Master's Thesis in Telematics
for the Award of the Academic Degree
Diplom Ingenieur
at the
Graz University of Technology

Aspects of a Configurable Component Based Framework for Simulating DSPs

---

submitted by:

Wolfgang Lazian

May 2005

Institute for Information Processing
and Computer Supported New Media

Supervisor: Univ.Doz. DI. Dr. techn. Klaus Schmaranz

Diplomarbeit aus Telematik
zur Verleihung des des Akademischen Grades
Diplom Ingenieur
an der
Technischen Universität Graz

Aspekte eines konfigurierbaren Komponenten Systems zur DSP Simulation

---

vorgelegt von:

Wolfgang Lazian

May 2005

Institut für Informationsverarbeitung
und Computergestützte Neue Medien (IICM)

Begutachter: Univ.Doz. DI. Dr. techn. Klaus Schmaranz

Abstract

Technical systems become more and more complex, which increases the need to ensure the correctness and quality the of parts of those systems as early as possible. Simulations are used to evaluate systems before any physical prototype exists. They allow building of virtual prototypes and permit conclusions about the final product. But the simulation environments of complex systems tend to become very complex on their own.

This thesis researches simulations of complex systems using digital signal processors as an example (DSP). To handle the complexity, a component based design approach has been chosen. The simulation model is created using relatively simple components and by using design methods like Design by Contract™ (DbC) high quality standards of composed systems are ensured.

The results of this work are the Coco libraries (Configurable Components), which define basic building blocks for component-based software. Using these the xSim application has been developed to simulate a state of the art DSP.

Kurzfassung

Im Zuge der ständigen Weiterentwicklung werden technische Systeme immer komplexer und aufwändiger. Dementsprechend rückt das Sicherstellen der Funktionalität und Korrektheit schon während der Entwicklung immer weiter in den Vordergrund. *"First time right"* Simulationen erlauben das Erstellen von Prototypen und lassen Rückschlüsse auf das fertige Produkt zu. Die Simulation eines komplexen Systems tendiert jedoch oft selbst zu einem zu werden.

Diese Arbeit untersucht die Simulation komplexer Systeme anhand Digitaler Signalprozessoren (DSPs). Um den Aufwand der Erstellung und Modellierung so gering wie möglich zu halten, wird ein komponentenorientierter Ansatz verfolgt. Um das System überschaubar zu halten wird das Simulationsmodel aus wenigen Komponenten mit geringer Komplexität zusammengesetzt. Design by Contract™ (DbC) orientierte Methoden stellen die Qualität der einzelnen Komponenten sicher.

Die Ergebnisse dieser Arbeit sind die Coco Libraries (Configurable Components), die als Basis Bibliotheken zur Erstellung von Komponenten dienen. Mit ihrer Hilfe ist die Applikation xSim erstellt worden, die einen State of the Art DSP simuliert.

Acknowledgments

First of all I want to thank my family, especially my parents who made all this possible and supported me throughout my whole studies.

I highly estimate the support of my girlfriend Alexandra which goes far beyond of what I expected.

Special thanks go to Christian Panis who gave me the chance to be part of the *xDSPCore* team.

I want to thank Klaus Schmaranz, the supervisor of the thesis and especially my friend Gunther Laure. Together we designed, implemented and tested the complete *Coco* framework.

Additionally I would like to express my gratitude to the following people for their support and assistance: Harald Krottmaier, Christoper Gallé and David Riebenbauer for the inspiring discussions and for proofreading my thesis.

I hereby certify that the work reported in this thesis is my own and that the work performed by others is appropriately cited.

Signature of the author:

Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die, dem Aufgabensteller bereits bekannte Hilfe selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten Anderer unverändert oder mit Abänderungen entnommen wurde.

Table of Contents

Abstract 1

Kurzfassung 2

Acknowledgments 3

1 Introduction 16

1.1 Motivation 16

1.2 Aspects of Simulation 20

1.3 A Real World Scenario 24

1.3.1 A Simple Processor 24

1.3.2 Instruction Set Simulation 26

1.3.3 Conclusions of the Real World Scenario 27

1.4 Motivation for Component based Design 29

1.5 Outlook of this Thesis 31

2 Background of Component Based Design 34

2.1 History of Production of Interchangeable Parts in America . . 35
2.2 Evolution of Software Development 36
2.3 Software Paradigms 38
2.3.1 Structural Programming 39
2.3.2 Object Oriented Programming 39
2.3.3 Meta Programming 40
2.3.4 Generic Programming 42
2.3.5 Component Based Programming 42
2.4 Concepts of Component Based Approaches 43
2.4.1 Pipes and Filters 44
2.4.2 Plugins 45
2.4.3 Component Models 46
2.4.4 Web services 55
2.5 Chapter Conclusions 57

3 Reusable Software Components 60

3.1 Component Definition 61
3.2 Software Quality 64
3.2.1 Definitions of Software Qualities 64
3.2.2 External Quality Factors 65
3.2.3 Internal Quality Factors 69
3.3 Approaches to Reusability 72

3.4 Testing 73

3.5 Composition 75

3.6 Design by Contract™(DbC) 77

3.6.1 Human contracts 77

3.6.2 Contracts in software 78

3.6.3 Contracts for Components 81

3.6.4 Limitations of Interface Definition Languages 83

3.6.5 Design by Contract™ versus Defensive Programming 84

3.7 Conclusions 85

4 Coco Component Framework 87

4.1 Coco-Component 88

4.1.1 Structure 88

4.1.2 Participants 88

4.1.3 Implementation 89

4.1.4 Design by Contract™ in Coco-Component 92

4.2 Coco-CompositeComponent 101

4.2.1 Structure 102

4.2.2 Participants 102

4.2.3 Implementation 102

4.2.4 Outlook 104

4.3 Coco-ComponentModel 105

4.3.1 Structure 105
4.3.2 Implementation 106
4.4 Coco-Marshaller 109
4.4.1 Implementation 109
4.4.2 Outlook 112
4.5 Component Organisation 112
4.5.1 Singleton 113
4.5.2 Coco-ComponentFactory 115
4.5.3 Coco-ComponentRegistry 117

5 Coco Soap Library 118

5.1 Simple Object Access Protocol (SOAP) 119
5.2 The gSOAP Toolkit 122
5.3 gSoap Code Generator 123
5.3.1 Generating Client Stub 123
5.3.2 Generating Server Skeleton 125
5.3.3 Client Example 127
5.4 Automated Facade Class Generation 128
5.5 Bidirectional Communication 129

6 Summary and Conclusions 132

References 136

Index **141**

List of Figures

1.1 The configuration file is center of the tool-chain 19

1.2 Simple processor 25

1.3 Simulator generation 29

2.1 Spagetti code 38

2.2 CORBA framework 47

2.3 COM component 49

2.4 COM interface structure 50

2.5 EJB: Home and Remote interface 54

2.6 Web service: Interaction between UDDI, WSDL and SOAP . 57

3.1 Simplified layers for easier correctness proving 66

3.2 Few interfaces: bad example 71

3.3 Few interfaces: good example 71

3.4 UML example 76

4.1 Component structure 89

4.2 Example for needed invariant check at method entry point . . 93
4.3 Structure of an observable component 100
4.4 MultihookUp as Observer 101
4.5 Spanning hull of a composite component 101
4.6 Composite component structure 102
4.7 Subtracter that is composed through an adder and inverter . 105
4.8 Structure of the component model 105
4.9 Sequence diagram of instantiating components 107
4.10 Sequence diagram of connecting components 108
4.11 Marshaller concept 110
4.12 Structure of the component factory 115

5.1 Constructing a client with gSOAP 124
5.2 Constructing a service with gSOAP 126
5.3 Wrapper for transparent function call on client 128
5.4 Automatic generation of client wrapper 129
5.5 Bidirectional communication between simulator and GUI . . 130

6.1 Replace MAC_1 unit with another MAC_2 unit 133

Listings

1.1 Simple assembler program 26

2.1 Factorial numbers with recursion 41

2.2 Factorial numbers with template metaprogramming 41

2.3 Simple Pipes and Filters example 44

3.1 Example with pre- and postconditions 79

3.2 Syntax of an invariant in Eiffel 80

3.3 CORBA IDL example 83

3.4 Defense programming example 85

4.1 Query interface for Coco::Component (a) 89

4.2 Query interface for Coco::Component (b) 90

4.3 connect method 91

4.4 is connectable to hook method 91

4.5 method with invariant check 92

4.6 template method example 94

4.7 template method screen dump 94

4.8 calling template method in constructor 95
4.9 Hook and Hookup Container 97
4.10 doRegisterInterfaces example 97
4.11 configuration by marshaller 98
4.12 configuration by marshaller 98
4.13 concrete composite component 103
4.14 methods to add or remove a component 103
4.15 query methods of composite component 104
4.16 method to build the component model 106
4.17 method to connect components 108
4.18 public methods of a marshaller 110
4.19 friend of derived class 111
4.20 example of tagging a member for marshalling 112
4.21 using singleton as proposed by Alexandrescu 113
4.22 instantiate singleton more than once 113
4.23 create singleton as proposed by Alexandrescu 113
4.24 singleton concept used in the framework 114
4.25 valid code without having a private copy constructor 114
4.26 methods of the component factory 115
4.27 Creator defined for the component factory 116
4.28 Example to register a product 116

4.29 methods of the component registry 117

5.1 SOAP envelope 120

5.2 SOAP message 120

5.3 SOAP message with schema typed values 121

5.4 stub routine 124

5.5 namespace mapping table (1) 125

5.6 namespace declaration of method 126

5.7 namespace mapping table (2) 126

5.8 SOAP client example 127

5.9 Wrapper code for transparent method invocation 128

Preface

The Coco Component framework was studied, designed and implemented by Wolfgang Lazian and Gunther Laure.

The following chapters originated in this cooperation and thus are found in both theses:

- Introduction
- Background and History of Component Based Design
- Small parts of the Coco Component Framework chapter

Wolfgang Lazian's thesis focuses on the comparisons of existing component technologies, the architecture of the Coco Framework in depth and distributed technologies based on SOAP.

Gunther Laure's thesis examines different techniques to incorporate Design by Contract^{TM} in C++, component oriented design patterns and the Coco XML language developed for the Coco framework.

Chapter 1

Introduction

This chapter gives a short introduction to system simulation. The goal of it is to develop an application that simulates the functionality of a digital signal processor (DSP). A real world example is used to describe the model we intend to simulate and which features are needed. The chapter concludes with our objective to choose a component based approach.

1.1 Motivation

This project started during an internship at Infineon Technologies in Villach. The branch of telecommunications had just redesigned one of its line card products that is used by telephone providers in their switching centers. Support for emerging broadband technologies like DSL[1] had to be included. The Alca[2] project was started to design a new digital signal processor, especially for this purpose. It should replace the previously used micro controllers and provide enough computing power for upcoming DSL algorithms.

Our task was to implement the basic software tool-chain for Alca, which consisted of an assembler, a linker and an instruction set simulator. After

[1] Digital Subscriber Line

[2] Advanced Linecard Adapter

a short development time, the tool-chain was able to implement DSP algorithms, to translate them to binary code, and to execute them using the simulator. It was possible to advance through the application code and to display changes in pipeline memory and registers.

The former Alca project became xDSP and an updated hardware design experienced many architectural improvements to move into the field of general purpose DSPs. The project's software developers (= internals) new task was to modify the software tool set to support the new hardware design.

The following are the tool set changes:

- Dual Harvard load-store architecture
- VLIW instruction set (Very Long Instruction Word)
- Instruction buffer
- Orthogonal register file
- 4 or 5 stage pipeline (fetch, align, decode, execute 1, execute 2)
- Multiply and Accumulate unit (MAC)

xDSP's architecture was designed to fulfil the above mentioned requirements and to enable the development of an optimizing C-compiler. The instruction simulator now had to emulate a variable sized instruction cache, a longer pipeline, more parallel working execution units, and last but not least a much larger assembler instruction set.

Alca's simulator followed a very simplistic design. Every assembler instruction was implemented as a C function with the provided operands. For the relatively small instruction set this approach was appropriate but the new instruction set was about four times bigger and the design did not scale well. Additionally the new MAC unit turned out to be additionally problematic. Depending on the core's clock frequency the MAC unit did not compute a result in one cycle. MAC instructions had to be separated into

at least two parts to replicate the timing properties of the new MAC unit design (*execute 1* and *execute 2* pipeline stages). The number of instruction implementations doubled. Still the necessary functions were written but the simulator's complexity became awkward. (Just think of the maintenance of the instruction decoder.)

Development of assembler, linker and simulator had been relatively independent of each other. No common design or code base was used and any documentation was manually written, sometimes by internals not directly related to the project. The information synchronization's among the tools and documentation became the main drawback. Imprecise instruction definitions resulted in different and incompatible implementations of the same instruction set. At times the assembler did not know of instructions that were described in the documentation and interpreted by the simulator and vice versa.

As the development of the xDSP proceeded, the idea of a configurable core emerged. Previously *one fits all* architecture tried to match all thinkable customer needs. This resulted in multipurpose cores that are generally usable but lack efficiency. In most cases, core features remained unused but resulted in additional power consumption and silicon area. A configurable core can be customized to the customer's needs. The customer can decide on the number of registers or the encoding and size of the instruction set.

The xDSP architecture became scalable and allowed the parametrization of various architectural features: [PHF+04]

- Data memory: Size, bus width, interleaving
- Program memory: Size, bus width
- Register file: Number, type and bit width of the registers
- Instruction set: Every instruction is adjustable in opcode encoding, operand number and types and execution timing (according to execution unit configuration)

- VLIW: Number of parallel executable instructions
- Instruction cache: Cache size, bus size and timing behaviour.
- Pipeline: Number of stages, to scale clock frequency

Some of these were simple to implement like adjustable memories or bus widths, but at that time the instruction set became adjustable too. A new approach had to be found. A simulator based on the Alca design could not fulfil this task. The development of a consistent and workable tool-chain for a static and unconfigurable core had already become quite complicated. Additionally the design of the software tools for a *moving target* needed a common approach for configuration support. A configuration to facilitate all parts of the tool-chain including the documentation of the instruction set was needed. The tool-chain (assembler, linker, simulator, and compiler) had to be synchronised in all cases (see figure 1.1).

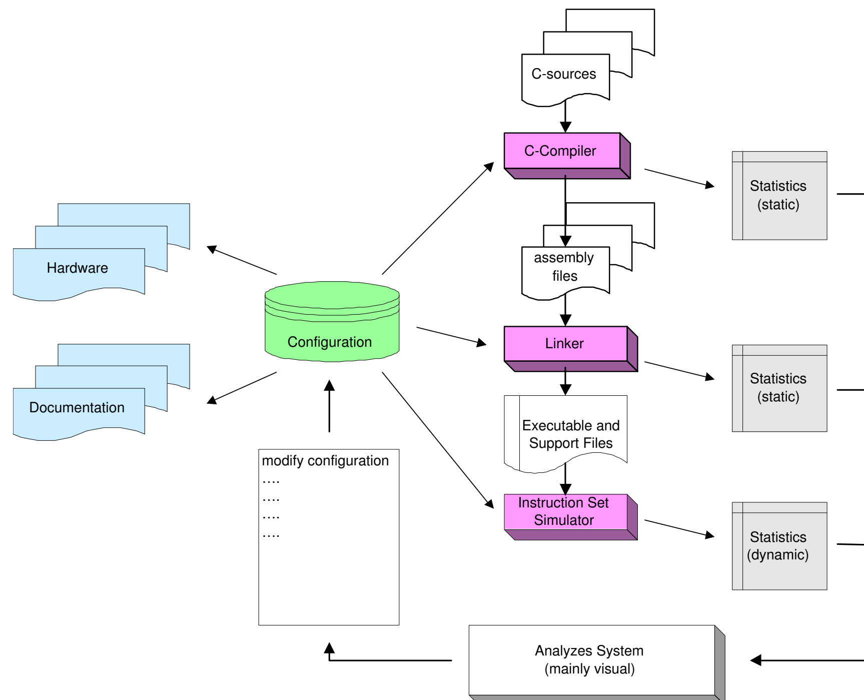

Figure 1.1: The configuration file is center of the tool-chain

The design requires that the configuration should be given by a configuration file. It must be humanly readable and editable, due to the fact that a customer should be able to change it. For the tools of the tool-chain a library had to be developed that allows access to the configuration file (see [Lau05]). Ideally all changes to the architecture should only be reflected in modifications of the configuration file and not affect any source code changes.

The above problems of exchange and modification of core parts but not recompiling anything, brought the component based development of the simulator to mind. Any simulation model should be assembled using small, but meaningful components which can be gathered in a library. The fact that there was no free framework available, which permit us to configure and connect components at runtime using a readable file (see section 2.4 on page 43) forced us to develop a new framework.

1.2 Aspects of Simulation

> "The technique of imitating the behaviour of some situation or process (whether economic, military, mechanical, etc.) by means of a suitably analogous situation or apparatus, especially for the purpose of study or personnel training."
> *Oxford English Dictionary*

Simulation is the process of designing a model of a real or imaginary system and conducting experiments with that model. One of the pioneers of simulation was John von Neumann. He had the idea to repeat a model multiple times to gather statistical information about it. With this information he derived the behaviour of the real system. His system is known as the "Monte Carlo method" [MRR+53].

Computer simulation was first used on large scale during the Manhattan Project in World War II. It was used to model the process of nuclear detonation and used a Monte Carlo based method.

If the system to simulate is simple, it may be represented and solved analytically. In this case, one or more mathematical equations defined the model. But most problems in the real world are more complex and a mathematical model cannot be found.

Reasons for using simulations:

- **Cost:** An experiment on a real system would be too expensive. This is also the case in the construction of DSP's or similar devices. When designing a new architecture for such a computing device, several specification data, such as maximum power consumption, are considered. It is very costly to produce a prototype every time the design should be tested whether or not it is compliant with the requirements. In this case simulations will give invaluable information about the core without building a hardware prototype.
- **Ethical, Dangerous:** In some situations it is ethically not justifiable or too dangerous to have real persons involved. For example car crash tests for the automotive industry.
- **Experiments:** Due to the fact that the simulator is constructed in software, it is easier to change something than in any real system. An example is the planning of a house. With a simulator it is easier to change the positions of some walls.
- **Speed:** If the real system is too fast, for instance an electric circuit, it is nearly impossible to see the effects as they occur.

In the case of processor simulation we follow the definition given by scientists from Princeton University:

> "(computer science) the technique of representing the real world by a computer program; a simulation should imitate the internal processes and not merely the results of the thing being simulated"
> *Princeton University, WordNet*

Generally, simulations of processors can be described as *discrete event simulations*, where a simulator holds a queue of events sorted by the simulated time they should occur (VHDL Simulator, SystemC). The simulator processes the queue and triggers new events as each event is processed. This type of simulation seldom executes in real-time, but it is possible to gather data to find defects in the logic design.

For DSP instruction set simulators we have to construct a model that behaves like the real DSP, but does not resemble the hardware design at the lowest level. This allows faster simulations while producing the same output for the same input. It is possible to use discrete event simulation but often it is too slow for DSP algorithms of large input data sets. In this case a static timing on clock cycle basis is used. Depending on the desired details and speed these are some aspects that have to be considered when constructing a simulator for a computing device:

- **Functional simulation:** For a functional simulation, only those things are implemented that the programmer sees. This is mainly the instruction set of the processor. It does not include internal architectural details but it must provide the same results as the processor. Due to the fact that internal unit simulation is not performed, it is very fast.

  A drawback of this strategy is the loss of internal information such as timing information or power consumption and it cannot be used for real time simulation.

  Emulation by interpretation is a standard approach. The disadvantage of this method is that it is quite slow and sometimes slower than normal execution. Another approach is using a compiled simulator [FKH] where a program for the DSP is translated into a high level language like C and then is compiled and executed on a standard personal computer. In this case every program has to be compiled in advance. The advantage of this method is the gain in performance as the execution speed is comparable and often faster than the program running on the original architecture.

- **Performance simulation:** When dealing with performance simulation, the micro architecture of the processor has to be implemented, too. The micro architecture is not visible to the DSP application programmer but is needed to emulate real-time behaviour. Performance simulation can further be divided into the following parts:

  - Instruction scheduler: The simulator schedules the instructions based on resource availability. Only one instruction at a time can be processed and is therefore not suitable for *very long instruction word (VLIW)*-DSP's.
  - Cycle timers: Here the simulator tracks the microarchitectural state of each cycle. So the simulator state corresponds to the microarchitectural state. Furthermore many instructions can be processed at any time which is needed when dealing with VLIW.

Profiling is also a serious part in the simulation process. Statistical data is gathered and displayed in an arbitrary fashion. The most important profiling methods are:

1. **Static profiling:** Static profiling includes all information that is still known and available before the program is executed. That includes for example the number of instructions, the number of different instructions and the code density only to mention a few of them.
2. **Dynamic profiling:** Dynamic profiling involves data that occurs when a program is executed. This is the number of cycles, the number of loop iterations and the number of instructions that are executed or not. This allows conclusions about the efficiency of the implemented algorithms depending on runtime data.

The more information that is gathered through the profiling process, the better the developer of the real system can optimize it.

Although there are some tools that help in developing a simulator (such

as the *SimpleScalar* [LLC] toolkit) most simulators that exist nowadays have the following problems in common:

- They are always developed with a specific architecture in mind. It is nearly impossible to support more than one architecture using their code base.
- They usually depend on specific platforms.
- Changes have to be made in the source code of the simulator. Therefore the person who has to change something, has to be experienced in a specific programming language.
- They are not free or freely available.

1.3 A Real World Scenario

The idea of developing a configurable DSP is not new. There are already some projects like the *DSPxPlore* [PHL+04] or the one from *Tensilica* [Inc]. These configurable DSP cores have the advantage of being adaptable and therefore usable in a wide field of applications. Customers have the ability to get a specific DSP that fits their personal needs.

In the following sections we discuss a typical real world scenario.

1.3.1 A Simple Processor

This example scenario describes a very simple processor with load-store Harvard architecture. The processor consists of following units:

Data Memory: A volatile storage for arbitrary data with 1kB memory.

Program Memory: This unit holds the program which should be executed by the processor in the form of machine instructions.

Register File: The register file is a container of registers. That means that every register is part of it. Our example processor has two registers with 16 bit width (d0 and d1).

Execution Unit: Performs something depending on the given instruction. It is connected to the data memory and the register file.

Instruction Decoder: The instruction decoder is used to detect binary opcodes that symbolise the processors instruction set. It generates signals that are used to control the execution unit.

Pipeline: The pipeline consists of the basic stages: fetch, decode and execute. Read and write operations of the instructions arguments are done in the execute stage.

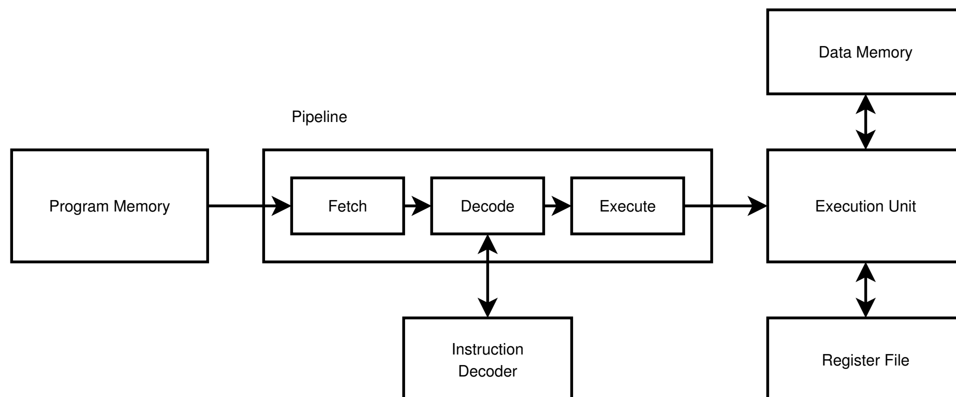

Figure 1.2: Simple processor

Figure 1.2 shows how the units are connected. An instruction is first fetched from the program memory. The next cycle decodes the instructions opcode to generate the control signals for the execution unit. In the following step, the instruction is executed. The execution unit reads the values addressed by the instructions operands, calculates the result using internal units and then stores the result in the designated register.

Our processor has a small instruction set consisting of 4 instructions only:

LOAD #address, reg : The first operand is a 16 bit unsigned value that represents an absolute address in data memory. The second operand is a register. This instruction reads the addressed value from the data memory and stores it in the given register.

STORE reg, #address : This is the reverse instruction to LOAD and stores a register value in the data memory.

ADD reg, reg : This instruction adds to register values and stores the result in the second register.

NEG reg : The given register value is inverted.

This minimalistic core may be sufficient for a specific task but it may not be usable for others. It is very likely that 1 kByte data memory or only two distinct registers may not enough and most algorithms may not be implemented efficiently using the small instruction set. But due to the fact that the core is configurable, it is no problem to alter parts of the architecture.

1.3.2 Instruction Set Simulation

In this section a simple test program for the processor defined in the previous section is developed and executed. The source code defined by listing 1.1 loads registers d0 and d1 with data memory values. Both registers are added and the calculated result is stored in the data memory.

```
1    ; load a 16 bit from data memory
2 LOAD 0x80, d0
3    ; load a 16 bit from data memory
4 LOAD 0x82, d1
5    ; Add both values
6 ADD d0, d1
7    ; Store the result
8 STORE d1, 0x84
```

Listing 1.1: Simple assembler program

We assume that the simulator provides a gdb[3] like user interface. After starting the instruction simulator, we *load* the assembler program which is stored in the simulator's program memory emulation. Interactive mode allows us to *set* the addressees data memory cells with predefined values (0x80=10, 0x82=ff). A *breakpoint* is set to line 8 to interrupt execution allowing to check register d1 for the calculation result. The *run* command triggers continuous execution until a breakpoint or the end of the program is reached. Alternatively the *step* command can be used to execute only one instruction. The *back step* command traces the execution backwards.

Various *views* and *watches* may be defined to inspect specified memory address, data registers or internal state registers. The built in profiler collects runtime statistics like register and memory usage and how often instructions of the program are executed.

Architectural modifications should not influence the general work-flow sketched above. If it's possible to propagate the changes during runtime or the applications startup phase, two additional registers (d2, d3) may be added without the need to change the simulator application.

1.3.3 Conclusions of the Real World Scenario

In the example scenario sketched above, features are outlined that need attention in the creation of a framework for simulating a configurable DSP. Keep in mind that the example scenario was a very simple abstraction of a real processor. Its intention was not to describe all the features and components involved in such a unit but to show key features a configurable framework should fulfil. These are:

- The framework is configured through a configuration file, that must be humanly readable so that the user of the framework can easily change it without in depth knowledge of how the framework works. This configuration file contains the configuration data of each individual

[3]GNU Debugger

component and the data of how they are connected.

- Having such a configuration file implies having a proper parser for it. The parsing and processing library developed throughout this thesis is described in [Lau05].
- Every component has to have some sort of marshalling and unmarshalling mechanism with which it is possible to store/read a state to/from any medium. With such a mechanism there is the ability to simulate to a desired point in time and to restart the simulation from that point. So that it is not necessary to restart the simulation from the beginning.
- Components can be added and removed. This should be possible without a recompilation step. Additionally this should be possible at runtime. Components therefore have to be available through some sort of library to which the user has access.
- There must be some sort of connection mechanism for the components. To realize such a mechanism, the involved components have to fulfil certain contracts and interfaces. How this mechanism is realized is outlined in [Lau05].
- The framework must be able to connect and initialise the components as they are given through the configuration file. *CocoComponent-Framework* is the underlying framework developed in this thesis and described in chapter 4 .

Figure 1.3 gives an overview of the key features and how a simulator for a specific configuration is constructed. The framework that reads the configuration file is able to construct components given by a component library and connects them properly. After that the simulator is ready to use them.

There are a lot of other features that have to be considered like the communication over a network if several parts are located on different machines. When talking about distributed computing, other things have to be taken

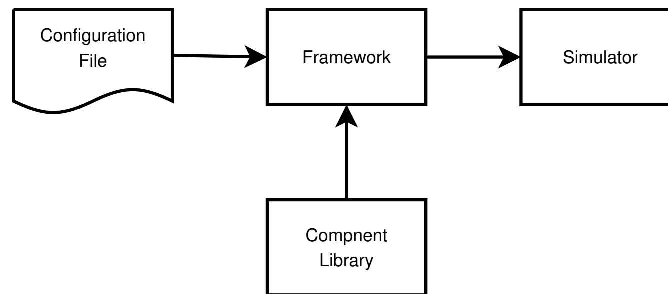

Figure 1.3: Simulator generation

into account like security issues. These aspects are beyond the scope of this thesis but some of them are discussed in [Has03].

For the user of such a simulation framework, it would be comfortable to have some sort of graphical user interface for configuring the framework and visualisation of the components data. Although this is not a key feature of the basic idea of developing a configurable simulation framework, some ideas and concepts are discussed in [Lau05].

1.4 Motivation for Component based Design

Software systems tend to become very complex, which is especially true for simulation applications. If such complex systems are realized in a monolithic fashion, they tend to become unwieldy beasts, impossible to maintain or extend. In addition to this drawback, only those who are familiar with the system can change it. If such a person leaves the development team, another person has to spend a great amount of time to understand the system. Bugs are sometimes very hard to find and fix. Such bug fixes often induce other bugs.

Breaking complex systems into smaller parts is a common way to overcome the whole problem[4]. A DSP can easily be regarded as a system con-

[4] divide and conquer

sisting of different parts. There we have the program memory which holds instructions, the data memory, which acts as storage or an execution unit which performs something depending on the instruction. These smaller parts can be solved independently of each other and be put together to solve the initial task. The advantage of that strategy is that the problems can be solved parallelly by different developing teams which decreases the development time. A disadvantage is that sometimes the interfaces is not well defined and therefore different components may not interact correctly.

Component based development can be a solution for the problem described above. The keywords for this software developing process are *extensible*, *platform independent*, *reusable* and *configurable*.

There are are several definitions of what a component is [com]. All definitions have in common that a component is software that performs some sort of service. Additionally each has a well defined interface for the interaction with other component. Components can be compared to LEGO™bricks. Different bricks and pieces fit together because they have the same interface.

Because of well defined interfaces, components should be exchangeable with another components. The register file of the real world scenario section 1.3 can be replaced with a register file having more registers.

Several techniques emerged that have components in mind:

- **CORBA**[5] was developed by the OMG[6] and is an infrastructure which enables applications, written in any language and by any vendor, to work together over the network
- **COM[7]/DCOM**[8] was developed by Microsoft™and has the same purpose as CORBA. Further techniques that evolved are *COM+*, which allows native C++ calls to be translated into the correct COM call,

[5]Common Object Request Broker Architecture [cora]
[6]Object Management Group (see `http://www.omg.org/`)
[7]Common Object Model [com05]
[8]Distributed Component Object Model [dco05]

and *.NET*, which is a framework for Web services and component software and is similar to *J2EE*[9]

- **EJB**[10] is Java's answer to the distributed component system although components can only be written in the Java™programming language.

A component model specifies the requirements a component has to fulfil. Due to the fact that every model has different requirements, a component can only be created for a specific model. If it should be of use in another model, it has to be adapted.

The above mentioned models have a serious constraint: they do not provide a runtime configuration and connection mechanism with a humanly readable file. In this thesis a component model is developed that overcomes this drawback.

1.5 Outlook of this Thesis

Chapter 2, Background of Component Based Design:

This chapter starts with a historical view of the evolution of the idea of interchangeable parts. Focusing on the beginning of the industrial age when it was not usual to have factories that produced high precision goods.

The evolution of software development is described. MacIlroy talked about a *software crisis* and the idea of mass-production techniques in software. Analogies between the industrial revolution and the software industrial revolution are shown.

The chapter concludes with an overview of concepts and techniques for using and developing software components.

[9] Jave 2 Platform, Enterprise Edition
[10] Enterprise Java Beans [ejb05]

Chapter 3, Software Components:

Component oriented software or component based software design are hot technologies often discussed in software engineering. Nevertheless there is no clear definitions of the terms *Software Component* and *Component Software* neither what they really are nor how they are correctly composed.

This chapter starts with some definitions of a Software component and tries to summarize them in to a general definition. Therefore an overview of software qualities and approaches to reusability are given.

Aspects such as Design by Contract^{TM} and testing strategies are explained.

Chapter 4, Coco Component Framework:

This chapter describes the design and implementation of all component related aspects. This includes the description of a component itself, the composite component, the component model and the marshaller for reading and writing states.

All developed parts have a *Coco* prefix which is an abbreviation for *ConfigurableComponent*.

The used language is C++. As with every programming language, there were a lot of problems that had to be solved in order to realize the design which is also discussed in this chapter.

Chapter 5, Coco Soap Library:

The *Hook* concept discussed in [Lau05] so far has the drawback that connected components have to be in the same address space. Components should not be bound to only one machine. Distributed communication is one of the main concepts of component based development.

As already discussed in section 2.4.3 component models have evolved over the years. Most of them communicate through a binary protocol. Version mismatches of those protocols sometimes hinder component communication with each other.

Web services are one concept that tries to use as many standardized techniques as possible. For communication the SOAP protocol is used which transfers XML-based messages between the communication endpoints. Customers can easily read the message stream as it is a humanly readable format.

Chapter 2

Background of Component Based Design

> "The paradigm that may launch the Information Age is similar to the one that launched the Manufacturing Age 200 years ago. It is a product-centric paradigm in which progress is measured by the accretion of standard, interchangeable, reusable components, and only secondarily by advancing the processes used to build them."
> *Brad J. Cox*

This chapter starts with a historical view of the evolution of the idea of interchangeable parts. Focusing on the beginning of the industrial age when it was not usual to have factories that produced high precision goods.

The evolution of software development is described. MacIlroy talked about a *software crisis* and the idea of mass-production techniques in software. Analogies between the industrial revolution and the software industrial revolution are shown.

The chapter concludes with an overview of concepts and techniques for using and developing software components.

2.1 History of Production of Interchangeable Parts in America

It was the army, that sponsored the idea of mass production of arms. Before this was realized, arms had to be built by skilled persons. Every weapon had an individual touch since all parts were made by hand. These were usually not produced to be exchangeable. As a consequence of this situation defective arms could hardly be repaired and skilled people were needed.

In the year 1785 Thomas Jefferson was in France. He was inspired by Honore le Blanc who produced guns, with such precision, that parts could be easily exchanged. Jefferson realized the advantage of building such arms because of the possibility to repair them. But he could not convince Blanc to move to America to demonstrate his technique.

1798 Eli Whitney stated that he was able to mass-produce muskets consisting of interchangeable parts. He had three principles to achieve this goal:

1. Use water-powered machinery
2. Produce identical parts
3. Use unskilled people to assemble the uniform parts

The government gave him a contract to produce 10.000 guns within two years. Due to the fact that no machinery or techniques for mass production were available Whitney needed eight years to fulfil the contract but was then able to produce another 15.000 muskets within another two years.

The American system of manufacturing [Smi] is an example of the fact, that new ideas and techniques need time to be realized and performed. A similar situation is given in the software industry. Everytime a new paradigm comes up, some years can go by until developers recognize its benefits. The reason is nearly always their value rigidity. Brad J. Cox said [Cox90]:

> “ Revolutions happen so slowly, and often displace one group by another, because of value rigidity, the inability to relax the pursuit of an older good to gain a newer one.”

2.2 Evolution of Software Development

The first programmers had to work with computers that resided in the same place where they were created. They were big clumsy machines that were very unreliable and slow. Besides they had memory capability constraints.

At the time, a competent programmer had to be curious and find clever tricks. Additionally there was the opinion, that programming was nothing more than optimizing the efficiency of the computational process due to the limitations of the machines. These limitations forced them to push the machines to their limit. Everyone thought, that if the machines become more powerful, programming them would no longer be a problem.

As the next generation of computers evolved Edsger W. Dijkstra stated [Dij72]:

> “ ...instead of finding ourselves in the state of eternal bliss with all programming problems solved, we found ourselves up to our necks in the software crisis.”

The problem of this crisis was the fact that the new machines were too powerful and reliable. Furthermore a lot of additional techniques such as interrupts appeared, making the life of a programmer more difficult. They had to think about problems and solutions they only dreamed of a few years earlier.

The software crisis appeared because of the need to write correct, understandable and verifiable computer programs due to the rapid increase of the

complexity of the problems that could be tackled by the upcoming, powerful computers. In the *The Humble Programmer* Dijkstra also said:

> “ The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.”

The first person who announced the term *software crisis* was M. D. McIlroy. In 1968, he talked at the NATO conference on software engineering in Garmisch about *Mass Produced Software Components* [McI69]. He said that the software industry is weakly founded. A reason for that is the absence of a software subindustry. Such industry should produce software components for any desired job. A customer should be able to take a catalogue of software components and order a specific one that fits his needs. The aim is that everybody can assemble a program out of prebuilt components. McIlroy compared that strategy with that given in the hardware industry where catalogues of standard parts are available.

Brad J. Cox spoke about the *software industrial revolution* [Cox90] in which programmers stop coding everything from scratch. They should not reinvent the wheel everytime. In his mind, programming should be the transformation from a cut-to-fit craft into manufacturing. Consumers should be able to solve software problems. He compared this with the plumbing problem where consumers assemble their own solutions out of off-the-shelf components which can be bought. The success of the plumbing supply market is the usage of a producer and consumer hierarchy. The plumbing supply market lets the plumber only solve problems of a distinct level in the producer/consumer hierarchy. It’s the task of the supplying market to think of lower level problems like the reinvention of a pipe.

2.3 Software Paradigms

In this section, the most important software paradigms are described which were invented to write readable, easily understandable and reusable programs.

Unstructured Programming

The whole code is written in a single continuous block. The logic moves from routine to routine without returning to a base point. To prevent code duplication, the *Goto* statement was used, which jumps to a desired point in the program specified by a label or a line number.

The disadvantage of this paradigm is that the programmer was encouraged to write nearly unmaintainable, unreadable and hardly understandable spaghetti code (see figure 2.1).

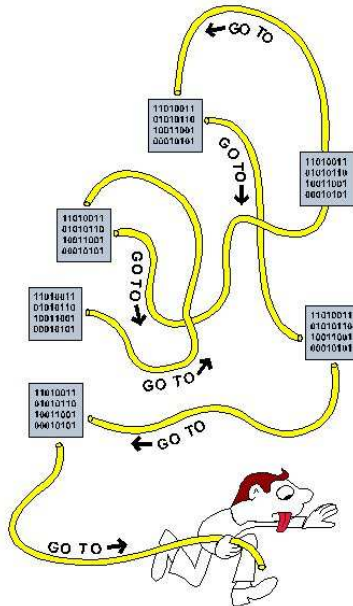

Figure 2.1: Spaghetti code (from Computer Desktop Encyclopedia ©1998 The Computer Language Co. Inc.)

2.3.1 Structural Programming

Structural programming means that a complex problem is broken into simpler parts that are small enough to be understood easily. These smaller problems are then implemented using hierarchical program flow structures like functions and procedures. Statements are organized in a specific manner to minimize error or misinterpretation. The use of the well known GOTO statement is discouraged.

This technique was the first step towards the creation of reusable and easy understandable programs. The programmer is able to use procedures created and tested by some other person. He must not be aware of the internal functionality. Only the fact that the function solves a desired problem is important and not how this is done.

Providing fundamental functions to others is often done by means of a library. A library is nothing more than a collection of a family of functions. An example of such a library is the mathematical library given in nearly every programming language. Mathematical functions solving sine or cosine computations can be taken from there.

The programs became more reliable using this paradigm. There is also a decrease in the time to market because the different parts of the program could be implemented and tested independently as long as the interfaces between the modules were clearly defined.

A disadvantage of structural programming was that data is regarded as something external. They are separated from the instructions.

2.3.2 Object Oriented Programming

Object oriented programming is the approach to combine data and its procedures. It consists of objects that interact with each other. The following concepts are emphasized:

- **Class:** A class describes an abstract data type and its partial or total implementation. An abstract data type combines data and its methods.
- **Encapsulation:** It is also known as information hiding. A client using the object is only able to use the methods that are available through the public interface. This ensures, that features used for internal purpose are hidden from the client.
- **Polymorphism:** This is the ability that different objects can have the same interface but behave in a different manner. Having the same interface, it is possible to interchange those objects with each other.
- **Inheritance:** Inheritance provides a classification mechanism to control the increasing complexity of software development. A class can inherit all properties and behaviour from an existing one.

The object oriented paradigm encourages the programmer to develop a rich model of the problem domain which did not yield an architecture that could be easily adapted to changing requirements. A further drawback is that there is no clear separation between computational and compositional aspects.

2.3.3 Meta Programming

Meta Programming is the technique of creating programs that in turn create programs. The aim is that things get done at compile time instead of at runtime. Thus at runtime more things can be done.

C++ has a built-in meta programming facility called *Template metaprogramming.*

The following two examples shows how factorial numbers can be calculated with either using recursion or template metaprogramming.

```
1  int factorial(int number)
2  {
3    if (number == 1)
4    return 1;
5    else
6    return number * factorial(number - 1);
7  }
```

Listing 2.1: Factorial numbers with recursion

```
1  template <int number>
2  struct Factorial
3  {
4    enum { value = number * Factorial<number - 1>::value };
5  };
6
7  template <>
8  struct Factorial<1>
9  {
10   enum { value = 1 };
11 };
```

Listing 2.2: Factorial numbers with template metaprogramming

Lets look at what happens, when the factorial of the number 4 is calculated in the two cases:

- **recursive:** factorial(4) evaluates to `(4 * 3 * 2 * 1)` which is 24. Three recursions are necessary to get the result.
- **template metaprogramming:** The line of code that is needed looks like this one: `Factorial<4>::value` which also returns 24. In this case this value is a constant value evaluated at compile time. No further runtime calculation is needed.

Andrei Alexandrescu is well known for his advanced usage of template metaprogramming [Ale01]. An example is the Typelist idiom explained in more detail in [Lau05].

2.3.4 Generic Programming

Generic programming is a technique to write programs as general as possible.

Krzysztof Czarnecki has the following definition for generic programming [CE00]:

> "... deals with finding abstract representations of efficient algorithms, data structures, and other software concepts ..."

A lot of object-oriented languages like *C++* or *Eiffel* provide the ability for generic programming. The *Standard Template Library*(STL) is a very popular library that extensively uses generic programming.

2.3.5 Component Based Programming

The main principles of component based programming are:

- **Composition:** Software should be assembled with prefabricated components like in the hardware industry.
- **Multiple-use:** Components should be constructed with reuse in mind. They should not be limited to a specific context.
- **Encapsulation:** Implementation details should be hidden from the user. Having a well defined standardized interface is desirable.

Component based paradigm has much in common with the object oriented paradigm. Betrand Meyer said the following about components [Mey00]:

> "It doesn't take long to realize that they will be very much like classes."

Szyperski raised to following question [SGM02]:

> "If classes are so similar to components, why did object technology not succeed in establishing significant component markets?"

There are two answers to this question:

- The definition of objects does not include notions of independence or late composition. This situation led to the construction of monolithic application
- Object technology tends to ignore the aspects of economics and markets. The idea of having catalogues full of class libraries failed. There are only a few vendors that provide software components but they want to sell something else with that product. An example of that is the MFC[1] library of Microsoft which simplifies the development of Windows™applications.

Although component based programming is an extension of the object oriented programming paradigm, it is not necessary that components have to be written with it, but object technology is probably one of the best ways to realize component technology.

Available component technologies are further described in more detail in chapter 3 .

2.4 Concepts of Component Based Approaches

This section describes some of the approaches to component based development. First we take a look at already realized techniques that can be seen to have component based design in mind.

[1] Microsoft Foundation Classes

2.4.1 Pipes and Filters

Douglas McIlroy was one of the founders of the Unix shells. He invented the pipes and filters concept and included it to Unix. It is one of his important developments.

Unix uses the following philosophy to write programs:

- That do one thing and do it well.
- To work together.
- To handle text streams, because that is a universal interface.

A Unix pipe connects two programs. The output of one program is sent as input to another one. In Unix the symbol for the pipe is "|". A sequence of programs connected together by pipes is known as a pipeline. All involved programs have access to the following three special files:

- **stdin:** the standard input file
- **stdout:** the standard output file
- **stderr:** the standard error file

A program performs some transformations to its input data and can therefore be regarded as a filter which filters the appropriate data.

The code (see Listing 2.3) shows a simple pipes and filters example.

```
1  ls | grep 'hugo'
```

Listing 2.3: Simple Pipes and Filters example

In the example above, the following Unix programs are used:

ls: lists the content of the current working directory

grep: does a search operation and returns all the lines that contain a given string-pattern.

The given pipes and filters examples lists all files in the current directory containing the string *hugo* in its name.

This technique can be seen as the first step towards software components. Every program can be interpreted as a component which encapsulates its implementation. The commands communicate through a well defined mechanism which is to write to standard output and/or read from standard input by using text streams.

2.4.2 Plugins

The plugin concept poses also a method of developing applications based on reusable components. *Computer Desktop Encyclopedia* gives the following description for a plugin[2]:

> "An auxiliary program that works with a major software package to enhance its capability."

A plugin is a program that provides a specific functionality, but is not meant to be a stand-alone application. Some base program is needed that provides a way to register and appropriately call them. Further mechanisms for data exchange must be given by them.

Plugins are different from **extensions**. While plugins add new features to a program, extensions only modify existing functions.

The following examples use a plugin technique:

- *Web browsers:* a plugin can be added to handle a new type of content

[2] see `http://www.answers.com/plugin`

(like the one from Adobe for viewing SVG images[3]).

- *e-mail clients:* encryption and decryption are nearly always done through a PGP plugin.
- *Eclipse:*[4] Eclipse is a platform-independent framework. It's principal role is to provide tools and mechanism for developers so that their plugins can be integrated.
- *Photoshop:* a filter can be added for some special effect.

2.4.3 Component Models

Common Object Request Broker Architecture (CORBA):

It is defined by the **Object Management Group (OMG)** and is currently in the third design generation.

Components can be written in any language and on any platform. CORBA provides a way distributed programs can communicate. Sometimes it is described as an *object bus* or *software bus*, because it is a software-based communication interface through which objects are located and accessed.

CORBA objects are described by an *Interface Definition Language (IDL)*. It defines the methods and the parameters that are sent and received. IDL-compilers generate stubs and skeletons from the description. Compilers for nearly every programming language are given which enables programmers to use their preferred language for the client and server.

The key element of the CORBA architecture is the *Object Request Broker*(ORB) which is responsible for the communication part. On the client side, the ORB provides proxy objects that give the illusion that they communicate with local objects instead of remote ones.

[3]see `http://www.adobe.com/svg/`

[4]see `http://www.eclipse.org/`

Figure 2.2 shows the basic concept. The request of a client is first sent to the local ORB which knows the real location of the desired object. A message is then sent to the remote server ORB which performs the request and returns a message back to the requesting client ORB. Messages to and from the remote ORB are performed over the *General Inter-ORB Protocol* (GIOP). Client and server have always a transparent view of the remote object. The communication part is done by the ORBs.

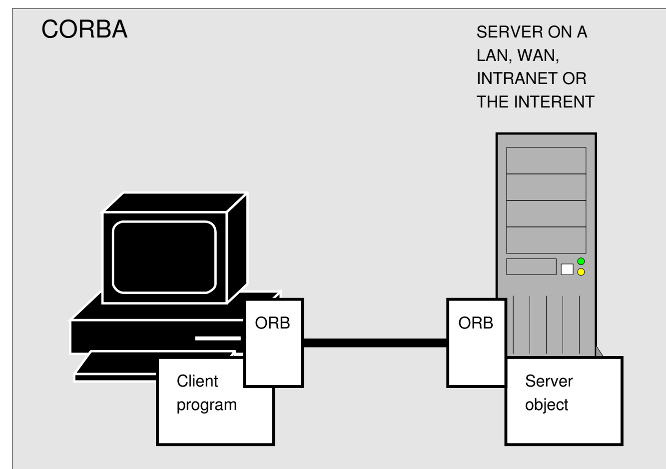

Figure 2.2: CORBA framework (from Computer Desktop Encyclopedia ©1999 The Computer Language Co. Inc.)

There are several implementations of the OMG's CORBA standard:

- *OmniORB* [Gri] for C++ and Python,
- *TAO* [corc] for C++ or
- *JacORB* [corb] for Java.

One of the most famous projects in the Unix world that uses CORBA for communication is *GNOME*[5]. On the one hand GNOME is an intuitive and attractive desktop for end-users, on the other hand it is a development platform.

[5]GNU Network Object Model Environment (see `http://www.gnome.org`)

An ORB called *ORBit*[6] was implemented because at the beginning of the GNOME project all free ORB implementations were either too resource intensive, too slow or did not implement the desired C bindings.

Further *Bonobo*[7] was developed which is a component model that uses CORBA for communication. It defines interfaces that are implemented in the GNOME desktop environment.

Component Object Model (COM):

First there was the idea that a user may be able to embed or link a document into another one. An example is the inclusion of an *Excel* spreadsheet into a *Word* text document.

Microsoft developed the *Object Linking and Embedding* (OLE) technology. OLE was based on *Dynamic Data Exchange* (DDE) which was very complicated to program with. Furthermore OLE's performance left much to be desired.

Due to these lacks, Microsoft invented COM in 1993, and the next version of OLE. COM was designed to be a general interaction mechanism. The drawback of OLE was that communication between different COM objects was only possible as long as they resided on the same machine.

The interfaces are described using the Microsoft COM extensions to the standard DCE[8] Interface Description Language (IDL). The Microsoft IDL (MIDL) compiler can generate header files and other files from the IDL description.

Different component types are identified by class IDs which are Globally Unique Identifiers (GUIDs). Each component exposes functionality through one or more interfaces. Different interfaces are distinguished using Interface IDs (IIDs). Components can be written in several programming languages.

[6]see `http://developer.gnome.org/doc/tutorials/#orbit`
[7]see `http://developer.gnome.org/doc/tutorials/#bonobo`
[8]Distributed Computing Environment see(`http://www.opengroup.org/dce/`)

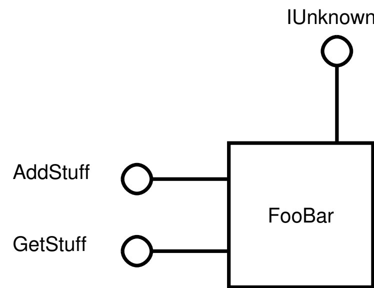

Figure 2.3: COM component (taken from Wikipedia `http://en.wikipedia.org/wiki/Software_component`)

Figure 2.3 shows the representation used in *UML* of a typical COM component. It shows the `FooBar` component having three interfaces:

- IUnknown
- AddStuff
- GetStuff

The communication is based on interfaces. COM defines several basic interfaces but it is also possible for a component-developer to define new ones. The `IUnknown` interface has to be implemented by every component. It consists of the following three methods:

- `AddRef` and `Release` which are intended for reference counting.
- `QueryInterface`: This methods allows a caller to retrieve a reference to a desired interface. This is similar to `dynamic_cast` operation in C++ or `casts` in Java.

When a component is instantiated, the client gets a pointer to the `IUnknown` interface. The client then can invoke the `QueryInterface` method to query the component for the desired interface. Every interface has it's own ID (IID).

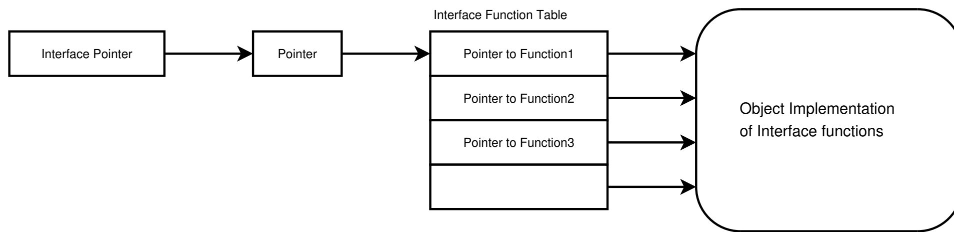

Figure 2.4: COM interface structure (taken from [Bro96])

A client does not get direct access to the component. Figure 2.4 shows the interface structure used for COM. The *Interface Pointer* is the pointer the client gets for a component. This pointer itself points to another pointer that points to a table that holds the addresses of each member function in the interface. This structure resembles the C++ virtual method table for classes.

There is no problem to use the above pointer in a situation when the client and the component-server are in the same process because the pointer is valid in the client's address space. On the other hand, the object's exact pointer value cannot be passed when local and remote objects are involved. COM uses a *marshalling* mechanism that overcomes that problem. This is described in more detail in [Bro96].

There are further developments of COM:

- *Distributed COM* (DCOM) appeared with the upcoming of Windows NT. DCOM made it possible that COM objects could communicate in a distributed environment.
- *ActiveX* is another extension based on COM that allow a web browser to view any kind of document.
- *COM+* was released with Windows 2000.

Containment and *Aggregation* are the ways components can be composed. They differ in the way the interface of the inner component is brought

to the outside [GT00].

JavaBeans™:[9]

In the Java world, a *bean* is a component consisting of a set of classes and resources.

After Java Applets, JavaBeans™is the second component model developed by Sun. The focus lies in the support of connection oriented programming.

The main aspects of the bean model are:

- **Events:** Events are objects created from an event source. It propagates to all registered event listeners. The communications have either multicast or unicast semantic. For the first case, multiple listeners can be notified from the source. The number of listeners can change while an event is propagating. When unicast semantic is used, an event source is connected to only one event listener.

  A listener needs to implement an interface that extends the `java.util.EventListener`[10] interface that has a receiving method for each event a listener listens to. Only one interface can be implemented. When a listener is registered with several event sources that all can generate the same event, the listener has to determine where an event came from and handle it appropriately. To avoid such switching, an adapter can be used that implements the listener interface and holds references for listeners. A separate adapter can be registered with each source and calls a different method of the listener.

  The event concept can be seen as a generalization of the *Observer* pattern as described in [Lau05].

- **Properties:** A property is a named attribute that can change the behaviour of the bean that defines it. Getter and setter methods are

[9] see `http://java.sun.com/products/javabeans/`

[10] see `http://java.sun.com/j2se/1.4.2/docs/api/java/util/EventListener.html`

used to access it. When a property is constrained, the setter method always throws an exception to indicate the registered vetoable-change listeners that a change of a constrained property was attempted. These listeners may throw another exception if the contract is broken.

A property can further be bound. Such properties generate an event if a change occurred.

- **Introspection:** This is the ability to inspect a bean to find its properties, events and methods.
- **Customization:** With customization the possibility of setting properties of a bean is intended.
- **Persistence:** This is the feature of a bean to save a state so that it can be reloaded later on.

Enterprise JavaBeans™(EJB):[11]

EJB is the third component model invented by Sun. It focuses on container-integrated services supporting EJB beans as their components and is part of Sun's J2EE[12].

The difference from JavaBeans™technology is that it does not provide a provision for connection oriented programming. It is weak at connection-oriented composition and strong at the level of contextual composition.

Contextual composition is the automatic composition of component instances with appropriate services and resources. As Enterprise JavaBeans™ are embedded in a container contextual composition means the placement of objects in appropriate containers based on attributes. Clemens Szyperski wrote: [Szy02a]

" Contextual composition is like using a cellular phone that

[11] Enterprise JavaBeans (see `http://java.sun.com/products/ejb/`)

[12] Java 2 Platform, Enterprise Edition see(`http://java.sun.com/j2ee/index.jsp`)

automatically interacts with a base station."

More about composition and contextual composition is discussed in section 3.5 .

As stated above, an EJB is embedded in a container. This container provides system level services. The containers help the application developer to concentrate on the problem. The following system services are provided:

- Persistence
- Security
- Transaction
- Component lifecycle management
- Threading

Containers are provided by EJB-Servers.

The communication between the client and the component is done through two interfaces (see: figure 2.5):

- *Home interface:* This interface defines methods for creating, finding and removing beans like `create()` and `remove()`. They are implemented by the container.
- *Remote interface:* The remote interface is specific to a EJB. It defines the business methods for the bean. In figure 2.5 the `BankAccountEJB` has the methods `deposit()` and `credit()`. A client can only invoke these two methods on the bean.

EJB-Server provides a runtime environment for one or more EJB-Container. An EJB component is embedded in an EJB-Container.

Remote methods can be grouped in the following ways:

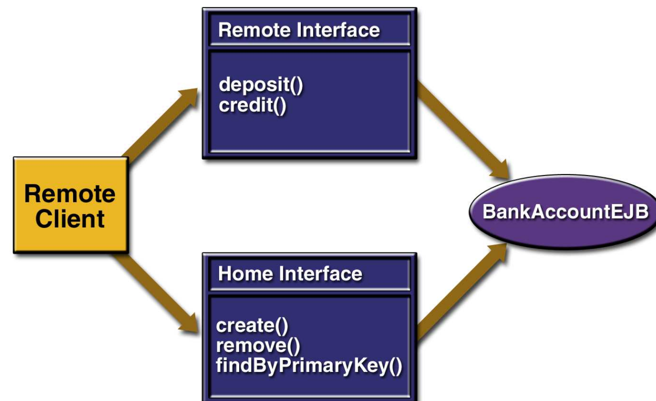

Figure 2.5: Home and Remote Interface taken from J2EE tutorial (see `http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBConcepts6.html`)

- Methods not tied to a specific instance (e.g. create an EJB instance) which are declared by the home interface.
- Methods tied to a specific instance which are defined by the remote interface.

An EJB container can hold four major categories of beans:

- *Session Beans:* A session bean performs something for the client. There are two types:
  1. Stateful: The state of a bean retains as long as there is a client-bean session.
  2. Stateless: contents of variables are not guaranteed to be preserved across method calls.
- *Entity Beans:* These are persistent components like a database. Their state is stored on non-volatile storage.
- *Message Driven Beans:* Provides a way to expose components to the *Java Metadata Interface (JMI)*[13].

[13] see `http://java.sun.com/products/jmi/`

2.4.4 Web services

A web service is a collection of open protocols for exchanging data. Several consortiums like OASIS[14] and W3C[15] are responsible for architecture and standardization. Programs residing on any platform and written in any programming language should have the ability to transfer messages over the internet. The following techniques are used:

- **XML:** XML is a method to format data. The advantage of this technique is that it is a text format and therefore humanly readable. Changes can easily be done using a simple text editor.

  If an application wants to use a web service it must have at least a XML parser and a XML generator so that incoming messages are understood and outgoing messages can be constructed.

- **SOAP:** The *Simple Object Access Protocol* is a simple and lightweight mechanism to exchange structured and typed data between peers. These data are written in XML syntax and piggybacked on a standard protocol like HTTP or SMTP.

  With the use of HTTP as transport protocol, messages can be transferred over firewalls without changes to their filtering rules. On the one hand this makes it possible for institutions behind a firewall to offer or use a web service. But on the other hand this posses a security problem too when a worm or Trojan horse uses this concept for interactions.

  SOAP in the context of the developed component framework is further discussed in chapter 5 .

- **WSDL:** The *Web Services Description Language* describes network services as collections of communication endpoints which are capable of exchanging messages. Further information about the location, supported operations (public interfaces) and the used messages format are

[14] see `http://www.oasis-open.org/`
[15] see `http://www.w3.org`

given. This description is done using XML.

WSDL uses SOAP as transport protocol.

- **UDDI:** The *Universal Description, Discovery and Integration* is an industry initiative for a *catalogue* of web services accessible through the internet.

  For service provider, the UDDI project made it possible to advertise their services. This offers small businesses the ability for reaching more customers. For the customers on the other hand, the UDDI offers a great searching environment to find the appropriate service.

  An example of a UDDI registry is the XMethods[16] web page. New services can be registers and existing ones can be tested online.

Figure 2.6 shows how the above mentioned parts of a web service interact. A service provider registers his service, by transferring the WSDL file of the service to a Service Broker who offers an UDDI environment.

A client searches for a desired service with the help of UDDI. If the needed service is found, UDDI transfers its WSDL file. With the help of this file, the client is able to use the offered service by directly sending SOAP messages to the provider.

SOAP messages can be either generated manually or by using a toolkit which automatically does this job. Toolkits for nearly every programming language are available. Some of them are:

- *Apache Axis*[17] for Java and C++
- *gSOAP*[18] for C++ which is also explained in more detail in [Laz05].
- *SOAP::lite*[19] for Perl

[16] see `http://www.xmethods.net/`
[17] see `http://ws.apache.org/axis/index.html`
[18] see `http://www.cs.fsu.edu/~engelen/soap.html`
[19] see `http://www.soaplite.com/`

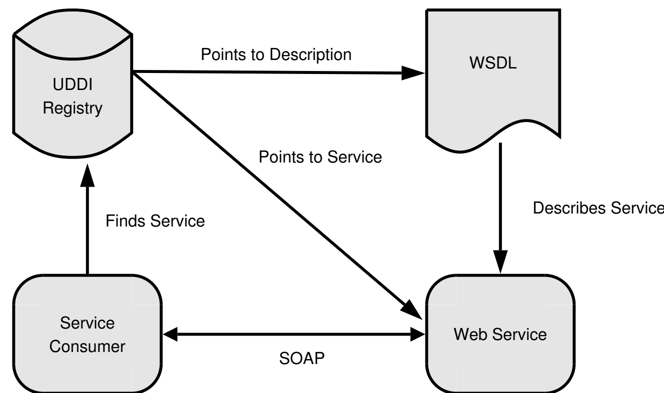

Figure 2.6: Web service: Interaction between UDDI, WSDL and SOAP (taken from [Cor03])

CORBA and web services do conceptually the same things as both technologies attempt to solve nearly the same problems.

Jean-Guy Schneider and Jun Han stated [SH04]:

> " Hence, it seems that the community tries to reinvent the wheel once again, and before doing so in the future, we should probably reflect more on the problems we have solved and how well these solutions worked."

2.5 Chapter Conclusions

Component based software development is an approach to reuse code that appears often in the development cycle. This can be compared with reusable parts of e.g. electronic circuits. In that field of development it is a very common procedure to choose a desired component (for example a resistor) from a catalogue.

Knowing the internals of a component is not necessary. A well specified component regarding the reaction on some input data is information enough

for the consumer to use it as part of his application.

New technologies like in this case the component based development of software take some time to be successful and accepted by the community. In this chapter this evolution was compared with the first steps of mass production of weapons. From the first idea to the realization, more than twenty years have past.

The software development techniques evolved over the years. First the developers programmed a machine by typing assembler code. Nobody spoke about reusable code at that time.

As machines have grown, software was able to execute things that machines were not seriously built for. Applications became bigger and bigger. An enhancement of software paradigm was inevitable. Programmers tend more and more to write structured code that is easier easily to maintain and read.

The next step in writing software was the object-oriented programming, generic programming and metaprogramming, all having a rather different idea of writing reusable software in mind.

Some techniques toward reusable software have already been developed. One of the first concepts of connecting components together through a well defined protocol was “pipes and filters” where applications communicate through well known text files. Every program that can handle those files can be put in a so called pipeline.

Component models like CORBA, COM or Enterprise JavaBeans™ are indented to be a mechanism which handles the communication between user defined components.

Web services is one of the keywords that are very popular nowadays. Components can be connected over the internet through a standardised protocol like http[20]. Data is transferred by means of xml files and the use

[20]hyptertext transfer protocol

of SOAP.

The method of programming application has changed over time. Component based development is surely the right way to help with the construction of reusable software out of reusable parts.

Chapter 3

Reusable Software Components

"If you build it, they will come."
W.P. Kinsella, Field of Dreams (1989)

Component oriented software or component based software design are hot technologies often discussed in software engineering. Nevertheless there is no clear definitions of the terms *Software Component* and *Component Software* neither what they really are nor how they are correctly composed.

This chapter starts with some definitions of a Software component and tries to summarize them in to a general definition. Therefore an overview of software qualities and approaches to reusability are given.

Therefore an overview of software qualities and approaches to reusability are given.

3.1 Component Definition

The definition of a component is one of the most discussed topics in current software technology. The most popular ones are given in this section.

- Clemens Szyperski states in his book *Component Software: Beyond Object-Oriented Programming* the following definition [SGM02]:

  > "A software component is a unit of composition with contextually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition."

- Bertand Meyer stated [Mey99]:

  > "A software component is a program element with the following properties:
  > – The element may be used by other program elements (*clients*)
  > – The clients and their authors do not need to be known to the element's authors."

- Philippe Krutchen from Rational Software said:

  > "A component is a nontrivial, nearly independent, and replaceable part of a system that fulfils a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces."

Above citations have some facts in common. The definitions mainly depend on the context that a component is used for. Some criteria can be outlined which all components should have in common [Mey00]:

- *Usable by other software elements.*

  Some may say that any program in the traditional sense can be regarded as being a component. Programs are indeed built to be *deployable* and used by humans. The main focus is human use, instead of serving other programs. To be usable for other software elements, programs must be *componentized*. Usually such constructed component has only one input and one output, but it can be used in other application.

- *Of interest to a broad range of "clients".*

  Much more effort in the construction of a component is needed than programming an arbitrary piece of code implementing the same functionality. A well defined interface has to be given and the code has to be tested as generally as possible because the place of action for the component can not be foreseen. This additional effort is only done if more than one client can be addressed in using the component. The more often a component is reused the more profitable it is.

- *Specification of dependencies.*

  In order to function as specified, the components dependencies have to be fulfilled. These can be a software platform, a special hardware or something else. Additional dependencies to other components must be specified if certain tasks can not be performed by the component alone.

- *Precise specification of the functionality.*

  A precise specification of what a component can do and its boundary conditions are very important.

  The explosion of the Ariane 5 launcher is a good example of a system failure through the usage of unspecified components. The error was caused by the *Intertial Reference System* (IRS) which was not needed after liftoff. Regardless, this software was taken and reused from the former Ariane 4 project. The IRS component assumed that a horizontal bias fits in a 16 bit integer and thus an overflow could never occur.

This analysis was true for Ariane 4 but not for Ariane 5 which had other trajectory parameters.

The IRS is a trajectory initialisation system that is used before lift off. After launch it could have been deactivated without consequences. Due to the fact that a resetting of this system could take several hours, it was decided it would be better to let the computation proceed in case the lift off would have been delayed. Right after lift off this completely unnecessary system had thrown an exception which was not caught and Ariane 5 had to be destroyed.

Bertand Meyer and Jean-Marc Jézéquel argued that a different choice of programming language would probably have avoided the problem [JM97] especially by using the principles of Design by Contract™ (DbC).

Ken Garlington on the other hand said that using DbC would have led to the explosion too [Gar98]. The reason for the error was the missing documentation that specified the bias of Ariane 5. Without this specification a wrong contract would have been written to IRS and the launch would have failed too.

- *Able to use on the basis of specification.*

  Components have to be usable on the sole basis of the given specification. Access to functions not given by the interface must be avoided. This can also be seen as the information hiding principle of a component.

- *Composable with other components.*

  A principle of component based design is: *Just do one thing but do it right.* Complex components are then compositions of those basic components. In principle a good component is usually a part of a more general component.

- *Can be integrated into a system.*

  Integration of a component into a system should be as fast and smooth as possible.

3.2 Software Quality

> "When quality is pursued, productivity follows"
> *K. Fujino (quoted in [GJM91])*

The construction of high quality components implies producing high quality software. This can be achieved by having software quality in mind when a piece of software is being developed. These qualities can be divided into *internal* and *external* quality factors.

Internal and external factors are closely coupled. Techniques to fulfil internal factors are needed to reach the external software qualities.

> "In general, users of the software only care about the external qualities, but it is the internal qualities - which deal largely with the structure of the software - that help developers achieve external qualities"
> *Ghezzi et al. [GJM91]*

3.2.1 Definitions of Software Qualities

There are some definitions given by international organizations about quality:

Definition:	Quality is the totality of features and characteristics of a product or a service that bear on its ability to satisfy given needs. *[ANSI-Standard (ANSI/ASQC A3-1978)]*

This definition was further extended by the *IEEE-Standard (IEEE Std 729-1983)*:

Definition:	• The totality of features and characteristics of a software product that bear on its ability to satisfy given needs; for example, confirm to specifications. • The degree to which software possesses a desired combination of attributes. • The degree to which a customer or user perceives that software meets his or her composite expectations. • The composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer.

3.2.2 External Quality Factors

External factors are factors that can be judged by the user. For example computational performance can be easily measured. Waiting a long time for results will surely annoy the person who must work with that piece of software. External quality is the quality of the finished product. It's the quality of the product as it appears to the external world.

The following definitions are taken from Bertrand Meyer's book *Object-Oriented Software Construction* [Mey97].

Correctness

Definition:	Correctness is the ability of software products to perform their exact tasks, as defined by their specification.

Correctness is the prime quality. If a program does not do what it is supposed to do, then there will be no customer who will pay for it.

To prove correctness some sort of specification is needed that comes from the user. A *User Requirement Document* (URD) can be such an example.

Use Cases are given that describe parts of “real” scenarios like opening or closing files. They explain the activities that have to be done by users to get the expected result. In a later development step those cases can be used as a basis for test cases.

A method for ensuring correctness is a layered approach. Each layer is correct on the assumption that the underlying layer is correct. This separation lets the developer concentrate on the problem of his application.

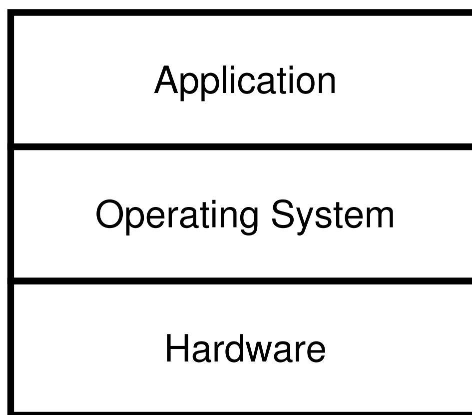

Figure 3.1: Simplified layers for easier correctness proving

Figure 3.1 shows a simplified layered system. When proving that an application is correct, it is assumed that the operating system itself functions correctly which in turn assumes that the hardware works due to it’s specification.

Robustness

Definition: Robustness is the ability of software systems to react appropriately to abnormal conditions.

Robustness complements correctness. While correctness covers the system behaviour given by the specification, robustness describes what happens in unspecified and unexpected cases. These are also known as *abnormal cases*.

When such situations arise, it is preferable that the system reacts properly and not in a system crash. The first flight of Ariane 5 is one of the most famous example, where an uncaught exception caused its explosion [Lio96].

Extendibility

Definition: Extendibility is the ease of adapting software products to changes of specification.

Programs tend to become monolithic giants. Software has the characteristic that an application can never be stated to be finished. Sure, a program can conform to the specification given at a distinct time (milestone) but changes to it are not inevitable. Some of these changes include the modification of requirements (the user wants additional features) or of algorithms that make some critical part more secure.

Big programs, which evolved over time, tend to be hard to adapt to the above mentioned changes. Sometimes "hacked" in fixes to the source code leads to unpredictable behaviour or even bugs. Bug fixes can also pose a source for other bugs. Little changes, if addable to the system at all, usually requires a lot of time and effort from experienced developers.

Two principles are essential for improving extendibility:

- *Simplicity:* It will be easier to change a program that consists of simple constructs than to change a complex, unmanageable one.
- *Decentralization:* Changes to nearly independent, autonomous modules will only affect a small number of other modules rather than triggering off a chain reaction of changes over the whole system.

Ease of use

Definition:	Ease of use is the ease with which people of various background and qualifications can learn to use software products and apply them to solve problems. It also covers the ease of installation, operation and monitoring.

This term focuses on the ability that somebody, experienced enough to understand the used programming language, understands the written code. On of the key concepts to achieve this goal is to have well structured code which can be easily read.

Programs should be designed for users with the assumption that their experience is limited. J. Hansen stated [Han84]:

> "Do not pretend you know the user; you don't."

Efficiency

Definition:	Efficiency is the ability of a software to place as few demands as possible on hardware resources, such as processor time, space occupied in internal and external memories, bandwidth used in communication devices.

Different attitudes towards efficiency are present. Some developers try to optimize nearly every line of code. The resulting code is often so extremely specialized that it is not possible to reuse parts of it or to extend it. On the one hand, efficiency is fulfilled with glamor but on the other hand the other software qualities were neglected. Further great performance does not matter much if the software is not correct. It's better to "make it right before you make it fast".

It does not make sense to make everything in a program as fast as possible. A better choice is to try to improve those parts that are most frequently used.

Compatibility

> **Definition:** Compatibility is the ease of combining software elements with others.

Often programs can not interact because the do not fit together. This can lead to very serious problems. Imagine programs designed for a specific operating system which then do not operate as desired on it because some false assumption of interaction were made. Needless to say, such software often does not care much about extendibility or other quality factors.

Having a well defined interface for inter-program communication is a must when an application is developed in teams. Enough projects have to be canceled at the time when the separately constructed parts should be put together.

Using standardized conventions such as text files that can be read by anyone, is preferable.

3.2.3 Internal Quality Factors

Internal factors are hidden from the user. They describe the way the software is constructed.

Simplicity

Simplicity means that the code should be written in a way that others can understand it easily. This principle is also known as *Keep It Small and Simple* (KISS). Some techniques exists to write a more readable code:

- A module/function does only one atomic task.
- Always document the things a module/function does.
- Avoid too much dependencies on other parts of the software.

- Use flat interfaces.

Modularity

> "... was to divide each of the difficulties which I would examine into as many parcels as it would be possible and required to solve it better."
> *René Descartes, Discourse on the Method(1637)*

To solve a complex problem, usually it is better to recursively break it into smaller, autonomous subproblems. These small pieces can then be implemented independently and be assembled to solve the complex problem. This small pieces are called *modules.*

Modules should be self-contained and organized in stable architectures in order to be of any advantage.

There are some rules that must be observed to ensure modularity:

- *Direct Mapping:* There should always be a correlation between the structure software-solution of a problem domain and the problem domain itself.
- *Few Interfaces:* There should be as few connections between the different modules as possible. Changes in one module will therefore only affect a few other modules.

  Figure 3.2 shows a situation in which every module is connected to every other existing one. A better situation is shown by figure 3.3 where a module only talks to its neighbors.
- *Small Interfaces:* Modules should exchange as little information as possible
- *Information Hiding:* Details about how the functionality is implemented should not be obvious to the world. Only a small subset of properties are available.

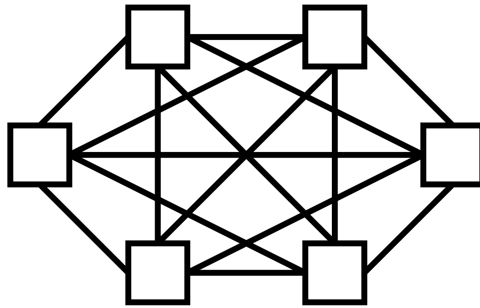

Figure 3.2: Few interfaces: bad example

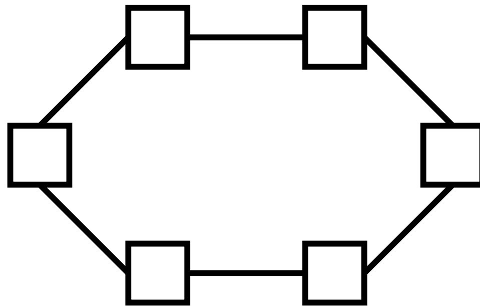

Figure 3.3: Few interfaces: good example

Reusability

Definition: Reusability is the ability of software elements to serve in the construction of many different applications.

Software systems often follow similar patterns. Instead of using existing code that provides common solutions, developers tend to "reinvent the wheel".

The advantage of reusability is that less software must be written and well tested code can be used. Besides, more effort may be devoted to improve other factors like correctness or robustness.

How reusability can be achieved is further discussed in section 3.3.

3.3 Approaches to Reusability

Reusability can be seen as one of the most important quality factors described in section 3.2. By enhancing it there is the potential of enhancing almost all of the other qualities.

Not so long ago, it was usual to build software from scratch. Software was produced in a monolithic fashion. There was no intention to produce pieces that could be reused in other projects. Another problem was the attitude of programmers towards reusable code. Brad J. Cox quoted in his article *Planing the Software Industrial Revolution* [Cox90] one of Apple's most creative programmers:

> "Reusing other people's code would prove that I don't care about my work. I would no more reuse code than Ernest Hemingway would have reused other authors' paragraphs."

Every programmer surely observed that software has a repetitive nature. Well defined algorithms, like searching in a binary tree, have been implemented again and again. Often developers run into the same mistakes which others have made before.

The following benefits are expected by using reusability:

- **Timeliness:** Using existing components means that the software can be built faster and time to market is sooner.
- **Decreased maintenance effort:** The maintenance of components can be given to the one who developed it and knows the internal functionality. The *competent developer's paradox* should be avoided which states that the more someone works, the more work he creates for himself. A trivial solution to that paradox is to become an incompetent developer which is not the preferred situation.

- **Reliability:** Every component guarantees to hold its specification. The author will have applied all the needed care including testing and validating. Further components will be already used in other applications which has the advantage that remaining bugs would be found and fixed.

  The developers can concentrate on enforcing reliability of the new things they have to do and which are not covered by any component.

  In short, reliability is the absence of bugs.

- **Efficiency:** The developer can use components that implement the best possible algorithms or have the fastest data structures.

- **Investment:** Making a piece of code reusable means to preserve the invested know-how that was put into it. Besides, this know-how can be turned into a permanent asset.

Although reusability is the key concept in producing reliable software, there are some companies that provide solutions that are purposely not reusable. They fear not getting the next job from the customer, because if the result of the current job is too widely applicable the customer may not need a next job.

3.4 Testing

> "The cost of program testing for the purpose of safeguarding that the specific requirements are met."
> *McCall*

A test is done in order to find failures or misbehavior in a program and not to show that everything functions correctly. Several levels of component testing are defined:

- **Item testing:** Every component is tested individually to ensure that it functions properly. This is also known as unit testing or white

box testing, because the test cases are constructed through the use of the source code. On the one hand developers do this type of testing because on the one hand they are familiar with the source code, on the other hand the source code is usually only available to them (e.g. component is given in binary form to the customer only).

- **Integration testing:** The individually tested components are connected and tested to show how they work in a distributed environment.
- **System testing:** The application as it should be delivered to the client is tested. The client is not involved. This level of testing is also known as black box testing. The source code is not viewed any longer because there is far too much code to rely on. Besides the testers usually do not have access to the source code.
- **Acceptance testing:** The client runs the new system to ensure that it complies with the original specification.

Information about the software's intention or the expected usage usually affects the selection of test cases. The situation is different when the component developer has no intuition where his constructed component will be used. It is possible that an application executes portions that others never use or which was an extremely unlikely scenario in the original system's behaviour.

Elaine J. Weyuker stated that software component developed for one project should not be used for another project without significant additional testing [Wey98]. She relates her argumentation to the Ariane 5 disaster where a component was taken from the former Ariane 4 project without testing the change in the specifications.

A lot of projects exist that specialise in testing of software components. One of them is the `CrashIt` [Val03] framework which can be used in component based java applications

3.5 Composition

Clemens Szyperski stated his book *Component Software: Beyond Object-Oriented Programming* [SGM02]:

> "One thing can stated with certainty: components are for composition."

Components are built to be composable with other components. The aim is to assemble a software with small well specified and tested components. Composition enables prefabricated "things" to be reused by rearranging them in ever new composites. Three composition approaches dominate the scene [Szy02b]:

- **Object oriented composition:** The object oriented composition does not really work. There is no problem using this composition if the object does not rely on any other object to perform its work. Further there is no problem if the object depends on other objects and this behaviour is well specified.

  The attention of the specification to real objects is paid to the abstract interface. This is usually done using pre- and postconditions. Incoming calls are specified but there is no way to find out what the outgoing calls do if there are any.

  As said above, object oriented composition is no problem as long as everything is specified which includes the incoming calls as well as the dependencies on other used objects.

  UML[1] defines a way in which the dependencies can be given. The object modeled with UML have well defined boundaries that cover both incoming and outgoing dependencies. These dependencies are given through connections between the objects.

[1] Unified Modeling Language

Figure 3.4 shows an simple example described in UML. The `ElevatorController` communicates with one or more `Buttons` and an `Elevator`. The controller can invoke:

- **Button:** `illuminate` or `status`
- **Elevator:** `move`, `stop` or `status`

The dependencies between the three objects are clear and it is further possible to replace either of them as long as the interfaces are the same.

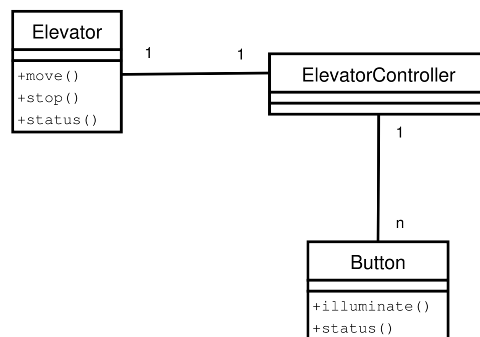

Figure 3.4: UML example

Declaring all dependencies to other components is called *external dependencies*.

To gain more flexibility the explicit dependencies should be replaced with parametric dependencies. This means that instead of making a class depend to another class, make it depend on an interface and equip it with a mechanism to connect to any class that implements that interface.

- **Connection oriented composition:** A component can only use what is has been connected to. The interaction is based on connections. Composition is done through a "wiring" principle and can be compared to coupling of electrical devices.
- **Container based composition:** This form of composition is also known as contextual composition. Components reside inside a con-

tainer and benefits from container supplied services and from container maintained abstraction.

Direct communication to the components are usually not possible. All calls from the world are intercepted by the container which can be seen as a hull over the internal components.

Typical examples are EJB containers.

3.6 Design by Contract™(DbC)

Design by Contract™(DbC) is a technique to make a software more reliable. Bertand Meyer is one of the pioneers who developed the DbC method. He was also the founder of the Eiffel programming language, which is one of the most famous, basically having DbC basically included.

First of all, a contract as known to be in use by humans is given. Then DbC in software development is described which finally leads to the requirements needed by DbC in the context of component design and development.

3.6.1 Human contracts

When two parties make a deal where each expects some benefits, a contract is always made to ensure this situation on a sheet of paper. The contract states that a one of them (*supplier*) has to perform some task to the other (*client*). Each of them has to accept some obligations. The following example 3.1 shows the situation of benefit and obligation from a client to the supplier (in this case between an airline and a customer) and vice versa.

The example shows that a contract protects the client and the supplier. The client can be aware that if he comes early enough to the airport and has only acceptable baggage and a paid ticket, the supplier (in this case the airline) will bring him to Chicago. The customer does not have to know how he will be transferred (although by plane will be surely preferred).

	Obligations	**Benefits**
Client	*(Must ensure precondition)* Be at the Santa Barbara airport at least 5 minutes before scheduled departure time. Bring only acceptable baggage. Pay ticket price.	*(May benefit from postcondition)* Reach Chicago
Supplier	*(Must ensure postcondition)* Bring customer to Chicago.	*(May assume precondition)* No need to carry passenger who is late, has unacceptable baggage, or has not paid ticket price.

Table 3.1: example of a human contract (taken from [Mey96])

The airline in turn can ignore the customer if its preconditions are not fulfilled. But have to be concerned about the transfer if everything is okay.

Resulting from the above example the following rules can be given:

Assertion violation rule

A precondition violation is the manifestation of a bug in the client.

A postcondition violation is the manifestation of a bug in the supplier.

3.6.2 Contracts in software

Bertand Meyer said that under DbC theory a software system is viewed as a set of communicating components whose interaction is based on precisely defined specifications [Mey96].

The same situation as given in the section above, section 3.6.1, is given to

the world of software. When for example a method (*supplier*) is called from a client, the client and the method itself have to assure a certain contract which is given by the specification. A simple example is the insertion of a given element to a container. The following code fragment (see Listing 3.1) shows how such a method could be realized using the Eiffel programming language[2].

```
1 put(element: Element) is
2 require
3   not element.empty
4 do
5   -- do insertion here
6 ensure
7   has (element)
8 end
```

Listing 3.1: Example with pre- and postconditions

The precondition is given through the **require** clause and the postcondition through the **ensure** clause. The method *has* is given by the container which tells whether it contains the element. It is assumed, that the container can have an arbitrary number of elements, because otherwise this circumstance must be considered in the pre and postcondition too.

When a client calls this method and assures that the given element is not empty, he can be sure that the element is inserted into the container.

DbC further helps the developer in debugging the code. If the precondition is not satisfied, the client has produced the error. Otherwise, if the postcondition indicates an error, the implementation of the method does not work properly.

As described above, pre- and postconditions apply to individual methods. Another kind of assertion that falls in this category is an **invariant** also known as **class invariant** in the context of object orientation. An in-

[2]the reason for using Eiffel is the fact that this language support DbC out of the box. Other languages need additional tools like *iContract* [Kra98] for Java or *Nana* [Fou] for C++

variant describes a property that must hold all the time. Lets assume that the container in the previous example has a distinct capacity. Adding an element requires that the number of the elements in the container is less or equal to the capacity. In Eiffel syntax this if described in the following manner:

```
invariant
  count <= capacity
```

Listing 3.2: Syntax of an invariant in Eiffel

Count is a property indicating the actual number of elements in the container.

The facts given up to now can be summarized by the following three questions that must always be considered when a method or component is written:

- What does it maintain? (*invariant*)
- What does it expect? (*precondition*)
- What does it guarantee? (*postcondition*)

A real advantage of the DbC is the documentation of the interfaces. In Eiffel for example it is possible to get a **short form** of a class which gives all information about invariant, pre- and postconditions of its members. With such a documentation the client is able to know what a special method semantically does and what it expects.

We can conclude, that DbC has the following benefits:

- Systematic approach to building bug-free systems
- Effective framework for debugging, testing and quality assurance.
- Method for documentation.
- Technique for dealing with abnormal cases.

Handling violations

When a contract is broken, the system has to react. There are different actions that can be undertaken:

- *Ignore:* No attention is paid to the contract handling. This is the same as disabling the contract checking mechanism.
- *Reject:* The operation where the violation occurred is stopped. If an assertion fails in C++ the program is terminated and a message about which assertion caused the violation is given to the client. DbC in C++ is further described in [Lau05].

  Another way to inform the client of such an issue is to raise an exception. This technique has the advantage that the client has the possibility to react to a failure by either quit or process.
- *Wait:* In this case the client waits until the contract becomes valid through and finished. This option has no application in sequential program but it makes sense for concurrent programs.
- *Negotiate:* The failing operation will be retried. For example in case of a network failure.

3.6.3 Contracts for Components

To trust a component, its user must know the specifics at to what is really done. One way toward trusting components is the use of contracts. The same three kind of specification as given above are needed in the design and implementation of software components. These are **invariants**, **preconditions** and **postconditions**.

Some challenges arise when contracts from the world of classical programming are brought to the world of component based development. These problems are:

- In the classical world the monitoring of the contracts is turned off for release versions. This is done due to performance reasons because checking the contracts needs some time and decrease the computational speed. But within the world of components the verification of the components must always be kept on.
- Semantics of contract changes in a concurrent environment where preconditions may be interpreted as wait conditions rather than correctness conditions.

On the last point, Antoine Beugnard et al. described in their article *Making Components Contract Aware* four level of contracts [BJPW99]:

- **Basic contracts:** The following things are specified:
  - The operations a component can perform.
  - The input and output parameters each component requires.
  - The possible exceptions that might be raised during operation.

  All these requirements are fulfilled by an *Interface definition language* (IDL) as defined for CORBA and COM, or by typed object oriented language.
- **Behavioral contracts:** Also known as semantic contracts which are described through per- and postconditions and invariants.
- **Synchronization contracts:** Dependencies between services which are provided by a component are described. This kind of contract is very important in an concurrent environment where more clients interact with one server. The contract guarantees to each client that, whatever the other clients request, the requested service will be executed correctly.
- **Quality of service contracts:** Common examples of quality of service parameters that may be exposed by a server are:
  - maximum response delay

  - average response
  - quality of result, expressed as precision
  - result throughput for multiobject answers such as streams

3.6.4 Limitations of Interface Definition Languages

The *Interface Definition Languages* (IDLs) given by CORBA and COM are quite similar and are nearly the same as the C++ header file. The intention of IDL is to give the user of a component information about the signature of operations without the implementation. The following example shows a CORBA IDL specification of a function from a component that can add an element to it's internal storage:

```
1  boolean addElement(in Element an_element) raise(OutOfMemory);
```

Listing 3.3: CORBA IDL example

It says that a component has an operation named *addElement* which takes as input parameter an element of type *Element* and returns a boolean value. Further an exception called *OutOfMemory* can be thrown.

The example above describes the syntax of the operation *addElement*. But does it really add an element? The user can only guess as semantic information is not explicitly stated. If the source code is given, someone can read it to find it out. But having only binary components, a walk through the source code is not possible. Another way of exploring the semantics is by regarding the exceptions. Bertand Meyer said of this practice the following:

> "... and of course it is rather paradoxical that we should specify to our clients what abnormal conditions can arise, but not take the trouble to tell them what the normal expected behavior is"

Using IDLs as used by CORBA and COM make it possible to connect components if they satisfy the specification. For some reason this may be enough but there are enough components that fit together but they "should not" be connected. A good example (although taken from real life) is the following: The electric outlet in Ecuador and in Estonia are the same but have a marginal difference, that the voltage in Ecuador is 110V and in Estonia 220V [3]. Electrical devices from one country fit in the plugging device of the other country and vice versa. Devices from Ecuador may surely be damaged without any adapter.

This example shows, that it is necessary to additionally have specification and semantic information about a component to use it correctly.

Another drawback of using a separate mechanism representing the interface is, that both software and it's interface description have to be synchronized everytime either of them is changed. Developers have to write the specification properties twice. To avoid this but be able to further use COM or CORBA, automatic tools have to be developed that wrap the code into a COM or CORBA. *EiffelCOM*[4] is one such tool which interactively wraps Eiffel applications into COM components.

3.6.5 Design by Contract™ versus Defensive Programming

Defensive programming can be viewed as reducing or eliminating *Murphy's Law*:

> "Any of certain humorous axioms stating that anything that can possibly go wrong, will go wrong."

In software this is expressed by adding additional code which checks for any possible failure. The example given for pre- and postconditions(see

[3] see `http://www.call220.com/Resources/electric_current.htm`

[4] see `http://eiffelzone.com/esd/ecom/`

Listing 3.1 on page 79) implemented in defense programming style can look like the following code:

```
1  put(element: Element) is
2  do
3    if element.empty
4      then
5        -- do error handling
6      else
7        -- do insertion here
8      end
9  end
```

Listing 3.4: Defense programming example

The supplier additionally checks the pre conditions. The advantage of DbC is that the check mechanism can be either turned on or off. Disabling the contracts are done for release versions to gain performance.

This is not true for defense programming. The error handling code will always be executed. Another drawback is that the software will become more complex and the added code may be error prone too. Writing complex software is abused in terms of software qualities.

3.7 Conclusions

Component based software construction has the reusability of parts of software in mind. They should be built in a manner that they can be interchangeable. The hardware industry poses how components can be reused.

Some quality factors are given in this chapter that help in the construction of reusable software. The advantages of reusable software are timeliness, decreased maintenance effort, more reliability and efficiency.

Components are built to compose them. Several approaches toward composition were out lined. Connection oriented composition is one of the most used form. By using this concept components are “wired” together.

Container based composition is another way to bring components in relation. They are put in a container that somehow spawns a hull around them. Communication to and from components must pass this hull. The container decides if messages are allowed to pass or not. The advantage of this form of composition is that the components can rely on services provided by the container like security.

Design by Contract™ (DbC) must also be taken in to consideration for constructing trusted components. This chapter discusses how a component can be bound to contracts and the limitations of current techniques for component based development toward DbC.

Chapter 4

Coco Component Framework

This chapter covers the design and implementation of the Coco component framework. The first section describes the *Coco-Component* that represents the basic building block in a component-based application. Composition of these basic elements is covered in the following section (Coco-Composite-Component). The Coco-Component-Model is explained next. It is a special composite providing the ability to build up and encapsulate all parts of a model. Component persistence is supported by dedicated Coco-Marshaller classes, that are described in the next section.

All parts of the framework are named with the *Coco* prefix which is an abbreviation for *ConfigurableComponent*. They also share an equally named namespace.

The used language is C++. As with every programming language, there exist a lot of problems that had to be solved in order to realise the design. The most important solutions are also discussed in this chapter.

4.1 Coco-Component

The class Coco-Component declares a uniform interface for all concrete component implementations usable in the framework. All instances of Coco-Component have the following properties:

- Every instance features a unique identifier.
- It returns the component type on request.
- The component's public interface is export-able using the *Hook* design pattern.
- Instances may access other instances using HookUps.
- Hooks and HookUps can be inspected at runtime, with no need to know the component's exact type.
- Observing is possible by inheriting from an *ObserverSubject* base class.
- All methods of Coco-Component feature checks for preconditions, postconditions and the object invariant.

Every component instance has an identifier which must be uniquely addressable in a component model. All components are arranged in the `ComponentRegistry`, a container allowing access to all used component instances. The registry is further explained section 4.5.3 .

4.1.1 Structure

Figure 4.1 illustrates the structure of the Coco-Component class.

4.1.2 Participants

- **ObserverSubject:** The ObserverSubject is the subject part of the *Observer* design pattern described in [GHJV95]. It provides the functionality to attach or detach any component which is derived from

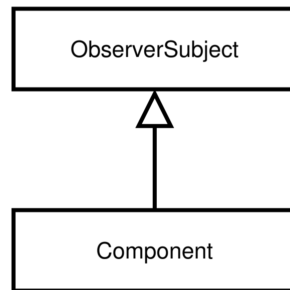

Figure 4.1: Component structure

Observer. Every component instance should be observable to notify it's marshaller of state changes allowing transparent object persistence.

- **Component:** Is the base class of all components. The provided methods are out lined below.

4.1.3 Implementation

Exporting Interfaces

Every component has a public interface which should be exposed to the world. This is done by using the *Hook* design pattern.

For the user of a component it is necessary not only to know the things a component "can do" but also the dependencies to other components if there are any. First of all the intended features need to be registered which is described in section *Initialization* on page 96.

After registration the exported interface can be queried from other components by using one of the following public methods:

```
1 bool hasHook(const std::string& hook_id) const;
2 HookBase& getHook(const std::string& hook_id) const;
3 template<class HookT>
4   HookT& getHook(const std::string& hook_id) const;
5
6 bool hasHookUp(const std::string& hookup_id) const;
7 HookUpBase& getHookUp(const std::string& hookup_id) const;
```

```
8   template<class HookUpT>
9     HookUpT& getHookUp(const std::string& hookup_id) const;
```

Listing 4.1: Query interface for Coco::Component (a)

A component can be queried for a particular *Hook* or *HookUp*. The given identifiers must correspond with the *Hook* and *HookUp* identifiers. If the `hasHook` method returns `true`, a *Hook* reference can be retrieved by either of the following methods:

- getHook as given in line 2: This method returns a reference of type `HookBase`. If the client wants a specific derived type, he must cast it to the one he needs.
- getHook as given in line 3-4: This template method returns the hook type that is given as template parameter. This method automatically executes the cast that otherwise has to be done by the client. First it fetches the *HookBase* by using the method of line two. Then it casts the reference to the requested Hook type. If the cast fails an exception is thrown.

Similar methods are given in respect to *HookUps*. Two additional methods are implemented in the component base class to get a list of either *Hooks* or *HookUps* of a component:

```
1   const HookIDList& getHookIDList() const;
2   const HookUpIDList& getHookUpIDList() const;
```

Listing 4.2: Query interface for Coco::Component (b)

Connections

As connection mechanism, the *Hook* design pattern is again of use (see [Lau05]). Components may access other components using *HookUps*. A *HookUp* can be connected to a *Hook* using the following method:

```
1 void connectWith(const std::string& hookup_id, HookBase& hook);
```

Listing 4.3: connect method

This method connects the *HookUp* given by its identifier, to a *Hook* of another component. For successful linking, some preconditions have to be fulfilled:

- The calling component must have a *HookUp* with the given identifier.
- *HookUp* and *Hook* must be connectable. This property can be checked using the method `isConnectAbleToHook` that will return true if a connection is possible (see listing 4.4). This method allows to test at runtime if a *Hook* and a *HookUp* can be connected.

```
1 bool isConnectAbleToHook(const std::string& hookup_id,
2                          HookBase& hook) const;
```

Listing 4.4: is connectable to hook method

Currently it is possible to check if connections are possible. Future development should enable the runtime inspection of the *Hook* and *HookUp* so that the number of parameters and their types can be retrieved.

A drawback of current *Interface Definition Languages* (IDLs) is the lack of semantic description as discussed in section 3.6.4 on page 83. The pre-, postconditions and the invariant must be given in some form to the client. In Eiffel this is done with the tool *short*.

The only suitable solution in C++ is to define contracts as comments preceding the function declarations. These contracts can be described by using the *Doxygen* syntax[1] providing the client with the needed information in different readable formats. Further development will include a mechanism for runtime inspection so that the client can ask a component directly about it's contracts. The developer does not need to uses separate documentation.

The great advantage of the Hook based connection concept is that components can be linked without the need of knowing their exact types.

[1] see `http://www.doxygen.org`

4.1.4 Design by Contract™ in Coco-Component

Design by Contract™ (DbC) poses a problem when C++ is used which was already discussed in [Lau05].

Consider the `doRegisterInterfaces` method as an example. This method is declared abstract in the component base class. Hence, every class that is derived from it must implement that method. As said above, it is intended to register the hooks and hookups of a component.

DbC requires that at least precondition, postcondition and an invariant should be checked everytime a method is called. Let's take the invariant as example and the problems that evolved by implementing it.

An invariant describes the valid states of an object that must be assured and checked before and after each method call. After every method call it must be checked. As C++ has no built in methods to check the invariant automatically, this has to be ensured in the software design.

Let's assume that the component base class has a method `checkInvariant`. A method that uses the invariant check can look like this one:

```
1  int Foo::doSomething()
2  {
3    assert (checkInvariant());
4    // do implementation
5    assert (checkInvariant());
6  }
```

Listing 4.5: method with invariant check

The invariant is checked at the entry and the exit point of the method. Doing it at the exit point is clear because the function body would violate the invariant rule. The checking at the entry point might be surprising. One may say that only the exported public methods could be used which are always checked against the invariant. Normally it is not possible that the object state can be changed between two method calls. But the following situation shown in figure 4.2 may arise:

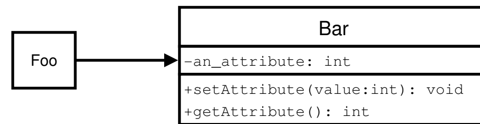

Figure 4.2: Example for needed invariant check at method entry point

An object `Bar` has a private feature. No direct manipulation is possible. The attribute can be set with the `setAttribute` method and the actual value can be retrieved by calling `getAttribute`. Both methods do invariant checking at the exit point.

For whatever reason lets say that `Foo` has a pointer to the private member of `Bar`. `Foo` can directly change `an_attribute` without using the supposed getter or setter method of `Bar`. `Bar's` internal state may therefore be changed between two method calls. This is the reason why the invariant must be checked at the methods entry point too.

So far we have said when the invariant must be checked. The question is if it is possible to check the invariant of inherited classes. In this case the following rule must be fulfilled (taken from [Mey97]):

Invariant inheritance rule

The invariant property of a class is the boolean **and** of the assertions appearing in its **invariant** clause and of the invariant properties of its parent if any.

This means that the invariant of any derived class and that of its base class must hold.

In C++ the use of the *Template Methods* pattern will provide a solution (discussed in [Lau05]).

Let's come back to the problem with the *doRegisterInterfaces* method. The programmer who implements this method should not be concerned with the invariant handling. The following listings shows how the *Template*

Method pattern is used in the component design to overcome this problem:

```
class Component
{
  public:
  void aTemplateMethod()
  {
    checkInvariant();
    doRegisterInterfaces();
    checkInvariant();
  }
  protected:
  virtual void  doRegisterInterfaces() = 0;
  virtual void checkInvariant()
  {
    cout << "check invariant of Component" << endl;
  }
};
class ConcreteComponent : public Component
{
  protected:
  virtual void doRegisterInterfaces()
  {
    cout << "ConcreteComponent::doRegisterInterfaces" << endl;
    // do some implementation
  };

  virtual void checkInvariant()
  {
    Component::checkInvariant();
    cout << "check invariant of ConcreteComponent" << endl;
  }
};

int main()
{
  ConcreteComponent concrete_component;
  concrete_component.aTemplateMethod();
  return 1;
}
```

Listing 4.6: template method example

The output of would look like the following one:

```
check invariant of Component
check invariant of ConcreteComponent
ConcreteComponent::doRegisterInterfaces
check invariant of Component
```

```
5   check invariant of ConcreteComponent
```

Listing 4.7: template method screen dump

The screen dump shows that the invariants of both, `Component` and `ConcreteComponent` are properly called. When a developer constructs a new component and an according invariant routine, he has to take care that the base class is called too. Line 37 of the code above shows how this can be done in C++. This pattern can also be applied to satisfy the pre- and postcondition checks.

As stated, *Template Methods* provide a way to divide a method in two parts:

- A static part, that does not change during inheritance. This is always implemented in the base class.
- An individual part, that is implemented by a derived class. Every inherited class can have another implementation of the method.

In the case of the invariant check, the static part consists of calling the `checkInvariant` at the beginning and the ending of the method. In between these two calls, the individual part is invoked which is the `doRegisterInterfaces`. This can be seen from line 5 to 12.

The developer of a component does not have to be concernde with the proper invariant check as this is done in the base class.

The client, who uses a `ConcreteComponent` object, has to invoke `aTemplateMethod` in order to initialize the interfaces. As the interface information is hard coded, the question arises if it is possible that the init routine `aTemplateMethod` is called at construction time, in the constructor of `Component`:

```
1 class Component
2 {
3   Component()
4   {
```

```
5        aTemplateMethod();
6      }
7      ...
8  };
```

Listing 4.8: calling template method in constructor

To proof that the given code will not work as expected, the construction procedure of a `ConcreteComponent` must be considered:

1. The constructor of `Component` is called and the *Virtual Method Table* (VMT) is built holding entries of all virtual methods of `Component`.

2. The constructor of `ConcreteComponent` is called an the entries in the VMT are updated.

When the template method is called in the constructor of the base class (as shown in line 5 of Listing 4.8), the VMT entry points to the `doRegisterInterfaces` method of `Component`. Thus this method is executed instead of the overridden one of `ConcreteComponent`. The VMT entry of `doRegisterInterfaces` is set in the constructor of `ConcreteComponent` which is too late because this happens after `doRegisterInterfaces` was called in `Component`.

There is no way in C++ to call a virtual method correctly at initialization time. Therefore a custom creation and init routine has to be called by the application that fulfil this task.

In the case of `Coco-Component`, the template method is "`initComponent`". If the component factory is used (see section 4.5.2 on page 115), this method is executed implicitly.

Initialization

The initialization of a component is divided into several parts:

- **Register Interfaces:** The interfaces which are exported have to be registered.

  Registering interfaces means to give a *Hook* or *HookUp* a name which is unique concerning the component. The registration is supported through the following methods:

```
1    void registerHook(HookBase* hook, const std::string& hook_id);
2    void registerHookUp(HookUpBase* hookup, const std::string&
         identifier);
```

Listing 4.9: Hook and Hookup Container

  - *registerHook:* The given *Hook* is registered with the given hook id.
  - *registerHookUp:* The given *HookUp* is registered with the given hookup id.

  Coco-Component has two independent containers for *Hooks* and *HookUps* therefore it is possible to register a *Hook* and a *HookUp* with the same identifier.

  The following code is an example of how this methods are accurately used:

```
1    registerHook(&result_, "result");
2    registerHookUp(&parameter_, "parameter");
```

Listing 4.10: doRegisterInterfaces example

  It is obvious that `result_` must be defined as a *Hook* and `parameter_` as a *HookUp*. Additionally every declared *Hook* has to be bound to the method it exports. This binding is usually done in the components constructor.

  The signature of a *Hook* and its bounding method have to match meaning that the return type as well as the number of the arguments and their types have to be the same in both cases.

- **Restoring internal state:** A marshaller initializes the internal states of a component. Members whose initial values can be derived from configurable ones are set by calling the following method after the marshaller:

```
1        virtual void configureComponentByMarshaller();
```

Listing 4.11: configuration by marshaller

- **init component using connected components:** To improve the performance and reduce unnecessary method calls the following method can be used to retrieve data from connected components:

```
1        virtual void initComponentUsingConnectedComponent();
```

Listing 4.12: configuration by marshaller

Howto construct a new component

This section summarizes the facts discussed and provides the information needed to design a new component.

First of all, the new component must be derived from the `Coco-Component` class. Some methods *must* be implemented because they are declared abstract in the base class. Besides there are other methods that can be implemented optionally.

The custom initializer `initComponent` has to be called after an object is instantiated. It calls the method that register the public interfaces.

Methods that must be implemented

method name and syntax	**short description**
std::string doGetType() const	returns the type of the component as string
void doRegisterInterfaces()	perform the registration of the interfaces that should be exported and public available. This include *Hooks* as well as *HookUps*. See section *Initialization* on page 96 for more information about registration.
virtual void configureComponentByMarshaller()	method that is called by the marshaller after the states have been read to calculate private members that depend on configurable parameters.

Methods that can optionally be implemented

method name and syntax	**short description**
virtual void initComponentUsingConnectedComponent()	get information of already connected components and cache them to redundant calls through the *Hook* mechanism.
bool checkInvariant() const	Method that performs checks of internal states that must be given at any time. Do not forget to call `checkInvariant` from the base class.

Outlook

The current design forces that every component is observable. This comes from the former thoughts that every component must have a marshaller that stores or restores state information.

A lot of components were developed that do not have any state information to be stored/restored or were intended to be observable. Nevertheless these components had the implemented functionality of being observable. To overcome this design flaw one of the following modifications can be incorporated:

- Introduce another indirection that defines an observable component as shown in figure 4.3.
- Use *MultiHookUp* as described in [Lau05]. Instead of connected to only one *Hook* a *MultiHookUp* can be connected to more *Hooks*. Figure 4.4 shows the structure of a *MultiHookUp*. The advantage of this technique is that the connection mechanism of components can be used to connect *Observer* and *ObserverSubject*.

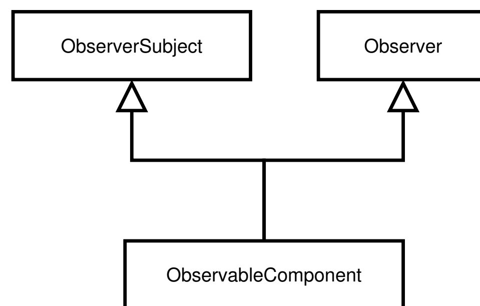

Figure 4.3: Structure of an observable component

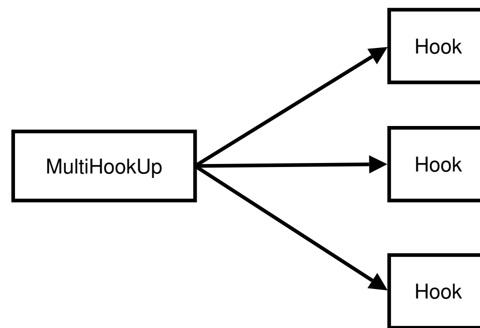

Figure 4.4: MultihookUp as Observer

4.2 Coco-CompositeComponent

A *Coco-Component* as described in section 4.1 can be connected to other components through the *Hook* mechanism. This poses a connection-oriented composition. The different forms of compositions are discussed in section 3.5 on page 75.

The *Coco* library also supports the contextual composition which is also known as container-based composition. The name already implies that the key concept of this form is to have containers that internally hold other components. This container forms a hull around its components which may intercepts any incoming and outgoing messages as shown in figure 4.5.

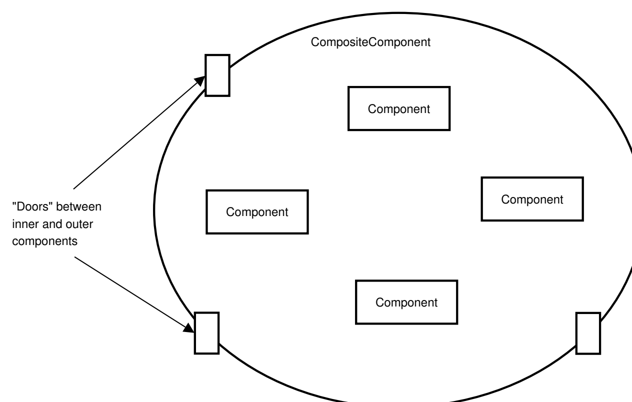

Figure 4.5: Spanning hull and communication “doors” of a composite component

Coco-CompositeComponent is the container developed to provide context-based composition. Again it is a component as it is derived from *Coco-Component*. Thus it can be treated as a usual component. Communication with the container is therefore connection oriented.

4.2.1 Structure

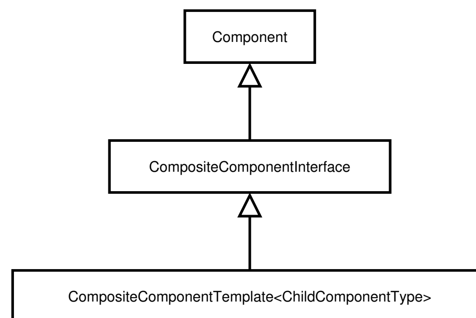

Figure 4.6: Composite component structure

4.2.2 Participants

- **Component:** Provides the functionality needed for a component.
- **CompositeComponentInterface:** Defines the interface for the client.
- **CompositeComponentTemplate:** Assures that the inner components have to be of a distinct type. It realizes the idea of constrained genericity.

4.2.3 Implementation

Constrained Genericity

Constrained genericity is an important technique to combine genericity and inheritance. Genericity is achieved by the generic programming technique described in section 2.3.4 on page 42.

A composite component should only have components of a specific type or a descendant of the given type. Therefore the *CompositeComponentTemplate* class is a template class with a template parameter that specifies the type of the inner components. An example of that usage is the *RegisterFile* used in the xDSP model. A *Registerfile* is nothing more than a container of registers.

To assure that a *RegisterFile* only contains *Register* components, it has to be derived from *CompositeComponentTemplate* in the following way:

```
1 class RegisterFile : public Coco::CompositeComponentTemplate<Register>
```

Listing 4.13: concrete composite component

Another example of a composite component is the *Coco-ComponentModel* further described in section 4.3.

Inner Component handling

There are additional methods for component handling. The following ones are intended to add or remove a component:

```
1 bool addComponent(const std::string& comp_id,
2     boost::shared_ptr<Component> component);
3 bool removeComponent(const std::string& comp_id);
```

Listing 4.14: methods to add or remove a component

- *addComponent:* adds the given component and registers it internally with the given identifier. It returns true, if the component could be added.
- *removeComponent:* removes the component with the given identifier. Returns true if the remove operation was successful.

For querying the composite component, the following methods are provided:

```
1 boost::shared_ptr<Component> getComponent(const std::string& comp_id);
2 bool hasComponent(const std::string& comp_id) const;
3 ComponentList getRegisteredComponents() const;
```

Listing 4.15: query methods of composite component

- *getComponent:* This method returns the component with the given identifier. The precondition claims that a component with such identifier is given.
- *hasComponent:* Returns true if a component with the given identifier is registered.
- *getRegisteredComponents:* A list of the registered components will be returned.

Howto construct a composite component

The construction of a composite component involves nearly the same steps as the construction of a "normal" component (see section 4.1.4). There is only one difference, namely that is has to be derived from *CompositeComponentTemplate* instead of *Component*. Further the developer has to define the type of the inner components and pass it as template parameter to the parent class. Listing 4.13 shows how this can be done.

4.2.4 Outlook

The public interface in the form of registered *Hooks* and *HookUps* of a composite component does not change if a component is added.

In some situation it is preferable to define a new component consisting of several other components. This new component has an interface that resembles the interfaces of its internal components. Figure 4.7 shows an example of a subtracter that is composed of an adder and an inverter, each being a component.

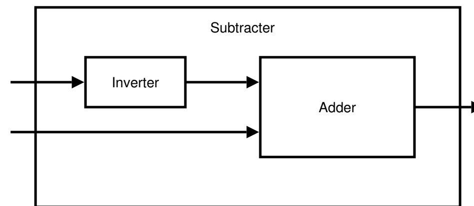

Figure 4.7: Subtracter that is composed through an adder and inverter

The adder has two *HookUps*, to get the parameters that should be added, and a *Hook* that returns the result of the addition. Only one *HookUp* and one *Hook* are defined for the inverter. The resulting subtracter has two *HookUps*, one from the inverter and one from the adder, and one *Hook* which is the one from the adder.

To avoid the need of rebuilding the subtracter of it's sub components each time, a mechanism to create new composed components has to be developed. Some sort of composition that exports *Hooks* and *HookUps* of the inner components is desired. This will enable to define a subtracter component only once and use it as often as it is needed.

4.3 Coco-ComponentModel

4.3.1 Structure

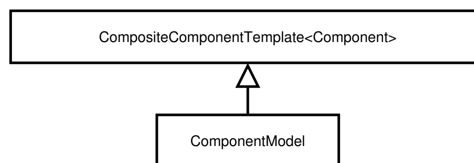

Figure 4.8: Structure of the component model

The intention of the component model is to have a container that holds all components of a given model. Due to the fact that it is derived from *CompositeComponent* it behaves as a component. Therefore it is possible

to connect it to any other component or component model.

Other components can use the following *Hooks*:

- *instantiateComponents:* Instantiates the components which identifier are given by a reader and adds them to the model. This *Hook* is bound to the *instantiateComponents* method described in section 4.3.2.
- *connectComponents:* The components that are given by the reader will be properly connected. See section *Connect components* for more details.

4.3.2 Implementation

The main tasks of the component model are:

1. Instantiate components
2. Connect components

Instantiate components

The following method instantiates components and adds them to the model:

```
1 void instantiateComponents(const ComponentStateReader& reader, unsigned
2     int depth);
```

Listing 4.16: method to build the component model

Collaborations:

The sequence diagram figure 4.9 shows the participants that are involved in the process of instantiating the components of a model:

- **ComponentFactory:** The component factory returns a component with the given identifier which is in this case a *std::string* that comes from the component state reader.

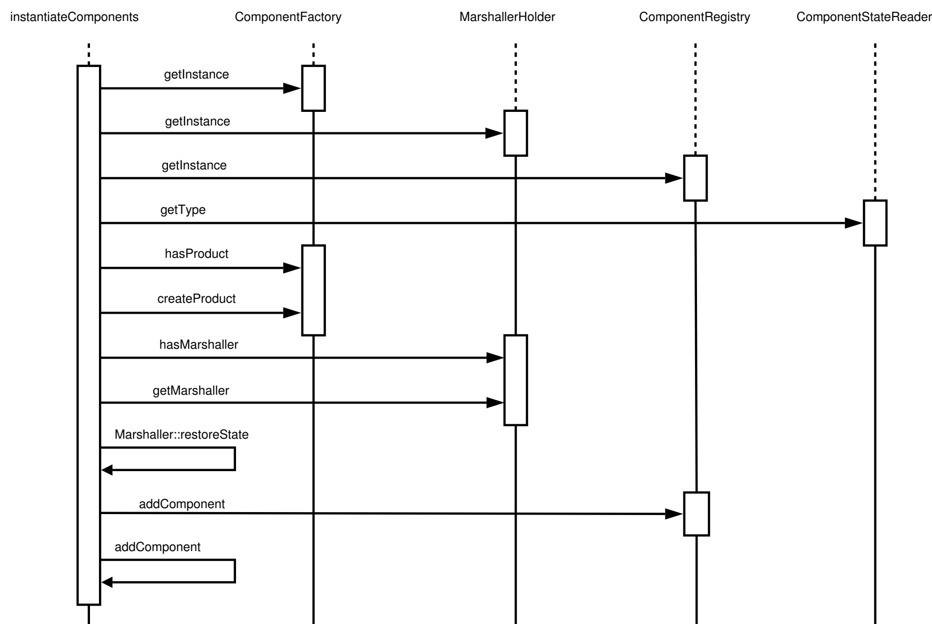

Figure 4.9: Sequence diagram of instantiating components

- **MarshallerHolder:** Does the same as the *ComponentFactory* but returns a *Marshaller* for a component instead of a component instance.
- **ComponentRegistry:** Is a global object registry that holds component instances of all models.
- **ComponentStateReader:** A reader that provides information needed to construct a component and its marshaller.

After a component is properly constructed and initialized by its marshaller, it is added to the model. Remember that the model is a composite component. Additionally it is added to the registry to be accessible throughout existing component models.

Connect components

Connections of components are done with the next method:

```
1 void connectComponents(const ComponentStateReader& reader, unsigned
2     int depth);
```

Listing 4.17: method to connect components

Collaborations:

The following sequence diagram figure 4.10 shows the procedure of connecting components of a component model:

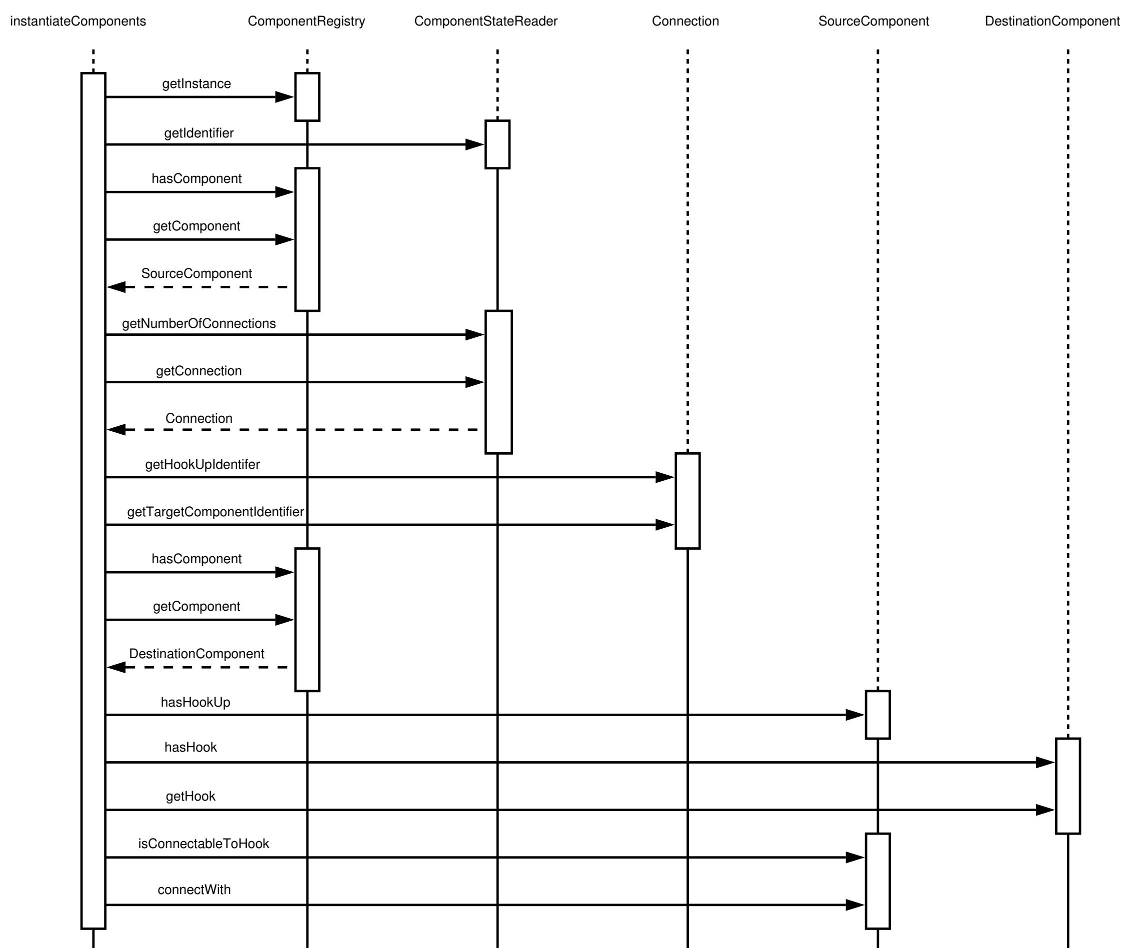

Figure 4.10: Sequence diagram of connecting components

- **ComponentRegistry:** The component registry is used to get the two components that should be connected. Some may argue that

the components that should be connected are already parts of the component model and therefore it is not needed to use the registry to retrieve any component.

This is true as long as only one model exists. If more than one model is defined, it is possible that a component from one model wants to connect to a model given by another one.

- **ComponentStateReader:** Gives information about the connection.
- **Connection:** Holds all information about which *HookUp* should be connected to which *Hook*.
- **SourceComponent:** Is the component which needs the some functionality of another component. The *HookUp* of this component is connected with the *Hook* of another one. This information is given by the *Connection* object.
- **DestinationComponent:** Is the component which provides functionality in form of one or more *Hooks*.

4.4 Coco-Marshaller

The marshaller concept is used to implement persistence for *Coco-Components*. A component does not have to be aware of how and where the states are stored/restored. Every component has its own marshaller that knows all its internals. Where the data is written to or read from depends on the writer or reader passed to the marshaller.

Storing and restoring states are needed to realize back stepping or in the sense of simulation: to start simulation at a desired point.

4.4.1 Implementation

A marshaller consists of the two public methods:

```
void restoreState(boost::shared_ptr<Component> component, const
ComponentStateReader& reader);
void storeState(boost::shared_ptr<Component> component,
boost::shared_ptr<ComponentStateWriter> writer);
```

Listing 4.18: public methods of a marshaller

- *restoreState:* Fetches data from the given reader and sets the appropriate members to that data.
- *storeState:* The state of the component is written to any format depending on the writer.

The marshaller concept is used in two different ways:

- The configuration is read from the configuration file. The component model directly gets a desired marshaller from the globally accessible *MarshallerHolder* and calls the appropriate method. This can be seen at the instantiation time of the component model in section 4.3.2.
- The marshaller is called by an observer which is notified somehow. This circumstance is shown in figure 4.11.

  This concrete observer passes a *ComponentStateReader* or *ComponentStateWriter* as parameter to the concrete marshaller.

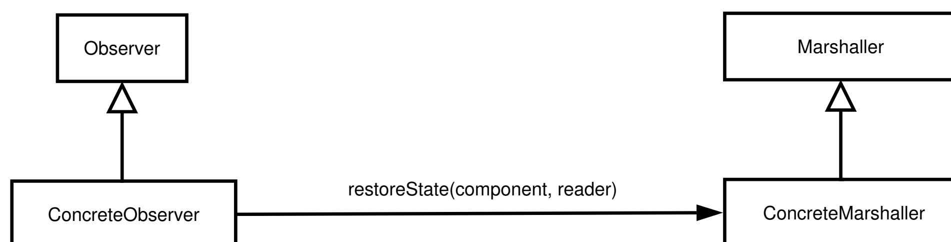

Figure 4.11: Marshaller concept

Friend concept

Due to store/restore the states of a component there must be a way to access the members. Object oriented programming forces information hiding so members are not accessible from any external component. A way to overcome this is to write getter and setter for every *private* state. Usually it is not preferred to give access to every internal member.

In C++ there is a way to allow a class of a specific type to have access to non-public members. A class has to declare the other as `friend`. A friend of a base class cannot access private members of a derived class. This is shown in the following code fragment:

```
class Base
{
  friend class Friend;
  private:
  int a_member;
};
class Derived : public Base
{
  private:
  int another_member;
};

class Friend
{
  public:
  void foo()
  {
    Derived derived;
    derived.a_member = 5;
    derived.another_member = 5; // that line is not allowed
  }
};
```

Listing 4.19: friend of derived class

The compiler will complain that line 20 is not valid. Any instance of `Friend` is allowed to change private members of `Base` but not the ones additionally declared in `Derived`.

A similar situation is given in the component design. The base class of all

components can only declare the base class of all marshallers to be a friend. Every concrete component has individual members that must be considered for serialization, so a dedicated marshaller is needed for it. This component must explicitly declare the marshaller as friend due to the drawback shown by the code example above.

4.4.2 Outlook

Constructing a marshaller is straight forward. It has to overwrite the methods *doRestoreState* and *doStoreState* which are called by *restoreState* and *storeState* of the base class. The *Template Method* is used here again to perform DbC.

In this method nothing more is done than to store the components state with a writer or to set the components internal members to the values given by a reader. According to the rules of a marshaller, it can be generated automatically. The states that should be taken into consideration can be marked by some meta data which a be easily identified by a parser. The following code is an example of how members can be marked in the doxygen description of a method:

```
1 /**
2   * @marshal
3   */
4 int a_member;
```

Listing 4.20: example of tagging a member for marshalling

4.5 Component Organisation

The need to have a central point for creation and registration of components has evolved in the design of the component framework.

This section describes the two parts which serve for the component organisation. On the one hand there is a factory for creating a component and

on the other hand there is a registry that is aware of any existing component in the framework.

4.5.1 Singleton

Both, the factory and the registry, should be unique for the lifetime of the framework. For this purpose they are implemented as singletons.

To achieve this goal, the *Loki Singleton* was used (see [Ale01]). Alexandrescu proposed to use his singleton in the following way:

```
1 class MyClass { ... };
2 typedef SingletonHolder<MyClass> MySingleClass;
```

Listing 4.21: using singleton as proposed by Alexandrescu

This strategy has the following drawbacks:

- Although *MyClass* is intended to be a singleton, it can be instantiated more than once by simply using the following lines of code:

  ```
  1 MyClass first_class;
  2 MyClass second_class;
  3 ...
  ```

  Listing 4.22: instantiate singleton more than once

  At first sight it would be satisfying to put the constructor of *MyClass* in the private scope. Compiling the above code would not be successful. As well as constructing a singleton through the singleton concept would not be successful. The given creation policy is not able to construct an object of type *Foo* because of its private constructor.

- The instance of the singleton is retrieved through the following code which is not very elegant:

  ```
  1 MyClass &my_class = MySingleClass::getInstance();
  ```

  Listing 4.23: create singleton as proposed by Alexandrescu

Using *MyClass* as singleton requires to know the type definition of *MySingleClass*. In cases where the singleton type definition does not have a similar sounding name, things will become more complicated to use.

To overcome the above mentioned drawbacks, a singleton class is constructed like the class *Foo* in the code below:

```
1 class Foo
2 {
3 public:
4     typedef SingletonHolder<Foo> Singleton;
5 private:
6   friend class CreateUsingNew<Foo>;
7   Foo(){};
8   Foo(const Foo &instance) {};
9 };
```

Listing 4.24: singleton concept used in the framework

It is not possible to instantiate *Foo* without using the singleton because the constructor is set private. It is also important to set the copy constructor private because otherwise it is possible to an object through the singleton mechanism and then to copy it.

Without a private copy constructor the following code is valid and will lead in having two *Foo* objects although as singleton only one is allowed:

```
1   Foo &first_foo = Foo::Singleton::getInstance();
2   Foo second_foo(first_foo);
```

Listing 4.25: valid code without having a private copy constructor

Line 1 of the code above shows how the instance of the singleton can be retrieved.

There is only one drawback: It is assumed that the used creation policy is *CreateUsingNew*. Declaring it as a friend (as shown in line 6 of Listing 4.24) makes it possible to construct an object of type *Foo* even though the constructor is private.

4.5.2 Coco-ComponentFactory

The component factory is able to instantiate new components on demand. As explained, it is implemented as a singleton.

Structure

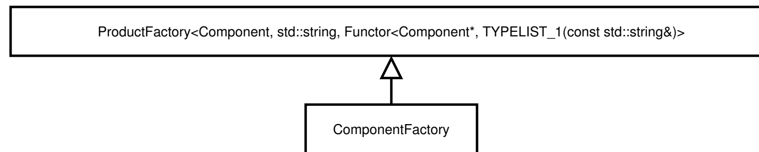

Figure 4.12: Structure of the component factory

A short description of the parameters of *ProductFactory*:

1. parameter is the base type for the objects that should be created.
2. parameter is the identifier type for the creating objects. In this case it is a string.
3. parameter is the creating policy of how an object should be created.

Implementation

The most important methods for the factory are the following one:

```
1 bool hasProduct(const IdentifierType& id) const;
2 bool registerProduct(const IdentifierType& id, ProductCreator creator
      );
3 bool unregisterProduct(const IdentifierType& id);
4 bool unregisterAllProducts();
5 std::auto_ptr<Component> createProduct(const std::string&
      component_type,
6 const std::string& component_id);
```

Listing 4.26: methods of the component factory

- *hasProduct:* Returns true if a product with the given identifier is registered.
- *registerProduct:* Registers a product with an identifier and product creator pair.
- *unregisterProduct:* With this method it is possible to unregister an already registered product with the given identifier.
- *unregisterAllProducts:* Unregisters all products.
- *createProduct:* Returns a new product with the given type and identifier.

Creator for Components

Everytime a product should be registered in the factory an additional creator for it must be passed as parameter. This creator must be a functor as it can be seen in figure 4.12. The component factory provides a template class that can be used as creator:

```
1 template <class DataType>
2 class Creator
3 {
4   public:
5   DataType* operator() (const std::string& identifier)
6   {
7     return new DataType(identifier);
8   }
9 };
```

Listing 4.27: Creator defined for the component factory

With this creator the registering of a product is a very simple process:

```
1 registerProduct("Foo", Coco::ComponentFactory::Creator<Foo>());
```

Listing 4.28: Example to register a product

4.5.3 Coco-ComponentRegistry

The Component Registry is the administration or management container for component instances. All created *Coco-Component* objects have to be registered in the registry to be dynamically connectable. If a component wants to hookup to another instance a registry must have the information if the target instance exists.

It has the following public methods:

```
1 bool hasComponent(const std::string& comp_id) const;
2 bool addComponent(boost::shared_ptr<Component> component);
3 bool removeComponent(const std::string& comp_id);
4 boost::shared_ptr<Component> getComponent(const std::string& comp_id)
      const;
```

Listing 4.29: methods of the component registry

- *hasComponent:* Returns true if a component with the given identifier is in the registry.
- *addComponent:* Adds the given component to the registry. Returns true if the operation was successful.
- *removeComponent:* Removes the component with the given identifier. Returns true on success.
- *getComponent:* Returns a shared point to the component with the given identifier.

Chapter 5

Coco Soap Library

The *Hook* concept discussed has the drawback that connected components have to be part of the same address space. Component base software should not be bound to only one machine, as distributed computation is one of the main concepts of component based development.

As already discussed component models have evolved over the years (section 2.4.3 on page 46). Most of them communicate through a binary protocol. Version mismatches of those protocols often hinders components to talk to each other.

Web services are one concept that tries to use as much standardized techniques as possible. For communication the SOAP protocol is used which transfers XML-based messages between the communication endpoints. Customers can easily read the message stream as it is a human readable format.

The advantages of a text base communication protocol convinced us to use SOAP and not any proprietary binary protocol.

5.1 Simple Object Access Protocol (SOAP)

SOAP was initially created by Microsoft, later on IBM, Lotus and Userland joined in the developing process.

The Simple Object Access Protocol (SOAP) [soa] is a simple and lightweight mechanism for exchanging structured and typed information between peers. It is language and platform neutral as the protocol is XML-based. There is no need to have a certain platform or programming language to work with SOAP.

SOAP aims to bridge the sharing of resources among disparate organizations which can possibly be located behind firewalls. Due to the fact, that the exchange mechanism is based on XML, there is no need to have the server and client be implemented in specific programming languages. But a XML parser and generator for parsing the incoming and constructing the outgoing messages have to exist.

There are toolkits for nearly every known programming language like a CGI-based perl version *SOAP::lite* [RJR] or the stubs and skeleton compiler *gSOAP* [vE] for C++ to mention some of them. The *gsoap* SDK toolkit will be explained later in more detail.

The SOAP messages are not bound to a specific protocol. So it can be sent by using either HTTP, SMTP, TCP or any other one. There is also the possibility to route a SOAP message, meaning that a message is sent to a router which in turn sends it to another machine. The router can even use another transport protocol. For increase security the HTTPS protocol can be used, which encrypts the header and the body of the message automatically.

By using the WSDL it is possible to get automatic generated SOAP stubs for the development of clients within arbitrary programming languages. These stubs are used to invoke the remote functions of the web service.

A SOAP message consists of a XML element with two children: header and body. The following code example shows the basic message structure (taken from [CDK+02]):

```
<SOAP:Envelope
  xmlns:SOAP=''http://schemas.xmlsoap.org/soap/envelope/''>
  <SOAP:Header>
    <!-- content of the header -->
  </SOAP:Header>
  <SOAP:Body>
    <!-- content of the body -->
  </SOAP:Body>
</SOAP:Envelope>
```

Listing 5.1: SOAP envelope

The following example shows a SOAP message that is transferred with the HTTP protocol. In this example a person called "John Smith" wants to get a flight info:

```
POST /travelservice
SOAPAction: ''http://www.acme-travel.com/checkin''
Content-Type: text/xml; charset= ''utf-8''
Content-Length: nnnn

<SOAP:Envelope
  xmlns:SOAP=''http://schemas.xmlsoap.org/soap/envelope/''>
  <SOAP:Body>
    <et:eTicket xmlns:et=
      ''http://www.acme-travel.com/eticket/schema''>
     <et:passengerName first=''Joe'' last=''Smith''/>
     <et:flightInfo airlineName=''AA''
                    flightNumber=''1111''
        departureDate=''2002-01-02''
        departureTime'=''1905''/>
    </et:eTicket>
  </SOAP:Body>
</SOAP:Envelope>
```

Listing 5.2: SOAP message

The message is transferred using *HTTP Post*. The next line defines a *SOAPAction*. This line is optional and only indicates the purpose of the message. A firewall can be configured to use this optional information to let SOAP messages pass.

Additionally, there must be a way for defining typed values. The typed values given in the SOAP message have to be converted to the appropriate values for the application on the other end point of the communication part. It must be possible to transfer an integer in such a manner, so that the receiving party can convert it to it's original integer representation. As SOAP is programming language independent a method must be used that is understood by everybody. The W3C's XML schema specification [xml] provides a standard language to solve this problem. It offers a way to define types using XML.

The above example using schema typed values looks like this:

```
1 POST /travelservice
2 SOAPAction: ``http://www.acme-travel.com/checkin''
3 Content-Type: text/xml; charset= ``utf-8''
4 Content-Length: nnnn
5
6 <SOAP:Envelope
7   xmlns:SOAP=''http://schemas.xmlsoap.org/soap/envelope/''>
8   <SOAP:Body>
9     <m:GetFlightInfo
10      xmlns:m=''http://www.acme-travel.com/flightinfo''
11      SOAP:encodingStyle=''http://schemas.xmlsoap.org/soap/encoding
          /W
12      xmlns:xsd=''http://www.w3.org/2001/XMLSchema''
13      xmlns:xsi=''http://www.w3.org/2001/XMLSchema-instance''>
14      <airlineName xsi:type=''xsd:string''>UL</airlineName>
15      <flightNumber xsi:type=''xsd:int''>506</flightNumber>
16    </m:GetFlightInfo>
17  </SOAP:Body>
18 </SOAP:Envelope>
```

Listing 5.3: SOAP message with schema typed values

The *encodingStyle* attribute defines a set of serialization rules. These can be used to deserialize the SOAP message.

SOAP only offers basic methods for communication. There is no possibility to find out what messages must be exchanged for successfully communication with a webservice. This task is fulfilled by the *Web Service Description Language*.

5.2 The gSOAP Toolkit

gSOAP [vE] is a stub and skeleton compiler to ease the development of webservices and applications for C and C++. Unlike other toolkits for webservice development, gSOAP hides the specific aspects of a networking library and specific data structures. Due to the fact that it is a pre compiler it automatically translates the data types given in the application to the semantically equivalent XML data types and vice versa. The advantage of this strategy is that the developer does not have to face SOAP specific logic.

Some of the features of gSOAP:

- support SOAP 1.1/1.2 and WSDL 1.1
- platform independent
- includes a WSDL generator.
- WSDL parser for automated client and server development.
- support gzip compression, SSL
- schema-specific XML pull parser
- supports the use of some STL Containers like *std::vector*, *std::list*, *std::set* and *std::deque*.

gSOAP performs the following mapping between C/C++ data types and SOAP XML schema types [xml]:

Basic Types : The primitive types like char* and float are encoded to the primitive XML schema types like *xsd:string*, *xsd:double*.

Enumeration : This type is encoded to the *enumeration* XML schema type.

Structs : Encoded to *complexType*

Classes : Encoded to *complexType*. All fields are recursively encoded and treated having public access privilege.

Pointer : Pointers are serialized and deserialized using the SOAP feature for multi-referencing of data. This ability avoids transferring an object more than one time, if more than one pointer refers to it. The object is serialized once and identified by an identifier attribute. All pointers are encoded through the use of *href* attribute.

The encoding for other types like *Arrays* or *Compound Types* are described in [vEG].

As stated above the gSOAP package consists of two tools:

- A WSDL parser called *wsdl2h* which converts a given WSDL file into a header file that can be interpreted by the stub and skeleton compiler.
- A stub and skeleton compiler called *soapcpp2*.

5.3 gSoap Code Generator

5.3.1 Generating Client Stub

figure 5.1 shows the steps to construct stub for a client out of a *WSDL* file:

- The *WSDL* file is processed by *gSOAP's WSDL* parser. This parser generates a C/C++ header file which declares the services.
- The *gSOAP* compiler produces the source code to implement the client application. Proxies for the client application are generated.

Using the generated stub the client has transparent access to the server. Serialization and deserialization is performed within the generated proxy classes.

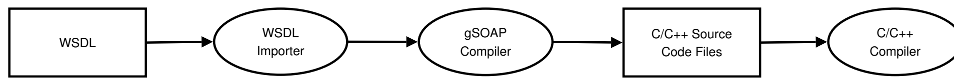

Figure 5.1: Constructing a client with gSOAP

The compiler generates the following files for C++ (generation for C only is done by adding the *-c* switch to the compiler):

soapC.cpp : contains the serializer and deserializer for the used data types.

soapClient.cpp : contains the stub routines that can be called by the client. The stub routine for a method that returns the temperature can look like this one:

```
1 int soap_call_ns1___getTemp(soap, endpoint, NULL, zipcode, &
      result)
```

Listing 5.4: stub routine

The following parameters are given to the method:

1. parameter is a run-time *gSOAP* environment.
2. parameter is the endpoint URL of the web service. If it is NULL, the one given by the WSDL will be used.
3. parameter indicates the SOAP action required by the web service.
4. parameter is the parameter for the remote method invocation.
5. parameter is the return parameter of the remote method. The return parameter is always given as the last one.

The returning integer indicates whether or not the soap call was successful.

gSOAP additionally generates skeleton files which are analogue to the ones described above.

5.3.2 Generating Server Skeleton

Nearly the same procedure as described above is necessary to implement a web service application. There are two possibilities as shown in figure 5.2:

- Do the same steps as for the implementation of a client. Instead of generating stubs, skeletons are created.
- Take a legacy C/C++ application and develop a service simply by entering the C/C++ web service method operations and data types into a header file. The *gSOAP* compiler generates the source code for the project and produces additionally a *WSDL* file. This header file must be understood by the stubs and skeleton compiler.

  Namespace handling by *gSOAP* is done by adding it as prefix to the methods. Two underlines separate the namespace from the method name. Listing 5.6 shows an example. In line 1 a method *getTemperature* is defined which resides under the namespace *ns1*. Furthermore a namespace mapping table must be given manually. The following example shows a namespace mapping table definition:

```
1 struct Namespace namespaces[] =
2 {   // {"ns-prefix", "ns-name"}
3         {"SOAP-ENV", "http://schemas.xmlsoap.org/soap/envelope/"},
4         {"SOAP-ENC", "http://schemas.xmlsoap.org/soap/encoding/"},
5         {"xsi", "http://www.w3.org/2001/XMLSchema-instance"},
6         {"xsd", "http://www.w3.org/2001/XMLSchema"},
7         {"ns1", "urn:xmethods-delayed-quotes"},
8         {NULL, NULL} // end of table
9 };
```

Listing 5.5: namespace mapping table (1)

  The first four parameters of the namespace mapping table consists of standard namespaces used by SOAP 1.1. Namespaces are important for the compiler to know which method should be invoked and where it is located. Every used namespace in the application must have an entry in this mapping table.

  Namespaces allow to define two or more methods with the same name but which are located on different servers.

```
1 int ns1__getTemperature(std::string zipcode, std::string&
      result);
2 int ns2__getTemperature(std::string city, std::string& result);
```

Listing 5.6: namespace declaration of method

The code above is an example of two methods with the same name but using different namespaces. An according mapping table looks like this:

```
1 struct Namespace namespaces[] =
2 {    ...
3   {"ns1", "http://www.a_server.com/schemas/namespace"},
4   {"ns2", "http://www.another_server.com/schemas/namespace"},
5     ...
```

Listing 5.7: namespace mapping table (2)

The clients do not have to be implemented using the same programming language as the server is implemented in (C++ in our case). Another tool like *Apache AXIS*[1] can be used to generate a stub class for the client.

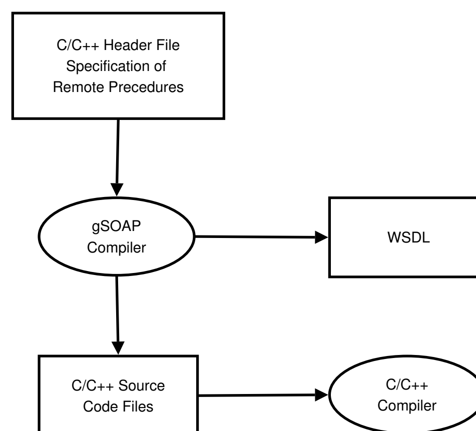

Figure 5.2: Constructing a service with gSOAP

[1]see `http://ws.apache.org/axis/`

5.3.3 Client Example

The client only consists of a few lines:

```
int main()
{
  TemperatureBinding binding;

  struct ns1__getTempResponse response;

  if(binding.ns1__getTemp("10001", response) == SOAP_OK)
  {
    std::cout << response.return_ << std::endl;
  }
  else
  {
    soap_print_fault(binding.soap, stderr);
  }
  return 0;
}
```

Listing 5.8: SOAP client example

The client above queries for the temperature at a place in the US that has the zip code 10001, which is part of New York. The unit of the result is Fahrenheit.

Although gSOAP generates stubs, the client has to call the specific methods that are generated for the stubs instead of the original defined method. For example, we have the following function which should be accessible:

```
int getValue();
```

As explained above (when constructing a server), we will get a function which look like the following one:

```
int soap_call_ns__getValue(struct soap* soap,
                           char* URL,
                           char* action,
                           int& result);
```

5.4 Automated Facade Class Generation

As listing 5.8 shows, the client has to know the namespace and some other information like the URL and the action of the web service that should be called.

For a client it would be preferable to have the same method syntax as it is offered by the server. Figure 5.3 shows that a wrapper can perform the necessary mapping between the special function calls of *gSOAP* and the original method syntax of the service. The client therefore has have a transparent view to the web service API.

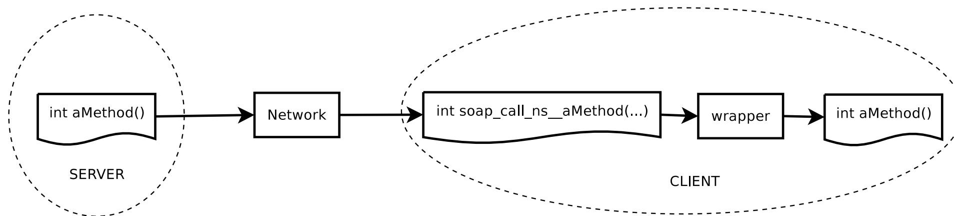

Figure 5.3: Wrapper for transparent function call on client

The wrapper code for the above function looks like this one:

```
1  int getValue()
2  {
3    struct soap* soap = soap_new();
4    int result = 0;
5    soap_call_ns__getValue(soap, ``an_endpoint'', ``an_action'', result
         );
6    return result;
7  }
```

Listing 5.9: Wrapper code for transparent method invocation

Throughout the development of *Coco-Soap*, a script was created that automatically creates a server implementation and a client wrapper facade for a given class. The following things are performed by the script (see also figure 5.4):

- implementation for the soap server

- wrapper for the client

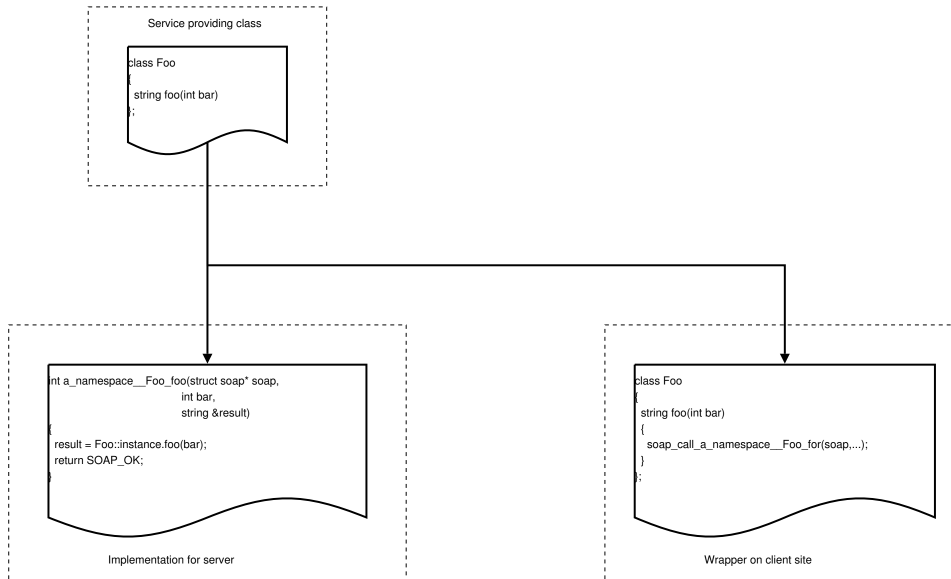

Figure 5.4: Automatic generation of client wrapper and server implementation of a given class

The only prerequisite to the input class is that is must be implemented as a singleton.

5.5 Bidirectional Communication

Suppose the following situation as shown in figure 5.5 on page 130. A simulator kernel and its *Graphical User Interface* (GUI), are internally constructed through component based design. Furthermore both reside on different machines meaning that communication between both of them has to be performed through a network layer.

The GUI retrieves data through the `getValue` method and acts as client. This client (GUI) server (simulator) relationship can be realized with the *gSOAP* toolkit described above.

Somehow, situations may occur when the simulator wants to inform it's

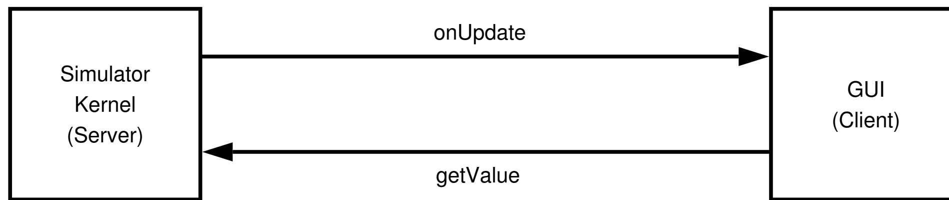

Figure 5.5: Bidirectional communication between a simulator and it's GUI

GUI. That may happen in cases when it is controlled by more than one client at the same time. Bidirectional communication is necessary.

This form of communication poses a problem when using SOAP as the communication is one way only. On the one hand it is possible that a client can talk with a server by invoking it's methods. But on the other hand, there is no chance that the server can notify the client of changes. SOAP just allows unidirectional communication.

Besides this constraint, bidirectional communication can be realized by one of the following solutions:

1. **Polling:** The client calls the server periodically. Using the return parameter the server then can inform the client of an event. This strategy results in a large number of method calls and therefore in an unnecessary network overhead.

2. **Client acts as server:** The client also has to act as server. By adding this functionality to the client, it is possible that other clients (or even a server that acts as client), can call it's public available methods.

   The firewall of the client has to be configured to allow method calls from outside.

If every value of the simulator in figure 5.5 is transferred to the GUI by a call of the remote method, there will be a high traffic overhead.

To avoid this situation, it's better to group related attributes and transfer them as one object. This sort of object is called a *Transfer Object* and is

a design pattern described in [tra]. The server constructs a transfer object and sends it to the requesting client. The client can query this object for needed data. This is done locally on the client side reducing need for remote calls.

Changes of the data on the client side are also done on the transfer object. After such changes, the server must be informed. Again this is done by transferring the transfer object back from the client to the server. The server then synchronizes its data with the new ones retrieved from the client.

Chapter 6

Summary and Conclusions

This thesis evolved from the implementation of an instruction set simulator for a *Digital Signal Processor* (DSP). As its customer needs changed over the time, the requirements for the simulator changed too.

An early simulator was constructed. This was a huge, monolithic application developed for the Windows^{TM} operating system. Techniques for reusability were not taken into account and the time came by where it was nearly impossible to maintain it for bugfixes or adapt it to new feature requirements. Further the architecture of the DSP was coded in a manner where changes could not be made without spending a lot of time to "hack" them in. Severe changes in the hardware could not easily be realized to the application.

The lack in the design and additional feature requirements which implementation could not be realized with the "old" simulator, forced us to develop a new simulator from scratch. The main requirements are the following ones:

- Configurable through a humanly readable file.
- Parts of the core, like the *Multiply and Accumulate* (MAC) unit should be exchangeable at runtime. Figure 6.1 shows an example where

MAC_1 will be replaced with *MAC_2*.

- *Graphical User Interface* (GUI).

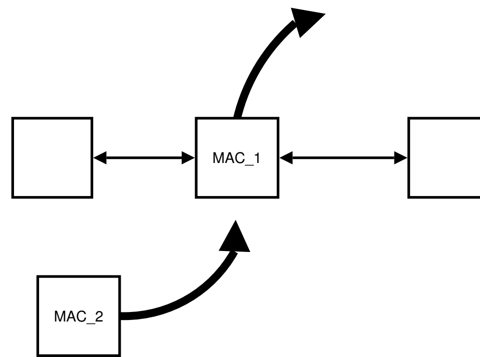

Figure 6.1: Replace MAC_1 unit with another MAC_2 unit

The replacement of parts of the core poses some problems have to be solved:

- *MAC_1* and *MAC_2* must have the same interface.
- It is preferable that a *MAC* unit do not heavily has dependencies to other parts which must be changed too.

Comparisons to current software paradigms (see section 2.3 on page 38) leaded to a realization of the simulator framework by using *Component Based Programming*. The principles of this concept provide a solution to all problems mentioned above.

In this work a new component model is constructed. It overcomes the nowadays existing component models like CORBA or COM by having a humanly readable configuration file. This file is in XML syntax and allows the following things to be defined:

- Definition of components that are given to a component model.

- Initialization of component states.
- Connection of components.

The developed component framework consists of the following parts:

- *Component:* Defines the rules for a component.
- *CompositeComponent:* Is a container holding components.
- *ComponentModel:* Is a CompositeComponent holding all components of a model. It is responsible for initialisation and correct connection of it's defined components.
- *Marshaller:* This is the concept of setting and retreiving attributes from a component for persistence storage or initialization.

As components are for composition, two composition techniques are involved:

- *Connection oriented composition* which is realized by using the *Hook* and *HookUp* concept further explained in [Lau05].
- *Container based composition* by using the above mentioned *CompositeComponent*.

The syntax of the configuration file is described in the master thesis of Gunther Laure [Lau05].

The framework is built with C++. To be able to construct a component in any programming language, the *Coco-Soap* library is developed. The communication is performed using the SOAP protocol which also enables distributed computation that can not be realized with the *Hook* concept alone.

To help the development of distributed components, a script has been constructed which generates stub and skeleton components for the client

and server. The communication is then automatically performed through the SOAP protocol.

References

[Ale01] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making Components Contract Aware. *Computer*, 32(7):38–45, 1999.

[Bro96] Kraig Brockschmidt. What OLE Is Really About. July 1996. Microsoft Corporation.

[CDK+02] Francisco Curbera, Matthew Dufler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Untraveling the web services web. March, April 2002.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.

[com] Component Definition. visited: 04-04-2005 `http://c2.com/cgi/wiki?ComponentDefinition`.

[com05] COM: Component Object Model Technologies. visited: 30-03-2005 `http://www.microsoft.com/com/default.mspx`, 2005.

[cora] CORBA. visited: 30-03-2005 `http://www.omg.org/`.

[corb] Jacorb the free java implementation of the omg's corba standard. visited: 09-04-2005 `http://www.jacorb.org/`.

[corc] Real-time corba with tao (the ace orb). visited: 09-04-2005 `http://www.cs.wustl.edu/~schmidt/TAO.html`.

[Cor03] Systinet Corp. Web services: A practical intoduction. 2003.

[Cox90] Brad J. Cox. Planning the Software Industrial Revolution. Software Technologies of the 1990's(November), 1990.

[dco05] DCOM Architecture. visited: 30-03-2005 `http://msdn.microsoft.com/library/default.asp?url=dcom`, 2005.

[Dij72] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972. Turing Award lecture.

[ejb05] J2EE Enterprise JavaBeans Technology. visited: 30-03-2005 `http://java.sun.com/products/ejb/`, 2005.

[FKH] Stefan Farfelder, Andreas Krall, and Nigel Horspool. Ultra Fast Cycle-Accurate Compiled Simulation of Inorder Pipelined Architectures.

[Fou] Free Software Foundation. GNU Nana: improved support for assertion checking and logging in GNU C/C++. visited: 16-04-2005 `http://www.gnu.org/software/nana/`.

[Gar98] Ken Garlington. Critique of "Put it in the contract: The lessons of Ariane". August 1998. Revised 16 March 1998; available at `http://www.flash.net/~kennieg/ariane.html` visited: 14-04-2005.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Pattern: Elements of Reusable Object-Oriented Software*. Addison-Wessley Professional Computing Series. Addison-Wessley, 1995. ISBN 0-201-63361-2.

[GJM91] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[Gri] Duncan Grisby. Free high performance orb. visited: 09-04-2005 `http://omniorb.sourceforge.net/`.

[GT00] Volker Gruhn and Andreas Thiel. *Komponentenmodelle*. Addison-Wesley, 2000.

[Han84] Wilfred J. Hansen. User engineering principles for interactive systems. In D. R. Barstow, H. E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, pages 217–231. McGraw-Hill, New York, 1984.

[Has03] Edmund Haselwanter. Aspects of component composition in distributed frameworks. Master's thesis, Graz University of Technology, October 2003.

[Inc] Tensilica Inc. Tensilica - The Configurable Processor Company. visited: 03-04-2005 `http://www.tensilica.com/`.

[JM97] Jean-Marc Jézéquel and Bertrand Meyer. Design by Contract: The Lessons of Ariane. *IEEE Computer*, 30(1):129–130, 1997.

[Kra98] R. Kramer. iContract - The Java(tm) Design by Contract(tm) Tool. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 295, Washington, DC, USA, 1998. IEEE Computer Society.

[Lau05] Gunther Laure. A Component Based Simulation Framework for Digital Signal Processors. Master's thesis, Graz University of Technology, May 2005.

[Laz05] Wolfgang Lazian. Communication layer for a decoupled model view controller in c++. March 2005.

[Lio96] Jacques-Luis Lions(Chairman). ARIANE 5 Flight 501 Failure. visited: 14-04-2005 `http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html`, July 1996. Report by the Inquiry Board.

[LLC] Simple Scalar LLC. Simple scalar llc to serve and project. visited: 03-04-2005 http://simplescalar.com/.

[McI69] M. D. McIlroy. "Mass produced" software components. In P. Naur and B. Randell, editors, *Software Engineering*, pages 138–155, Brussels, 1969. Scientific Affairs Division, NATO. Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968.

[Mey96] Bertrand Meyer. Building bug-free O-O software: An introduction to Design by Contract(TM). Technical report, ISE Inc., Santa Barbara (California), Octoeber 1996. available on http://www.eiffel.com/doc/manuals/technology/contract/ visited: 16-04-2005.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[Mey99] Bertrand Meyer. On to components. *Computer*, 32(1):139–140, 1999.

[Mey00] Bertrand Meyer. What to Compose. *Software Development Magazine*, March 2000. part of "Beyond Objects" also available on http://archive.eiffel.com/doc/manuals/technology/bmarticles/sd/compose.html visited: 15-04-2005.

[MRR+53] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of State Calculations by Fast Computing Machines. *The journal of chemical physics*, 21(6), June 1953.

[PHF+04] Christian Panis, Ulrich Hirnschrott, Stefan Farfeleder, Andreas Krall, Gunther Laure, Wolfgang Lazian, and Jari Nurmi. A Scalable Embedded DSP Core for SoC Applications. In *2004 International Symposium on System-on-Chip*, pages 85–88, New York, NY, USA, 2004. IEEE Catalog.

[PHL+04] Christian Panis, Ulrich Hirnschrott, Gunther Laure, Wolfgang Lazian, and Jari Nurmi. Dspxplore: design space exploration methodology for an embedded dsp core. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 876–883, New York, NY, USA, 2004. ACM Press.

[RJR] Paval Kulchenko Randy J. Ray. Soap::lite for perl. `http://www.soaplite.com/`.

[SGM02] Clements Szyperski, Dominik Gruntz, and Stephen Murer. *Component Software: Beyond Object-Oriented Programming.* COMPONENT SOFTWARE SERIES. ACM PRESS ADDISON-WESLEY, second Edition edition, 2002.

[SH04] Jean-Guy Schneider and Jun Han. Components - the Past, the Present, and the Future. June 2004. Ninth International Workshop on Computer-Oriented Programming.

[Smi] Merrit Roe Smith. Eli Whitney and the American System of Manufacturing.

[soa] Soap. `http://www.w3.org/TR/soap/`.

[Szy02a] Clemens Szyperski. Back to the Universe. *Software Development*, September 2002.

[Szy02b] Clemens Szyperski. Universe of Composition. *Software Development*, August 2002.

[tra] Core j2ee patterns - transfer object. visited 29-01-05 `http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html`.

[Val03] Egon Valentini. Development of a Framework for Contract-Based Testing of Software-Components. Master's thesis, Graz University of Technology, May 2003.

[vE] Robert A. van Engelen. gsoap: C/c++ web services and clients. visited 27-01-05 `http://www.cs.fsu.edu/~engelen/soap.html`.

[vEG] Robert A. van Engelen and Kyle A. Gallivan. The soap c/c++ stub and skeleton compiler sdk for deploying legacy applications in soap web services and peer-to-peer computing networks.

[Wey98] Elaine J. Weyuker. Testing Component-Based Software: A Cautionary Tale. *IEEE Software*, 15(5):54–59, 1998.

[xml] Xml schema. `http://www.w3.org/XML/Schema`.

Index

Design by Contract^{TM} versus Defensive Programming, 84

Acceptance testing, 74
ALCA, 16
Approaches to Reusability, 72
Automated Facade Class Generation, 128

Basic contracts, 82
Behavioral contracts, 82
Bidirectional Communication, 129
Bonobo, 48

Coco Component Framework, 87
Coco Soap Library, 118
Coco-Component, 88
Coco-CompositComponent, 101
Coco-Marshaller, 109
COM, 48
Compatibility, 69
Component Definition, 61
Component initialization, 96
Component Models, 46
Component Organisation, 112
Composition, 75
connection-oriented composition, 76
connections between components, 90
Constrained Genericity, 102
container-based composition, 76
Contracts for Components, 81
Contracts in software, 78
CORBA, 46
Correctness, 65

Definitions of Software Qualities, 64
Design by Contract, 77
DSL, 16
Dynamic profiling, 23

Ease of use, 68
Eclipse, 46
Efficiency, 68
Enterprise JavaBeans^{TM}, 52
Extendibility, 67
External Quality Factors, 65

Functional simulation, 22

Generic programming, 42
GIOP, 47
gSOAP, 122

Handling violations, 81
Howto construct a new component, 98
Human contracts, 77

Integration testing, 74
Internal Quality Factors, 69
Item testing, 73

JavaBeans, 51

Limitations of IDLs, 83

Meta programming, 40
Modularity, 70

object-oriented composition, 75
ORBit, 48

Performance simulation, 23
plugins, 45

Quality-of-service contracts, 82

Reusability, 71
Robustness, 66

Singleton, 113
SOAP, 55, 119
Software Quality, 64
Static profiling, 23
Synchronization contracts, 82
System testing, 74

Testing, 73

UDDI, 56

web services, 55
WSDL, 55