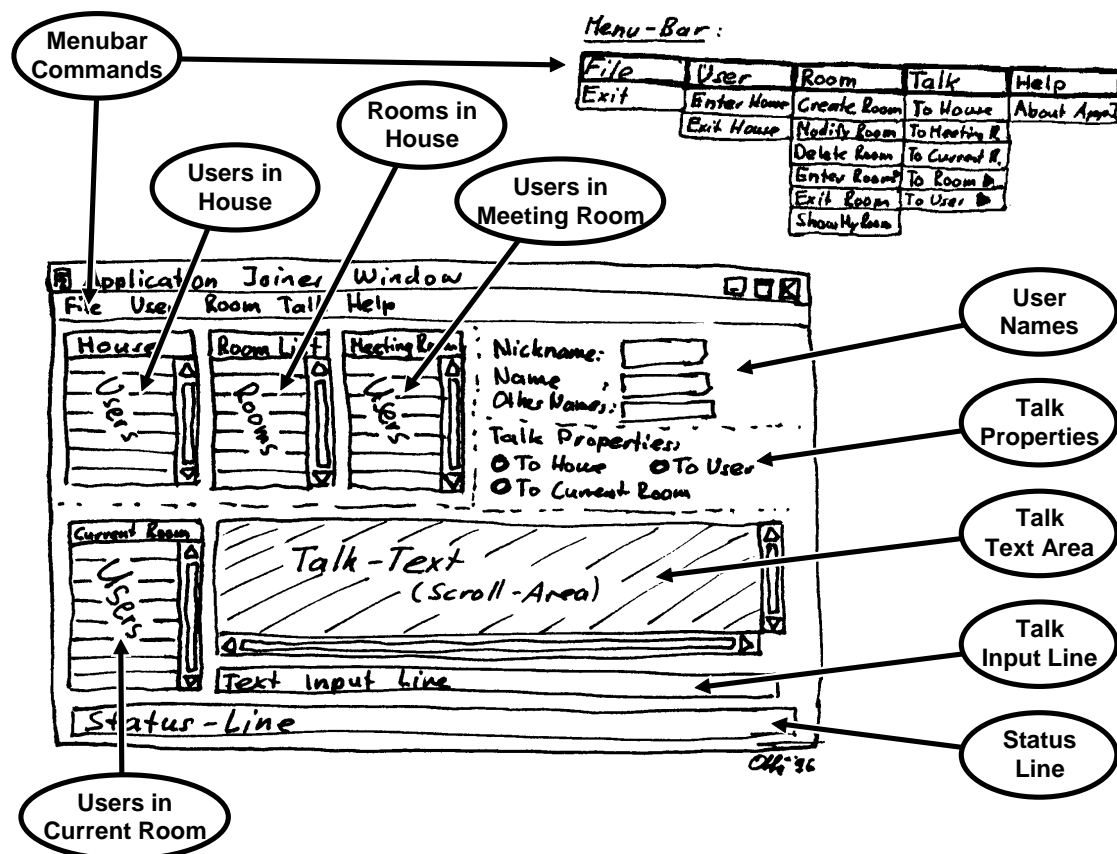


Figure 6.4: A sketch of the Application Joiner user interface

Figure 6.3 also shows that the classes “HouseProxy”, “MeetingRoomProxy” and “RoomListProxy” are implemented as “well-known” sharable classes. That means, these three classes have predefined shared object identifiers. A newly connected user “sees” through the instances of these three classes the “inside” of the house.

6.2.2.2 Specialisation of the User Interface Classes

The object server of the Application Joiner program only needs to connect to a SSORB broker and have access to the proxy classes described in the previous section (for instantiation). On the other hand, client applications of the Application Joiner program need to present the shared objects to their user.

A sketch of the graphical user interface components usually helps to identify the required classes of a GUI application. Figure 6.4 shows a sketch of the user interface for the Application Joiner program and its menu bar.

The menu bar provides commands for various purposes. The commands are grouped into the following five “pull-down” menus:

- **File:** Exiting the Application Joiner program.
- **User:** Entering and exiting the house.
- **Room:** Creating, modifying, deleting, entering and exiting of rooms.
- **Talk:** Talk commands for choosing a specific user or a room to talk to.
- **Help:** Information about the copyright information of the Application Joiner program.

The Application Joiner window includes four scrollable list boxes that show the users in the house, the users in the meeting room, the users in the current room and the available rooms in the house. The users’ attributes *nickname*, *name* and *other names* are shown in separate entry fields (User Names). The area “Talk Properties” contains tick items that let the user choose to talk to all users in the house, to all users in a specific room or to a specific user. The “Talk Text Area” displays the messages that are sent and received by the user. Talk messages are entered into the field “Talk Input Line”. The field “Status Line” is used to display status messages to the user (e.g. if one wants to enter a room that user limit has already been reached).

Using the drawn sketch of Figure 6.4, the GUI classes of the Application Joiner program can be specialised. The classes provided by the AWT package of Java development kit are used for this specialisation.

Figure 6.5 illustrates the class hierarchy of the GUI classes of the Application Joiner program. The classes with grey background are standard classes of the AWT package. The specialised “Frame” class “AppJoinClient” is the actual Application Joiner Window. It’s instance instantiates all other GUI components.

The class “ThingListPanel” is used, because the class “List” of the Java development kit did not work properly in version 1.0. The class also includes some additional features for displaying instances of class “SSOProxy”. The four subclasses of class “ThingListPanel” represent the scrollable lists of users and rooms (see Figure 6.4). The class “UserPanel” is responsible for displaying and modifying the user ob-

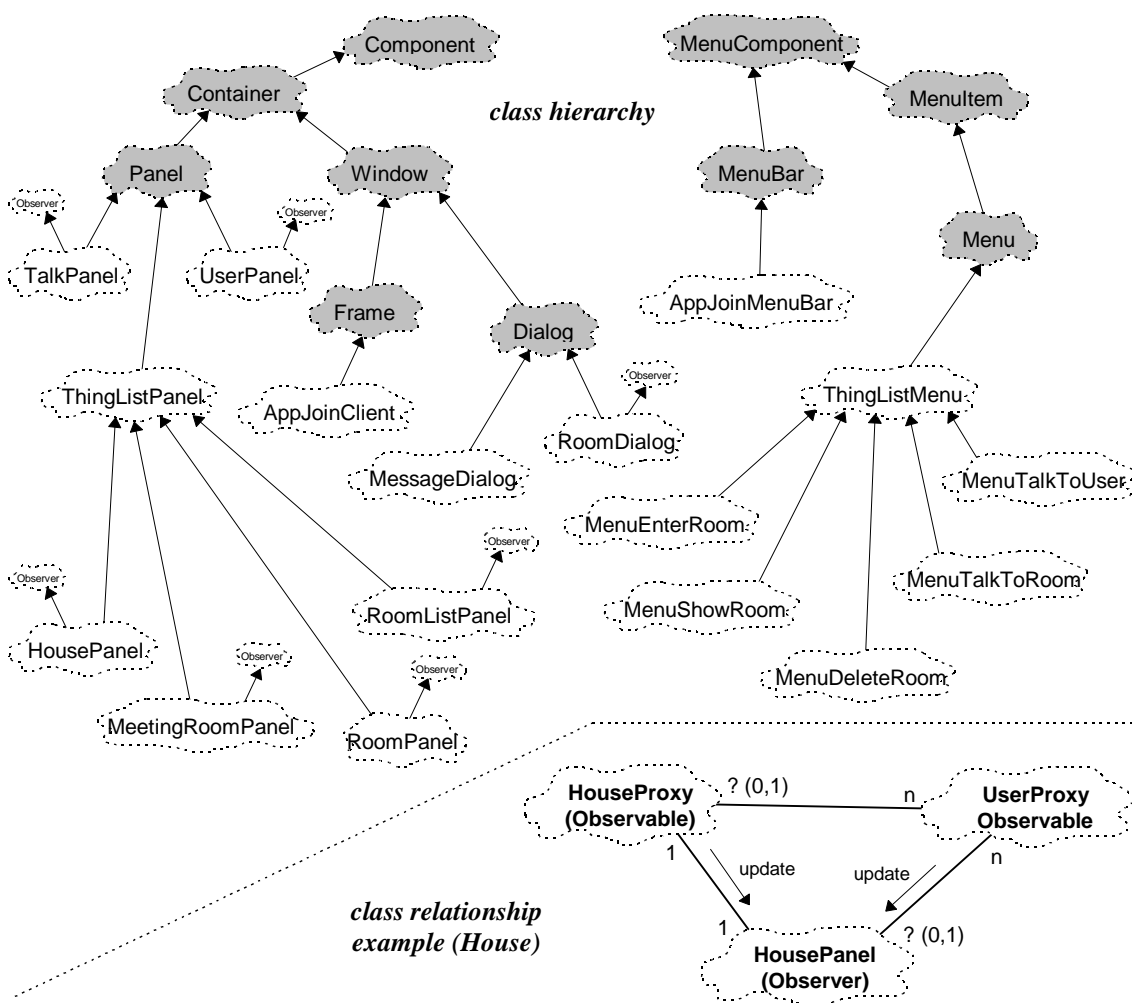


Figure 6.5: The GUI classes of the Application Joiner program

ject (user names). The class “TalkPanel” manages the tick items of the talk properties, the input field and the talk text area.

The program contains two specialised dialog classes. The class “MessageDialog” is responsible for prompting messages to the user that have to be acknowledged by clicking on an “OK” button. The class “RoomDialog” is used when a user creates a new room (requesting of a room name and a user limit for the room).

The class “AppJoinMenuBar” provides the pull-down menus for the Application Joiner program and uses the specialised “Menu” classes.

The relationship example at the bottom of Figure 6.5 shows how the class “HousePanel” is connected to its sharable data classes “HouseProxy” and “UserProxy”. The instance of class “HousePanel” observes the instance of class

“HouseProxy” (the list) as well as instances of class “UserProxy” (the entries). Thus, the list is updated whenever a user enters or exits the house and when a user changes one of the names (*name* and *other names*).

The relationships of the other observer classes such as “MeetingRoomPanel”, “RoomListPanel” and “RoomPanel” are similar to that of the shown class “HousePanel”.

6.2.3 Evolution

The subsection 6.2.3.1 describes how a specialised sharable proxy class can be implemented. Only one of the sharable proxy classes of the Application Joiner program has been picked out to describe the implementation of a specialised “SSOProxy” class. The chosen class is the class “UserProxy”, because it contains normal data members as well as shared proxy references.

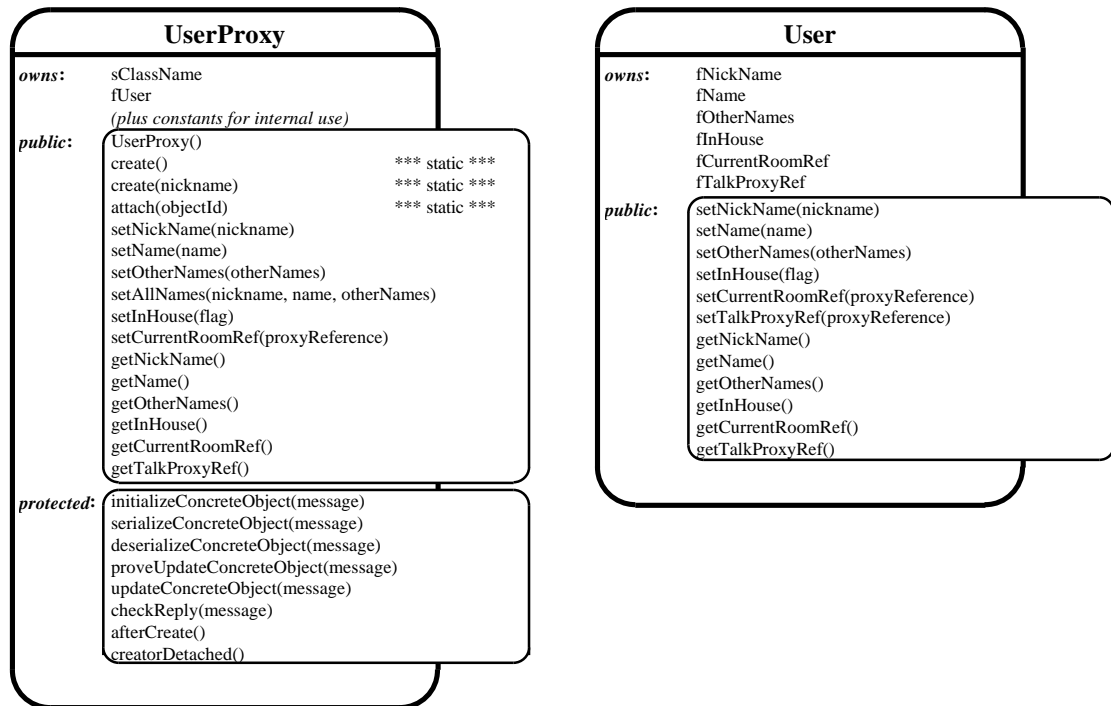
The subsection 6.2.3.2 shows some screen dumps of a posed example of a CSCW collaboration. The graphical user interface is shown on two different windows system that are X-Windows (Open Windows) and Windows 95.

6.2.3.1 Implementation of a Specialised Sharable Proxy Class

The implementation of a specialised sharable proxy class is described with the class “UserProxy”. The description describes all method needed for sharing “User” objects in the Application Joiner program.

A user is divided into two classes, the class “User” (concrete objects) and its proxy class “UserProxy” (proxy objects). The class “User” is only responsible for holding the data members of a user. The class contains six attributes, that are a nickname, a name, other names, a boolean that defines whether or not the user in the house, a reference to the current room and a reference to the talk proxy of the user. The class and the six attributes are declared as follows:

```
class User {
    String      fNickName      = "";
    String      fName          = "";
    String      fOtherNames    = "";
    boolean     fInHouse       = false;
    SSOProxyRef fCurrentRoomRef = new SSOProxyRef();
}
```

Figure 6.6: The structures of the classes `User` and `UserProxy`

```

SSOProxyRef fTalkProxyRef = new SSOProxyRef();
}

```

Figure 6.6 illustrates the structures of the classes “`User`” and “`UserProxy`”. The class “`User`” is only accessed by its proxy class “`UserProxy`”. The class “`User`” provides the access methods for all its attributes (`set...()` and `get...()`). The implementation of these access methods is very simple and consists only of one line of code for each method. The following code shows the access methods for setting the attributes of class “`User`”.

```

public void setNickName(String nickName) {
    fNickName = nickName;
}
public void setName(String name) {
    fName = name;
}
// ...and similar for other access methods

```

The following code shows the access methods for reading the attributes of class “`User`”.

```

public String getNickName() {
    return fNickName;
}

```

```

}
public String getName() {
    return fName;
}

// ...and similar for other access methods

```

The class “UserProxy” has two major attributes; these are a static data member for the name of the class (sClassName) and an instance of class “User” (fUser). The class also declares a few constant data members (final integers) for message and reply type identification. The class “UserProxy” and its attributes are declared as follows:

```

public class UserProxy extends sso.SSOProxy {
    static final int kSetNickName      = 1;
    static final int kSetName          = 2;
    static final int kSetOtherNames    = 3;
    static final int kSetInHouse       = 4;
    static final int kSetCurrentRoomRef = 5;
    static final int kSetTalkProxyRef  = 6;

    static final int kReplyAlreadyInHouse = -1;
    static final int kReplyNotInHouse    = -2;
    static final int kReplyNickNameAlreadyExists = -3;
    static final int kReplyNoNickName    = -4;

    static String sClassName = "appJoin.UserProxy";
    appJoin.User  fUser      = new appJoin.User();
}

```

The default constructor of class “UserProxy” needs to be declared as public, because the classes of package “sso” need to access this class for instantiation. The application, however, should never use this default constructor. Rather, the application must use the static member functions “create()” and “attach()”, that are offered by the class. The class provides two “create()” methods, one without arguments for normal instantiation and another with an argument that allows to specify an initial nickname for the new user object. The nickname argument is serialised into a message object and given to its superclass for instantiation. The constructor, create and attach methods are implemented as follows:

```

public UserProxy() {
}

static public UserProxy create() throws sso.SSOException {
    return (UserProxy) sso.SSOProxy.generalCreate(sClassName);
}

static public UserProxy create(String nickName)
                                throws sso.SSOException {
    sso.SSOMessage msg = new sso.SSOMessage();
}

```

```

    msg.writeInt(kSetNickName);
    msg.writeUTF(nickName);
    return (UserProxy) sso.SSOProxy.generalCreate(sClassName, msg);
}

static public UserProxy attach(long objectId)
                                throws sso.SSOException {
    return (UserProxy) sso.SSOProxy.generalAttach(sClassName, objectId);
}

```

The methods modifying a user object need to serialise their arguments into message objects and send the message objects to the object server by calling the method “sendUpdatePrimary()”. Thus, all “set...()” methods are implemented similar. The following code shows all provided “set...()” methods:

```

public void setNickName(String nickName) throws sso.SSOException {
    sso.SSOMessage msg = new sso.SSOMessage();
    msg.writeInt(kSetNickName);
    msg.writeUTF(nickName);
    sendUpdatePrimary(msg);
}

public void setName(String name) throws sso.SSOException {
    sso.SSOMessage msg = new sso.SSOMessage();
    msg.writeInt(kSetName);
    msg.writeUTF(name);
    sendUpdatePrimary(msg);
}

public void setOtherNames(String otherNames) throws sso.SSOException {
    sso.SSOMessage msg = new sso.SSOMessage();
    msg.writeInt(kSetOtherNames);
    msg.writeUTF(otherNames);
    sendUpdatePrimary(msg);
}

public void setAllNames(String nName, String name, String oName)
                        throws sso.SSOException {
    sso.SSOMessage msg = new sso.SSOMessage();
    msg.writeInt(kSetNickName);
    msg.writeUTF(nName);
    msg.writeInt(kSetName);
    msg.writeUTF(name);
    msg.writeInt(kSetOtherNames);
    msg.writeUTF(oName);
    sendUpdatePrimary(msg);
}

public void setInHouse(boolean flag) throws sso.SSOException,
                                     sso.SSOReplyException {
    sso.SSOMessage msg = new sso.SSOMessage();
    msg.writeInt(kSetInHouse);
    msg.writeBoolean(flag);
    checkReply(sendUpdatePrimary(msg));
}

public void setCurrentRoomRef(SSOProxyRef ref)
                        throws sso.SSOException {
    sso.SSOMessage msg = new sso.SSOMessage();
}

```

```
msg.writeInt(kSetCurrentRoomRef);
msg.writeSSOProxyRef(ref);
sendUpdatePrimary(msg);
}
```

All access methods reading user attributes (`get...()`) of the class “UserProxy” simply call the corresponding access methods of the class “User”. Thus, the implementation of these “`get...()`” methods is very simple and looks like follows:

```
public String getNickName() {
    return fUser.getNickName();
}

public String getName() {
    return fUser.getName();
}

public String getOtherNames() {
    return fUser.getOtherNames();
}

public boolean getInHouse() {
    return fUser.getInHouse();
}

public SSOProxyRef getCurrentRoomRef() {
    return fUser.getCurrentRoomRef();
}

public SSOProxyRef getTalkProxyRef() {
    return fUser.getTalkProxyRef();
}
```

As previously described, the class “UserProxy” provides a create methods that allows initialisation of the nickname attribute. Therefore, the predefined protected member function “`initializeConcreteObject()`” needs to be specialised for initialisation of the nickname attribute. The method de-serialises the nickname argument and sets the nickname attribute of the user object. The method “`initializeConcreteObject()`” is implemented as follows:

```
protected boolean initializeConcreteObject(sso.SSOMessage msg)
    throws sso.SSOException {
    switch(msg.readInt()) {
        case kSetNickName:
            fUser.setNickName(msg.readUTF());
            break;
        default:
            return false;
    }
    return true;
}
```

The specialised method “`serializeConcreteObject()`” simply serialises all attributes of the user object. This method is called whenever an object needs to be transferred into another address space. The method is implemented as follows:

```
protected void serializeConcreteObject(sso.SSOMessage msg) {
    msg.writeInt(kSetNickName);
    msg.writeUTF(getNickName());
    msg.writeInt(kSetName);
    msg.writeUTF(getName());
    msg.writeInt(kSetOtherNames);
    msg.writeUTF(getOtherNames());
    msg.writeInt(kSetInHouse);
    msg.writeBoolean(getInHouse());
    msg.writeInt(kSetCurrentRoomRef);
    msg.writeSSOProxyRef(getCurrentRoomRef());
    msg.writeInt(kSetTalkProxyRef);
    msg.writeSSOProxyRef(getTalkProxyRef());
}
```

The specialised method “`deserializeConcreteObject()`” is responsible for de-serialisation of a whole user object. The serialisation of a whole user object is implemented in the same way as the serialisation of a single attribute of a user object. Therefore, the method “`updateConcreteObject()`” can be used for de-serialisation of a whole user object as well as for de-serialisation of a single user attribute. The following implementation shows that the methods “`deserializeConcreteObject()`” simply calls the methods “`updateConcreteObject()`”:

```
protected void deserializeConcreteObject(sso.SSOMessage msg)
    throws sso.SSOException {
    updateConcreteObject(msg);
}
```

The method “`proveUpdateConcreteObject()`” is called by the object server of the Application Joiner program and is responsible for proving whether or not an update arguments are valid. It should be noted that all possible attributes needs to be de-serialised, even if there is not need for proving of a specific attribute. This is necessary, because the prove method is called for all update messages. The method is implemented as follows:

```
protected int proveUpdateConcreteObject(sso.SSOMessage msg)
    throws sso.SSOException {
    while(msg.available() != 0) {
        int field = msg.readInt();
        switch(field) {
            case kSetNickName:
            case kSetName:
            case kSetOtherNames:
```

```

        msg.readUTF();
        break;
    case kSetInHouse:
        try {
            boolean inHouse = msg.readBoolean();
            HouseProxy hp = HouseProxy.attach();
            if (inHouse) {
                if (getNickName().length() <= 0)
                    return kReplyNoNickName;
                if (getInHouse())
                    return kReplyAlreadyInHouse;
                if (hp.doesNickNameExist(getNickName()))
                    return kReplyNickNameAlreadyExists;
                hp.addSSOProxy(this);
            } else {
                if (!getInHouse())
                    return kReplyNotInHouse;
                hp.removeSSOProxy(this);
            }
        } catch (sso.SSOException e1) {           // should not happen!
        }
        break;
    case kSetCurrentRoomRef:
        msg.readSSOProxyRef();
        break;
    case kSetTalkProxyRef:
        msg.readSSOProxyRef();
        break;
    default:
        break;
    }
}
return 0;
}

```

The method “updateConcreteObject()” is only called with valid attributes (serialised in a message object). Therefore the method is able to update a user object directly without any proving. The method simply calls the required “set...()” methods of the class “User”. At the end of the method a call of “setChanged()” (Observable) initiates an update of all registered observers. The method “updateConcreteObject()” is implemented as follows:

```

protected void updateConcreteObject(sso.SSOMessage msg)
                                throws sso.SSOException {
    while(msg.available() != 0) {
        int field = msg.readInt();
        switch(field) {
            case kSetNickName:
                fUser.setNickName(msg.readUTF());
                break;
            case kSetName:
                fUser.setName(msg.readUTF());
                break;
            case kSetOtherNames:
                fUser.setOtherNames(msg.readUTF());
                break;
        }
    }
}

```

```

    case kSetInHouse:
        fUser.setInHouse(msg.readBoolean());
        break;
    case kSetCurrentRoomRef:
        try {
            if (isServer()) {
                SSOProxyList pl = (SSOProxyList) fUser.
                    getCurrentRoomRef().getProxy();
                if (pl != null) {
                    pl.removeSSOProxy(this);
                }
            }
        } catch (SSOException e) { // should not happen!
        }
        fUser.setCurrentRoomRef(msg.readSSOProxyRef());
        break;
    case kSetTalkProxyRef:
        fUser.setTalkProxyRef(msg.readSSOProxyRef());
        break;
    default:
        break;
}
}
setChanged();
}

```

The method “checkReply()” is responsible for converting reply identifiers (set by the prove method “proveUpdateConcreteObject()”) into plain text. The method simply throws an SSOReplyException object if the given reply identifier matches with one of the class internal “error-identifiers”. The method is implemented as follows:

```

protected void checkReply(int reply) throws sso.SSOReplyException {
    switch(reply) {
        case kReplyAlreadyInHouse:
            throw new sso.SSOReplyException(reply,
                "You are already in the house, can not enter");
        case kReplyNotInHouse:
            throw new sso.SSOReplyException(reply,
                "You are not in the house, can not exit");
        case kReplyNickNameAlreadyExists:
            throw new sso.SSOReplyException(reply,
                "Your chosen Nickname is already used, can not enter");
        case kReplyNoNickName:
            throw new sso.SSOReplyException(reply,
                "You have no Nickname, can not enter");
        default:
            break;
    }
    super.checkReply(reply);
}

```

The specialised notification method “afterCreate()” ensures that every user object refers to its own instance of class “TalkProxy”. The method is implemented as follows:

```
protected void afterCreate() {
    if (isServer()) {
        try {
            TalkProxy tp = TalkProxy.create(this);
            fUser.setTalkProxyRef(new SSOProxyRef(tp));
        } catch (sso.SSOException e) {
        }
    }
    super.afterCreate();
}
```

Finally, the specialised notification method “creatorDetached()” is responsible to guarantee the deregistration of a disappearing user object from the user lists “house” and “current room”. The method also ensures the elimination of the reference to the talk proxy object. The following implementation of the method “creatorDetached()” finishes the explanation of the classes “User” and “UserProxy”.

```
protected void creatorDetached() {
    try {
        if (getInHouse()) {
            HouseProxy hp = HouseProxy.attach();
            hp.removeSSOProxy(this);
        }
        SSOProxyList pl = (SSOProxyList) getCurrentRoomRef().getProxy();
        if (pl != null)
            pl.removeSSOProxy(this);
        fUser.setTalkProxyRef(null);
    } catch (sso.SSOException e1) { // should not happen!
    }
    super.creatorDetached();
}
```

6.2.3.2 The Application Joiner Program - A Posed Example

Now it is time to show how the Application Joiner program works in practice. Suppose a SSORB broker is running on a well known TCP/IP port and an object server for playing chess offers its service. A few users connect to the Application Joiner program via a World Wide Web page by using a Java enabled browser such as Netscape Navigator. The connected users are able to communicate and meet in private rooms (limited number of users). Then, the users choose an offered CSCW application for collaborations.

Figure 6.7 and Figure 6.8 show a posed example of the meeting process of two users which decide to play a party of chess in a private room. One users' Application Joiner runs on a Unix system with X-Windows and the other users' Application Joiner runs on a PC with Windows 95. The figures show a possible dialog between the two

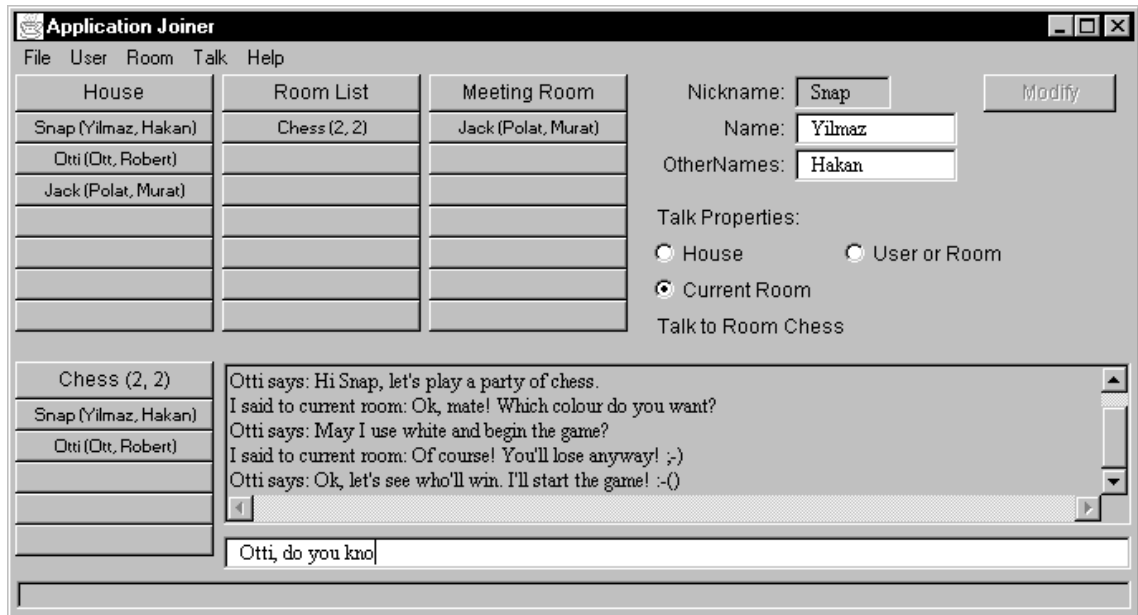


Figure 6.7: A screen dump from the sample SSORB application (on Windows 95)
User “Snap” in room “Chess” talking to user “Otti”

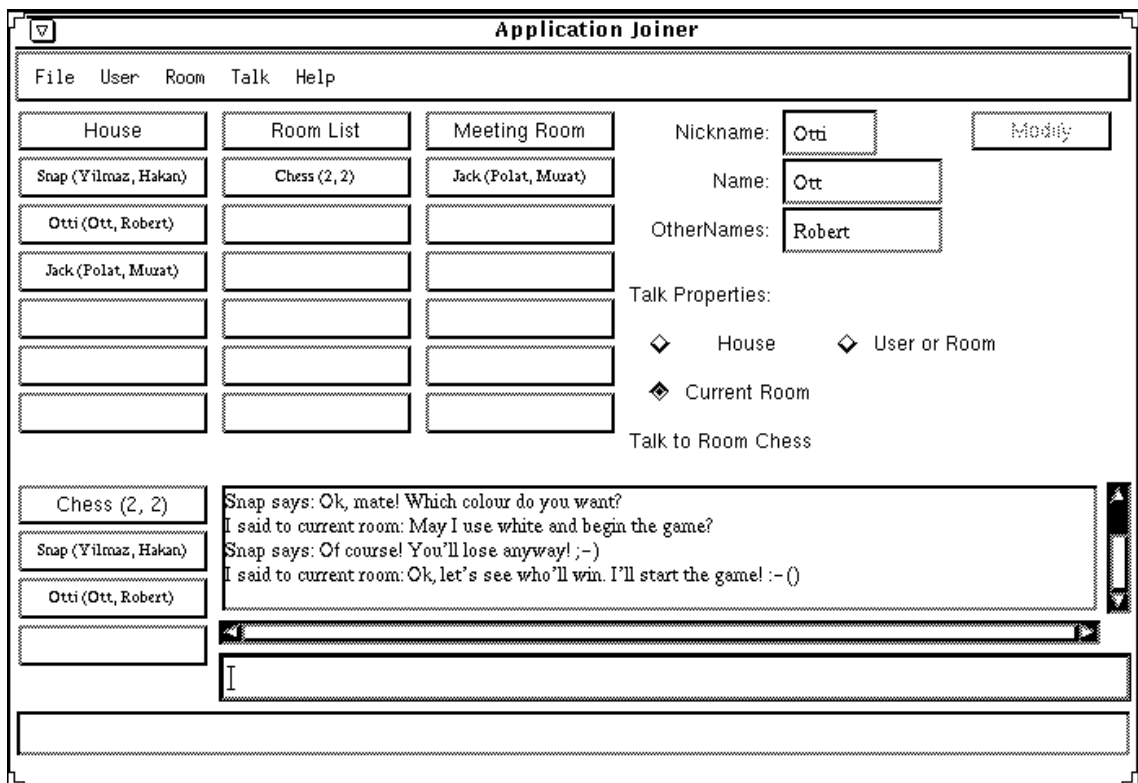


Figure 6.8: A screen dump from the sample SSORB application (on Unix)
User “Otti” in room “Chess” talking to user “Snap”

users “Snap” and “Otti”. A third user “Jack” waits in the meeting room for new connecting CSCW partners.

6.2.4 Modification

Booch defines the modification phase as the last of the four software development phases analysis, design, evolution and modification [Booch94]. He uses the citation of Lehman and Belady [Leh+89] regarding the maturation of a developed software system for the definition of the modification phase. This section describes what modification could be required in an advanced Application Joiner program.

The functionality of this sample Application Joiner program is very limited. A productive version should also support the following requirements:

- A collaborator should be able to save a dialog into a local file (communication protocol).
- A collaborator should be able to refuse talk messages from different sources as he deems fit.
- A list of available CSCW applications should be presented to each user.
- After a group of collaborators has been formed, the group should be able to dispatch from the Application Joiner program for the execution of a chosen CSCW application.

This requirement list does not include all possible requirements for an Application Joiner program. However, it covers the major functionality's required for starting a CSCW application by a number of collaborators.

6.3 Evaluation

An evaluation of a software system can be done in various ways. The aim of design patterns is to describe problems that occur over and over again in an environment and to describe the core of the solutions to that problems (restatement of Alexanders' definition of patterns [Alex+77]). This thesis describes the two design pattern “Simple Shared Object” and “Deputy” that follow the design patterns guideline defined by Erich Gamma

et al. [Gam+95]. The SSORB system itself could also be seen as a design pattern. However, the provided functionality and the size of the system classifies the SSORB system more likely as a framework for object sharing.

The evaluation of the SSORB system is divided into three factors:

- The first factor is **portability**. The success of a system for sharing objects facilitating the implementation of CSCW application depends on the degree of the portability of the system.
- The second factor is the **performance** of the system. A shared object system for distributed programming is based on the exchange of messages between concurrent running programs. Therefore, an evaluation of the number of messages exchanged is fairly important the performance of a system.
- The third factor is **reusability** of the system. The degree of reusability of a software system depends mainly on the design of the system. The evaluation of this factor shows in what aspects the SSORB system provides reusability.

6.3.1 Portability

Portability of software systems became important since users are able to choose on which operating system they want to use their software products. In the past, users were usually obliged to use that operating system on which ever a software product was implemented. This obligation resulted often in office desks full with computers running different operating systems. Since software products became portable, a user can do most of the required tasks by using a single computer.

CSCW applications are used by individuals at different locations using different operating systems. Therefore, a system for sharing objects across address spaces such as the SSORB system has to provide a high degree on portability.

Most portability of the SSORB system is provided by the Java programming language. The Java programming language is already available for the four operating systems Windows 95, Windows NT, Macintosh OS and Sun Unix. The source code of the Java programming language is also provided by Sun Microsystems Inc. meaning a Java interpreter can easily be implemented on most operating systems.

The SSORB system does not need any additional low level libraries. It only uses the Java classes provided in the standard version. The SSORB system uses the network package “`java.net`” of Java development kit. Therefore, a network must be available even if a CSCW application runs only on a single multi user system.

A graphical user interface is not required by the SSORB system. As a result, object can be shared with the SSORB system even if no graphical user interface is available. The use of the Abstract Windows Toolkit “`java.awt`” is totally up to the implementation of object servers and client applications.

While implementing the sample SSORB program “Application Joiner”, some portability problems have been faces in designing the graphical user interface components for different windows environment such as Windows 95 and Open Windows. In order to minimise these problems, an additional class called SSOStandards has been implemented. This class can be asked for standards of windows components such as standard button with/height, standard font/font size and menu bar height. However, this class does not actually belong to the SSORB system, rather it is just an additional class providing portability for implementing graphical user interfaces for CSCW applications.

Overall, the Java programming language has been experienced as an extremely operating system independent foundation for the SSORB system.

6.3.2 Performance

The performance of a software is hard to measure, because it depends on many factors such as processor power, memory availability, operating system, number of users on a system and network protocols. This section analyses the number of messages exchanged on a running SSORB application.

There are eight distinct tasks in the SSORB system, each task involves exchange of a number of request/response messages. The tasks are: registration of an object server, registration of a client application, creation of a shared object, attachment to a shared object, update of a shared object, handling of a message by an object server, disconnection of a client application and disconnection of an object server (see also Figure 5.2, page 89, “Interaction table for message exchanges within the SSORB system”).

Message Tasks	Number of Messages	Graph																		
		0	1	2	3	4	5	6	7	8	9									
Registration of an object server with a request broker (Connecting)	2		█	█																
Registration of a client application with a request broker (Connecting)	2		█	█																
Creation of a shared object (e.g. initiated by a client application)	4		█	█	█	█														
Attaching to a shared object (e.g. initiated by a client application)	4		█	█	█	█														
Update of a shared object (e.g. initiated by a client application)	$6 + n^*$		█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█
Handling of a message by a shared object (e.g. initiated by a client application)	4		█	█	█	█														
Disconnecting of a client application	3		█	█	█															
Disconnecting of an object server	$1 + n^*$		█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█	█

Legend: * n = number of clients using a shared object concurrently

Figure 6.9: Number of messages exchanged, classified by message tasks

Figure 6.9 illustrates the number of messages (requests and replies) exchanged for the eight message tasks.

The message tasks for registration and disconnection of object servers and client applications are not critical for the performance of a CSCW application using the SSORB system. The three message tasks for creation, attachment and update of a shared object and the task “handle message” are exchanged frequently within an application. Therefore, it could be interesting to optimise the exchange of these message tasks.

As described in detail in chapter 4 and 5, all message exchanges between object servers and client applications are passed through a broker. It seems to be possible to evade the involvement of the broker for the exchange of messages once a client application knows its object server. Direct TCP/IP connections between an object server and its client applications would cut the number of message exchanges in half.

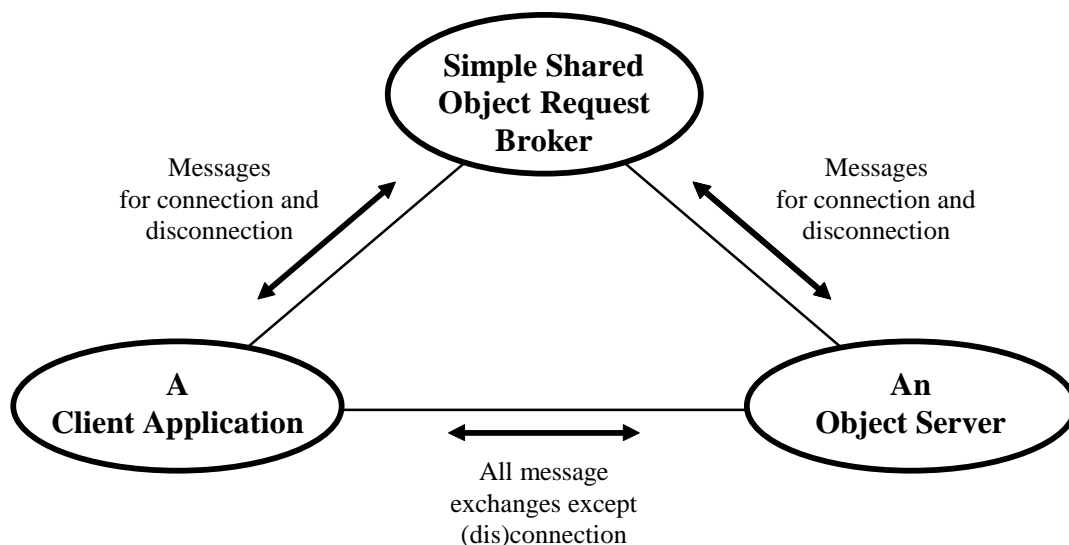


Figure 6.10: Direct connection between an object server and a client application

Figure 6.10 illustrates an object server and a client application with a direct connection to each others. The broker is only involved for connection (register customer) and disconnection of object servers and client applications. This connection model could work, if it could be guaranteed that all SSORB components such as broker, object servers and client applications are never running within a browser such as Netscape Navigator. However, the model would fail because of a security restriction within browsers. Java classes running in a browser are only allowed to connect to TCP/IP sockets of that Web address from which the Java classes have been loaded (on which the broker is running).

Direct connections between object servers and client applications could be established even within a browser by creating a dynamic web page on that address on which an object server is running. A client could then reload the communication classes from the dynamic web page and would be able to connect directly to the object server.

Therefore, the SSORB system does not support direct connection between client applications and object servers and it uses the more expensive way of exchanging messages via the broker.

6.3.3 Reusability

An important phase in a software development process is the maintenance of the software. In practice, however, this phase is sometimes faced as a disturbing side task of a software product. Object-oriented approaches can help to maintain and extend finished software products. The following paragraphs describe in what aspects object sharing with the SSORB system supports the reusability of a CSCW application:

- **Decoupling of application specific classes from network handling:** The SSORB system decouples CSCW applications from network handling for inter process communications. In the future, the SSORB system could be replaced by a newer system providing the same functionality. As a result, already written CSCW applications using the SSORB system do probably not require changes if the basic functionality of the SSORB system is provided similar by the newer system.
- **Sharable classes are the same for object servers and client applications:** The base class of all application specific proxy classes for object servers and client application is the same (in contrast to the Simple Shared Object pattern). Therefore, only one common proxy class needs to be specialised. The methods “`isServer()`” and “`isClient()`” within the class “`SSOProxy`” allows the specialised proxy classes to act differently on object servers and client applications. Modification of proxy classes can be done in the common classes for object servers and client applications.
- **Proxies do not know how they are presented to a user:** Through the implementation of sharable proxy classes as observable, the Model-View (or Observable-Observer) constellation can be applied for separation of the data classes from their representation views. This constellation provides a high degree of reusability when the presentation of objects needs to be changed. The SSORB system is designed so that all presentations to a user are handled by specialised observer classes.

In short, it can be said that the SSORB system is designed to provide reusability. The SSORB classes are implemented as application independent as possible and they provide much flexibility for implementing specialised applications classes. However, it should be considered that the SSORB system has only been tested by implementing one particular and small application program (Application Joiner). Therefore, it should be

noted that the SSORB is implemented as a prototype system and it is probably far away from a finished product providing object sharing across address spaces. Rather, it should be seen as a design for sharing objects across address spaces that can be specialised for different purposes such as integration into frameworks.

6.4 Summary

This chapter has described a sample SSORB application thus providing a concrete example for the SSORB system described in chapter 4 and 5. Furthermore, the SSORB system has been evaluated in terms of portability, performance and reusability.

The explanation of the sample application called “Application Joiner” helps to understand how SSORB applications are meant to be implemented. One sharable proxy class (UserProxy) of the sample application has been chosen to describe an example of a sharable proxy class in detail. Some screen dumps of the sample application illustrate how the shared objects are presented to a user. Furthermore, some additional features that would be necessary in a productive version of such an application joiner program are explained at the end of the description of the sample application.

The evaluation of the SSORB system describes in what degree the system can be ported to different operating systems. Then, the number of message exchanges for object synchronisation has been analysed to provide some information about the performance of the system. Finally, the section reusability explains in what ways the SSORB system provides the reuse of application specific classes.

Chapter Seven

7. Conclusions and Future Research

7.1 Conclusions

The conclusions of this thesis are summarised in two subsections. The first section (Contributions) describes the fields of object-oriented programming to which this thesis contributes. The second section (Reflections) shows what I have learned during this research and the ways in which I expect to apply the resulting experience in future software projects.

7.1.1 Contributions

The following subsections describe the contributions to object-oriented programming in terms of design patterns, distributed programming and the usage of the Java programming language.

7.1.1.1 Contributions to Design Patterns

This thesis describes two design patterns that can be used to extend design pattern collections. The first, “Simple Shared Object”, described in chapter 3 provides a model for sharing objects across address spaces. Chapter 4 describes the second pattern, “Deputy”, and how it contributes to multi-threaded programming techniques.

Erich Gamma *et al.* [Gam+95] suggested that the implementation and sample code parts of design patterns should be described in common object-oriented programming languages such as C++ and Smalltalk. The “Deputy” pattern shows that the Java programming language is useful for describing patterns related to multi-threaded programming and Internet programming. The “Deputy” pattern is one of the first patterns showing implementation code in the Java programming language.

7.1.1.2 Contributions to Distributed Programming

There are many aspects to distributed programming including parallelism of program tasks, resource sharing, and data distribution. This thesis concentrated on sharing of objects across address spaces; and thus includes both resource sharing and data distribution aspects. The SSORB system facilitates especially the implementation of CSCW applications; such applications are becoming increasingly important in companies and tertiary institutions.

Distributed applications using the SSORB system do not have to deal with the exchange of messages for updating mirrored copies of shared objects. The contribution to distributed programming is that the system provides a great degree of reusability. That means, once a sharable proxy class is implemented using the SSORB system for one distributed application, the class can be reused in other applications. For instance, a sharable user class is likely to be used in different distributed applications.

7.1.1.3 Contributions to Applications Using Java

The Java programming language provides the features promised in Sun's definition of the language: "*Java: A simple, object-oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, and dynamic language.*". The SSORB system can be used as an additional Java package for object sharing across address spaces. The OMG has committed itself (Java mailing list, February 1996) to the implementation of only the client part of the CORBA system in Java, the SSORB system can be seen as a complete implementation of a broker based system for object sharing in Java. Even though the functionality of the SSORB system is not as wide as the functionality of a system like CORBA, the SSORB system can satisfy the requirements for many distributed applications.

The SSORB system is fully implemented in Java. The system can be extended or changed in many ways to suit specific requirements of CSCW applications. Some applications may use the SSORB broker simply as a communication utility between components of a distributed application. Other applications may change the SSORB system and use direct communication connections between object servers and client applica-

tions. Therefore, the SSORB system contributes to applications using the Java programming language as an open, extendable and reusable class library.

7.1.2 Reflections

This section summarises my reflections of this research on “Design Patterns for Distributed Programming”. The major conclusions I made are described in the following four subsections:

7.1.2.1 Keeping a Catalogue of Design Patterns

I began this research without knowing what a “Design Pattern” was. Even though I had implemented software systems using object-oriented programming languages for a number of years prior to this research, I had missed the benefits of using design patterns in conjunction with object-oriented programming techniques. Conducting a research on design patterns opened new ideas and ways for future software projects. I learned to feel how the design of an application program can be improved in describing reusable parts of the application as design patterns. Many improvements and generalisations can be made in the process of describing reusable object constellations as design patterns.

7.1.2.2 Java: An Alternative to C++

The implementation of the SSORB system in the Java programming language has led me to the conclusion that the language provides a great range of functionality even in its early version 1.0. I can unhesitating recommend a consideration of the Java programming language as an alternative to other object-oriented programming languages such as C++ and Smalltalk.

Even though some bugs have been encountered during the implementation of the SSORB system, the number of unsolvable problems was null. I remember other development tools that, even in second and third releases, were considerably less complete and effective.

The Java programming language is in its syntax closely related to C++, while it is an interpreted language like Smalltalk. These similarities of Java to already existing languages simplify a switch to the Java programming language for suitable applications.

7.1.2.3 Distributed Software Systems

The increasing availability of networked computer systems and programming environments facilitating the implementation of networked applications will lead to powerful software system for many kind of collaborations. CSCW applications are likely to lead to new ways of collaborating within and between companies. However, the performance and the security of today's networks needs to be improved for real collaborations across country borders.

7.1.2.4 The World Wide Web: An Important Means for Research

I began this research with the usual way of searching in articles, conference abstracts and books related information about the topic I was researching. After a few weeks, I tried to use the World Wide Web to find some additional information about design patterns and distributed programming. It was immediately obvious that the information provided through the World Wide Web was enormous in scope, even though it is sometimes harder to find a specific Web page then finding a way though a big maze.

Once I had implemented some small prototype applications using the Simple Shared Object pattern, I put my first results on the Web server of the computer science department in our university. Within a few weeks, some e-mail messages started to arrive in response to the pattern I provided. Such a rapid response can not be achieved when using other means of publication such as journals or conferences. Subsequently, I kept my Web pages up-do-date and used the World Wide Web as a strategic instrument for getting response throughout my research.

In short, I can strongly recommend the publication of research results through the World Wide Web by both, research students and academic staff at universities.

7.2 Future Research

Design patterns and distributed programming are topics that will be increasingly important in future research and software development. This research focused on ways for sharing objects across networks. The Simple Shared Object pattern and the SSORB system are deliberately simple and need specialisations and extensions for specific software solutions. Possible future research could explore the following:

- **Refinement of the Deputy pattern:** The Deputy patterns has been extracted from common needs on broker and customer side of the SSORB system. Some generalisations have been made to provide a flexible pattern. However, there might be additional requirements when delegating commands from one thread to another. Therefore, a refinement of the Deputy pattern could provide additional functionality and increase the reusability of the pattern.

Just at the time of finishing this thesis, Doug Lea [Lea96] provided in his Web pages a valuable collection of design patterns on concurrency for the Java programming language. The collection includes a pattern named “Runnable Commands” that is related to the Deputy pattern.

- **More applications using the SSORB system:** The SSORB system has only been applied in the sample application described in this thesis (see section 6.2). Other applications will probably reveal additional requirements. Therefore, future research could use and extend the functionality of the SSORB system by considering requirements of other CSCW applications. It could also be interesting to analyse how the SSORB system behaves when the number of concurrently connected customers (object servers and client applications) increases.

- **Implementation of the SSORB system in languages such as C++ and Smalltalk:** The SSORB system has been designed relatively programming language independent. Therefore, an implementation of the system in a programming language other than Java should be easy. It could be interesting to compare the performance of the existing Java implementation with an implementation in an non-interpreted language such as C++. Another possibility could be to implement only the SSORB broker side in a programming language like C++.

- **Integration of a derivative of the SSORB system into the Java development kit:** The Java programming language is well suited to the implementation of a system for sharing objects such as the SSORB system. It would be advantageous if the Java programming language supported object sharing by default. An implementation of object sharing into the Java kernel would allow a greater degree of encapsulation of the shareable classes from application specific classes.

- **Integration of security classes into the SSORB system:** Java (within browsers such as Netscape Navigator) has an integrated security system that allows client programs only to connect to TCP/IP ports of the host machine from which Java classes have been loaded. The central broker of the SSORB system relaxes this restriction and allows indirect communications between client applications and object servers via broker. Such functionality could be misused for data transfer. For instance, files from one client could be transferred to other clients without that a user being aware of the transfer. Therefore, future research could implement security classes into the SSORB system to allow restrictions on transfers. However, such a security system should still allow limited communications between client applications for collaboration purposes.

Appendix A

Abbreviations

ANSI	American National Standards Institution.
ASCII	American Standard Code for Information Interchange. An 8 bit definition of most commonly used characters in Latin letters.
AWT	Abstract Window Toolkit. A package within the Java development kit that provides peers for graphical user interfaces such as Windows 95, Windows NT, X-Windows and Macintosh-GUI.
CASE	Computer Aided Software Engineering (tool).
CORBA	Common Object Request Broker Architecture, endorsed by the Object Management Group (OMG).
CSCW	Computer Support for Co-operative Work.
FTP	File Transfer Protocol.
GUI	Graphical User Interface.
HTTP	HyperText Transfer Protocol.
IDL	Interface Definition Language.
LAN	Local Area Network.
MFC	Microsoft Foundation Classes. A C++ class library facilitating Windows 95 and Windows NT application programming.
MS	Microsoft Corporation.
MVC	Model View Controller. A constellation of classes within the Smalltalk-80 framework.
OLE	Object Linking and Embedding.
OMG	Object Management Group, an organisation that includes major computer manufactures (e.g., Digital, Sun, Hawlett-Packard, IBM) and software providers (e.g., Microsoft, SunSoft, Object Design).
ORB	Object Request Broker.
OSF	Open Software Foundation.
PC	Personal Computer. Mostly used for IBM compatible computers with Intel processors.

RDBMS	Relational Database Management System.
SSO	Simple Shared Object pattern. A design pattern for object sharing across address spaces that is presented in this thesis.
SSORB	Simple Shared Object Request Broker. A system for sharing object across address spaces, based on the Simple Shared Object pattern. The system uses an application independent request broker for message exchange and the system is presented as main part in this thesis.
URL	Uniform Resource Locators.
WAN	Wide Area Network.
WWW	World Wide Web.

Appendix B

Trademarks and Terms

Applet	An Applet is a component within a World Wide Web page that includes compiled Java code. An Applet is able to animate graphics, play sound files and it can even contain a whole GUI application.
HotJava	A trademark of Sun Microsystems Incorporated and a World Wide Web browser.
INFORMIX	A trademark of Informix Incorporated and a relational database management system.
INGRESS	A trademark of Ingress Corporation and a relational database management system.
Java	A trademark of Sun Microsystems Incorporated and an object-oriented programming language for distributed programming.
MacApp	A trademark of Apple Computer Incorporated and a development environment, based on the C++ and Object Pascal, for creating GUI applications for the Macintosh computer family.
Macintosh	A trademark of Apple Computer Incorporated and a general term for the Macintosh computer family.
Motif	A trademark of Open Software Foundation Incorporated.
MS-Word	A trademark of Microsoft Corporation.
Netscape Navigator	A trademark of Netscape Communications Corporation and a World Wide Web browser.
Object Studio	A trademark of Easel Corporation and a development environment, based on the Smalltalk language, for creating GUI applications client/server database applications.
ORACLE	A trademark of Oracle Corporation and a relational database management system.
OS/2	A trademark of International Business Machines.
Presentation Manager	A trademark of International Business Machines.
Smalltalk-80	A trademark of ParcPlace Systems.

SYBASE	A trademark of Sybase Incorporated and a relational database management system.
Unicode	Unicode is a superset of ASCII. It defines most characters that are used all around the world. A Unicode character is stored in a 16 bit value. For more information about Unicode, see <i>The Unicode Standard: Worldwide Character Encoding</i> , Version 1.0, Volume 1 (ISBN 0-201-56788-1) and Volume 1 (ISBN 0-201-60845-6), and the additional information about Unicode 1.1 (at ftp://unicode.org .)
UNIX	A trademark of AT&T Technologies, Incorporated.
Windows 95	A trademark of Microsoft Corporation.
Windows NT	A trademark of Microsoft Corporation.
Word Perfect	A trademark of Word Perfect Corporation.

Appendix C

Java Code Listings

Class DPYDeputy

```
package deputy;

import java.lang.*;
import java.util.*;
import java.io.*;

public class DPYDeputy implements Runnable {
    Vector    fDPYSerialCommandList = new Vector();
    Vector    fDPYParallelCommandList = new Vector();
    Thread    fSerialThread = new Thread(this, "DPYDeputy-Serial");
    Thread    fParallelThread = new Thread(this, "DPYDeputy-Parallel");
    Hashtable fParallelExecutingList = new Hashtable();
    int       fParallelLimit = 0;

    public DPYDeputy() {
        this(0);
    }

    public DPYDeputy(int parallelThreadLimit) {
        fParallelLimit = parallelThreadLimit;
        fSerialThread.start();
        fParallelThread.start();
    }

    public void setParallelThreadLimit(int parallelThreadLimit) {
        fParallelLimit = parallelThreadLimit;
    }

    public int getParallelThreadLimit() {
        return fParallelLimit;
    }

    private void handleSerialCommands() {
        while(true) {
            DPYCommand cmd = null;
            synchronized (fDPYSerialCommandList) {
                try {
                    cmd = (DPYCommand) fDPYSerialCommandList.elementAt(0);
                    fDPYSerialCommandList.removeElementAt(0);
                } catch (ArrayIndexOutOfBoundsException e) {
                }
            }
            if (cmd != null)
                handleCommand(cmd);
        }
    }
}
```

```

        if (fDPYSerialCommandList.isEmpty())
            Thread.currentThread().suspend();
    }
}

private void handleParallelCommands() {
    while(true) {
        DPYCommand cmd = null;
        synchronized (fDPYParallelCommandList) {
            if ((fParallelLimit <= 0) ||
                (fParallelExecutingList.size() < fParallelLimit)) {
                try {
                    cmd = (DPYCommand) fDPYParallelCommandList.elementAt(0);
                    fDPYParallelCommandList.removeElementAt(0);
                    Thread thr = new Thread(this, "DPYDeputy-Command");
                    fParallelExecutingList.put(thr, cmd);
                    thr.start();
                } catch (ArrayIndexOutOfBoundsException e) {
                }
            }
        }
        if (cmd == null)
            Thread.currentThread().suspend();
    }
}

private void handleCommand(DPYCommand cmd) {
    if (cmd == null)
        return;
    DPYReply reply = null;
    if (cmd instanceof DPYDumbCommand)
        reply = handleDPYDumbCommand((DPYDumbCommand) cmd);
    if (cmd instanceof DPYSmartCommand)
        reply = ((DPYSmartCommand) cmd).execute();
    if (cmd.getDPYDelegator() != null) {
        if (reply == null)
            reply = new DPYReply(0);
        reply.setDPYCommand(cmd);
        cmd.getDPYDelegator().handleDPYReply(reply);
    }
}

public final void run() {
    if (Thread.currentThread() == fSerialThread) {
        handleSerialCommands();
        return;
    }
    if (Thread.currentThread() == fParallelThread) {
        handleParallelCommands();
        return;
    }
    DPYCommand cmd = (DPYCommand) fParallelExecutingList.
        get(Thread.currentThread());
    if (cmd != null) {
        handleCommand(cmd);
        fParallelExecutingList.remove(Thread.currentThread());
    }
    fParallelThread.resume();
}

protected DPYReply handleDPYDumbCommand(DPYDumbCommand command) {
    return null;
}

```

```
public final void doDPYCommand(DPYCommand cmd, boolean inSequence,
                               DPYDelegator delegator) {
    if (cmd == null)
        return;
    cmd.setDPYDelegator(delegator);
    if (inSequence) {
        synchronized(fDPYSerialCommandList) {
            fDPYSerialCommandList.addElement(cmd);
            fSerialThread.resume();
        }
    } else {
        synchronized(fDPYParallelCommandList) {
            fDPYParallelCommandList.addElement(cmd);
            fParallelThread.resume();
        }
    }
}

public final void doDPYCommand(DPYCommand cmd, boolean inSequence) {
    doDPYCommand(cmd, inSequence, (DPYDelegator) null);
}

public final void doDPYCommand(DPYCommand cmd) {
    doDPYCommand(cmd, false, (DPYDelegator) null);
}
}
```

Bibliography

- [Alex+77] Alexander C., Ishikawa S., Silverstein M., Jacobson M., Fiksdahl-King I. and Angel S. (1977) *A Pattern Language*, Oxford University Press, New York
- [Booch94] Booch G. (1994) *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings Publishing Company Inc., Redwood City, CA
- [Brow+94] Brown A. W., Carney D. J., Morris E. J., Smith D. B. and Zarrella P. F. (1994) *Principles of CASE Tool Integration*, Oxford University Press, New York
- [Brown94] Brown C. (1994) *UNIX Distributed Programming*, Prentice-Hall, Englewood Cliffs, New Jersey
- [Burn+95] Burnett M. M., Goldberg A. and Lewis T. G. (1995) *Visual Object-Oriented Programming: Concepts and Environments*, Manning Publications Corporation, Greenwich
- [Copl92] Coplien J. O. (1992) *Advanced C++ Programming Styles and Idioms*, Addison-Wesley Publishing Company, Reading, MA
- [CORB92] Manual (1992) *The Common Object Request Broker: Architecture and Specification*, Published by Object Management Group and X/Open, Framingham, MA
- [Ellis+90] Ellis M. A. and Stroustrup B. (1990) *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company, Reading, MA
- [Fried+94] Friedman F. L. and Koffman E. B. (1994) *Problem Solving, Abstraction, and Design Using C++*, Addison-Wesley Publishing Company, Reading, MA
- [Gam+95] Gamma E., Helm R., Johnson R. and Vlissides J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company, Reading, MA
- [Gam92] Gamma E. (1992) "Objektorientierte Software-Entwicklung am Beispiel von ET++: Design Muster, Klassenbibliothek, Werkzeuge", *Doctoral Thesis*, University of Zürich, Springer-Verlag
- [Gold+83] Goldberg A. J. and Robson D. (1983) *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley Publishing Company, Reading, MA
- [Gold94] Goldsmith D. (1994) *Taligent's Guide To Designing Programs: Well-Mannered Object-Oriented Design in C++*, Addison-Wesley Publishing Company, Reading, MA

- [Gosl+95] Gosling J. and McGilton H. (1995) *The Java Language: A White Paper* and *The Java Language Environment: A White Paper*, Sun Microsystems Computer Company, Mountain View, CA, Edition May 1995 (available at <http://java.sun.com/whitePapers/>)
- [Grah93] Graham I. (1993) *Object Oriented Methods*, Addison-Wesley Publishing Company, Reading, MA
- [Gray95] Gray N.A.B. (1995) "Patterns for a Framework for Distributed Programs", *Tools Pacific 95*, Melbourne, November 27-30
- [Green95] Green H. (1995) "Microsoft to License Sun Java Program", *New York Times Syndicate*, December 7 (available at http://nytsyn.com/live/News3/341_120795_134125_6450.html)
- [IBMPress95] IBM Corporation (1995) "IBM licenses Java technology from Sun Microsystems for use in Internet products." *IBM Press release*, December 6 (available at <http://www.ibm.com/News/javapr.html>)
- [Koba91] Kobara S. (1991) *Visual Design with OSF/Motif*, Addison-Wesley Publishing Company, Reading, MA
- [Koen95] Koenig A. (1995) "Patterns and Antipatterns", *Journal of Object-Oriented Programming*, March-April, pp. 46-48, SIGS Publications, CO
- [Kras+88] Krasner G. E. and Pope S. T. (1988) "A cookbook for using the Model-View-Controller user interface paradigm", *Journal of Object-Oriented Programming*, 1 (3), SIGS Publications, CO
- [Lea96] Lea D. (1996) *Concurrent Programming in Java: Tutorials and Design Patterns* (available at <http://g.oswego.edu/dl/pats/aopintro.html>)
- [Leh+89] Lehman and Belady (1989) as quoted in Sommerville, *Software Engineering*, 3rd ed. Wokingham, England: Addison-Wesley, pp. 546
- [Lew+95] Lewis T., Rosenstein L., Pree W., Weinand A., Gamma E., Calder P., Andert G., Vlissides J. and Schmucke K. (1995) *Object-Oriented Application Frameworks*, Manning Publications Co., Greenwich
- [Love91] Love T. (1991) "Timeless Design of Information Systems", *Object Magazine*, November-December, pp. 46
- [Mesz95] Meszaros G. (1995) "Pattern: Half-Object + Protocol", *Pattern Languages of Program Design*, Coplien J. O. and Schmidt D. C. (eds), Addison-Wesley Publishing Company, Reading, MA
- [Neve+95] Neves F. and Garrido A. (1995) "Warden: A Pattern for Object Distribution" Papers of the Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems, OOPSLA '95 (available at <http://www.cs.wustl.edu/~schmidt/OOPSLA-95>)

- [NSPress95] Netscape Communications Corporation (1995) "Netscape to License Sun's Java Programming Language" *Netscape Press release*, May 23 (available at <http://home.netscape.com/newsref/pr/newsrelease25.html>)
- [Ott96] Ott R. (1996) "Simple Shared Object: An Architectural Pattern for Simple Object Sharing" in *Object Currents*, Hathaway B. (ed), SIGS Publications, 1(3), March 1996 (available at <http://www.sigs.com/objectcurrents/>)
- [Pins+88] Pinson L. J. and Wiener R. S. (1988) *An Introduction to Object-Oriented Programming and Smalltalk*, Addison-Wesley Publishing Company, Reading, MA
- [Pree94] Pree W. (1994) *Design Patterns for Object-Oriented Software Development*, Addison-Wesley Publishing Company, Reading, MA
- [Rich95] Richardson P. (1995) "Distributed Object Computing", *Tools Pacific 95*, Melbourne, November 27-30
- [Rumb+91] Rumbaugh J., Blaha M., Premierlani W., Eddi F. and Lorenzen W. (1991) *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, New Jersey
- [Rumb95] Rumbaugh J. (1995) "OMT: The function model", *Journal of Object-Oriented Programming*, March-April, pp 10-14, SIGS Publications, CO
- [Schm+94] Schmidt D. C. and Stephenson P. (1994) "Achieving Reuse Through Design Patterns: A Case Study of Evolving Object-Oriented System Software Across OS Platforms", *Proceedings of the 3rd SIGS C++ World Conference 1994* (available at <http://www.cs.wustl.edu/~schmidt/>)
- [Schm95] Schmidt D. C. (1995) "Reactor: An Object Behavioural Pattern for Concurrent Event Demultiplexing and Dispatching", *Pattern Languages of Program Design*, Coplien J. O. and Schmidt D. C. (eds), Addison-Wesley Publishing Company, Reading, MA
- [Selm+96] Selm R. and Hanselmann M. (1996) *Qualitätsfähigkeit*, Verlag Paul Haupt, Berne
- [Sole92] Soley R. M.. (1992) *Object Management Architecture Guide*, Published by Object Management Group and X/Open, Framingham, MA
- [Soukup94] Soukup J. (1994) *Taming C++: Pattern Classes and Persistence for Large Projects*, Addison-Wesley Publishing Company, Reading, MA
- [Spurr+94] Spurr K., Layzell P., Jennison L. and Richards N. (1994) *Computer Support for Co-operative Work*, John Wiley & Sons Ltd, Chichester, UK

- [Stal95] Stal M. (1995) "The Broker Architectural Framework", Papers of the Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems, OOPSLA '95
(available at <http://www.cs.wustl.edu/~schmidt/OOPSLA-95>)
- [Trew+91] Trew A. and Wilson G. (1991) *Past, Present, Parallel: A survey of available parallel computing systems*, Springer-Verlag, New York
- [Wein+88] Weinand A., Gamma E. and Marty R. (1988) "ET++: an object oriented application framework in C++", *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, pp. 45-57, ACM, September
- [Wils+90] Wilson D. A., Rosenstein L. S. and Shafer D. (1990) *C++ Programming with MacApp*, Addison-Wesley Publishing Company, Reading, MA
- [Your94] Yourdon E. (1994) *Object-Oriented Systems Design: An Integrated Approach*, Prentice-Hall, Englewood Cliffs, New Jersey

-
- Alex+77 Alexander C., Ishikawa S., Silverstein M., Jacobson M., Fiksdahl-King I. and Angel S. (1977) *A Pattern Language*, Oxford University Press, New York
- Booch94 Booch G. (1994) *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings Publishing Company Inc., Redwood City, CA
- Brow+94 Brown A. W., Carney D. J., Morris E. J., Smith D. B. and Zarrella P. F. (1994) *Principles of CASE Tool Integration*, Oxford University Press, New York
- Brown94 Brown C. (1994) *UNIX Distributed Programming*, Prentice-Hall, Englewood Cliffs, New Jersey
- Burn+95 Burnett M. M., Goldberg A. and Lewis T. G. (1995) *Visual Object-Oriented Programming: Concepts and Environments*, Manning Publications Corporation, Greenwich
- Copl92 Coplien J. O. (1992) *Advanced C++ Programming Styles and Idioms*, Addison-Wesley Publishing Company, Reading, MA
- CORB92 Manual (1992) *The Common Object Request Broker: Architecture and Specification*, Published by Object Management Group and X/Open, Framingham, MA
- Ellis+90 Ellis M. A. and Stroustrup B. (1990) *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company, Reading, MA
- Fried+94 Friedman F. L. and Koffman E. B. (1994) *Problem Solving, Abstraction, and Design Using C++*, Addison-Wesley Publishing Company, Reading, MA
- Gam+95 Gamma E., Helm R., Johnson R. and Vlissides J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company, Reading, MA
- Gam92 Gamma E. (1992) "Objektorientierte Software-Entwicklung am Beispiel von ET++: Design Muster, Klassenbibliothek, Werkzeuge", *Doctoral Thesis*, University of Zürich, Springer-Verlag
- Gold+83 Goldberg A. J. and Robson D. (1983) *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley Publishing Company, Reading, MA
- Gold94 Goldsmith D. (1994) *Taligent's Guide To Designing Programs: Well-Mannered Object-Oriented Design in C++*, Addison-Wesley Publishing Company, Reading, MA
- Gosl+95 Gosling J. and McGilton H. (1995) *The Java Language: A White Paper* and *The Java Language Environment: A White Paper*, Sun Microsystems Computer Company, Mountain View, CA, Edition May 1995 (available at <http://java.sun.com/whitePapers/>)

- Grah93 Graham I. (1993) *Object Oriented Methods*, Addison-Wesley Publishing Company, Reading, MA
- Gray95 Gray N.A.B. (1995) "Patterns for a Framework for Distributed Programs", *Tools Pacific 95*, Melbourne, November 27-30
- Green95 Green H. (1995) "Microsoft to License Sun Java Program", *New York Times Syndicate*, December 7 (available at http://nytsyn.com/live/News3/341_120795_134125_6450.html)
- IBMPress95 IBM Corporation (1995) "IBM licenses Java technology from Sun Microsystems for use in Internet products" *IBM Press release*, December 6 (available at <http://www.ibm.com/News/javapr.html>)
- Koba91 Kobara S. (1991) *Visual Design with OSF/Motif*, Addison-Wesley Publishing Company, Reading, MA
- Koen95 Koenig A. (1995) "Patterns and Antipatterns", *Journal of Object-Oriented Programming*, March-April, pp 46-48, SIGS Publications, CO
- Kras+88 Krasner G. E. and Pope S. T. (1988) "A cookbock for using the Model-View-Controller user interface paradigm", *Journal of Object-Oriented Programming*, 1 (3), SIGS Publications, CO
- Lea96 Lea D. (1996) *Concurrent Programming in Java: Tutorials and Design Patterns*, (available at <http://g.oswego.edu/dl/pats/aopintro.html>)
- Leh+89 Lehman and Belady (1989) as quoted in Sommerville, *Software Engineering*, 3rd ed. Wokingham, England: Addison-Wesley, pp. 546
- Lew+95 Lewis T., Rosenstein L., Pree W., Weinand A., Gamma E., Calder P., Andert G., Vlissides J. and Schmucke K. (1995) *Object-Oriented Application Framworks*, Manning Publications Co., Greenwich
- Love91 Love T. (1991) "Timeless Design of Information Systems", *Object Magazine*, November-December, pp. 46
- Mesz95 Meszaros G. (1995) "Pattern: Half-Object + Protocol", *Pattern Languages of Program Design*, Coplien J. O. and Schmidt D. C. (eds), Addison-Wesley Publishing Company, Reading, MA
- Neve+95 Neves F. and Garrido A. (1995) "Warden: A Pattern for Object Distribution" *Papers of the Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems*, OOPSLA '95 (available at <http://www.cs.wustl.edu/~schmidt/OOPSLA-95>)
- NSPress95 Netscape Communications Corporation (1995) "Netscape to License Sun's Java Programming Language" *Netscape Press release*, May 23 (available at <http://home.netscape.com/newsref/pr/newsrelease25.html>)
- Ott96 Ott R. (1996) "Simple Shared Object: An Architetural Pattern for Simple Object Sharing" in *Object Currents*, Hathaway B. (ed), SIGS Publications, 1(3), March 1996 (available at <http://www.sigs.com/objectcurrents/>)

-
- Pins+88 Pinson L. J. and Wiener R. S. (1988) *An Introduction to Object-Oriented Programming and Smalltalk*, ++, Addison-Wesley Publishing Company, Reading, MA
- Pree94 Pree W. (1994) *Design Patterns for Object-Oriented Software Development*, Addison-Wesley Publishing Company, Reading, MA
- Rich95 Richardson P. (1995) "Distributed Object Computing", *Tools Pacific 95*, Melbourne, November 27-30
- Rumb+91 Rumbaugh J., Blaha M., Premierlani W., Eddi F. and Lorenzen W. (1991) *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, New Jersey
- Rumb95 Rumbaugh J. (1995) "OMT: The function model", *Journal of Object-Oriented Programming*, March-April, pp. 10-14, SIGS Publications, CO
- Schm+94 Schmidt D. C. and Stephenson P. (1994) "Achieving Reuse Through Design Patterns: A Case Study of Evolving Object-Oriented System Software Across OS Platforms", *Proceedings of the 3rd SIGS C++ World Conference 1994*, (available at <http://www.cs.wustl.edu/~schmidt/>)
- Schm95 Schmidt D. C. (1995) "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Dispatching", *Pattern Languages of Program Design*, Coplien J. O. and Schmidt D. C. (eds), Addison-Wesley Publishing Company, Reading, MA
- Selm+96 Selm R. and Hanselmann M. (1996) *Qualitätsfähigkeit*, Verlag Paul Haupt, Berne
- Sole92 Soley R. M.. (1992) *Object Management Architecture Guide*, Published by Object Management Group and X/Open, Framingham, MA
- Souk94 Soukup J. (1994) *Taming C++: Pattern Classes and Persistence for Large Projects*, Addison-Wesley Publishing Company, Reading, MA
- Spur+94 Spurr K., Layzell P., Jennison L. and Richards N. (1994) *Computer Support for Co-operative Work*, John Wiley & Sons Ltd, Chichester, UK
- Stal95 Stal M. (1995) "The Broker Architectural Framework", *Papers of the Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems*, OOPSLA '95 (available at <http://www.cs.wustl.edu/~schmidt/OOPSLA-95>)
- Trew+91 Trew A. and Wilson G. (1991) *Past, Present, Parallel: A survey of available parallel computing systems*, Springer-Verlag, New York
- Wein+88 Weinand A., Gamma E. and Marty R. (1988) "ET++: an object oriented application framework in C++", *Proceedings of the Object-Oriented Programming Systems, Languages and Applications Conference*, pp. 45-57, ACM, September
- Wils+90 Wilson D. A., Rosenstein L. S. and Shafer D. (1990) *C++ Programming with MacApp*, Addison-Wesley Publishing Company, Reading, MA

-
- Your94 Yourdon E. (1994) *Object-Oriented Systems Design: An Integrated Approach*, Prentice-Hall, Englewood Cliffs, New Jersey