

CHALMERS



Single Source XML Data Structure Integration in Java Applications and Swing Graphical User Interfaces

- A coherent exposition of standards, XML, Java, and Swing

MAGNUS ARNEVALL

Master's Thesis

Computer Science and Engineering Program

CHALMERS UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering

Division of Computer Engineering

Göteborg 2006

All rights reserved. This publication is protected by law in accordance with the Swedish "Lagen om Upphovsrätt, 1960:729". No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior permission of the author.

© Magnus Arnevall, Göteborg/Munich 2006-2007.

Acknowledgement

I would like to thank my supervisor Dr. Cyprian Grassman at Infineon Technologies and Dr. Katarina Blom at Chalmers Technical University for their valuable feedback and good advice.

Göteborg, April 2007

Magnus Arnevall

Abstract

Software is virtual. That is the main reason behind its incredible success as well as for its shortcomings and failures. The problems associated with software, such as spreading uncontrollably or mutating beyond recognition, can be reduced by imposing standards.

At Infineon Technologies, the microchip design process was rendered more effective and coherent by developing a new software system. The industry standards XML and Java were used to build a single source XML data model with an API as a gateway to access the data. A Graphical User Interface (GUI) was implemented using Java's Swing package. The system with its three layers; XML data model, API, and GUI, was packaged together and could be run as a stand-alone application.

Before the new system was developed there were issues of inconsistencies, inefficiencies, and errors. With the new single source data, inconsistencies were eliminated. By using the GUI and automated generators, human errors could be avoided just as manpower and typing efforts were reduced.

The biggest challenge when designing the GUI was that the prerequisites were not known from the start. Another conclusion is that Swing is not an uncomplicated graphical toolkit. It is complex and difficult to fully master.

Sammanfattning

Mjukvara är virtuell. Det är huvudskälet både till dess enorma framgång såväl som dess brister och misslyckanden. Problemen associerade med mjukvara, såsom okontrollerad spridning eller mutering bortom igenkänning, kan reduceras med införandet av standarder.

På Infineon Technologies effektiviserades och sammanfördes design processen för mikrochip genom utvecklandet av ett nytt datasystem. Industristandarderna XML och Java användes för att bygga en singulärt lokaliserad XML datamodell med ett API som portgång gentemot datan. Ett grafiskt användargränssnitt (GUI) implementerades med Javas Swing paket. Systemet med dess tre lager; XML datamodell, API och GUI, paketerades samman och kunde köras som en fristående applikation.

Innan det nya systemet hade utvecklats förelåg vissa problem med inkonsistens, ineffektivitet och fel i datan. Med den nya singulärt lokaliserade datan kunde inkonsistenser elimineras. Genom att använda GUI:t och automatiserade generatorer kunde användarfel undvikas, samt att arbetskraft och manuellt skrivande reducerades.

Den största utmaningen med att designa GUI:t var att kraven inte var kända från början. Ytterligare en slutsats är att Swing inte är en okomplicerad uppsättning grafiska verktyg. Den är komplex och svårt att bemästra fullt ut.

Keywords

Java, Swing, GUI, XML, graphical user interface, JAXB, standards, single source

Preface

In October 2005, a project was initiated at Infineon Technologies in Munich. The goal of the project was to derive a company wide solution for maintenance, communication, and storage of design data, related to microchip manufacturing. The developed solution is meant to be used in real production.

The project was hosted at the System Design Methodology Department at Infineon AG in Munich. It was staffed with the headcount of two and a half employees, two Master Thesis students, and one intern student. This thesis is the result of six months' work of one of the two Master Thesis students, between September 2006 and February 2007.

The company project is scheduled to be completed in the fall of 2007.

Table of Contents

1	Introduction	4
1.1	Background	4
1.1.1	The Chip Construction Process Simplified	4
1.2	Problem and Solution	5
1.2.1	Pre-existing Solutions	6
1.2.2	Single Source Solution	6
1.3	Purpose of this Report	8
1.4	Disposition	9
1.4.1	Introduction	9
1.4.2	Theory	9
1.4.3	Method	9
1.4.4	Design Choices	9
1.4.5	XML Data Model and Java Custom API	9
1.4.6	Designing the GUI	9
1.4.7	Results and Conclusions	10
1.5	Definitions	10
1.6	Delimitations	10
1.6.1	Implementation and Coding	10
1.6.2	Scientific Method	11
1.6.3	Code Review	11
1.6.4	Additional Features not Covered	11
2	Theory	12
2.1	Software and Standards	12
2.2	XML	13
2.2.1	XML – Extensible Markup Language	13
2.2.2	XSD – XML Schema Definition	14
2.2.3	Namespaces	16
2.2.4	SPIRIT	17
2.3	Java	17
2.3.1	The Java Technology	18
2.3.2	The Java Programming Language	18
2.3.3	Useful Features of Java	18
2.3.4	The Java API	21
2.3.5	Serialization	22
2.4	Java and XML	22
2.4.1	JAXP, SAX, DOM	23
2.4.2	JDOM – Java Document Object Model	24
2.4.3	JAXB	24
2.5	Swing – Building GUIs in Java	25
2.5.1	Introduction to Swing, AWT, and JFC	25
2.5.2	Start Building from the Root - JFrame	26
2.5.3	Laying out the Components – JPanel	27
2.5.4	Making Everything Fit – JScrollBar and JScrollPane	27
2.5.5	Nesting of Components	28
2.5.6	Dynamic Tables – JTable	29
2.5.7	Dynamic Object Trees – JTree	29
2.5.8	Building Graphical Tools – Custom Painting of JPanel	31

3 Method	32
3.1. Methodology in a Software Project.....	32
3.2 Gathering Background Information	32
3.2.1 Secondary Data	32
3.2.2 Primary Data	32
3.2.3 GUI Tracer Bullet.....	33
3.3 Code Review	33
3.4 Development Tools	34
4 Design Choices.....	35
4.1 Conforming to Standards	35
4.1.1 Why Standards are Important.....	35
4.1.2 Problems with standards.....	35
4.1.3 Conclusion.....	36
4.2 XML Data Model	37
4.2.1 Using XML	37
4.2.2 The SPIRIT Standard	37
4.2.3 Elements vs. Attributes	38
4.3 Java.....	38
4.3.1 JAXB.....	38
4.3.2 On-demand import of classes.....	38
5 XML Data Model and Custom Java API	40
5.1 XML Data Model	40
5.2 Results of the Data Model Implementation.....	40
5.2.1 Object Instantiation and Detachment	41
5.2.2 Arithmetic String Expressions	41
5.2.3 Mark-up Language Documentation	42
5.2.4 Many-to-Many Relations	42
5.3 Java Custom API.....	43
5.3.1 Purpose of the Custom API.....	43
5.3.2 Extending JAXB	43
6 Designing the GUI	45
6.1 Design Challenges and User Requirements	45
6.1.1 End-user Requirements	45
6.1.2. Structural Requirements.....	45
6.2 Setting up the layout.....	46
6.3 Dynamic Object Trees – JTree.....	47
6.4 Component Editor	49
6.5 General Input Fields and HTML Documentation Editor	49
6.6 Dynamic Editing Tables – JTable	50
6.7 Building Customized Graphical Tools - Extending JPanel.....	51
6.7.1 Bitfield Overview Tool	51
6.7.2 System Composition Tool.....	53
6.8 Input Groups.....	54
6.9 Multiple Layouts	54
6.10 Class Separation and Scalability	56
6.10.1 Input Group Return Values	56
6.11 Propagating Size Changes.....	58
7 Conclusions	60
7.1 Requirement Uncertainties.....	60
7.2 Swing Complexity.....	60

7.3 Consistency, Efficiency, and Correctness	61
7.4. The Nature of Software	61
References	63
Printed Books	63
Articles On-line	63
Other Internet Sources.....	63
Appendix A – XML Data Model	1
Appendix B – HTML Code Sample.....	2
B.1 System Specification Overview.....	2
B.2 Detailed System Specification.....	2

1 Introduction

This thesis is based on the project work that was done from the informatics aspect of chip design and Intellectual Property (IP). Concerning what was implemented in terms of software, the characteristics and properties of the microchip components were stored in an XML data model. Above that, a custom Application Programming Interface (API) was built to serve as the gateway towards the data structure. In the topmost layer, a Graphical User Interface (GUI) was developed that utilized the custom API for creation and manipulation of chip components in the data structure.

All parts of the software system are discussed in this report, but since all programming done by the author of this report was that of the graphical user interface, the GUI is the only part that is detailed in terms of code implementation.

1.1 Background

Infineon Technologies AG is a chip manufacturing company. The construction of a chip, from its conception to the final end product, is a complicated, time consuming, and costly business. Before an actual, physical chip emerges, the development is done virtually by means of software. Currently, the people involved in the different stages of the chip construction use different data structures, different programming languages, and the data for the same chip component is stored in several different locations.¹

From Infineon Technologies' part, there was a wish to solidify and unify the information flow related to the microchip construction process.

1.1.1 The Chip Construction Process Simplified

To construct and produce a modern-day microchip, hundreds of people might be involved and the cost of the first physical version of the chip can easily amount to millions of Euros. It is an extremely complicated process. It is not the purpose of this thesis to dig into the details of that process, but in order to better understand the problematic issues that brought forth the work with which this thesis is involved, a simplified explanation could be useful.¹

There are different approaches, so called design flows, regarding how to construct a chip, and one of them is called semi-custom design flow. Simply put, there are three different types of engineers involved in the construction process; the Concept Engineer, the Design Engineer, and the Test Engineer. The Concept Engineer is the person who writes the specification of the chip. The specification delineates the concept and so describes the desired behavior and structure of the chip and its components. The Design Engineer takes the specification and translates it into a Hardware Description Language (HDL). The HDL can then be compiled into a netlist, which is a semantic description of electronic connections between cells. Then the Design Engineer uses the netlist and performs Place and Rout, which decides how and where physical connections will be drawn on the chip. Finally the Test Engineer performs validation of the functionality of the system and the components. The chip design process is iterative and in each iteration, the Test Engineer runs tests on simulations of the chip on different levels.¹

¹ Information based on informal discussions with expert people at Infineon Technologies AG.

1.2 Problem and Solution

Throughout the product development process, partially redundant data are generated in different formats, stored in different locations, and indeterminately updated locally. Primarily, this induces a risk of *inconsistencies*. Secondly, it involves some *inefficiencies* since the manual conversions and modifications involved are time consuming and error prone.

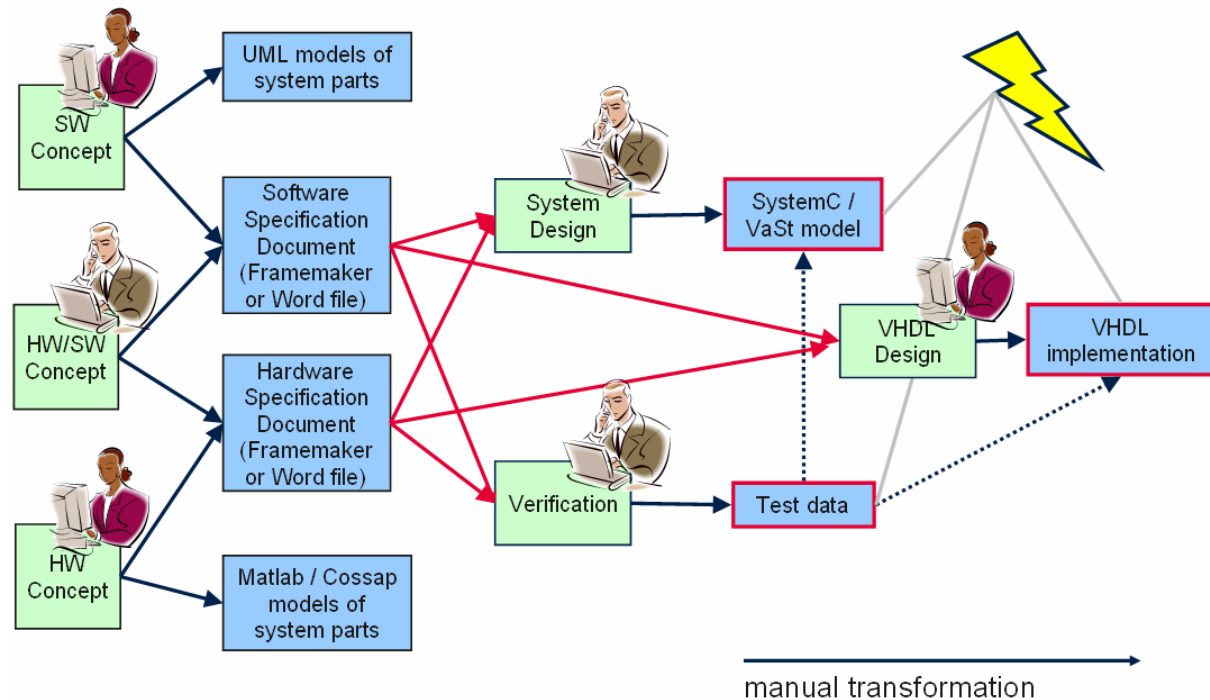


Figure 1

Practically all the information in the design process is informal, i.e. written by hand or containing pictures, so that it cannot be automatically read or generated. The Concept Engineer writes a specification that the Design Engineer and the Test Engineer use to implement and test the system. This specification also constitutes the base of the system documentation.

Since the specification is written as informal text three issues. First, errors can arise due to *misinterpretations* by the human reader. It could be a simple reading error or an actual misunderstanding of the intent. Often, engineers use completely different ways to describe some technique or solution even if they address the same thing. The misinterpretations of the text lead to inconsistencies between the specification and the final design. Secondly, the information contained in the specification needs to be *re-typed several times*. This is both time consuming and a source of new errors. Thirdly, during the design phase, a design parameter sometimes has to be changed. If the Design Engineer then modifies the HDL files, that change needs to be propagated to all other documents involved. Even the initial specification has to be updated, and these *distributed updates must be done manually*. Headers for firmware, HDL files, test patterns for verifications, and documentation, all contain basically the same information and they must all be updated to remain coherent in the event of some change.

The most problematic consequence of the misinterpretations, the re-typing, and the manual updates, is concerning the verification. It is time consuming and difficult to ensure that the

specified product, the designed product, and the produced product are actually the same. Even though it is outside of the scope of this thesis, it is also worth mentioning that the problem does not stop at the design phase, but continues on all the way to the production and maintenance phases. The problem is the same throughout the whole life cycle of the chip.

1.2.1 Pre-existing Solutions

Partial solutions to the previously stated problems were paradoxically also a part of the problem. Several similar kinds of solutions were already developed and used within the company. They were focused on specific parts of the problem and not fully compatible with each other. Even taken as a whole, they only solve the problem partially because they merely addressed the generation of design data output from structured data, not primarily the exchange of the data. Because of that, these partial solutions could not be used to realize a single solution to the complete problem.

There were also a few commercial solutions available on the market, but they did not solve the entire problem either. The functionality and structure necessary to address all issues involved were not provided. Besides economical disadvantages of license fees, the pre-existing commercial solutions were deemed unsuitable due to the incomplete coverage of the problem domain.

1.2.2 Single Source Solution

The solution to the entire problem sphere was to build a completely new system in-house. The information should be stored in a single source design database, so that it is no longer redundant, and data should be formal where possible to enable automated generations. By addressing the previously inherent problems, the new solution should lead to efficiency, consistency, and flexibility. In Figure 2 the workflow together with the single source data base is illustrated.

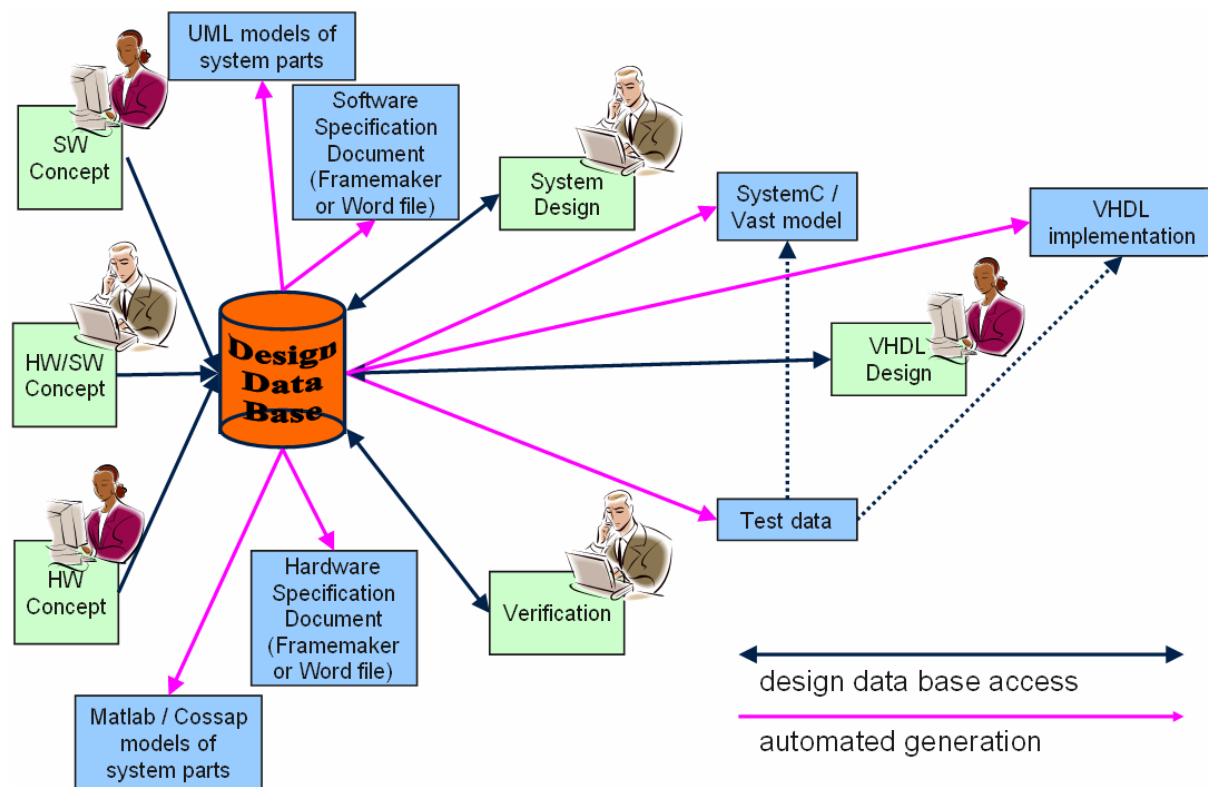


Figure 2

In order to enable the users to efficiently and consistently communicate with the database, a complete software system should be developed. The solution should consist of the single source database, a custom built API which should function as a layer around the database and which applications can utilize for communicating with the database, and in the topmost layer there should be a graphical user interface to provide an easy and efficient way for the involved engineers to carry out their work. The system should also run as a stand-alone application. Figure 3 shows the different parts and layers of the system. (The unlabeled blocks in the figure represent generators and other external applications communicating with the Custom API.)

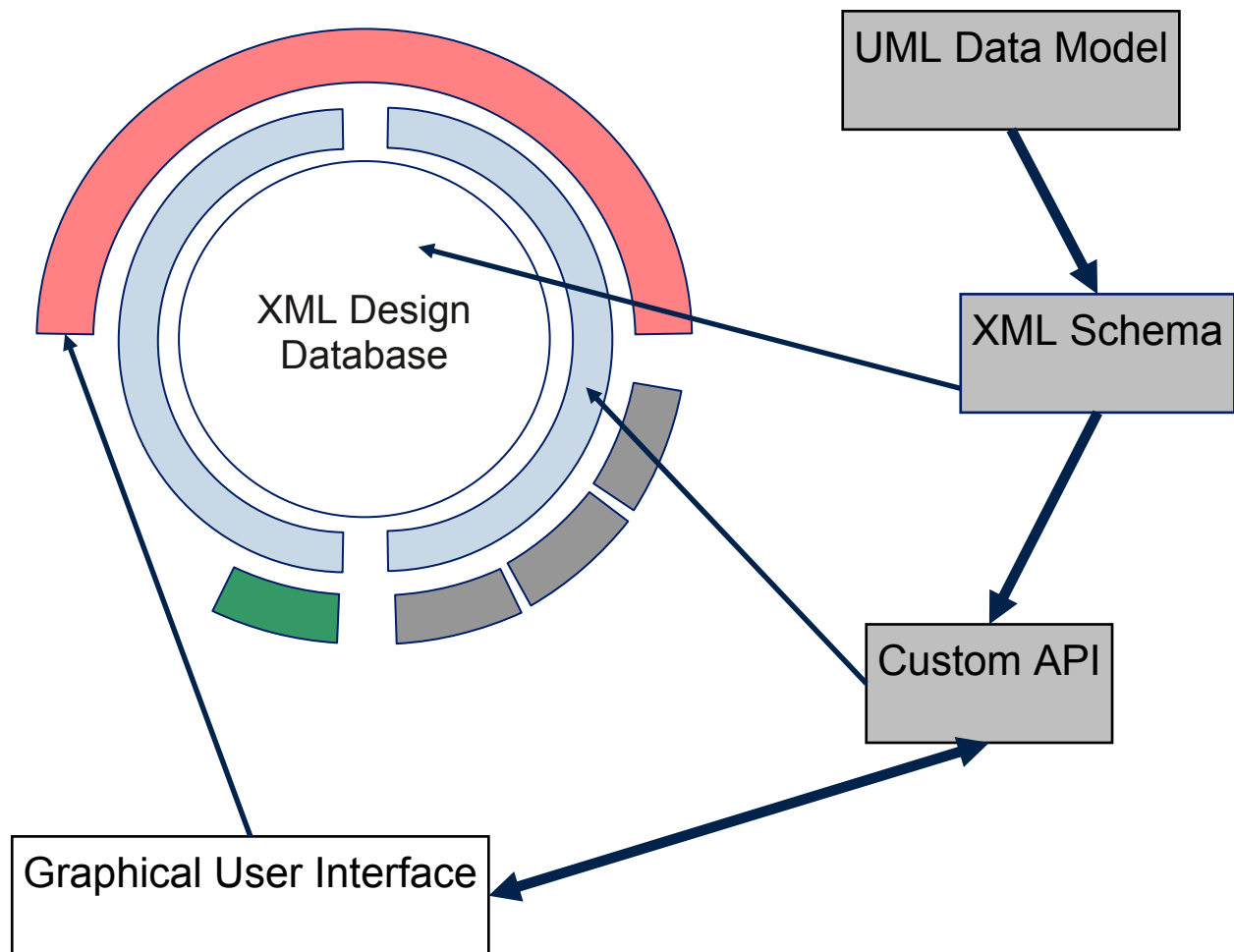


Figure 3

1.3 Purpose of this Report

One thing is important to point out concerning this thesis; its main purpose is not to outline the finer details of which classes and methods were used to realize a software product. That information is already contained in the source code which incidentally is proprietary and not publicly available. Numerous sources on the web provide abundant support for those who wish to learn more about programming techniques or how to solve common problems. Instead the perspective is from a higher level in terms of which techniques and languages were ultimately chosen to solve the task, and more importantly, *why* these particular techniques were chosen. By reasoning about software and standards from a wider perspective this thesis can hopefully provide meaningful substance to the decision making phases of other software projects, without going into coding details which may or may not be applicable in other contexts. The experiences and conclusions drawn from the project work are expressed in general terms and not tied to specific or perhaps even unique details of this particular project.

The thesis aims to present relevant and interesting results that have emerged from the work throughout the project. But for these results to have a context, a deeper theoretical background beyond how certain tasks were solved is needed. A rather extensive literature study and theoretic overview related to Java and XML thus precede the results. A main goal of the thesis and the purpose of the theoretic material is to provide something more than just a static reference to used technologies. By discussing up-to-date and relevant theory more thoroughly

throughout the report the ambition is that a reader interested in standards, XML, Java, and Swing might have a chance to learn something about these subjects.

1.4 Disposition

1.4.1 Introduction

Chapter 1 introduces the reader to the problem statement and the proposed solution. It contains background information about the project and clarifies the aim of the report itself. Delimitations and some important definitions are stated.

1.4.2 Theory

After the introduction chapter a theoretic overview based on a literature study is given in chapter 2. The theory is presented in a bottom up fashion. First, underlying reasons behind chosen technologies are presented. Then, theory relevant for the XML data model, the Custom API, and finally the GUI follows. The reason is that the purpose of the application and how it was built will probably be better understood if its layers are presented in a bottom-up fashion.

The reader who is already well knowledgeable in XML or Java might be able to skip the introductory sections of these technologies. However, simple basics or coding details are not presented and the chapter focuses on interesting features that were used within the system, so even readers familiar with these technologies might find the chapter useful.

1.4.3 Method

In the method chapter the scientific methods of the project and the thesis are explained. The chapter outlines methodologies in software projects, a code review that was held in the project, and the technical tools that were used in the software development process.

1.4.4 Design Choices

In chapter 4, important design choices that were made in the project are detailed. It was decided that the project should try to conform to standards, and that XML and Java should be used for the implementation. From this chapter and onwards, the text is based on discussions and reasoning.

1.4.5 XML Data Model and Java Custom API

Chapter 5 summarizes the parts that the XML Data Model and the Java Custom API played in the application as a whole. Since this thesis does not cover the actual coding implementation of these parts of the system, the perspective in chapter 5 is from a high level. Instead of coding details, the results and effects of the data model and Custom API are discussed.

Although not going into implementation details, this chapter is still quite technical. So for the reader who is only interested in Swing and the GUI design, this chapter does not have to be fully understood.

1.4.6 Designing the GUI

As a consequence of the design choice to develop the application in Java, using the Swing Graphical Toolkit to design the GUI was directly implied. In chapter 6 the GUI as a whole is

detailed as well as specific features or functionalities. Some initial challenges of specifying the main layout are discussed. At the end of the chapter some more technical sections follow focused on code structure and layout complications.

1.4.7 Results and Conclusions

In the final chapter the results and conclusions from the project as a whole are summarized. The chapter begins with conclusions related to the graphical user interface and then continues on with conclusion drawn from the Custom API and XML data model. Finally, software in general and the use of standards are discussed.

1.5 Definitions

<i>Data Model:</i>	The underlying data structure in which information about chip components are stored. The data model is XML based and was developed within the project group.
<i>Java API:</i>	The publicly available Application Programming Interface provided by Sun Microsystems which describes the features of the Java language. A requirement in the project and for running the end-user application was to have Java version 5 installed (as JRE).
<i>Custom API:</i>	The API that was developed by the project group itself and explains how to interact with the software system that was built. This custom API was provided in the C++ and Java languages, but this report is only concerned with the Java version. It will henceforth be referred to as the Custom API to distinguish it from the official Java API.
<i>chip design:</i>	The process of putting microchip components such as memories or registers together to form a Solution on Chip (SoC).
<i>SoC:</i>	Solution on Chip – A more or less sophisticated solution for some application domain encapsulated on a microchip.
<i>IP Block:</i>	Intellectual Property Blocks - The SoCs that we refer to here are IP Blocks or collections of IP Blocks put together into a larger system called IP System. In the GUI the name IP Component was also used.
<i>component:</i>	Generally refers to a chip component, such as a register or memory. It should not be confused with the Component class in the Java Swing package.
<i>GUI element:</i>	A Swing GUI object such as a radio button, text field, drop-down list, or check box.

1.6 Delimitations

1.6.1 Implementation and Coding

The conceptual design and architecture of the whole system was conceived as a collaborative effort of the whole project group. Decisions were taken that specified the sought solution mainly by means of meetings and informal discussions. However, the actual implementation into real code could, due to time limitations, not be made collaborative. As a result the implementation and design of the client application and GUI was made by the author of this thesis Magnus Arnevall, the implementation of the Java/JAXB Custom API was made by the intern student in the project group, and the part of the data model for informal information and generation of documentation was made by the other Master Thesis student.

The documentation generation and informal data aspect of the project will not be discussed at all in this thesis. The use of JAXB, on the other hand, will be discussed in the following chapters since it constitutes an integral part of the logical link between standards, XML, and Java.

1.6.2 Scientific Method

In section 3.2, the kind of primary research data that was a part of the scientific process is described. The actual data itself, in terms of facts and figures, is not included in this report as further explained under that section. The decision making process and exactly why one standard or technical solution was chosen by the company, based on this primary data, is also beyond the scope of this thesis.

1.6.3 Code Review

The feedback from the code review mentioned in section 3.3 was in the form of verbal coding details and company notes taken by the supervisor. Neither is of relevance for reading this report and that information is not included. The actual purpose and outcome of the code review, on the other hand, are mentioned in chapter 3.

1.6.4 Additional Features not Covered

Although parts of the system implementation, the following specific GUI features are not covered:

- Events
- Menus, short-keys, general mouse input handling
- Splash Screens (The splash screen of the application was implemented using the techniques available up to Java 5. As of the recently released Java 6 a special splash screen functionality is already built in, simplifying the addition of splash screens significantly.)
- Simple Swing GUI elements (buttons, labels, drop-down lists, etc.)

2 Theory

2.1 Software and Standards

According to Burd (2002) there are three main reasons why software does not work:

1. There is always more than one way to express a solution to a programming problem.
2. Software is virtual, not physical.
3. Without rigorous standards, software is not useful in more than one context.

When writing a computer program, the solution is not necessarily an unambiguous sequence of imperative statements. In an object oriented language, for example, you can often construct different objects in arbitrary order, before letting them work together to produce some result. In the *Code Conventions for the Java™ Programming Language* (Code Conventions for the Java Programming Language 1999) numerous standards concerning how to write your code are proposed in order to decrease the amount of variations of expressing the same thing. One recommendation is to declare all your variables at the top of a method even though from the compilers point of view they could just as well be declared right before they are first used. In response to software being virtual, complex, and generally unreliable, one approach (with arguably mixed success) is to have a standardized discipline for analyzing, designing, coding, and maintaining a software project (Burd 2002).

Another problem with software is that because it is virtual, it can quickly become extremely complex. Since you can create, rebuild, reuse and destroy software extremely fast, the growth and mutation of software is unmatched by most of mankind's other creations. Burd (2002) uses an interesting example with bridges in which he points out that if we could build bridges as quickly as we build computer programs, then we would have billions of miles of bridges. The average hobbyist could build a bridge in minutes – and most of the world's bridges would be completely unusable.

The internet and its functionality is what it is today due to the first users adopting HTML. They did so because they expected benefits from being early adopters to this new standard (Antoniou & van Harmelen 2004). Others followed as more and better web tools became available and soon HTML was a universally excepted standard. In 2004 we could see the same early adoption of XML (Antoniou & van Harmelen 2004). Because the software was standardized it could be used across platforms all over the world. When everybody uses the same standard format for presenting data, programs can communicate with each other and the data has become portable. Experience has shown that industry-wide standards make it easier to work with software and informatics (Burd 2002).

Standards, however useful and necessary, have one major challenge. They need to be adopted to serve their purpose. According to Antoniou & Harmelen (2004) the greatest challenge concerning the XML and RDF standards for the semantic web, is not scientific but rather one of technology adoption.

Another application of the use standards are coding conventions. Coding conventions, however, are never a complete set of truths but can vary from one organization to another, and in different situations slight variations from a standard can be motivated. According to Oskarsson (1994), many of the rules are arbitrary conventions and it is appropriate that every

organization chooses its own set of rules. The purpose of coding conventions is to create easily readable code (*Code Conventions for the Java Programming Language* 1999).

2.2 XML

A complete reference or introduction to XML will not be given in this section. Numerous sources already exist on the internet for that purpose (see the end of section 2.2.1). Instead an overview of XML is presented along with two aspects of XML that were relevant for this project; XSD and SPIRIT.

2.2.1 XML – Extensible Markup Language

In its short lifetime, XML has become the international standard for representing structured, self-describing data (Burd 2002). XML has formats for describing healthcare, financial data, arts, human resources, and much more (*XML Applications and Initiatives* 2005). The XML standard can encapsulate almost any kind of data in a way that is flexible, extensible, and easy to maintain.

XML, like HTML, was derived from SGML (Standard Generalized Markup Language), an international device- and system-independent standard (ISO 8879) (Antoniou & van Harmelen 2004). Languages conforming to SGML are called SGML applications and represent both human- and machine-readable code. XML was developed in part to address the weakness of HTML, and much of the drive for XML has been the desire to share data (Burd 2002). Standards for representing both human- and machine-readable information are important because they enable effective communication, and therefore support technological progress and business collaboration (Antoniou & van Harmelen 2004).

An XML document consists of a prolog, a number of elements, and an optional epilog. The prolog is an XML declaration and an optional reference to external structuring documents (Antoniou & van Harmelen 2004). Here is an example:

```
<?xml version="1.0" encoding="UTF-16" standalone="no" ?>
<!DOCTYPE component SYSTEM "component.dtd">
```

XML elements represent the things that the XML document describes, for example microchips, registers, memories, and buses (Antoniou & van Harmelen 2004). We can think of them as a kind of objects. An element consists of an opening tag, its contents, and a closing tag. The contents can be other elements in a nested fashion or simple text. For example,

```
<register>My 32-bit Register</register>
```

XML attributes can be used to assign certain attributes to an element. If an element describes a register, an attribute could be the bit width of that particular register. An attribute is a name-value pair inside the opening tag of an element (Antoniou & van Harmelen 2004):

```
<register bitWidth="32">My Register</register>
```

The same information could also be represented as:

```
<register bitWidth="32" name="My Register"></register>
```

Or as:

```
<register>
  <bitWidth>32</bitWidth>
  <name>My Register</name>
</register>
```

When to use elements and when to use attributes to represent something is often a matter of taste (Antoniou & van Harmelen 2004). It should be noted however that attributes cannot be nested.

XML is strictly hierarchical, which means that the data model is always a tree, i.e. an acyclic graph (Burd 2002). This imposes a problem if the data model you want to express with XML happens to be relational so that cycles exist in its graph representation. Expressing overlapping (non-hierarchical) data structures in XML requires extra effort (*XML – Wikipedia* 2006). The problem arises when, starting from one element, you want to reference another element that does not exist directly above or below in the hierarchy, since that would create a graph cycle and violate the tree structure. There is simply no straight-forward way of doing this in XML. Mapping XML to the relational or object oriented paradigms is often cumbersome, although the reverse is typically easy (*XML – Wikipedia* 2006).

(To learn more or to get an introduction to XML, see the tutorials at *Website 1* in the reference list.)

2.2.2 XSD – XML Schema Definition

If an XML document conforms to the syntactic rules of XML, it is said to be *well-formed* (Antoniou & van Harmelen 2004). This syntactic correctness says nothing however, about the structure of the data in the document. Syntactically one could correctly define that fruit is a kind of apple or a subgroup of apple, even though it is well known that it should be the opposite. Another absurd example would be to declare that dog is a kind of fruit. It is therefore reasonable to restrict the structure or hierarchy of an XML document to ensure that documents conform to what they are actually trying to describe. That way we can make sure that, for example, dog is a sub-element of animal, just as apple is a sub-element of fruit in our XML representation.

There are two ways to define the structure of XML documents: DTD (Document Type Definition) and XSD (XML Schema Definition). DTD is older and more restricted, whereas XSD is more powerful and offers extended possibilities, mainly for the definition of data types (Antoniou & van Harmelen 2004). We will not discuss DTD much further, but instead focus the attention on the XML Schema (XSD) which offers a significantly richer language for defining the structure of XML documents (Antoniou & van Harmelen 2004).

One characteristic of the XML Schema is that its syntax is based on XML itself. Besides making the schema much easier to read, this design decision also, and more importantly, allows significant reuse of technology. Contrary to DTD, it is with XSD not necessary to write separate parsers, editors, pretty printers, etc., to obtain a separate syntax. An even more important improvement is the possibility of reusing and refining schemas. XSD permits definition of new types by extending or restricting already existing schemas. Together with an XML-based syntax, this means that schemas can be built from other schemas, which reduces

workload and saves time. Finally, XML Schema provides a sophisticated set of data types that can be used in XML documents. (Antoniou & van Harmelen 2004)

An XML Schema is an element with an opening tag that can look something like this:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
version="1.0" >
```

Element Types

An element type is a tag that can have a number of optional attributes, such as types, or cardinality constraints. (The slash sign at the end is an abbreviated form of an empty tag in XML.)

```
<element name="emailHeader" minOccurs="1" maxOccurs="1" />
```

Attribute Types

An attribute type is a tag that can have a number of optional attributes, such as types, existence, or a default value.

```
<attribute name="id" use="required" />
```

Data Types

A key weakness of DTDs is its very restricted selection of data types. XML Schema on the other hand provides powerful capabilities for defining data types. Following is a list of a few types: (Antoniou & van Harmelen 2004)

- Numerical data types, including Integer, Short Byte, Long, Float, Decimal
- String data types, including String, ID, IDREF, CDATA, Language
- Date and time data types, including Time, Date, Month, Year

In addition to the above list, there are also *user-defined data types*. They can be either *simple data types*, which cannot use elements or attributes, or *complex data types*, which can use both elements and attributes. Complex types are defined from already existing data types by defining some attributes together with one of the following tags: (Antoniou & van Harmelen 2004)

- *sequence*, a sequence of existing data type elements in a predefined order
- *all*, a collection of elements that must appear but in arbitrary order
- *choice*, a collection of elements, from which exactly one will be chosen

Going back to the above examples for email header elements and id attributes, but with types added we get a complete example (for simplicity the body of the email is just a string):

```
<element name="email" type="emailType" />

<complexType name="emailType">
  <sequence>
    <element name="head" type="headType">
      <element name="body" type="string">
    </sequence>
```

```

</complexType>

<complexType name="headType">
  <sequence>
    <element name="from" type="mailAddress">
      <element name="to" type="mailAddress" minOccurs="1"
maxOccurs="unbounded">
        <element name="subject" type="string">
      </sequence>
    </complexType>

    <complexType name="mailAddress">
      <attribute name="name" type="string" use="optional" />
      <attribute name="address" type="string" use="required" />
    </complexType>

```

Existing data types may also be extended or restricted with the *extension* or *restriction* tags and using the *base* attribute to refer to what type is being extended or restricted:

```

<complexType name="...">
  <extension base="emailType">
    ...
  </extension>
</complexType>

```

For further information about XML Schema, see the end of section 2.2.1.

2.2.3 Namespaces

In Java and most other programming languages, you can declare variables with the same name in different methods without them conflicting with each other. It is possible because the variables are declared locally inside the methods, and hence the scopes of the variables do not overlap. In XML, as you build large documents, or especially as you combine existing documents, it might also be necessary to differentiate the environment or scope in which a name exists, in order to avoid name clashes.

One of the main advantages of XML as a universal markup language is that information from various sources can be accessed (Antoniou & van Harmelen 2004). That means that an XML document may use more than one schema. But since each structuring document or schema was developed independently, name clashes appear inevitably (Antoniou & van Harmelen 2004). The solution is to use *namespaces*, and they qualify XML elements and attributes (Burd 2002). Technically, disambiguation is simply achieved by using a different prefix for each schema, and separating the prefix from the local name by a colon (Antoniou & van Harmelen 2004). For example, with namespaces you can have two different elements with the same name “ip”, using the namespace colon-notation:

```

<myChipNamespace:ip />
<myNetworkingNamespace:ip />

```

In the `myChipNamespace` above the name “ip” actually refers to Intellectual Property as a chip component, and in the `myNetworkingNamespace` “ip” means Internet Protocol.

Namespaces were not in the original XML specification, and lots of code was written before namespaces existed. To keep this old non-namespace code from crashing when namespaces were introduced, the whole namespace routine had to be added in a way that would look grammatically correct to old programs. Because of that, a so called namespace-unaware parser perceives `myChipNamespace:ip` to be just one long name that happens to contain the colon character. (Burd 2002)

2.2.4 SPIRIT

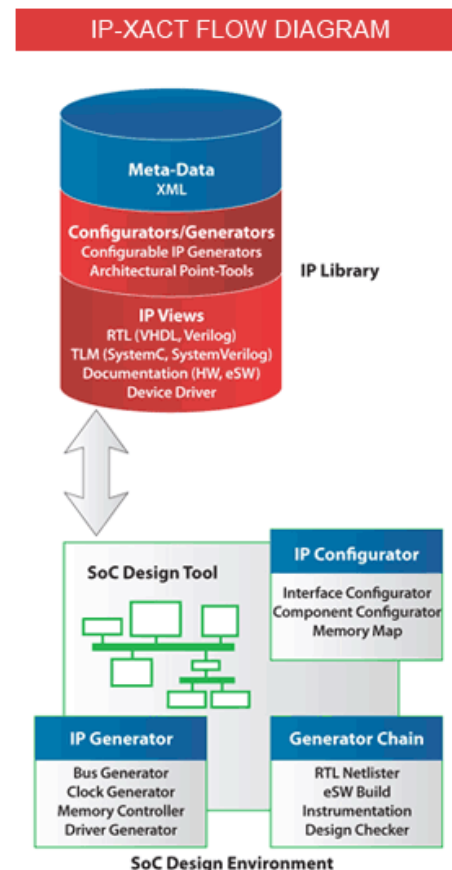
The SPIRIT Consortium is an independent non-profit organization which was incorporated in July 2006. The consortium is an international standardization organization whose targeted engagements involve debugging, hardware constraints, documentation, and register-description formats. SPIRIT currently has 54 member companies, of which Infineon Technologies is a contributing member. For the sake of this report, the consortium's official specifications for IP (Intellectual Property) meta-data, called IP-XACT, was the primary interest. These specifications have now been released to the public and are currently under review of IEEE Working Group P1685 (*IEEE P1685* 2006). IP-XACT is a standard to describe chip components or SoC (Solution on Chip) by means of XML and XSD, along with an API to provide tool access to the schema. (*The SPIRIT Consortium* 2006)

The original vision of the SPIRIT consortium is to: "Achieve an open standard for a development framework upon which an SoC development flow, from components to chip, can be built allowing distribution and use of IP from varied sources as well as the free choice of tools used in the SoC development." (*The SPIRIT Consortium* 2006)

IP-XACT is the name of the meta-data specifications from the consortium. The goal is to build on the existing XML (W3C) standard and to synchronize with Eclipse, OSCI, Si2, VSIA, etc. SPIRIT also wants to standardize one way to describe IP and one API for generators, to enable configuration and integration of multi-sourced IP. The SPIRIT Schema includes a component schema for cores, peripherals, buses and components, and a design schema for systems, component instancing and connectivity. Other so-called deliverables from SPIRIT are bus definitions and generator interfaces. (*The SPIRIT Consortium* 2006)

2.3 Java

An explanation of how to practically write, compile and run Java code will not be given in this report. Instead this chapter will cover the theory of some interesting, and for this project relevant, aspects of the Java programming language and the Java API.



2.3.1 The Java Technology

Java technology is both a programming language and a *platform*. A platform is the hardware or software environment in which a program runs. Most platforms can be described as a combination of an operating system and underlying hardware. The Java platform contains the Java *Virtual Machine* and the Java *Application Programming Interface* (API) and differs from most other platforms in that it is a software-only platform. The Java platform runs on top of other hardware-based platforms. (*The Java™ Tutorials*:1 2006)

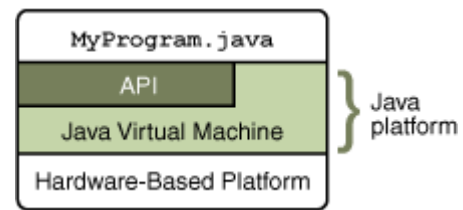


Figure 4

As long as the virtual machine is installed on a hardware-based platform (see Figure 4), the same Java program can be run on many different computer architectures and operating systems such as MS Windows, Solaris OS, Linux, or Mac OS. The Java platform is hence a platform independent environment. As such, the Java platform can be a bit slower than native code. However, advances in compiler and virtual machine technologies are bringing performance close to that of native code without some of the problems related to portability. (*The Java™ Tutorials*:1 2006) Java code is reusable and the platform independence is one reason for that (Burd 2002).

2.3.2 The Java Programming Language

*You know you've achieved perfection in design,
Not when you have nothing more to add,
But when you have nothing more to take away.*
- Antoine de Saint Exupery.

Seamless integration of parts from many sources to build large, reliable software systems, has been the goal of computing for the past several decades (Burd 2002). The Java™ programming language can assist in reaching these goals since it is designed for application development in the context of heterogeneous, network-wide distributed environments. (Gosling & McGilton 1996)

The Java programming language began as part of a research project to develop advanced software for a wide variety of network devices and embedded systems. The goal was to develop a small, reliable, portable, distributed, real-time operating platform. When the project started, C++ was the generally preferred programming language. According to Gosling & McGilton (1996), the difficulties that were encountered over time with C++ grew to a point where the problems could best be addressed by creating an entirely new language platform. The design and architecture decisions, of what was to become the Java language, drew from a variety of other languages such as Eiffel, SmallTalk, Objective C, and Cedar/Mesa. The result today is a language platform that has proven well suited for developing secure, distributed, network-based end-user applications, in environments such as network-embedded devices, the World-Wide Web and the desktop. (Gosling & McGilton 1996)

2.3.3 Useful Features of Java

Java is object oriented but at the same time it is simple to implement. In comparison to some other languages the development cycle is much faster because Java technology is interpreted. It means that the compile-link-load cycle is obsolete and only a compile and run is needed.

Besides being easy to work with, Java also means that the applications are portable across multiple platforms. There is no need to port Java applications since they can run without modification on multiple operating systems and hardware architectures. Because the Java runtime environment manages the memory automatically, coding errors related to memory allocation are avoided, which makes the applications more robust. For interactive graphical applications, Java provides high performance because multiple concurrent threads of activity in applications are supported by the multithreading built into the Java programming language and the runtime platform. Java applications are also adaptable to changing environments since you can dynamically download code modules from anywhere on the network. End users of Java applications always get a certain amount of security, even though they are downloading code from all over the Internet, because the Java runtime environment has built-in protection against viruses and tempering. (Gosling & McGilton 1996)

To summarize, the Java programming language is a high-level language characterized by being: (*The Java™ Tutorials*:1 2006)

- Simple
- Object oriented
- Distributed
- Multithreaded
- Dynamic
- Architecture neutral
- Portable
- High performance
- Robust
- Secure

Simple

The fundamental concepts of Java technology can be grasped quickly. The language syntax is based on the syntax of other widespread programming languages, so it is familiar to people who have seen a few other languages. Programmers can quickly become productive. (*The Java™ Tutorials*:1 2006)

Object Oriented

The Java programming language is designed to be completely *object oriented*, and Java technology provides an efficient object-based development platform. The needs of distributed, client-server based systems coincide with the encapsulated, message-passing paradigms of object-based software. To function within increasingly complex, network-based environments, programming systems should adopt object-oriented concepts. The object-oriented paradigm fit well together with the needs of client-server and distributed software. To be truly considered object oriented, a programming language should support at least four characteristics: (Gosling & McGilton 1996)

- *Encapsulation*: implements information hiding and modularity
- *Polymorphism*: the same message sent to different objects results in behavior that is dependent on the nature of the object receiving the message
- *Inheritance*: new classes can be defined by extending existing classes to obtain code re-use and code organization
- *Dynamic binding*: objects could come from anywhere, possibly across the network. A message must be able to be sent to an object without knowing the specific object type at the time the code is written. Dynamic binding provides flexibility while a program is executing. (Gosling & McGilton 1996)

Java meets these requirements nicely, as well as adding considerable run-time support (Gosling & McGilton 1996).

Multithreaded, Distributed, and Dynamic

Java's multithreading capability makes it possible to build applications with many concurrent threads of activity. Multithreading can thus provide a higher degree of interactivity for the end user. The Java platform supports multithreading at the language level with the addition of sophisticated synchronization primitives. Java technology's high-level system libraries have been written to be thread-safe. As a consequence, the functionality provided by the libraries is available without conflict to multiple concurrent threads of execution. (Gosling & McGilton 1996)

Java provides a simple solution to the problem of distributing your applications across heterogeneous network-based platforms. It is tailored to fit into distributed client-server applications. (Gosling & McGilton 1996)

Java is also a dynamic programming language. While the Java Compiler is strict in its compile-time static checking, the language and run-time system are dynamic in their linking stages. Classes are linked only when they are needed. Furthermore, new code modules can be linked in on-demand from a variety of sources, even from sources across a network. (Gosling & McGilton 1996)

Architecture Neutral and Portable

If a program is to be run on different computer architectures without being recompiled in-between it needs to function in a way that is independent of the underlying architecture. Java is architecture neutral and is designed to support applications that will be deployed into heterogeneous network environments. (Gosling & McGilton 1996)

Architecture neutrality is a major step towards being portable, but it is just one part of a truly portable system. Java technology implies additional portability because it is strict in its definition of the basic language. Java specifies the sizes of all its primitive data types and the behavior of its arithmetic operators. The strict language definitions help avoid data type incompatibilities across hardware and software architectures. (Gosling & McGilton 1996)

C and C++ both carry the problematic of designating many fundamental data types as implementation dependent. The programmers themselves have to ensure that applications are portable across architectures by programming to a lowest common denominator. Java eliminates the issue of implementation dependent fundamental data types by defining standard behavior that will apply to the data types across all platforms. (Gosling & McGilton 1996)

The Java environment itself is readily portable to new architectures and operating systems. Compiled Java language programs are portable to any system on which the Java interpreter and run-time system have been implemented. (Gosling & McGilton 1996)

High Performance

Performance is usually important in software environments. The Java platform achieves high performance by adopting a scheme by which the interpreter can run at full speed without having to check the run-time environment. The Java automatic garbage collector is in charge of memory management. It runs as a low-priority background thread, ensuring a high probability that memory is available when required, leading to better performance.

Applications requiring large amounts of computer power can be designed so that compute-intensive sections can be rewritten in native machine code and interfaced with the Java platform. According to Gosling & McGilton (1996), users generally perceive interactive Java applications to respond quickly even though they are interpreted. (Gosling & McGilton 1996)

Robust and Secure

The Java programming language is designed for creating highly reliable software. The Java compiler employs extensive and stringent compile-time checking so that syntax-related errors can be detected early. The compile-time checking is then followed by a second level of run-time checking. (Gosling & McGilton 1996)

The Java coding conventions, provided by Sun Microsystems, also encourage the use of a programming style that has had positive effects in the past (*Code Conventions for the Java Programming Language* 1999). According to Gosling and McGilton (1996) the language features guide programmers towards reliable programming habits.

The memory management model of Java is extremely simple. The programmer is not responsible for allocating or freeing up memory, since that is all handled by the memory management model. The system will find many errors quickly, which prevents problems in the code from lying dormant without being noticed. (Gosling & McGilton 1996)

C and C++ programmers are by now accustomed to the tedious work of explicitly managing memory. Memory needs to be both allocated and released by the programmer who also needs to keep track of what memory can be freed at what time. Explicit memory management has proved to be a fruitful source of bugs, crashes, memory leaks, and poor performance. The single biggest difference between Java and C or C++ is that Java's memory model eliminates the possibility of overwriting memory and corrupting data. Instead of pointer arithmetic, Java has true arrays and strings, which means that the interpreter can check array and string indexes. The memory management model is based on objects and references to objects. There are no pointers at all. Instead, all references to allocated storage are made through symbolic handles. When an object has no more references, the object becomes a candidate for automatic removal by the garbage collector. Java technology completely removes the memory management load from the programmer.

Since Java has security features designed into the language and run-time system, you can construct applications that cannot be invaded from outside. In a network environment, applications written in the Java programming language are secure from intrusion by unauthorized code attempting to create viruses or invade file systems. (Gosling & McGilton 1996)

The Java Features Combined

Taken individually, the characteristics discussed above can be found in a variety of software development platforms. What is completely new is the manner in which Java technology and its runtime environment have combined them to produce a flexible and powerful programming system. (Gosling & McGilton 1996)

2.3.4 The Java API

In chapter 2.2.3 the concept of namespaces was introduced. A namespace can be thought of as a boundary, within which a name or a label has a certain defined meaning. Outside the namespace boundary the same name can be undefined or have a different meaning.

A *package* is a namespace that organizes a set of related classes and interfaces. Because software written in the Java programming language can be composed of hundreds or

thousands of individual classes, it makes sense to keep things organized by placing related classes and interfaces into packages. A set of packages is called a *class library*. (*The Java™ Tutorials:2* 2006)

The Java platform provides an enormous class library called the *Java API* (Application Programming Interface) (Burd 2002). (The corresponding feature in C++ is called the Standard Template Library, STL.) The API packages provide the core functionality of the Java programming language, and span everything from basic objects to networking, security, XML generation, database access, and more (*The Java™ Tutorials:1* 2006). Java libraries can also be extended to provide new behavior (Gosling & McGilton 1996). (The API for the Java platform SE 6 can be found at *Website 2* in the reference list.)

2.3.5 Serialization

Serialization is a mechanism that, among other things, enables you to store Java objects in a disk file, or conversely to read objects back into memory based on the information contained in a file. (Besides a file on a hard disk, the source or destination could be another program, a peripheral device, a network socket, or an array.) The term *serial* in this case means that simple data types and objects are being read or written one item at a time in a *stream*. A stream is a sequence of data. (Greanier 2000) (*The Java™ Tutorials:3* 2006)

Classes whose objects can be serialized must implement the *Serializable* interface (*Serializable (Java Platform SE 6)* 2006). Since objects can contain references to other objects, all classes being utilized by a serializable class must in turn be serializable as well (*Serializable (Java Platform SE 6)* 2006). Most, but not all, standard classes support serialization (*The Java™ Tutorials:3* 2006). Figure 5 demonstrates serializability of an object a, which contains references to other serializable objects.

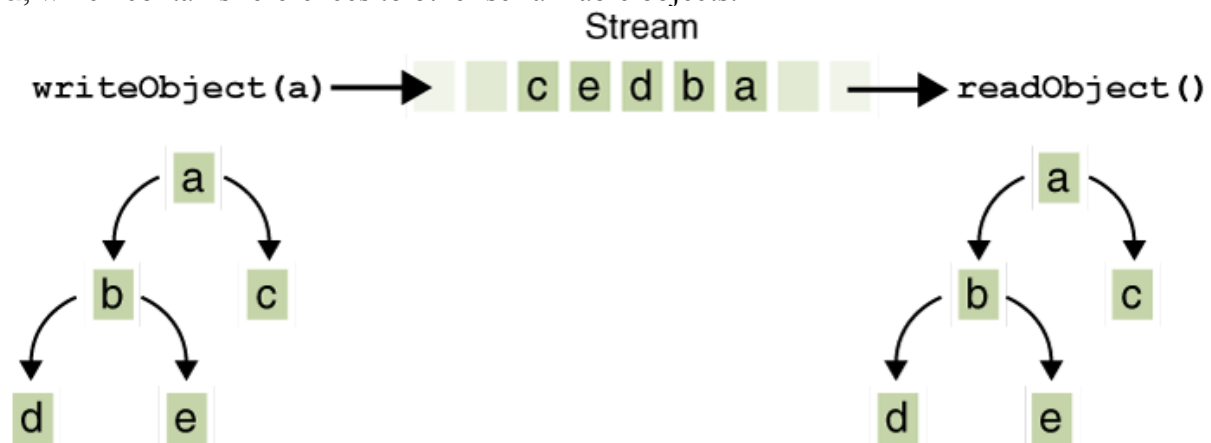


Figure 5

2.4 Java and XML

Java and XML work well together. According to Burd (2002), much of the code created for processing XML is in fact written in Java. One good reason why the XML developer community chooses Java could be that both Java and XML are streamlined for the internet. Furthermore, Java and XML are both industrial de-facto standards. Without standards, today's modern world could never have been. If one computer did not have a common way of communicating with another, there would be no internet at all for example. (Burd 2002)

Another reason why Java and XML blend so nicely together is because of portability. Java is portable, XML is portable, and in the realm of portability the chain is only as strong as its weakest link. If one link is not portable – the chain is not portable either. In a scenario of using C++ together with XML, problems could easily emerge due to changes in the software environment because the data would be portable, but the code would not (Burd 2002).

Starting with version 1.4, Java's core API included packages devoted exclusively to the processing of XML documents.

2.4.1 JAXP, SAX, DOM

JAXP is the collective term for several of Java's XML tools that form the backbone of the Java XML strategy (Burd 2002). JAXP is short for Java API for XML Processing. It is a collection of APIs in particular containing *SAX*, *DOM*, and *XSLT*, which are all included in Sun's JAXP package (Burd 2002). We will have a quick look at the two former APIs; SAX and DOM.

SAX

SAX stands for Simple API for XML. (If one would wish to further elaborate on this acronymic tangle, the full name would actually be "Simple Application Programming Interface for eXtensible Markup Language".) SAX is a low-level, general-purpose approach to handling XML documents (Burd 2002). Because SAX is very low-level you can do almost anything with it, and perhaps for this reason it has been given its nickname; QDAX – The Quick-and-Dirty API for XML (Burd 2002).

Sax views an XML document as nothing more and nothing less than a sequence of tags. You assign an action to each kind of tag, and perform the appropriate action for every tag in an XML document. Nothing gets stored over long-term in memory, and no part of the document gets processed along with any other part. This approach, to simply iterate through the tags, makes SAX processing extremely fast. According to Burd (2002), SAX should be used when the XML document represents linear data, when you are dealing with very large XML documents, or when you need fine-tuned control. (Burd 2002)

DOM

DOM stands for Document Object Model, and like SAX it is an all-purpose tool. Because the DOM API is not targeted towards any specific XML applications, you can perform almost any task with it. Unlike SAX's linear view of the XML data, DOM has a holistic view of the document. With SAX you could say that you are scanning data from top to bottom – with DOM you are scanning data from the inside out. (Burd 2002)

While SAX views a document as just a sequence of tags to iterate through, DOM handles the entire XML document at once, and instead of just seeing tags it recognizes XML elements. To do that, a DOM program makes a big copy of the document and stores it in the computer's memory. Because of this approach, to build up a tree structure of all the elements in memory regardless of what work might have to be done, DOM is not fast. According to Burd (2002), you should avoid DOM when you expect the XML document to be very large, but you should use DOM when the document makes heavy use of element nesting, when you need a global view of the data, or (as with SAX) when you need fine-tuned control. (Burd 2002)

2.4.2 JDOM – Java Document Object Model

As the name Java Document Object Model indicates, *JDOM* has got something to do with Java. Neither SAX nor DOM is specific to Java. You can write SAX/DOM programs in C++, Perl, and a number of other programming languages. If you would write a DOM program in Java, the core Java API would not be fully utilized since DOM must remain generic for all languages. If some feature is available in Java but not in C++, then DOM does not use that feature. Neither SAX nor DOM take advantage of any language's pre-established library of classes and methods. Because of that DOM can be awkward and cumbersome to use. (Burd 2002)

To remedy this and other DOM shortcomings, Jason Hunter and Brett McLaughlin created JDOM. The big difference is that JDOM takes full advantage of the power of Java, and uses a sleek and intuitive tree structure that is missing from DOM. (JDOM is not part of Sun's Java toolset, but it can be downloaded from www.jdom.org.) According to Burd (2002), you should use JDOM to create brand new XML documents, as a lightweight alternative to DOM, if you want clean, elegant code, or if your organization is committed to Java. (Burd 2002)

2.4.3 JAXB

With SAX or DOM you create an XML processing program. The program reads an XML document and uses the document to do some useful work, but it makes no assumptions about what is inside the document. All that is known is that the XML document has a root element, and possibly some child elements. Any special assumptions besides that actually narrow the usefulness of the code. But making few assumptions and being versatile comes with drawbacks, including the possibility of very high overhead. A DOM program, for example, has to parse an entire XML document and then put a representation of the document's tree in memory. If the document is very large, then the representation is large. The memory can get bloated with a lot of temporary data just as the code becomes slow. DOM is simply not customized to the needs of a specific situation, and that can easily lead to unnecessary resource consumption. (Burd 2002)

JAXB stands for Java API for XML Binding. With JAXB you can turn an XML document into a Java class or vice versa. When going from XML to Java, you get Java source code (in the form of a readable .java file) containing classes corresponding to the elements in the XML document. If the document contains a `Register` element, then the Java code can have a `Register` class. If the `Register` element has an XML attribute called `bitWidth`, then the Java `Register` class gets the methods `getBitWidth` and `setBitWidth`. The connection between a part of an XML document and a part of a Java class is called a *binding*. With these bindings, an instance of the class represents a single XML document. (Burd 2002)

The idea behind JAXB is to create a custom-tailored Java class that is streamlined to work with particular XML documents. This custom class is generated by running either a DTD or an XSD together with an optional *binding schema* through a special program called *schema compiler* (*Java™ Architecture for XML Binding* 2005). The generated class is optimized for the type of XML documents that is defined by the XSD/DTD, and mapped into Java classes in a way that is described by the binding schema. (Burd 2002)

Binding Schema

How XSD works has been detailed in chapter 2.2.2. The binding schema describes the way in which the Java class' values match up with the data in the XML document (Burd 2002). For example, in the binding schema you could instruct the schema compiler that the `chipWeight` element matches a variable of simple type, instead of a class, and that this variable is of type `double`. In the binding schema file (`xml-java-binding-schema.xml`), this instruction would be expressed by means of XML in the following way:

```
<element name="chipWeight" type="value" convert="double" />
```

Marshalling and Unmarshalling

Once you have the Java classes corresponding to a certain XML structure you need a way to populate the class instances to contain the values in the XML document. You also need to be able to write the values contained in a class instance into an XML file. These processes are called *unmarshalling* and *marshalling* respectively. The generated Java class automatically gets two methods named `unmarshal` and `marshal` for these purposes. (Burd 2002)

A typical JAXB example could be as follows: You start with an XML document containing some data. First, you unmarshal this into a Java object. Then you perform some calculations or manipulation of the data using Java. Finally, you marshal the new data back into the same or into a different XML file, where the data gets stored.

JDK and JAXB

Sun's XML suit currently has five subcategories (XML 2006). As of Java JDK 6, JAXB 2.0 is one of them (so is JAXP) and part of the standard distribution, so no separate download of JAXB is necessary (Java SE 6 Key Features 2007).

2.5 Swing – Building GUIs in Java

2.5.1 Introduction to Swing, AWT, and JFC

AWT stands for Abstract Window Toolkit and is the part of Java designed for creating user interfaces and painting graphics and images. AWT is a part of the Java Foundation Classes (JFC), which is a comprehensive set of GUI components and services (Java Foundation Classes (JFC) 2006). JFC contains five major parts: AWT, Swing, Accessibility, Java 2D, and Drag & Drop. These five parts, however, are certainly not mutually exclusive. Swing and Java 2D are User Interface Toolkits, and make it possible to create sophisticated Graphical User Interfaces (GUIs) (The Java™ Tutorials:1 2006). (Robinson & Vorobiev 2003)

Almost all Swing² components are derived from a single parent named `JComponent` which extends the `AWT Container` class. (Basically every AWT component has a Swing equivalent that begins with the prefix “J”, although many Swing classes do not have AWT counterparts.) Swing is built as a layer on top of AWT and is expected to merge more deeply with AWT in future versions of Java. (Robinson & Vorobiev 2003)

The Swing toolkit includes a large set of components for building GUIs and adding interactivity to Java applications. The Swing components range from the very simple, such as

² The name “Swing” comes from the code name of the project that was responsible for developing these new components for the Java API. Although unofficial, the name Swing is frequently used to refer to the new components and the related API. (The Java™ Tutorials:5 2006)

labels or buttons, to the very complex, such as tables, trees, and styled text documents (Robinson & Vorobiev 2003). Besides graphical components, Swing also includes rich undo support, a highly customizable text package, integrated internationalization, and accessibility support. To truly leverage the cross-platform capabilities of the Java platform, Swing supports numerous look and feels, for instance one matching the look and feel of the underlying operating system. Other basic user interface primitives such as drag and drop, event handling, customizable painting, and window management are part of the toolkit as well. (*The Java™ Tutorials:4* 2006)

The most remarkable thing about Swing components is that they are written 100% in Java³ and do not directly rely on peer components, as most AWT components do. It means that a Swing button or text area can look and function identically on Macintosh, Linux, and Windows platforms, which reduces the need to test and debug applications on each target platform. (Robinson & Vorobiev 2003)

Swing is far from a simple component toolkit (*The Java™ Tutorials:4* 2006). According to James Gosling, Vice President and Fellow at Sun Microsystems, Swing is an extraordinarily sophisticated user interface toolkit that gives great power to developers. This power is, according to Gosling, also the reason behind the biggest problem with Swing; that although there is logic to it all, its wide variety of facilities can be intimidating and confusing. Swing is a very complex library and few people master all aspects and areas of its extent. (Robinson & Vorobiev 2003)

2.5.2 Start Building from the Root - JFrame

The first thing you need in order to start building your GUI is a top-level foundation from which you can start adding layers and components. Every Swing GUI component must be part of a containment hierarchy, which is a tree of components with a top-level container as its root (*The Java™ Tutorials:7* 2006). Swing provides three generally useful top-level container classes: `JFrame`, `JDialog`, and `JApplet` (*The Java™ Tutorials:7* 2006).

`JApplet` is used for building applets that can be run inside a browser. We do not concern ourselves with applets in this case, but instead with a standalone application, so instead of `JApplet`, our main containment hierarchy will have a `JFrame` as its root. Dialogs are used to create a dialog with the user, either in a standalone application together with a `JFrame` or in an applet. Each `JDialog` has its own containment hierarchy separate from the hierarchy of the `JFrame` or `JApplet` where the `JDialog` object is the hierarchy root. (*The Java™ Tutorials:7* 2006)

³ The only exceptions are four heavyweight Swing components that are direct subclasses of AWT classes, and rely on platform-dependent peers: `JApplet`, `JDialog`, `JFrame`, and `JWindow`.

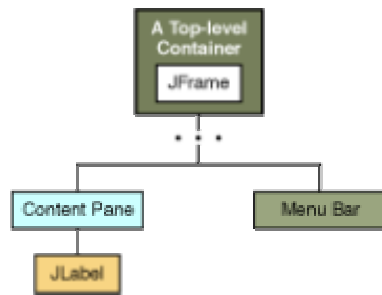


Figure 6

2.5.3 Laying out the Components – `JPanel`

Once you have a `JFrame` as the containment hierarchy root you can start placing out GUI components, such as buttons or text areas. However, only in the absolute simplest case would it suffice to simply place everything out in the content pane of the `JFrame` root.

In a normal GUI, the components have to be arranged in very specific ways. To achieve that, you need a sophisticated way to lay out the components. For this purpose you can use *panels* which in Java are placeholders for components or other panels. The `JPanel` class provides a generic lightweight container, commonly used to organize a group or groups of child components (Robinson & Vorobiev 2003). By default, panels do not paint anything except for their background, but you can easily add borders to them and customize the way they paint their area (*The Java™ Tutorials*:6 2006).

Every `JPanel`'s child components are managed by a *layout manager* (Robinson & Vorobiev 2003). A layout manager controls the size and location of each child inside a container (Robinson & Vorobiev 2003). There are a number of different layout managers available and they lay out and resize the child components in different ways and according to different criteria. By combining and layering `JPanels` on top of each other, and possibly using different layout managers for the panels, very specific positioning of the GUI objects can be achieved.

2.5.4 Making Everything Fit – `JScrollBar` and `JScrollPane`

In most GUIs you want to have *scrollbars* in one form or another. In the simple case, scrollbars can be used for text areas when the size of the text is larger than the size of the viewable area. In the more complicated scenario, scrollbars come into play when the gathered contents of all the components and sub-panels in an area are bigger than the area itself. Without scrollbars the only other way for the user to see all contents would be to resize the application window in a way that makes the area at least as big as the size taken up by the components. In many GUIs the total size of all input fields, buttons, text areas, and other components are larger than the size of the entire computer screen, making scrollbars absolutely imperative. In Swing, scrollbars are implemented by the `JScrollBar` class. (*The Java™ Tutorials*:8 2006)

To customize the way a component, such as a `JPanel`, interacts with its scrollbar, you can make the component implement the `Scrollable` interface (*The Java™ Tutorials*:8 2006). By implementing this interface we can specify how many pixels are scrolled when a scroll bar button or scroll bar paging area (the empty region between the scroll bar thumb and the buttons) is pressed (Robinson & Vorobiev 2003). If you cannot or do not want to implement a scrollable component, you can specify the unit and block increments using the

`setUnitIncrement` and `setBlockIncrement` methods of `JScrollBar` (*The Java™ Tutorials*:8 2006). The effect is to change how big leaps the viewport makes over the viewable area when the user clicks on the scrollbar.

Usability studies have shown that readers can find data quickly when they are scrolling vertically. Scrolling horizontally, on the other hand, is laborious and difficult when dealing with text and should be avoided. With visual information, such as tables of information, horizontal scrolling may be more motivated, but both vertical and horizontal scrolling should not be mixed together. Whenever possible, scrolling should either be avoided completely or introduced in only one direction. (Robinson & Vorobiev 2003)

When you want a part of the GUI to be scrollable, you use a `JScrollPane`. `JScrollpanes` can create a scrollable view of any component or container, for example text areas, pictures, or panels (Robinson & Vorobiev 2003). In the case of scrolling one single component, such as a text area, the use of scrollpanes is normally very simple and we need not concern ourselves further with that here (Robinson & Vorobiev 2003). However, when correct scrolling is needed over a complex nesting of panels and their contained components, the issue becomes far more complicated as illustrated in the following section. (*The Java™ Tutorials*:8 2006)

2.5.5 Nesting of Components

The layout manager for a panel is responsible for figuring out the panel's preferred size. Sometimes you need to customize the size hints that a component provides to its container's layout manager for the component to be laid out well. You can do this by means of setting a component's preferred size property. Many layout managers, however, do not pay attention to a component's requested maximum size. The grid layout, for example, forces all components to be the same size, and it tries to make them as wide as the widest component's preferred width and as high as highest one's preferred height. (*The Java™ Tutorials*:9 2006)

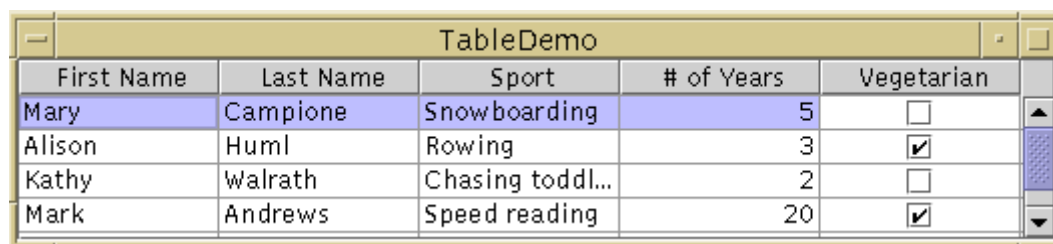
Another issue arises with components that dynamically change size. In the simple case of scrolling over one single component, changing the size of the component is a two-step process. First, set the component's preferred size. Then, call `revalidate` on the component to let the scroll pane know that it should update itself and its scroll bars. In the case of nested panels a third step must be added. The size-change must also be propagated upwards through the hierarchy so that the change update reaches the panel where the scroll bars are located. (*The Java™ Tutorials*:9 2006)

So the reason why scrolling over one single component is simple to implement is because it is fairly obvious what the size of that component should be. One simply calls the `getPreferredSize()` method of the component (Robinson & Vorobiev 2003). However, finding the preferred size of a collection of nested components and panels, where some components size may change dynamically as the GUI is being used, cannot be done simply by asking for the preferred size of the parent panel. Since the parent component does not necessarily respect the preferred size of its children, the component sizes need to spread upwards to reach the panel holding the scrollbars. If not, adding new components to a sub-panel could easily make the sub-panel expand out of view because the scrolling panel remains the same size and thus some components are placed outside of the scrollable view.

2.5.6 Dynamic Tables – JTable

With the `JTable` class you can display tables of data, optionally allowing the user to edit the contents (*The Java™ Tutorials:11* 2006). `JTable` is extremely useful for displaying, navigating, and editing tabular data. Because of its complex nature, `JTable` has a whole package devoted to it (`javax.swing.table`) (Robinson & Vorobiev 2003).

Each column in the table has an associated *cell renderer*, *cell editor*, and *table header*. When a `JTable` is placed in a `JScrollPane`, these headers are placed in the scroll pane's header viewport displaying the column names on top of the table. The columns can be dragged and resized to reorder and change the size of columns. Even if the user scrolls down, the column names remain visible at the top of the viewing area. Figure 7 shows how a typical table might look. (*The Java™ Tutorials:11* 2006) (Robinson & Vorobiev 2003)



First Name	Last Name	Sport	# of Years	Vegetarian
Mary	Campione	Snowboarding	5	<input type="checkbox"/>
Alison	Huml	Rowing	3	<input checked="" type="checkbox"/>
Kathy	Walrath	Chasing toddl...	2	<input type="checkbox"/>
Mark	Andrews	Speed reading	20	<input checked="" type="checkbox"/>

Figure 7

`JTable` does not contain or cache the data - it is simply a view of it. For performance reasons, the cells in Swing tables are not components themselves. Instead, the cell renderer is used to draw all of the cells that contain the same type of data. The cell renderer can be thought of as a configurable ink stamp that the table uses to stamp appropriately formatted data onto each cell. When a user starts to edit a cell's data, a cell editor takes over the cell, controlling the cell's editing behavior. The component returned by the cell editor is completely interactive (Robinson & Vorobiev 2003). (*The Java™ Tutorials:11* 2006)

If no renderer or editor is explicitly assigned, default versions will be used based on the class type of the column data (Robinson & Vorobiev 2003). For instance, by default, the cell renderer for a number-containing column uses a single `JLabel` instance to draw the appropriate numbers, right-aligned, on the column's cells. If the user begins editing one of the cells, the default cell editor uses a right-aligned `JTextField` to control the cell editing. `JTable` has a predefined list of classes for which special purposes editors and renderers are used. If the class of the objects in a column does not belong to that list, the object's `toString` method is used for display by the renderer. (*The Java™ Tutorials:11* 2006)

In Swing, some components allow us to define custom cell renderers and editors used to display and accept specific data, respectively. We can, for example, have the columns of a `JTable` rendered with custom icons, alignments, and colors. `JTable` is one of the most complex Swing components. Keeping track of its constituents and how they interact is a challenge. (Robinson & Vorobiev 2003)

2.5.7 Dynamic Object Trees – JTree

The tree data structure is very important and heavily used throughout computer science. Among other applied areas, it is used in compiler design, graphics, and artificial intelligence. (Robinson & Vorobiev 2003)

The tree data structure consists of a logically arranged set of *nodes*, which are containers of data. Each tree contains exactly one root node, which serves as that tree's top-most node. However, every node in a tree can also be viewed as the root node of the sub-tree rooted at that node. Any node can have an arbitrary number of child nodes. Each node is connected by an *edge*, which signifies the relationship between two nodes. A node's direct predecessor is called its *parent* node. A node that has no child nodes is called a *leaf* node, and a node that contains children is called a *branch* node. A *path* from one node to another is a sequence of nodes with edges from one node to the next. In graph theory, a tree is a connected *acyclic graph* (Tree (data structure) 2007). This means that edges between nodes must only exist between a parent and its direct children, not between a child and its parent's parents (*ancestors*). It also means that for every two nodes in the graph there must a path between them. (Robinson & Vorobiev 2003)

Java's `JTree` is a great tool for the display, navigation, and editing of hierarchical data. It can improve usability by easing the process of finding something within such a data set. Just as with `JTable` described in the previous chapter, `JTree` has a whole package devoted to it due to its complexity (`javax.swing.tree`). It provides us the ability to perform *preorder*, *inorder*, and *postorder* traversals of the tree. These are three distinct algorithms that visit every node in the tree once, but in different orders. (Robinson & Vorobiev 2003)

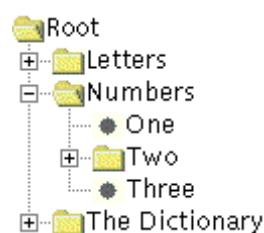


Figure 8

As Figure 8 shows, `JTree` displays its data vertically. Each row displayed by the tree contains exactly one node. Each tree cell that is not a leaf node is shown as being either *expanded* or *collapsed*, and typically, the user can expand and collapse nodes by clicking them. Expanded nodes show their sub-tree nodes, collapsed nodes hide what is underneath them (Robinson & Vorobiev 2003). The tree conventionally displays an icon and some text for each node. The default icon displayed by a node is determined by whether the node is a leaf, and if not, whether it is expanded or collapsed. (*The Java™ Tutorials*:10 2006)

Every node constitutes a cell in the `JTree`. Analogous to `JTables`, each cell can be rendered with a custom renderer and can be edited with a custom editor (Robinson & Vorobiev 2003). The renderer specifies how icons and text are displayed, and the editor specifies how the text can be edited directly in the tree. You can change the default icon used for leafs, expanded branches, or collapsed branches, by either instantiating or extending the class `DefaultTreeCellRenderer` and using it as the tree's cell renderer. The text being displayed after the icon of each node is the return value of the `toString` method of the object contained inside the node. (*The Java™ Tutorials*:10 2006)

`JTree` implements the `Scrollable` interface and is intended to be placed in a `JScrollPane` (Robinson & Vorobiev 2003).

2.5.8 Building Graphical Tools – Custom Painting of JPanel

If you cannot find a way to make a component look and behave the way you want it to, using icons, styled text, or borders, then you might need to perform custom painting (*The Java™ Tutorials:12* 2006). With custom painting, of for instance a `JPanel`, you can create a canvas area on which you can draw graphics that change dynamically in accordance with interaction from the user. That way you can create special-purpose interactive graphical tools.



Figure 9

When a Swing GUI needs to paint itself, whether for the first time or because it needs to reflect a change in the program's state, it starts with the highest component that needs to be repainted and works its way down the containment hierarchy. This process is orchestrated by the AWT painting system, and made more efficient and smooth by Swing. Swing components generally repaint themselves whenever necessary. (*The Java™ Tutorials:12* 2006)

Custom painting is not the same thing in Swing as it is in AWT. In AWT you typically override a `Component`'s `paint` method to do rendering. You also override the `update` method for implementing your own double-buffering or for filling the background before `paint` is called. With Swing, component rendering is much more complex. Even though Swing's `JComponent` is a subclass of AWT's `Component`, it uses the `paint` and `update` methods for different reasons. In fact, the `update` method is never invoked at all with Swing. Furthermore, there are five additional stages of painting that normally occur from within the `paint` method. We will not discuss the intricacies of the Swing painting process here, but suffice it to say that any `JComponent` subclass that wants to take control of its own rendering should override the `paintComponent` method and not the `paint` method. Additionally, the overridden method should always begin with a call to `super.paintComponent`. Knowing only this, it is quite easy to build a `JComponent` that acts as your own canvas on which you can draw graphics. You just have to subclass it and then do all drawing inside the overridden `paintComponent` method. (This is the approach for simple custom Swing components such as `JPanel`, however, do not attempt this with other more complex components because UI delegates are in charge of their rendering.) A component that is a specialized container should probably extend `JPanel` (*The Java™ Tutorials:12* 2006). (Robinson & Vorobiev 2003)

Inside the `paintComponent` method, you have access to that component's `Graphics` object (which should immediately be casted to a `Graphics2D` object). The `Graphics` class defines many methods that you can use to paint shapes and draw lines and text, using various fonts and colors. (Robinson & Vorobiev 2003)

If the component's size or position also needs to change, a call to `revalidate` precedes the one to repaint. (*The Java™ Tutorials:12* 2006)

3 Method

3.1. Methodology in a Software Project

An extensive discussion of methodology in its traditional sense is, due to the nature of the project in question, somewhat hard to motivate. As far as the usage of certain software methods are concerned, there is no clear distinction between what is presented in this report as theory, design decisions, or implementation. Instead they overlap to form the main coverage of methodology. What is meant by this is that Java, for example, is a method that can be used to reach scientific results, and Java as a tool or method is present throughout the whole thesis. Java is a means to an end method that can bring us the results and conclusions we search for.

The two main methods chosen in this project, Java and XML, are introduced and described in the theory chapter. There, the most important features that make Java and XML useful in this case are explained. The usage of these particular tools is then further analyzed and motivated in succeeding chapters.

One area of methodology, however, must take place in any sound research project, be it a software project or not. That is the gathering of necessary background information and the following sections outline that.

3.2 Gathering Background Information

Before any actual coding took place, the roadmap towards solving the given problem begun with a literature study in the field of Java & XML. As the coding begun, the literature study continued in parallel mainly with the subject of Java's Swing toolkit. Backman (1998) claims that the literature study of a thesis or report could be the most important phase in the entire research process. It allows the authors to gain insight in how previous surveys and studies have been conducted, and these are necessary prerequisites for the continued work (Backman, 1998). According to Jacobsen (2002), research data can be divided in primary and secondary data.

3.2.1 Secondary Data

Secondary data is information that has been collected by other people than the researcher himself, and has been done so for a different purpose than that of the researcher using the secondary data (Jacobsen 2002). Secondary data was used in this project in the form of the literature study summarized in chapter 2.

3.2.2 Primary Data

Primary data is such data that the researcher gathers from scratch and could be tailored for a specific purpose. Primary data can be collected by means of interviews, observations, or questionnaires (Jacobsen 2002). Primary data was used while defining the underlying XML data model. The idea and basic foundation behind the data model was based on secondary data (i.e. derived from the SPIRIT standard), but due to limitations of this standard it had to be further extended and modified. This extension and reconfiguration of the XML data model was influenced by primary data gained by means of meetings and informal discussions with people within the industry. Since this author did not partake in how that primary data was gathered and later used, it is not a part of this report. Although the implementations that

concern this thesis utilize the XML data model extensively, the question of how the data model was refined through qualitative primary data is outside the scope of this thesis.

3.2.3 GUI Tracer Bullet

Another source of primary data was a user requirements analysis that was initiated at the start of the project. The end users of the system and the GUI would be people working at the company in question. The purpose of the analysis was therefore to sit down with these people and discuss how they would want the GUI to look and how it should function. The difficulty with the analysis, and the reason why it was hard to formalize or to document, was that no system or even prototype existed to show the users. That made it difficult for them to know what to expect or to demand from the future GUI. Another big problem was that it was not clear at such an early stage exactly how the data model would look later on, so it was not yet known what features the GUI needed to have.

In order to tackle these issues, as well as to concretize the qualitative feedback from the users, it was decided to start working on a so-called tracer bullet. A tracer bullet is not a prototype. The purpose of a prototype is not to be used in real-world production. A car prototype of a new car model might be built in wood and nobody would install an engine in it and try to sell it as a real car to consumers. Nor should software prototypes morph their *raison d'être* into a continued existence they were not meant for. If a prototype grows into a finished product it was, or at least should have been, a tracer bullet instead. (Hunt & Thomas 1999)

The analogy with a tracer bullet comes from the military. Tracer bullets are blank signal flares that are mixed in between the real bullets in an automatic weapon. Where the tracer bullets flare up and mark the spot, is where the real bullets will hit as well. In a software system, a tracer bullet is not supposed to a complete system. It is supposed to be a skeleton or backbone with which some functionality is included. As more and more functionality is added to the rudimentary foundation of the tracer bullet, developers get iterative and continuous feedback from the users. (Hunt & Thomas 1999)

A tracer bullet for the GUI was introduced to the company users. That provided something tangible and visual to present to the users which helped them express their thoughts and requests. The tracer bullet evolved iteratively together with new user feedback until the user requirements and the visions of the developer had merged together into a real product. Thanks to this iterative process the user feedback automatically became integrated in the GUI although it was qualitative and could not be easily documented. Since an RCS was used from the start, early versions of the tracer bullet could be recovered if the need thereof should arise.

3.3 Code Review

Towards the end of the project, a code review on individual basis was held together with one of the supervisors. The review gave the opportunity to constructively analyze and evaluate the written code, as a way of finding improvements and clarifying the documentation.

The code review turned out to become particularly valuable for the company in question. By the end of the time period for this thesis, there was not yet someone else to take over the work related to the GUI. By having done a code review, the supervisor had gotten insight and understanding of the code and could thus in turn explain the code to any successors in the project.

3.4 Development Tools

During the project a number of software tools for developing, testing, and debugging of the system were used.

As Java development environment *Eclipse* was used. Eclipse is a powerful tool comprised of extensible frameworks, tools and runtimes for building, deploying, and managing software across the lifecycle. Eclipse is an open source development platform that can be downloaded freely from www.eclipse.org.

IBM Rational ClearCase was used for software revision control. In software projects it is extremely common for multiple versions of the same software to be deployed in different sites, and for the software's developers to be working simultaneously on updates. Software tools for revision control are increasingly recognized as being necessary for almost all software development projects these days. The choice of this particular Revision Control System (RCS) came from the fact that it was already used within the company. (*Revision control* 2007)

Sparx System Enterprise Architect was used for defining the XML data model.

4 Design Choices

4.1 Conforming to Standards

4.1.1 Why Standards are Important

Some approaches to tackle the problems with software discussed in chapter 2.1 are known today. They are widely spread and they are all related to the use of standards. The strength *and* the threat of being able to express solutions in different ways increase as the level of the programming language decreases. In a low-level language you have access to “smaller” and less restricted components close to or inside the operating system. You have more freedom to minutely tune the behavior of the computer, at your own risk of course. In a higher-level programming language you usually do not access these fine grained controls directly, but instead you access the end of a chain of controls that are designed to help you achieve your tasks. Because you use higher-level functions that have a standard approach towards solving common tasks, you have fewer alternatives of solutions to the same problem, and thus higher level programming languages reduces the problems coupled with too much expressiveness.

The connection that this has to the use of standards is subtle but clear. Naturally there exist standards for low-level just as well as for high-level programming languages. The important role of standards lies in the very use of taking one step up. When designing a high-level language a choice must be taken on how to solve common tasks in order to provide the programmer with the high-level functionality. Therein lies the creation of a new standard; the people responsible for the language in question must choose one solution out of all possible ones, and we have reason to hope that they will choose one of the better ones. This choice of solution is provided to the user and becomes a standard, simply because there is no other way of doing it.

Another way of dealing with the problems of software is the use of naming and coding conventions. For instance, if it had not been decided that the top-level domain name for every country should consist of two letters that refer to the name of that country, it would have been quite difficult to learn which domain name belonged to which country. Its much easier to learn that United Kingdom has the extension *.uk* instead of say *.3}m*. Coding conventions have the same effect; it is easier to guess what a function in a program does if its named *getTotalSumInEuros()* instead of *mYfuNction37()*. The coding convention standard also increases readability if the user can find things in code simply because he *knows* where to look for it. Because you can freely declare your variables at different places in the code, it is recommended, as a standard, that you do it at the top of you method, and not where the variable is first used.

4.1.2 Problems with standards

As many as the advantages of rigorous standards might be, there are some drawbacks as well. Concerning the internet and HTML, this de-facto standard is so strong today that it actually holds back and hinders significant improvements in the World Wide Web.

When the web was first created and HTML began to be utilized, it had a structure that with modern software awareness seems like mixing pears and apples unjustifiably in the same basket. In HTML you have both data, structure, and layout mixed up when it should really

better be separated. Some tags like the table cell tags hold only structural information, others like the font-color tag deal with graphical presentation of data, and raw text between tags constitutes actual data. Some tags have a closing tag, others do not. The easiest way to comment something out in HTML is often to just add a character like 'x' to the tag name rendering it incorrect which in turn makes it invisible in the browser. You can really abuse the HTML code without getting any errors or warnings and the average browser will simply take a guess or quietly ignore faulty code. In some cases, like with the form tag, you actually *have* to deliberately write incorrect code in order to force some browsers to respond correctly. MS Internet Explorer will for example produce line brakes due to the form tag – a visual arbitrariness which has absolutely nothing to do with declaring the boundaries of an HTML form (Appendix B). Mozilla Firefox also produces line breaks due to form tags but not in the same way and with a different visual result! A workaround for this is to hide the form tags between the row tags of a table – a terrible hack, yet effective. See Appendix B for HTML code samples and screenshots of this bizarre phenomenon.

Obviously, the lack of a standard way of graphically presenting HTML code among browser vendors create tremendous headaches for web designers all over the world. That the remedy of this illness should be to exploit the defects of the HTML language only proves what a poor language HTML is by today's measurements. It makes you wonder why HTML is still being used at all. Better standards for writing code on the internet already exist, such as the XML-based RDF and OWL languages (Antoniou & van Harmelen 2004), but because the HTML standard is *so* widespread, the friction and cost of improvement restrain this progress to a slow crawl. The standard on the web has grown so strong that it prevents the transition from an obsolete and defective language over to a superior alternative.

4.1.3 Conclusion

As seen in the previous section there are situations where a standard has obvious flaws but has become so strong and widespread that a better alternative has difficulties taking over. However, an old and poor standard that out of sheer size is able to fight off a stronger competitor, has gained that massive size for a reason. Let us remember that the standard itself is inanimate and has no will of its own to survive – it continues to live on because people nourish it for some various reasons. For whatever reason, be it cost of change, unawareness, fear of change, or politics, there is an explanation for why people cling to a strong standard.

In conclusion we can identify three cases in which one has to make up ones mind about what to do with standards.

In the first case, there are no standards available yet. Then you try to come up with the best solutions yourself and hope that it is good enough to spread and become a standard. Even if it does not, no other choice but our own creation was available so we have lost nothing by trying.

In the second case there are several standards available that are somewhat equally appealing. One standard that is not actually the best might be able to compete with the most optimal standard simply because the inferior standard is backwards compatible if that happens to be a requirement. In this case one has to evaluate the different standards and decide from case to case what to use. An example of this could be concerning web pages. If creating a webpage for a modern artist we should be inclined to use more modern technologies, such as Flash, to deliver a complex aesthetic message. On the other hand, if creating a webpage for online

banking, we would probably choose older, more compatible, and reliable web techniques to deliver a stable and secure banking service.

In the third case, there is only one standard available or there is one standard that stands out from rest to such an extent that the preferred choice is pretty obvious.

In any case, not conforming to any standard, if there are widespread and *suitable* standards available, would require a very good motive for doing so. In this project there were no reasons not to strive towards conforming to standards, and hence it was decided to base the work on current industry standards.

4.2 XML Data Model

4.2.1 Using XML

In accordance with the previous discussion it was decided that the project work should try to conform to industry standards as far as possible. As a result, XML was chosen as representation language for the underlying data structure. Three main reasons made XML the preferred choice due to it being a *standard*, *human readable*, and transparent *file based*. As far as being an accepted standard, XML was certainly not the only contestant for the data structure. Using a relational database would also have been possible. A relational database, however, is not human readable.

Furthermore, with a transparent file based data structure we could decide ourselves at what granularity level objects should be encapsulated in a file of its own on the hard drive. That way, small sub-objects that never exist autonomously could be contained in the file of a larger parent object. The file on the disk thus represented a top level object that was self contained and an entity of its own. We could for instance say that an IP Component and all its sub-components should be contained in one separate file. This had relevance in conjunction with the Revision Control System since that was file based as well. The user only checked out the top level object file and when reverting to previous representations of an IP Component, the version handling was done at the appropriate hierarchy level. This arbitrary file separation is not possible in a relational database, where the data on the hard drive is usually grouped in database and table levels instead.

4.2.2 The SPIRIT Standard

In the end, the SPIRIT data model as described in chapter 2.2.4 was not adopted in the project. Even though the content of that data model was quite similar to the XML data model that was developed and used instead, SPIRIT turned out to be inadequate in our case. The important difference was that in our case the data model was used as *specification* whereas SPIRIT used it as a *description*. In SPIRIT you start with a component and then use the standard to describe what you already have. In our case the procedure was the opposite; we wanted to start by describing what we wanted (at the conceptual design level) in order to produce a component or chip in the end. Besides the need of a top-down approach for design flow, our data model also had to be extended beyond what the SPIRIT standard supported. (Cross-references between certain chip components were for example not supported by SPIRIT.)

Much of the rudimentary data model specification was kept in conformity with the SPIRIT standard, however. By having as much as possible in common, export and import features to and from SPIRIT could be implemented relatively easy.

4.2.3 Elements vs. Attributes

Concerning the use of elements vs. attributes in XML in the case where both can be used, it is often a matter of taste which one is preferred over the other. It seems intuitive, though, to try to use elements to represent objects, such as registers, and to use attributes for non-complex characteristics such as bit width or the weight of the register, where the attributes are simple values rather than objects in themselves. To return to our previous example with registers, and introducing the sub-component bit field we get:

```
<register bitWidth="32" name="My Register" resetValue="0xffffffff">  
  <bitField startingBit="7" length="16"></bitField>  
</register>
```

4.3 Java

Various programming languages would have been possible to use for building the API around the XML data model and to build the Graphical User Interface. To ease maintenance and future development of the product it should be written in a language that was an acknowledged and well known standard in the industry. Beyond that, a platform independent language was advantageous since engineers at the company in question used many different operating systems. Another important aspect was how well the language would work together with XML. Finally the programming language of choice should be suitable for creating a GUI that could run and function consistently in different desktop environments. Java is platform independent, its JAXB and JAXP packages makes it ideal for use with XML, and Swing is a powerful Java toolkit for making multiplatform GUIs. All in all, Java stood out as the best choice of programming language.

4.3.1 JAXB

After the literature study and analysis of the different Java packages available for working with XML, it was concluded that JAXB would best suit the needs of the project. SAX and DOM were simply too low-level to be practical in our case. JDOM was discussed together with JAXB as a candidate but was eventually rejected as the requirements of the project materialized. Besides the downside of having to read in a whole XML files into a data structure in memory, JDOM was not as suitable as JAXB to build the Custom API that were to surround the XML data model. JAXB is tailored for precisely such application domains and could be used to generate the Java code for the entire Custom API based on the structure of the XML data model.

4.3.2 On-demand import of classes

It was decided not to use on-demand import of classes. One reason to use on-demand import is that it can save a little bit of time in the spur of the new coding moment; you do not have to specify every class in the package that you might need separately but instead import what is needed when it is needed. That way you can quickly get a good idea down into java code quickly. On the other hand, some people may use on-demand import purely for laziness reasons which is a less convincing motivation. The downside of using on-demand import is that it could create incompatibilities if names clash later on. When using the standard Java

packages, that is not likely to occur very often, but when writing GUIs you often go beyond the standard packages. You often utilize `java.awt` and `java.util` packages in the same class which is a reason to be more careful (*Packages* 2006). For these reasons together with the clearer overview you get by explicitly writing out exactly those packages being utilized, on-demand import has not been used. In addition, since the Eclipse developing environment was used throughout the coding process and it automatically detects what classes are missing, explicitly importing only the needed classes as they came along was just a mouse-click away.

Another good reason why on-demand import was not used was because of the way the Custom Java API was implemented in the middle layer. Whenever a component in the XML data model could contain multiple sub-components of some kind, the Custom Java API provided access to these sub-components by returning an `Iterator` from the parent object. For reasons beyond the scope of this thesis, that `Iterator` had to be extended to provide more functionality than the interface `java.util.Iterator` found in the standard Java API. (The `Iterator` in the Custom API had to provide a non-incremental `next()` method for instance.) Since the `java.util` package contains many other common classes, such as `Vector`, it is frequently used and with on-demand import of the entire package an ambiguity would have arisen between `java.util.Iterator` and the project specific `infineon.util.Iterator`.

When deciding upon using on-demand import or not, it should be noted that since import declarations do not actually import anything into a Java program (as opposed to the C-style `#include <foobar.h>`), any difference in compile time is very small. According to an experiment using thousands of class names it showed only a negligible change in compilation speed. So compilation performance should probably not be a factor to consider when choosing import style. (*JDC Tech Tips* 2000)

5 XML Data Model and Custom Java API

As more thoroughly described under *Delimitations* in chapter one, different members of the project group were responsible for different parts of the system implementation. The implementation of the data model and the Custom API were outside the responsibilities of this author so this chapter will not go into implementation details. But since the data model and the Custom API were sketched out and defined collaboratively by all group members and because they were an intrinsic part of the GUI, presenting them from a high-level perspective is still most relevant and of interest for understanding the whole system.

5.1 XML Data Model

The work of defining and implementing the XML data model started in august 2006. At the moment of writing, the data model is still changing. New requirements for the data model have continuously arisen throughout the project as the software system matured. Creating the data model was a dynamic and iterative development process which is likely to continue as the project carries on.

The data model describes chip design specifications. These specifications are called SoC - System on Chip, which is an integration of a whole computer system on a chip. An SoC is more complex than the earlier discrete circuits and usually contains its own processor.

The System on Chip is a hierarchical structure of sub-systems. Leaf nodes of this hierarchy are IP Blocks. IP Blocks have a clear interface via ports to the SoC, and they provide some functionality. They can be viewed as small sub-systems or sub-solutions. A USB-interface is an example of an IP Block. Basically, IP Blocks are a collections of registers, memories and combinatorial logic.

The XML data model specifies the formal structure of these SoCs and IP Blocks. It stores the parameters and attributes of all the chip components that together form a chip solution. Additional informal information about what the system does is also stored in the model. The latter cannot be automatically used by computers, however, since it is made up of informal descriptions. It contains normal language, such as “This IP Block calculates the Fast Fourier Transform (FFT)”, and is only be read and understood by humans.

5.2 Results of the Data Model Implementation

The question of how the XML data model should be specified was a collaborative issue within the project group. Influences from other parts of the project, such as the GUI, the Custom API, and the informal chip documentation, all affected the requirements of the underlying data model. Sometimes the implementation of the GUI had to be changed to accommodate the data model - sometimes the data model had to be re-defined to meet the requirements of the GUI. The specification of the data model was an integral part of the whole project and it affected the entire software system being built. Therefore it is worthwhile to present some important results of the XML data model that was developed. Since the actual implementation of the data model into XML code was done by another person in the project group we shall skip over any implementation details here. The data model is presented as-is in Figure 10 (and in larger format in Appendix A) and we proceed by looking at some of its more prominent features.

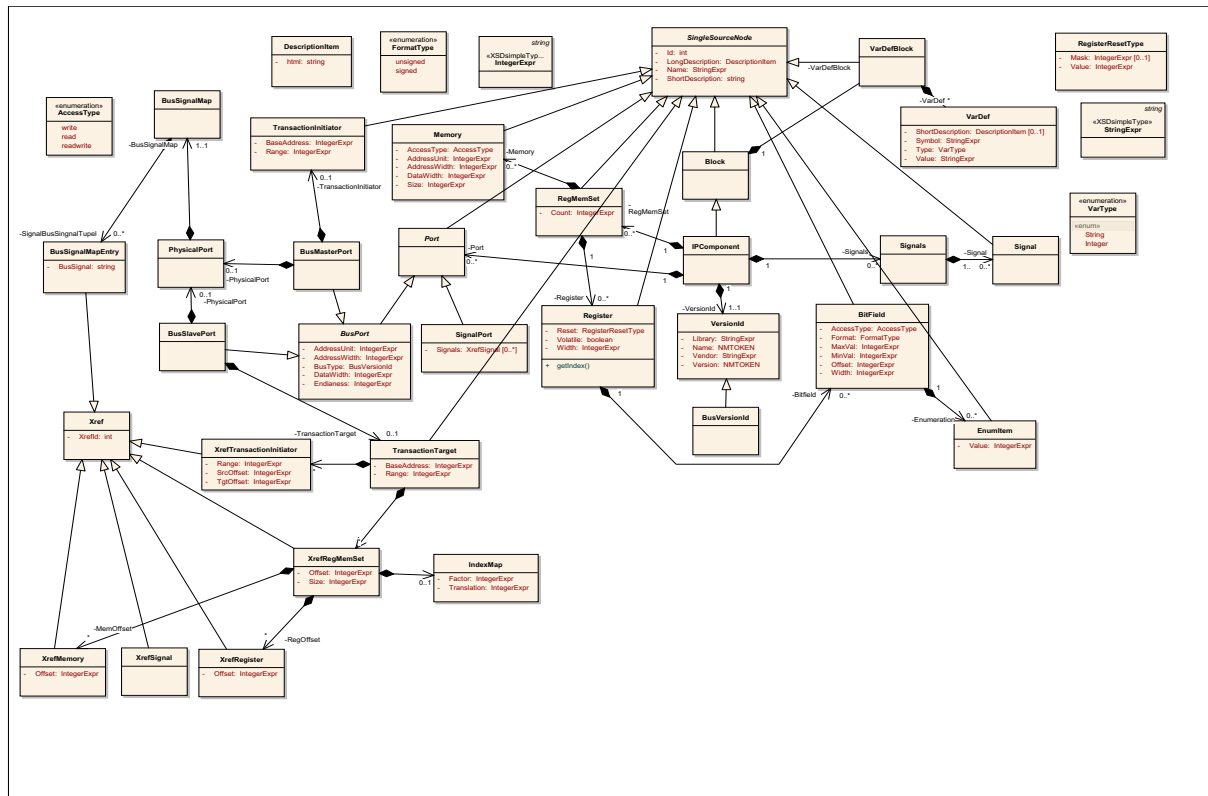


Figure 10

5.2.1 Object Instantiation and Detachment

As opposed to the SPIRIT data model discussed earlier, our data model provided the possibility of generating parameterized register or memory instances. In SPIRIT, sets of registers or memories were not supported nor part of the data model. In our model registers and memories could be combined arbitrarily into so-called register-memory-sets providing the ability to count and iterate over lists of registers or memories. Since references could be provided to detached and self-contained objects, the same register or memory could be instantiated and used in multiple places in the data model. With SPIRIT, one would have had to duplicate identical registers and memories since they were not detached from their parent objects and had to be created anew in every instance.

Furthermore, our data model could map arbitrary subsets of registers or memories to ports. What that means is that out of all available registers or memories, each port could be linked to an arbitrary subset of them. If, for example, there are 10 registers then one port can have a subset of five of them, another port can have two of them, and a third port might have all 10 of them. In our model, this was done by referencing existing registers or memories from within the ports, instead of defining registers as part of the address space related to a specific port. Semantically, our solution was not expressible in the SPIRIT standard.

5.2.2 Arithmetic String Expressions

Another advantage in our model was that it was not restricted to literal parameters. Instead of having literal parameters for specification parameters we had arithmetic string expressions. By introducing variable definitions into the data model, these expressions could also contain predefined variables. As an example, this enabled expressions of the form “ $3 * \text{variable1} + 14$ ” to be assigned to the bit width of a register. In SPIRIT every such assignment had to be a literal constant.

5.2.3 Mark-up Language Documentation

The data model also had more sophisticated support for informal documentation of the chip components. The informal documentation is normal “prose” being written and read by human users of the system. Instead of just having a simple raw text description field, the data model enabled XML based documentation conforming to the HTML standard for tags. The documentation was entered by the user through an HTML editor that was developed and integrated in the GUI. Thanks to the mark-up language based documentation we could use HREF links to create references to other chip components. Within the textual description, formal and structured data belonging to other components could be referenced wherever it was motivated by some relationship. (The references could be created directly in the GUI and by following a reference, the GUI brought up the component being referenced.)

The HTML based documentation also meant that formatting of the text such as superscript, bold font, etc., could be stored in the data model. These typographic possibilities for the documentation helped the design engineers to better express their descriptions. By storing all text formatting in the data model, it remained available at later stages when documentation was generated. As a result, more informative and appealing documentation could be generated for the chip components.

5.2.4 Many-to-Many Relations

The only major problem that was encountered as a consequence of using XML to realize the data model was that XML cannot be used to express arbitrary relational data models. Data is strictly hierarchical in XML. In a relational database you can have $m:n$ relations, meaning that several entities can have relations to several other entities. In a hierarchical data structure you only have $1:n$ relations, meaning that while one entity can have relations to several other entities it can only be referenced itself by one single entity. In XML there are only relations between a parent and its children in a graph presentation. We needed to be able to have relations between any components in the data model as is possible with a relational database.

Because of this inherent problem with the hierarchical structure of XML, it was necessary to find a way of referencing arbitrary elements other than what the data model itself provided. The strictly hierarchical data model was insufficient since there was a subset of elements which had to be able to relate to elements of another subset. A completely general solution, in which any element could reference any other element, was not needed however. That would, in fact, have been an unnecessarily powerful solution with overhead in the data model as a result. To restrict the cross-referencing of elements to only those cases where it was actually needed was also positive from a usage perspective since it would help prevent end-users from creating irrational references by mistake.

The solution was to part the set of elements in the data model into two disjoint subsets where relations only needed to exist between the subsets and not between elements in the same subset. The elements in one of the subsets acted as sources and in the other subset the elements represented targets. The targets had ids and the sources had cross-reference ids. To create a relation, an element from the source subset would get the same value for its cross-reference id as some other element in the target subset had as id.

In the data model the elements in the target subset all had unique ids and could not be related to each other or to elements from the other subset. The elements in the source subset could have

any valid id value for its cross-reference id, and so we had created a *:1 (many-to-one) relation between the two subsets.

One issue of having such relations between nodes in the data model was concerning deletion of elements. Since the cross-references were not part of the hierarchical data structure itself, but instead implemented by means of XML attributes, a mechanism for handling the references when elements were deleted also had to be implemented. If an element in the target subset was deleted, then every element in the source subset that had a reference to that element had to be removed. The existence of source elements with a corresponding reference id had no function or meaning if the target element was deleted. If an element from the source subset was deleted no other element had to be deleted from any of the two subsets since the effect of deleting a source element only meant that the relation was removed.

The difficulty with the implementation came from the fact that relations are always directed. The relation (a, b) does not imply a relation (b, a). As a more tangible example; just because Bob likes Denise it does not necessarily mean that Denise likes Bob back. So when a target element was deleted there was no direct way of knowing which source elements with affected relations had to be deleted. One solution would have been to search among the source subset to find elements that had to be deleted, but such searches would obviously have made the data model inefficient. Instead the target elements were augmented with a list of reference to all source elements that had a relation to it. That way, source elements could easily be removed when a target element was deleted, but on the other, the reference list also had to be maintained so that if a source element was removed the list under the target element was shortened by one and kept up-to-date.

5.3 Java Custom API

As mentioned earlier in chapter 4.3 it was decided that Java and the JAXB package should be used to build the layer between the XML data model and the GUI. The implementation of the Custom API by means of JAXB was made by another person in the project group and so we will not go into specific implementation details here, but instead present a high-level overview of what the Custom API is and how it was used.

5.3.1 Purpose of the Custom API

What we refer to here as the *Custom* (Java) API has nothing to do with the official Java™ API as distinguished under Definitions in chapter 1.5. The Custom API is the package of Java code that was developed within the project to act as a layer above the XML data model. Users never manipulate the data model directly, but instead use Java classes and methods in the Custom API to communicate with the model and to populate it with data. Most users would utilize the GUI in their work, and in that case the GUI used the Custom API to access the data model. In some cases however, a user might want to write his or her own programs or to automate some kind of generation based on the information in the data model, and then the Custom API could be imported as a package in other Java applications.

5.3.2 Extending JAXB

In a simpler situation with a small and uncomplicated data model, JAXB could have been used without modification to generate the Custom API. JAXB would automatically create Java classes containing variables and access methods corresponding to the elements and attributes in the XML files. Marshalling and Unmarshalling would then be used to send and

receive data between the Java classes and the XML data model. Unfortunately, the situation at hand and the requirements of the data model and the Custom API were too complex for the JAXB package to be used in its original form. It had to be modified and extended.

There were several reasons why JAXB had to be extended. One example was the issue with the cross-references as discussed in section 5.2.4. JAXB did not understand the relation between cross-reference ids and real element ids. If a cross-reference id attribute matches another elements id attribute that should impose a relation between the elements, but JAXB could not handle that without modification.

Another reason why JAXB had to be extended was due to the arithmetic expressions, as described in section 5.2.2. In the data model a string expression is nothing more than a string. In our case we had additional semantics that the string had to be checked against. The string expression had to be handled both as a string and as an evaluated value by the Custom API. In fact, there were 3 different types of expressions that all had to be handled by the API: HTML expressions, string expressions, and integer expressions. Derivation was performed when expression values were set and evaluation was performed when the interpreted value was asked for. The derivation proves that the string is part of the context free language by checking against a context free grammar (CFG). The derivation tries to construct the string using the rules from the grammar and if that succeeds then the string expression is correct.

There was also some special code (called recursive descent parser) that needed to be called by the generated API. Without extension, JAXB would normally never call such code.

After the appropriate extension, JAXB could be used to generate the Java code for the Custom API. The API had all the necessary methods for creating new chip components, modifying their attributes, creating references between components and communicating with the data model. With unmarshalling entire IP Components could be read into java objects from the stored XML files, and conversely newly created or modified objects could be written back into the data structure using marshalling.

JAXB has a plug-in mechanism for extending the package which made the extension easier.

6 Designing the GUI

In the core of the system the XML data model served as data structure for storing the chip components. Surrounding the data model, the Custom API served as communicative gateway for any application utilizing the data model. As a topmost layer and necessary tool for human users of the system a graphical user interface was developed in Java using the Swing graphical toolkit. This section describes some of the more interesting features of that GUI. Since all the coding of the GUI was done by the author of this report, this section contains that Java implementation in more details.

6.1 Design Challenges and User Requirements

As noted in chapter 3, a user requirements analysis was performed early in the project in order to get a picture of what the typical user wanted from the GUI. One of the biggest challenges when designing the GUI was to try to anticipate how the user requirements and preferences would change as the data model grew and new ideas for functionality emerged. In the early stage when the rough drafts for the GUI had to be made only a small part of what the GUI would later have to encompass was known by the project team, and hence the end users could not be asked how they would want everything to look and function. By adopting an iterative development process, new requirements and user preferences could be integrated in the solution, but some crude assumptions about the structure and general layout had to be made. These fundamentals had to be decided to carry the system forward and preferably they should not have to be re-made later on. The fundamental requirements could be separated into end-user and structural requirements.

6.1.1 End-user Requirements

Based on the early user requirements analysis together with estimations of what complexity the data model would finally reach, some fundamental requirements could be identified. For one thing, all the components belonging to some IP Block had to be visible and selectable. Secondly, new components of various types had to be able to be added and deleted in the component hierarchy. Thirdly, when selecting a specific component, an editing environment for that component type should be available for modifying all its attributes in a convenient way. The editing environment should include some editing tool for writing informal descriptions of components. The components, their attributes, and all their modified values should reside in the data model and the GUI should only communicate with it through the Custom API. The state of the components that the user worked with in a certain project needed to be saved to the hard disk so that it could be reloaded later on by opening the project files.

6.1.2. Structural Requirements

Other requirements besides those of the users also had to be taken into account. The company had some guidelines as to how the GUI should be implemented. First of all, the code should be separated into logical units that would make it easier to extend the GUI in the future. It should also be grouped in such a way that local modifications could be made without having to change more classes than necessary.

6.2 Setting up the Layout

One of the first things to be done in the GUI was to decide how the application window should be partitioned. When the contents of the application window, or a part of the window, becomes too big scrolling is necessary. Generally, horizontal scrolling should be avoided in a GUI in favor of vertical scrolling.

Somewhere in the GUI the object tree had to reside. Dynamic trees primarily expand vertically in Swing. Although they might also expand horizontally to some degree when a node is expanded and long descriptions are revealed in lower levels, the biggest size impact is vertical. Furthermore, vertical expansion is usually more critical than horizontal since horizontal expansion usually just means that the text label expand out of view whereas the node still remains selectable. A fully expanded tree takes up one vertical row unit for every node in the tree. In the case of 16x16 pixel icons, an expanded tree with n nodes would require $16*n$ pixels of vertical space. A tree with 50 nodes takes up more than the entire computer screen in a 1024x768 resolution.

To assign as much vertical space as possible to the object tree, it was decided to dedicate the full vertical height of the application window to the tree. Trees are often placed to the left on the screen, as is the case with most file system browsers for example. To make the average user feel familiar with the user interface and be able to grasp the functionalities in the layout quickly, the tree was placed on the leftmost side of the window. Since the tree did not require any extreme amounts of space horizontally, the window was split once along a vertical axis more to the left on the screen. That way the right part of the window, which was given to the editing environment of components, would have as much horizontal space as possible. Since the editing environment had to handle a vast amount of component input/output features, horizontal space was valuable.

A small area to display information to the user from the system was useful. It should be used for notifying the user of such things as what filename might have been accessed, that a component had been copied to the memory, or if a user tried to perform a faulty action such as trying to paste a sub-component where it did not belong in the data model. The nature of these textual pieces of event information made them ideal to put in a wide area with small height. Furthermore, that information was normally not vital to the user so it did not need to be right in the sight of the user. It could be placed in a less exposed part of the GUI and looked for if the information was needed. A status panel displaying the event information was inserted below the right editing environment panel by doing second layout split horizontally. The editing panel was likely to contain so many GUI features that it would have to scroll vertically anyway, and therefore the small amount of vertical space lost to the status information panel was negligible. The basic layout of the three main panels called *Component Selector*, *Component Editor*, and *Event Viewer* is shown in Figure 11.

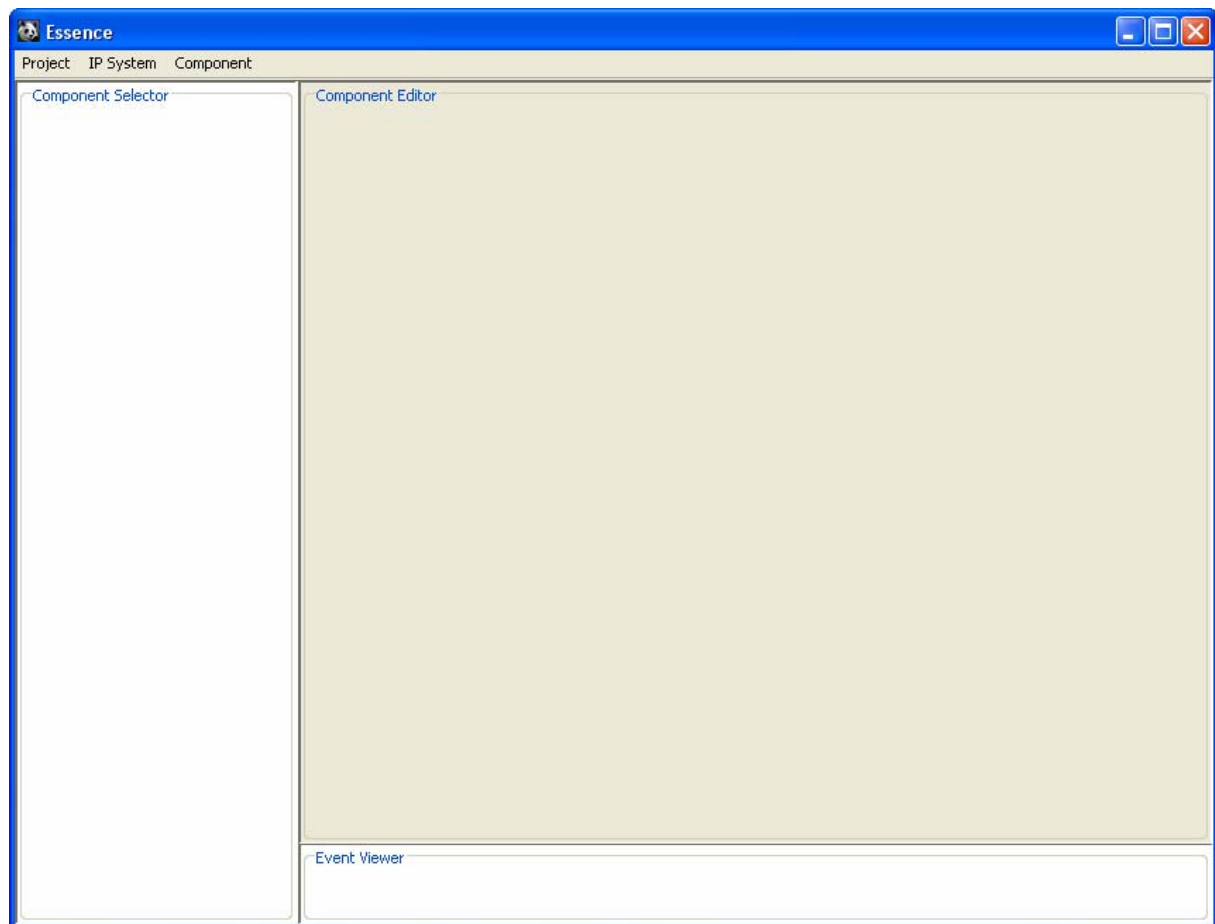


Figure 11

To navigate through components the mouse should be used to point and click in the object tree or when needed on special areas inside the editing panel to the right. Changing attributes or other information for components should be able to be done by typing into fields or using the mouse on special tools in the editing panel. High level functionality such as opening new files, saving the project state etc., should be done via a standard menu at the top of the screen. Other functionalities and commands, such as adding new sub-components, copying or deleting a component, should be available both as commands under the top menu with short keys and as pop-up-menus by right-clicking components in the object tree. (As stated in chapter 1.4, menus, short-keys, and the handling of standard mouse-clicks are not further discussed in this thesis.)

6.3 Dynamic Object Trees – *JTree*

The first step after having divided up the window into the three main panels was to implement the dynamic object tree inside the Component Selector panel. The tree should visually represent the hierarchical data and enable single selection of its nodes. The tree was dynamic in that it would constantly change as new components were added to or deleted from the hierarchy.

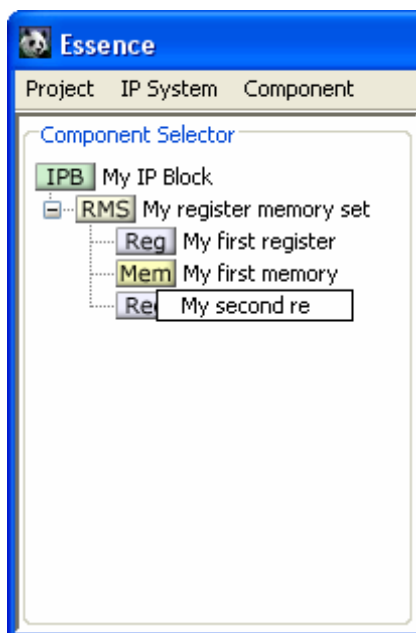
Fortunately Swing provides a suitable class for displaying dynamic trees graphically based on hierarchical data. Unfortunately this class can only be used without extensions if the objects in tree are very simple. In fact, the simple example provided by the Java Tutorial (*The Java™*

Tutorials:10 2006) only works satisfactory when the object is a string with an accompanying icon, as in Figure 8.

The reason behind this limitation comes from how the default tree cell renderer and editor are implemented. They only support the editing and rendering of strings (except for the rendering of the icon). The rendering of an arbitrary object into a textual string following the icon is easy enough to work around though. By overriding the `toString()` method of the object it will display the string provided by this method in the tree. Handling the editing of tree cell is more tedious.

`JTree` supports editing the name of an object in the tree. By pressing F12 for example, textual editing of the selected object in the tree will start. After completing editing the text, the object wrapped inside the selected tree cell will be replaced by this text as a string object, regardless of what kind of object it was originally! After such a name change, it is most likely that a `ClassCastException` will occur later on if something other than a string object is expected to be returned by the tree class. The reason why this switch of object type can occur without raising any exceptions immediately is because how the objects are wrapped in the tree. `JTree` uses a class called `DefaultMutableTreeNode` to wrap the objects. This class takes an object of type `Object` which is why it can be replaced by anything, `String` objects included.

So for the more complex object types that were used in this system to function in the tree the editor, the renderer, and the class that wrapped the objects had to be extended. Although this extension was quite straight forward and did not require too much coding effort, care also had to be taken to handle the icons. The extended classes must set the right icons otherwise they will not be shown at all or disappear when editing of the name of the object.



After having these extensions in place, providing custom icons to the component objects was also possible. Each component type was given a color coded icon so that different component types could easily be distinguished and perceived visually.

Figure 12 shows the component tree whilst the last object being renamed. (After enter is pressed the new name is immediately sent to the data model and the component is renamed.)

Figure 12

As mentioned, one requirement was that components in the tree had to be able to be copied. Preferably, that should imply that the whole sub-hierarchy from that component and below should be copied. A copy of the sub-tree had to be placed in memory for later pasting where the user wanted it. That functionality was not trivial to implement.

It was decided that the actual duplication of the sub-tree should be done by the Custom API, since this functionality could be wanted from scripts or other applications not connected to the GUI. The API thus provided a method that recursively copied and returned everything under a certain component. The problem for the GUI was that the objects inside this sub-tree were not wrapped into tree nodes by the class that `JTree` required. It was a component hierarchy consisting of only those classes defined by the data model. This had to be solved inside the GUI by traversing the sub-tree copy received from the Custom API and wrapping each component by the extended wrapper class and creating node connections between the wrapped parents and children. Only with this (to `JTree` somewhat anonymous) hierarchy based on nodes and parent-child relations could `JTree` efficiently handle the tree structure since it knew nothing of how to handle the diverse classes returned by the Custom API and the non-uniform relations between different component classes. Since the sub-tree copy from the API would contain different component classes based on what was in the original, the recursive wrapping could not be done in the same way throughout the copy. For example, if the object being copied was a register-memory-set, the sub-tree would contain register and memory objects. If the object being copied was a variable definition block, it would contain variable definition objects. So for each step in the recursion the object at that level inside the sub-tree copy had to be checked for its class type to know which child classes to ask for a reference to, and then wrap those children.

6.4 Component Editor

With the object tree functional in the Component Selector panel, objects could be created, copied, and selected. When selecting an object, the Component Editor opened up an editing environment tailored for the type of component currently selected. In this panel the main work of the user was performed and it held the most advanced GUI features. Many different input/output elements had to be implemented in the Component Editor to reflect the data model appropriately. Text fields, radio buttons, drop-down lists, as well as other much more advanced user interface elements, were combined in order to provide an intuitive and efficient editing environment. We will not discuss simple Swing features such as radio buttons or text fields, but the more advanced parts of the Component Editor will be outlined throughout the rest of this chapter.

All input to the Component Editor was sent to the data model immediately upon completion, without the necessity of pressing any buttons for the changes to take effect. The input was considered finished either when the user pressed the Enter-key (in single-line fields) or when the cursor or focus left the input field (in single- or multi-line). The idea behind this approach was to speed up the chip design process. When designing an IP Block a plethora of input fields has to be filled out and having to press a “save” or “update” button for each sub-component would have wasted a lot of time. Changes could always be undone later on by keeping the XML files of the data model in the RCS and reverting to older versions.

6.5 General Input Fields and HTML Documentation Editor

There were some attribute fields that every component object in the data model contained. These fields were: *Name*, *Description*, and *Short Description*. In order to avoid duplicating code, as well as making it easier to extend the GUI to handle new component types, these common fields were separated out and put at the top of the Component Editor.

As mentioned earlier the Description-field, which was used to write informal documentation about components, should support formatting of the text and reference functionality. To provide that, a simple HTML editor was built into the GUI. With the editor, the text could easily be formatted by the user, and references could quickly be created by clicking the linkage button at the top right and then selecting a component in the tree to the left. Figure 13 shows the general input fields and the HTML Editor positioned at the top of the Component Editor.

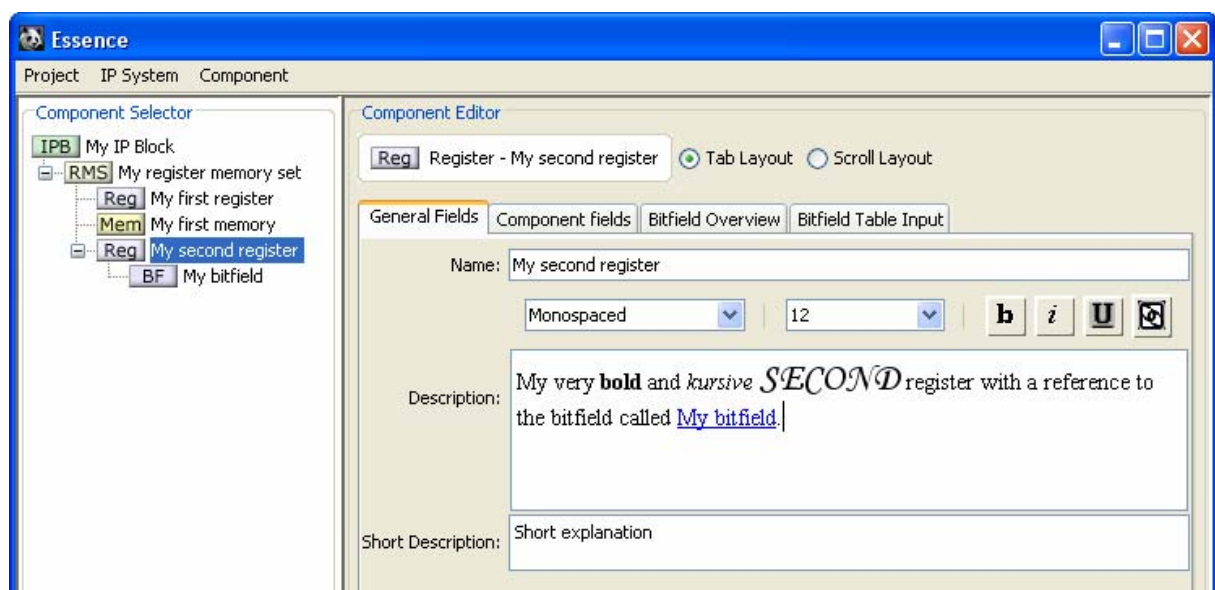


Figure 13

6.6 Dynamic Editing Tables – JTable

In some cases where a component might contain a large number of sub-components, it was desirable to have a quicker and more convenient way of editing the sub-components than having to select each one in the tree and then edit it. For that reason, dynamic tables were implemented as a part of the editor for some component types. These tables never modified any attributes of the parent component to which it belonged in the GUI. Instead they were used to edit and add sub-components of a different component type under the selected point in the hierarchy. The rows of the tables corresponded to all present child-component residing under the selected parent, and the columns corresponded to the attributes defined for that child-component type in the data model.

For the purpose of implementing the dynamic tables, Swing's `JTable` class was used. Although `JTable` works very well for showing and editing simple data types, the cell editor and renderer of the table model had to be overridden for more complicated data types, just as in the case with the dynamic object tree. For strings, numbers, and even booleans, `JTable` has good built in support which automatically provides nice layout and functionality in the table cells, but for more complex objects the editor and renderer has to be extended and

implemented specifically. One case in which extending the column type functionality was needed was with enumerations. Figure 14 shows how the column called “Access Type” was implemented to properly handle the enumerated list of options for that attribute by means of a selectable drop-down-list.

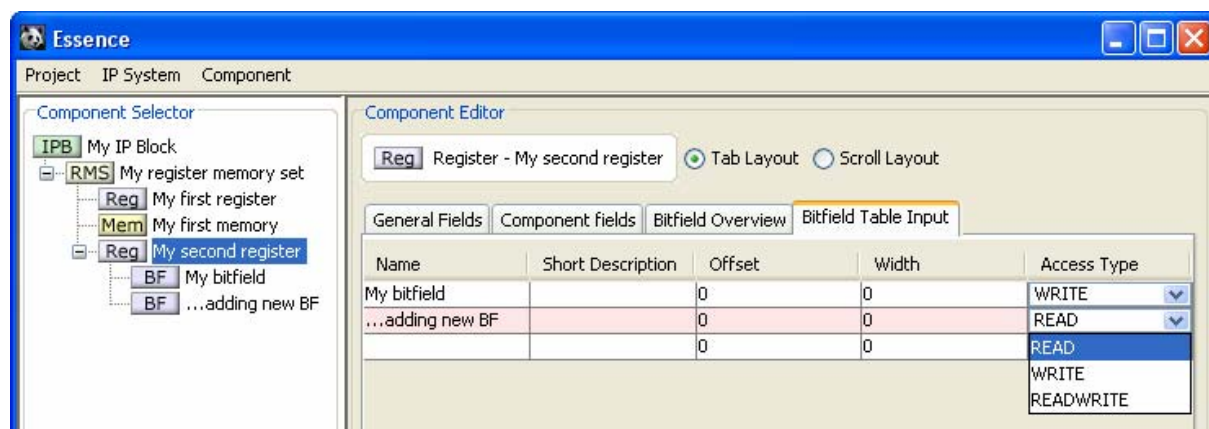


Figure 14

By simply typing in a new value for the name column in the last row, a new sub-component was automatically added and the table extended by another row. In Figure 14 this has just been accomplished by typing in the name “...adding new BF” in the highlighted row and the new bitfield object has appeared in the component tree to the left as well.

6.7 Building Customized Graphical Tools - Extending JPanel1

As the data model grew and the component specifications became more extensive, the demand for some more sophisticated editing tools than just fields or tables came up. On the highest level, where the user worked with IP Systems, a tool for connecting IP Blocks to busses was needed. In the case of the register component, a tool was requested for managing how bitfields residing under a register were placed out in terms of offset and bit width relative to the register.

6.7.1 Bitfield Overview Tool

The idea was that the tool should provide a way of efficiently adding new bitfields and changing the bit width and offset, while giving an easy to grasp overview of how all the bitfields overlapped in the register’s memory space. To provide this overview without the user having to compare offsets and bit widths numerically, a graphical tool was implemented as opposed to a normal table. Figure 15 shows how the Bitfield Overview Tool looked. At the top of the tool a row showing the bits of the register gave a graphical overview of the reset value of the register. By clicking on the ones and zeros in the row, the bit value of that particular bit position was inverted to give a new reset value. (The register reset value could also be filled in directly as an attribute in binary, decimal, or hexadecimal form in a normal input field.)

Figure 15 and Figure 16 shows how a new bitfield could be created simply by marking an area in the blue colored field with the mouse.

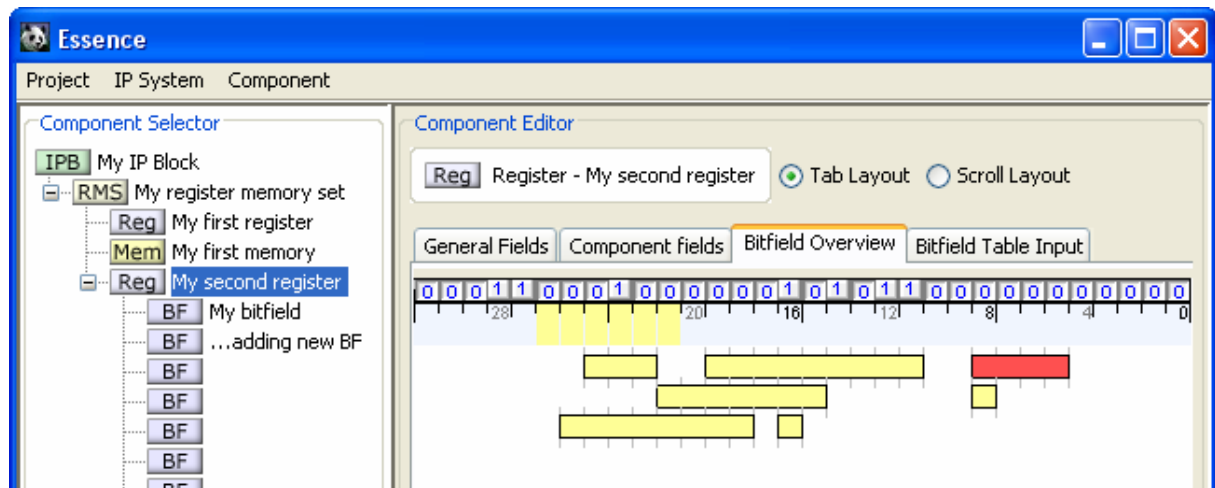


Figure 15

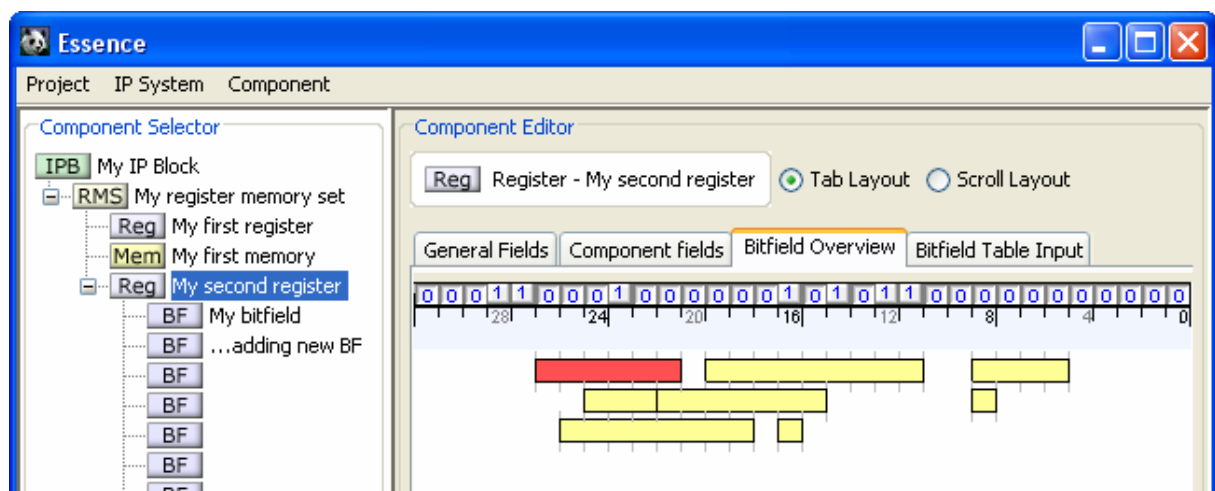


Figure 16

In the first figure the yellow marking in the blue area is where the user wants the bitfield in terms of register offset and bitfield bit width. The second figure shows how such a bitfield has been created at that position immediately after the user released the mouse button. The red color tells which bitfield is the currently selected one. Notice how the visual layout of the bitfield arrangements has changed in the second picture. This is due to how the vertical placement of overlapping bitfields was calculated by means of an ordering algorithm developed for the graphical tool. The bitfields that overlap were ranked according to smallest offset, largest bit width, and earliest creation time, in descending priority. The one with the highest rank took the highest position in the window. The bitfields were all rearranged when necessary. The vertical placement only had an aesthetic meaning in the GUI and was not stored in the data model.

The bitfields could also be moved sideways and thereby changing the bit offset. By pressing and holding down the mouse over a bitfield, the user could move the bitfield to any offset position in the register. The size of the bitfield could also be changed graphically. By placing the mouse over any of the two ends the user could drag one side of the bitfield and thereby changing the bit width when dragging the left side, or changing the offset and bit width when dragging the right side. The effects were transferred to the data model when the user released the mouse button, which also trigger a vertical rearrangement of the bitfields if necessary.

The dynamic bitfield table and the graphical tool in the register components enabled the user to modify bitfields in of three different ways. Changes could be made by using the mouse in the graphical tool, by entering values in the table, or by selecting a particular bitfield component and entering the values in its attribute fields.

The Bitfield Overview Tool was made by extending the graphical area of the common `JPanel` class. In order to work well with the layout in a GUI, a custom painted `JPanel` should return reasonable size information. In particular, if the component changes size dynamically with user input, these size changes need to be handled. Specifically, you should either override the `getMinimumSize`, `getPreferredSize`, and `getMaximumSize` methods or make sure that your component's super-class supplies values that are appropriate. If you do not want to override these methods, the corresponding `setMinimumSize`, `setPreferredSize`, and `setMaximumSize` methods can be used to update the super-class' size values.

6.7.2 System Composition Tool

The other graphical tool that was developed was the System Composition Tool. It was used to graphically layout IP Blocks and buses and to create connections between them. The connections were made between the available ports of the IP Blocks and the buses. Figure 17 shows the System Composition Tool in a simple case with three IP Blocks (without any ports) and two buses.

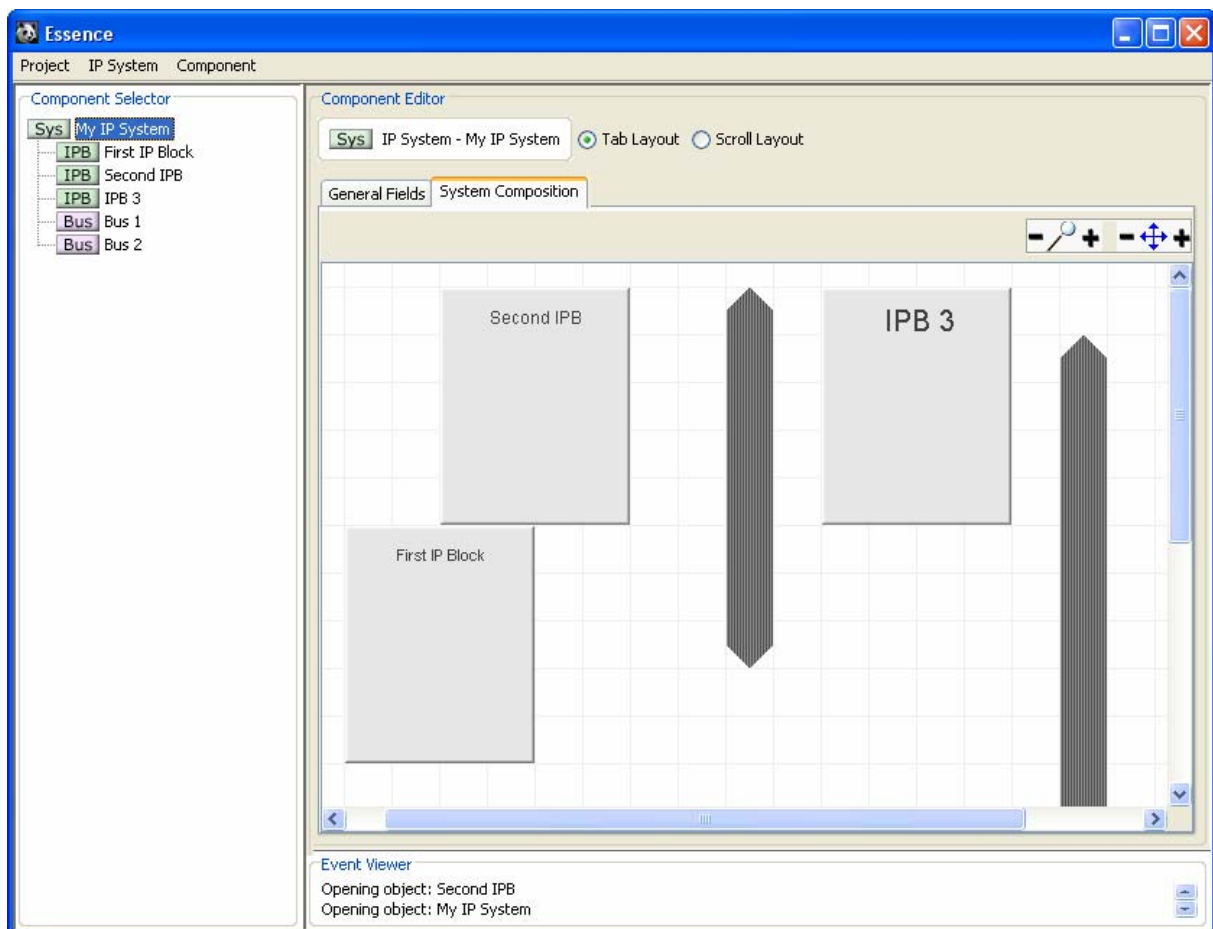


Figure 17

The Bitfield Overview Tool did not need to save any graphical layout information. All that was needed to perform the algorithm that arranged the bitfields vertically was available in the data model. The System Composition Tool however needed more information. When placing out components graphically, a situation was created where some information had to be saved that did not belong in the data model. The data was as such things as where components were placed on the grid, how big the background grid should be, how much the view was zoomed in, etc.

To handle the graphical state data and enable the user to reload previous system composition layouts, graphical state classes were created. These classes only held graphical information such as X and Y placement in the grid to be as light weight as possible. By then serializing these classes, the graphical state of the entire project could be stored and reloaded from the hard drive by using Java's built-in serializability feature.

6.8 Input Groups

The commonly shared fields described previously in section 6.4 made out a logical group of input objects. They belonged together because they were shared by all components and hence, would be displayed frequently to the user. For increased usability these input objects should appear at the same place within each component and in the same order, so that they were easily distinguishable. That way the user could quickly edit them or disregard them, depending on where in the component design process he or she was.

Input tables with their sets of cells, column types, and functionality, could also intuitively be considered to be a logically separated group of input. In fact, all the input/output objects within the components GUI could favorably be grouped together based on similar characteristics. They could, for instance, be grouped by hierarchy/locality (as with variable definition tables) or by functionality (for example the graphical tools). These logical separations of input objects will henceforth be referred to as *input groups*.

The purpose of the input groups, however, was not merely to graphically group GUI elements that belonged together in the layout. It was also a way of separating classes and handling multiple layouts.

6.9 Multiple Layouts

As the GUI progressed, it became apparent that the space required to display the entire Component Editor could potentially be very large for some component types. Some input groups contained many elements, some grew dynamically with new user input, and most components contained several input groups. Then there was also the consideration of the graphical tools. Although the graphical tools could very well be bundled together with the other input groups, a minimum and maximum size had to be set for their designated area. If too small, you would not be able to start drawing in the tool, and if too large it would take too much of the valuable vertical space from the other input groups.

To decide how valuable a graphical tool is to the user, and analogous, how much space the tool deserves is a difficult question for a developer to answer beforehand. Some users might prefer to do all the work with the graphical tools, whereas others might prefer to do the same work using an input table. As described earlier that was the case with editing of bitfields

under the register component. Another issue is that if you allow the graphical tool to expand dynamically in the layout, it makes the rest of the interface below the graphical tool to jump up and down. In the case of adding new components by means of a table, it would result in the user clicking in a cell which is immediately moved downwards because of the vertical expansion of the affected graphical tool as the new component gets created, leaving the freshly clicked mouse pointer over some other cell at a higher row in the table. For these reasons it was decided that it would be better to lock the vertical area given to the graphical tool at some reasonable height, and use additional scroll bars to navigate within that area. The fixed height was different for each graphical tool and was decided upon how much space would be needed in an average case, without stealing too much space from other input groups.

Due to the issue of the limited vertical space in the GUI, an alternative way of working with input groups that used much space was requested. In the normal layout all the input groups were stacked after each other vertically inside the Component Editor. This layout was called the *Scroll Layout* and the user simply used the scroll bar to navigate up and down if the input groups extended beyond the height of the window. The alternative way in which the user could navigate within input groups when a large working area was preferable, was given by introducing a second layout called the *Tab Layout*. In the Tab Layout, only one input group at a time was displayed, allowing it to fill out almost the entire editing area in the right panel. Figure 17 is an example of this, where the graphical tool is the only input group shown, and other input groups such as the general fields are hidden behind tabs at the top of the screen. The user could switch between the tab layout and the scroll layout by clicking the radio buttons at the top of the Component Editor. Figure 18 shows the scroll layout for the register component.

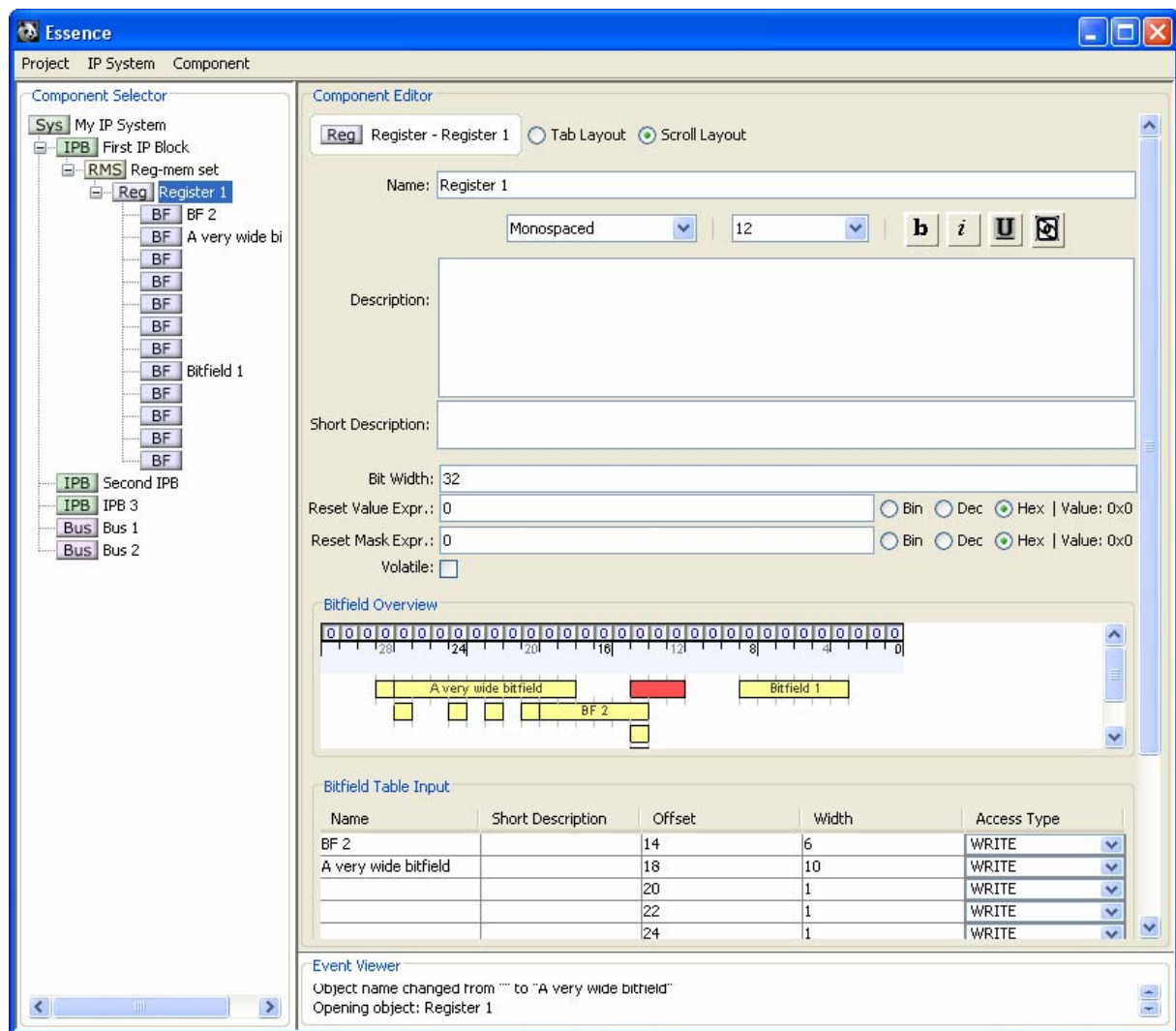


Figure 18

6.10 Class Separation and Scalability

One important design decisions was that the software system had to be extendable in the future. The implication of that, and the fact that each component was different from the others, was that the editor interface of each component should reside in a separate object. This helped ease addition of new components, as well as addition of new input fields to existing components in case new attributes were added. However, to end the class separation at the component level would not have been enough. Input groups could also be shared by many components, as with the general fields input group, and these groups should naturally not be created separately for each component object. Because of that, and again in order to ease future addition of new shared fields, the general fields input group was separated into its own class and was instantiated in each components top level GUI class.

6.10.1 Input Group Return Values

Separating the editing interface of the different components types into separate classes meant that each such class had to deliver its output in some unified way to the Component Editor. If we were to disregard having two separate layouts, one could have simply stated that the returned result from each component type layout class should be a `JPanel` that covered the editor area. A `JPanel` is a container in which GUI components can be grouped and laid out

according to various layout schemes. That would have worked well for the *Scroll Layout* since we could have simply packed all the input groups after each other in a panel and displayed them. However, because of the Tab Layout, putting everything inside a `JPanel` would not have been satisfactory.

Some input objects belonged inside a `JScrollPane` so that they could have their own local scrollbars. Examples were the multi-line text input fields and the graphical tools. The multi-line text input obviously needed to scroll to accommodate an arbitrary number of text rows being entered, and the graphical tools also needed to scroll since they had an area where component design was being made and that area had to be able to expand (or contract) as the graphical schemata were being altered. One might argue that a good simple solution would have been to wrap the `JScrollPane` in a `JPanel` to achieve the common return object. That would not have been a great solution though, since first of all, it would have meant a rather unmotivated wrapper. To simply wrap a scroll pane in an empty panel really does not make much sense other than for the sake of compatibility.

Secondly, the input groups should not look the same in the two layouts. In the scroll layout where all the input groups were laid out in a long column, some input groups should have a border and a title to distinguish which fields belong together and what their purpose was. Some components had multiple input tables for addition of subcomponents. These tables should be separated clearly and the border title could be used to tell what kind of components the table contained. In the case of the *variable definition block* components, the titles were simply “Local Variable Definitions” and “Global Variable Definitions” (see Figure 19). Without these titles the user would have had to analyze the two tables and spot some difference, such as that one was editable and the other was not, in order to decide which one was which.

Type	Name	Value	Go to
STRING	var1	abc	Variable Definition Block 1
STRING	var2	def	Variable Definition Block 1
INTEGER	var3	123	Variable Definition Block 1

Type	Name	Value	Action
INTEGER	local var 1	88	Delete
INTEGER	local var 2	99	Delete
STRING ▼			

Figure 19

Clearly the possibility of having a border with a title was desirable in some cases in the Scroll Layout. But what about the Tab Layout? Did the same rational apply there? It did not. First of all the border was unnecessary because the logical place where the border went in the Scroll Layout became the outer edges in the Tab Layout. Another border directly inside the tab boundaries would have served no useful purpose and merely been overkill that would have worsened the layout. Secondly, the title of the border is also useless in the Tab Layout since the title is already written on the actual tab.

The conclusion was that because of the existence of both the Scroll Layout and the Tab Layout, a more sophisticated wrapper than a `JPanel` was needed to help handling the Input Groups. This led to the introduction of a new class called `InputGroup` as a wrapper for the contents of an input group. The class could handle the getting and setting of borders, titles, scroll panes, and the core panels, as well as serve as a unified result type for returning input group objects to the Component Editor. The Component Editor built the layout by first placing out the commonly shared general fields input group, and then looping over the array of `InputGroups` returned by the user interface class of every component type. If Scroll Layout was chosen the input groups were simply placed after each other vertically inside another panel. If Tab Layout was selected the input groups were placed in separate tabs. Methods of the `InputGroup` class provided titles for the tabs and borders.

The GUI became substantially more complex with the possibility of switching between different layouts. The main benefit of the `InputGroup` class was to help managing the complexity that arose as a result of all the possible combinations of how borders, scrolling, and titles should be display in different contexts. Most scrolling objects, for example, should have their own scroll pane in the scroll layout, but in the tab layout they should instead use the scroll pane of the Component Editor where the entire tab was displayed. The System Composition Tool was an exception from that since it also had a control panel toolbar at the top. This toolbar should always be visible to the user and so the System Composition drawing area needed to have its own scrollbars in addition to the scrollbars of the Component Editor.

The System Composition Tool points out a variation to the normal case that had to be taken care of. In most cases where there was a scrollable component the title and border could be added to a `JScrollPane` and then that scroll pane could be shown in the scroll layout and not shown tab layout. However in the case of the System Composition where the input group contained a toolbar the border and title should naturally surround both the toolbar and the drawing area. Since the scrollbars should appear around the drawing area, but the title and border should surround the input group, the border and the scrolling could not be implemented on the same Swing object anymore. In order to handle that, the `InputGroup` class had to be extended so that it provided functionality for getting and setting the border, title, and scroll pane independently.

6.11 Propagating Size Changes

As the GUI progressed into having more and more functionality, handling the layout became increasingly complicated. The Component Editor consisted of input groups which could contain anything from simple text fields to tables and graphical tools, and every input group had to be displayed correctly in both the scroll layout and tab layout. Furthermore several panels had to be nested within each other to create the wanted layout.

One interesting and complicated aspect concerning nested panels is the issue of the scrollbars. In order for the scroll bars to appear correctly, and in the right panel, the sizes of all the nested panels must be handled. The underlying factor that governs how sizes are calculated by Swing is that a panel does not respect the size of the other Swing components which it contains, such as other panels. The parent panel tries to force the child panels to adjust their sizes according to the preferred size of the parent. This imposes a problem in the case where you do not want to respect the default preferred size of the parent, but instead want every subcomponents size to propagate upwards so that the topmost parent expands accordingly, and thus affects the scrollbars.

The problem with a GUI that grows dynamically as new chip components are added by the user is that the size change does not always reach the parent panel where the scrolling is supposed to take place. The effect can be that a child panel simply expands out of view and there is no other way of seeing all the contents then enlarging the application window. The parent panel simply has a preferred size value that is too small and as long as it is less than or equal to the size it has been assigned by the top level window, no scroll bars will appear. Setting the preferred size of the parent scrolling panel very large would solve the problem but would not be a good solution, since it would force the scrollbars to appear even when everything fits in the current window.

The way of handling scrolling as the size of some parts of the GUI dynamically change is to always make sure that the top level panel in which the scrolling should occur always has the correct preferred size values. This was solved by using Swing listeners to detect changes in the GUI and then notifying the top level panel of the size change. If a new chip component was added in a dynamic table, for example, the table fired a change event which was used to tell the panel in which the table resided to increase its preferred size by the same amount that the table expanded.

The event handling of size changes was implemented in the `InputGroup` class. A motivation for letting the input groups handle that was that simply knowing that a panel's size had changed was not good enough. It would have meant that the whole window would have had to be redrawn to compensate for it. It would not have been known what the effects were of the change from how it was before compared to how it became afterwards, so the whole layout would have had to be recalculated, causing a lot of processing overhead. Instead, when a size change event occurred, the old size of the input group size was saved within the `InputGroup` class before the new preferred size was updated. That way the top panel could be notified of the change and just increase its own preferred size by the difference of the old and new size of the input group that caused the size change event to occur.

7 Conclusions

7.1 Requirement Uncertainties

In conclusion, the single biggest challenge when designing the GUI was that the prerequisites were not known at the start of the project. In a software project, a sort of catch-22 can often arise because you have nothing concrete to show to the users in the beginning. That can make it difficult for the users to express their wishes and demands which in turn makes it difficult to start building anything concrete. The users have nothing visual to associate their thoughts with and they have no user interface infra-structure to help them explain to the developers where they want to have what and how it should function. The developers on the other hand have difficulties getting started because they may be stuck on square one without any concrete end-user feedback at all.

To break this dead-lock and to get the wheels moving it was probably a good idea to take the Tracer Bullet approach for the GUI and get an iterative development process with user feedback started. In fact, in the early stages of the project, the data model was defined to a very limited degree and there was no Custom API available to deliver functionality based on the data model. If temporary pseudo classes that could mimic the predicted behavior of the future Custom API had not been developed inside the GUI right from the start, too much GUI development time would have been wasted just waiting for the underlying structures to emerge. Instead, by mimicking API and data model functionality and when doing so only spending time on key features that had a great impact on how the GUI would look and function, meaningful and expressive versions of the GUI could be demonstrated to end-users at a very early stage. These early releases and the user feedback that they resulted in were invaluable for how well the final GUI turned out. If the major development of the GUI had waited until the data model and Custom API were essentially finished, the GUI would have encompassed only a fraction of the functionality that it provides today.

7.2 Swing Complexity

Another highly relevant conclusion related to graphical user interfaces is that Swing is not an uncomplicated toolkit. Building GUIs with Swing can be very difficult. Swing is powerful and complicated but that is not due to some blunder related to Java or the Swing package. Building GUIs simply is difficult in any programming language if, at the same time, you want the possibility of designing it exactly the way you envision it. For those who do not care about having a GUI that looks and behaves *exactly* as one would want, but only *more or less* that way, life is easier. Then tools and programming languages can be used that let you design GUIs quickly and easily. However, such easy-to-use approaches can be extremely time-consuming if somewhere along the road precision and exactness is required. Although not used to create GUIs, Microsoft Power Point is an illustrative example of this phenomenon. Power Point is very undemanding and a presentation can be created in a matter of minutes. It is well suited for creating *almost* what you want, but can be extremely time-consuming and tedious for creating *exactly* what you want. Java and Swing, on the other hand, are designed to be expressive enough to let developers design a GUI exactly how they want it while remaining a high-level language. The price of that power is the fact the Swing is a Java package that is difficult to master.

The negative side of Swing being such a complex package can however be minimized and sometimes even neutralized. In some cases Swing provides a GUI feature available in a varying level of complexity. That way one class that is easy to use can be chosen in simple cases and more complex classes can be chosen to suit more complicated situations. One frequent example in this project was the use of layout managers to decide how GUI objects were placed inside a panel. In many situations the most powerful layout managers would have been overcomplicated and their expressiveness unnecessary. In other situations the simpler layout managers could not possibly create the desired layout and only the most powerful managers were adequate. The lesson is that one should not use more complicated Swing classes than necessary. If alternatives are present, as in the case with layout managers, the simplest alternative that can get the job done should be chosen.

7.3 Consistency, Efficiency, and Correctness

Besides providing an easy-to-use and aesthetic end-user application, the system solved the main problems with which the project was concerned. The GUI could be used to easily and efficiently design micro chips and behind the scenes the data was validated and communicated through the Custom API to the XML Data Model. The single source nature of the data model solved three problems of inconsistency, errors, and use of manpower. Inconsistencies were eliminated since manual updates no longer had to be done in several places. Updates were only allowed in one place which preserved the consistency of the data. Errors could be avoided by using generators working with the single source data. The output could be headers for firmware, HDL, test patterns for verification, or documentation of chip components. The generation only had to be formally proven once for each generator. Then by formal theory paradigm the constructed result is correct if the constructor is correct. Manpower and typing effort was also reduced. The generators saved time by automatically creating documents based on the data source that previously had to be typed by hand. The updates only occurring once and in one single place also saved manual labor as well as increased flexibility by the possibility of concurrent modifications of a consistent design data base.

7.4. The Nature of Software

The most important feature of software, and the cause of most of its strengths and weaknesses, is that it is virtual. If software were something physical we would never have had the same issues or possibilities. One example only too well known to most of us, are the floods of spam email washing over the virtual reality every day. How much of all this spam would be sent out if emails were physical and cost 0.1 cent to deliver?

But it is not up to us to judge whether or not it is good or bad that software happens to be virtual. It *is* virtual, and we should focus on taking advantage of the possibilities and opportunities, as well as try to minimize the downsides and threats, that come from that. It seems clear that the use of standards is a potent response to the sometimes uncontrollable growth and evolution of software.

It did seem as if the productivity decreases slightly as the project progressed and the software system grew. The reason for that is that when system complexity grows, perceived productivity usually goes down. The underlying factor is quite natural; when a system is small and simple, new additions are more isolated and thus carry fewer complications with them. When a system is large and complex, on the other hand, new additions can have more

implications in other parts of the system, and so affecting other areas of the code. Since new code development usually is less isolated as the system grows, expected and unexpected code effects must be taken care of. For this reason the productivity, in terms of new features - not in terms of lines of code, could be perceived as decreasing slightly as the project went along.

References

Printed Books

- Antoniou, G. & van Harmelen, F. 2004, *A Semantic Web Primer*, The MIT Press, Massachusetts
- Backman, J. 1998, *Rapporter och uppsatser*, Studentlitteratur, Lund
- Burd, B. 2002, *Java & XML For Dummies*, Wiley Publishing Inc., Indianapolis
- Hunt, A. & Thomas, D. 1999, *The Pragmatic Programmer*, Addison-Wesley, U.S.A.
- Jacobsen D.I. 2002, *Vad, hur och varför? Om metodval i företagsekonomi och andra samhällsvetenskapliga ämnen*, Studentlitteratur, Lund
- Oskarsson, Ö. 1994, *Programutveckling i liten skala - En praktisk handbok*, Studentlitteratur, Lund
- Robinson M. & Vorobiev P. 2003, *Swing 2nd Edition*, Manning Publications Co., Greenwich

Articles On-line

- Gosling, J. & McGilton H 1996, *The Java Language Environment*, Sun Microsystems, viewed 15 January 2007, <<http://java.sun.com/docs/white/langenv/>>

Other Internet Sources

- The Java™ Tutorials:1* 2006, Sun Microsystems, viewed 20 November 2006, <<http://java.sun.com/docs/books/tutorial/>>
- The Java™ Tutorials:2* 2006, Sun Microsystems, viewed 20 November 2006, <<http://java.sun.com/docs/books/tutorial/java/concepts/package.html>>
- The Java™ Tutorials:3* 2006, Sun Microsystems, viewed 20 November 2006, <<http://java.sun.com/docs/books/tutorial/essential/io/index.html>>
- The Java™ Tutorials:4* 2006, Sun Microsystems, viewed 5 December 2006, <<http://java.sun.com/docs/books/tutorial/ui/overview/intro.html>>
- The Java™ Tutorials:5* 2006, Sun Microsystems, viewed 5 December 2006, <<http://java.sun.com/docs/books/tutorial/uiswing/index.html>>
- The Java™ Tutorials:6* 2006, Sun Microsystems, viewed 11 December 2006, <<http://java.sun.com/docs/books/tutorial/uiswing/components/componentlist.html>>
- The Java™ Tutorials:7* 2006, Sun Microsystems, viewed 11 December 2006, <<http://java.sun.com/docs/books/tutorial/uiswing/components/toplevel.html>>

The Java™ Tutorials:8 2006, Sun Microsystems, viewed 11 December 2006,
<<http://java.sun.com/docs/books/tutorial/uiswing/components/scrollpane.html>>

The Java™ Tutorials:9 2006, Sun Microsystems, viewed 5 January 2007,
<<http://java.sun.com/docs/books/tutorial/uiswing/layout/index.html>>

The Java™ Tutorials:10 2006, Sun Microsystems, viewed 8 January 2007,
<<http://java.sun.com/docs/books/tutorial/uiswing/components/tree.html>>

The Java™ Tutorials:11 2006, Sun Microsystems, viewed 8 January 2007,
<<http://java.sun.com/docs/books/tutorial/uiswing/components/table.html>>

The Java™ Tutorials:12 2006, Sun Microsystems, viewed 8 January 2007,
<<http://java.sun.com/docs/books/tutorial/uiswing/painting/index.html>>

Java™ Architecture for XML Binding 2005, viewed 10 January 2007,
<<http://java.sun.com/webservices/docs/2.0/jaxb/xjc.html>>

Serializable (Java Platform SE 6) 2006, Sun Microsystems, viewed 10 January 2007,
<<http://java.sun.com/javase/6/docs/api/java/io/Serializable.html>>

The SPIRIT Consortium 2006, The SPIRIT Consortium, 11 September 2006,
<<http://www.spiritconsortium.org>>

IEEE P1685 2006, IEEE, 11 September 2006, <<http://www.eda.org/spirit-p1685/>>

Greanier, T. 2000, *Discover the secrets of the Java Serialization API*, Sun Microsystems, viewed 20 November 2006,
<<http://java.sun.com/developer/technicalArticles/Programming/serialization/>>

Java Foundation Classes (JFC) 2006, Sun Microsystems, 5 December 2006,
<<http://java.sun.com/products/jfc/reference/faqs/index.html>>

XML 2006, Sun Microsystems, viewed 10 October 2006, <<http://java.sun.com/xml/>>

Code Conventions for the Java Programming Language 1999, Sun Microsystems, viewed 10 October 2006, <<http://java.sun.com/docs/codeconv/>>

Java SE 6 Key Features 2007, Sun Microsystems, viewed 10 January 2007,
<<http://java.sun.com/javase/6/features.jsp>>

JDC Tech Tips 2000, Sun Microsystems, viewed 10 January 2007,
<<http://java.sun.com/developer/TechTips/2000/tt0110.html>>

Tree (data structure) 2007, Wikipedia, viewed 12 February 2007,
<[http://en.wikipedia.org/wiki/Tree_\(computing\)](http://en.wikipedia.org/wiki/Tree_(computing))>

Revision control 2007, Wikipedia, viewed 12 February 2007,
<http://en.wikipedia.org/wiki/Revision_control>

XML – Wikipedia 2006, Wikipedia, viewed 20 November 2006,
<<http://en.wikipedia.org/wiki/XML>>

Packages 2006, Unknown, viewed 10 January 2007,
<<http://www.yoda.arachsys.com/java/packages.html>>

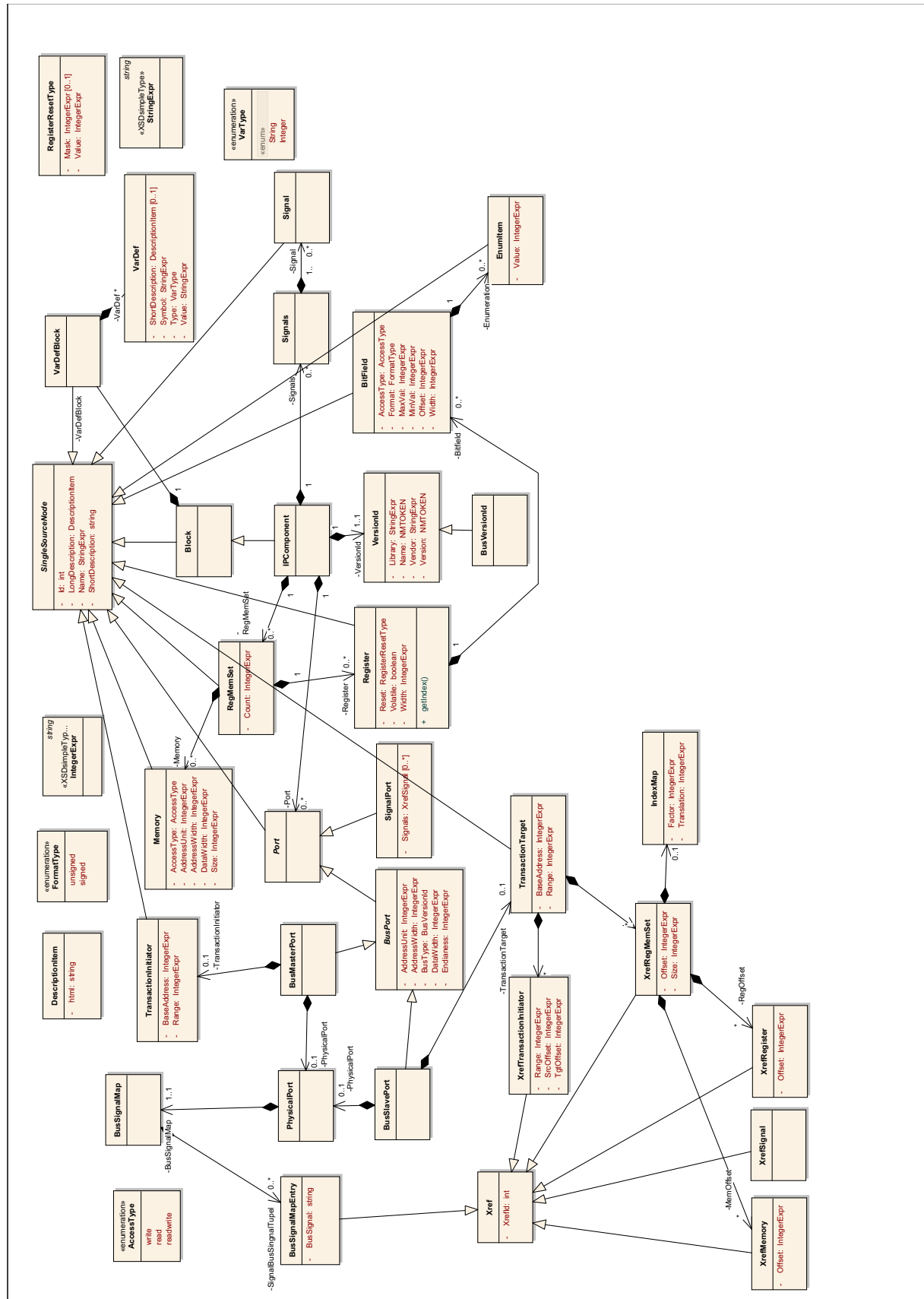
XML Applications and Initiatives 2005, Cover Pages, viewed 12 February 2007,
<<http://xml.coverpages.org/xmlApplications.html>>

Websites

Website 1, <<http://www.w3schools.com/xml>>

Website 2, <<http://java.sun.com/javase/6/docs/api/index.html>>

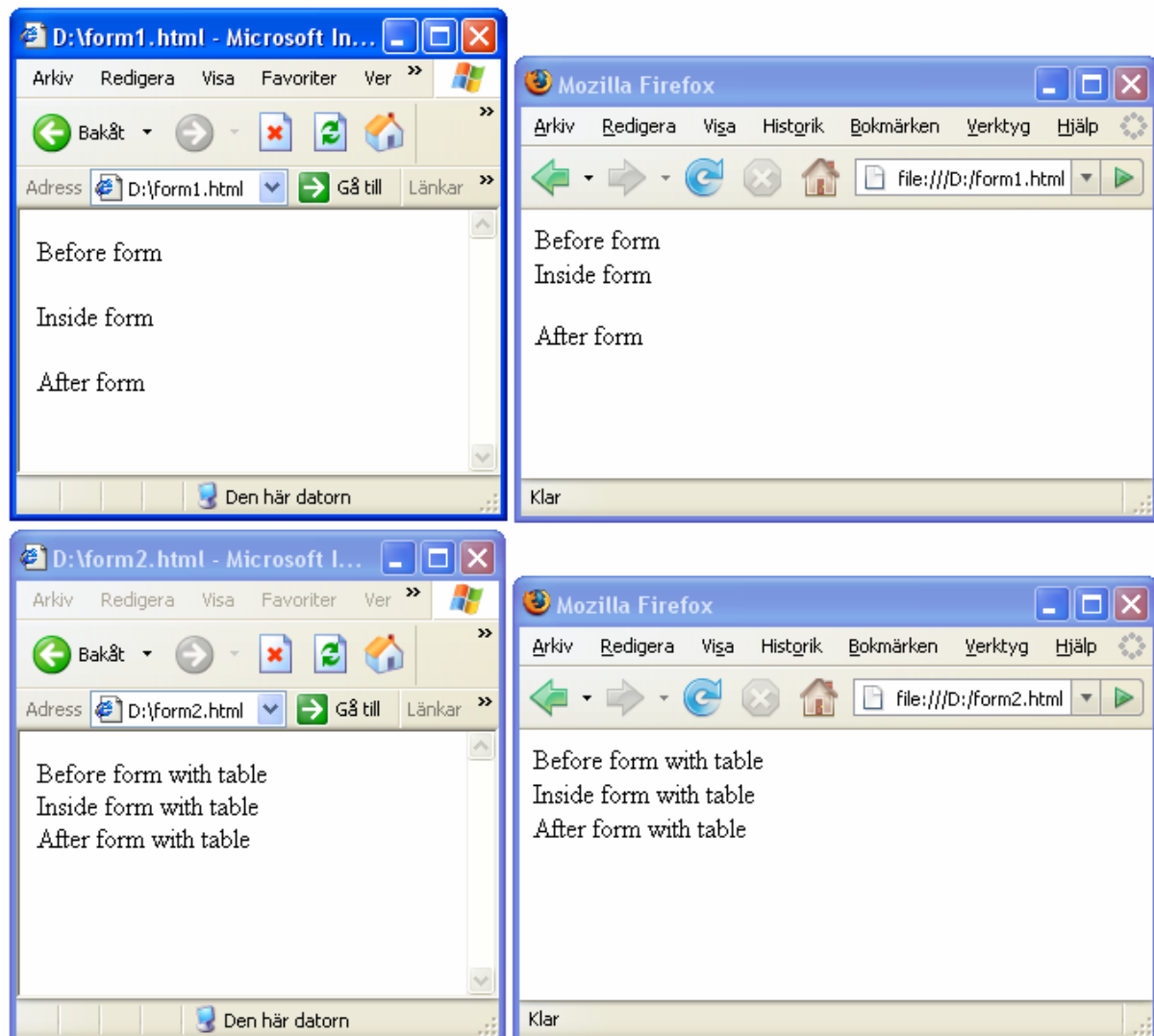
Appendix A – XML Data Model



Appendix B – HTML Code Sample

B.1 System Specification Overview

B.2 Detailed System Specification



```
<html>
<body>
  Before form
  <form>Inside form</form>
  After form
</body>
</html>
```

```
<html>
<body>
  Before form with table
  <table cellpadding="0" cellspacing="0">
    <tr>
      <form>
        <td>Inside form with table</td>
      </form>
    </tr>
  </table>
  After form with table
</body>
</html>
```