TDT4735 Depth study project

A framework for creating pluggable graphical user interfaces for ImproSculpt

Rune Flobakk

22nd December, 2006

Supervisor: Professor Letizia Jaccheri

Norwegian University of Technology and Science

Department of Computer and Information Science
Faculty of Information Technology, Mathematics and Electrical Engineering

Abstract

This document describes the process and results of a depth project investigating how XML-based declarative configuration of a graphical user interface (GUI) is applicable to the algorithmic composition and performance software ImproSculpt, developed by Øyvind Brandtsegg. The approach has been to design a proof-of-concept prototype that demonstrates a strategy using XSLT and a framework implemented in Python for building a GUI using XML. The framework and XML language are evaluated to see if it meets the stated goals.

Preface

This report is the result of my first encounter with the interdisciplinary research field concerning software and art. It has in many ways been a new experience, both because of the required focus on research as opposed to pure engineering, but also because of the unconventional domain of combining art with software and computer science.

The work is part of the master's degree graduate level course "TDT4735 Software Engineering Depth Study" at the Department of Computer and Information Science, Norwegian University of Science and Technology (NTNU). It counts 15 of the course's 23,5 ECTS credits.

Acknowledgments

I want to thank my supervisor Professor Letizia Jaccheri for the help she has provided me during the project and also for giving me the opportunity to work with such an exotic subject. Also, I would like to thank Øyvind Brandtsegg for letting me work with his software project on algorithmic composition, ImproSculpt. In addition, Associate Professor Hallvard Trætteberg has been a great help for me to be able to define my project goals.

Trondheim, Friday 22nd December, 2006

Rune Flobakk

Contents

1 Introduction **1**
1.1 About the project 1
1.2 Motivation 2
1.3 Goals 2
1.3.1 Contribution 3
1.4 Chapter guide 3
1.5 The project context 4

2 Research Method **5**
2.1 Limitations 5
2.2 Design science research 5
2.2.1 About the method 6
2.2.2 Purpose of the method 6
2.2.3 Differentiating routine design from research design 6
2.2.4 Guidelines 7
2.2.5 Evaluation 8
2.3 A considered alternative 9

3 Process **10**
3.1 Choosing of project 10
3.2 Project beginning 11
3.3 Search for project and research goals 11
3.4 Tasks 12

3.5 Miscellaneous learnings 13

4 Existing solutions **14**

4.1 XHTML 14

4.1.1 User interfaces 15

4.1.2 The layout problem 15

4.1.2.1 Cascading Style Sheets 16

4.2 Extensible Stylesheet Language 16

4.2.1 XSL Formatting Objects 16

4.2.2 XSL Transformations 17

4.2.3 About complexity 18

4.3 XUL 18

4.3.1 XUL organization 19

4.3.1.1 Content package 19

4.3.1.2 Skin package 20

4.3.1.3 Locale package 20

4.3.2 The XUL box model 20

4.3.3 Behaviour 21

4.3.3.1 Commands 21

4.3.3.2 Broadcasters and Observers 21

4.3.4 XBL 21

4.4 Non-XML languages of interest 22

4.4.1 OGDL 22

4.4.2 YAML 23

5 Solution proposal **25**

5.1 Name 25

5.2 Common motivation 25

5.2.1 GUIs are declarative by nature 26

5.2.2 Actions - a clarification 26

5.2.3 Separation of GUI from the remaining application 27

5.2.3.1 Aesthetic computing 27
5.2.4 Architecture and GUI 27
5.3 The framework 28
5.3.1 Handling XML in Python 28
5.3.2 Chosen strategy 28
5.3.2.1 Evolution of the idea 28
5.3.2.2 Process description 29
5.3.3 Using the framework 30
5.3.4 Using the XML Schema 30
5.4 XML language 31
5.4.1 The generic language 31
5.4.1.1 The root element: `gui` 32
5.4.1.2 The configuration section: `config` 33
5.4.1.3 The component elements: `container` and `widget` . . 33
5.5 Evaluation 34
5.5.1 Discussion on language readability 34

6 Conclusion and future work **36**
6.1 Future work 37
6.1.1 Missing functionality 37
6.1.2 GUI toolkit support 37
6.1.3 Cooperation with an artist 37
6.1.4 Investigating new scripting features of Java 6 38

References **39**

Index **43**

A Requirements specification **45**
A.1 Artefacts 45
A.2 Proof-of-concept framework 45
A.2.1 Expressiveness 45

A.2.2 Widgets 46
A.3 Language specification 46
A.3.1 Language design 46
A.3.2 Independence 47

B XML Schema **48**

C Code examples **50**
C.1 “BorderLayout” in XUL 50
C.2 Examples on XML data utilized by pyVisage 51
C.2.1 A simple GUI specification 51
C.2.2 Auto generated generic GUI specification 51
C.2.3 XSLT document 51

List of Figures

4.1 Java's BorderLayout implemented in XUL 20
4.2 Drop-down list represented as a graph model. 23

5.1 Diagram showing the 2-step processing of XML data to finally be able to set up a graphical user interface. 29
5.2 Example of a GUI set up using the pyVisage framework. 31
5.3 Writing a GUI description with assistance from an XML Schema . . . 32

Listings

4.1 Drop-down list implemented in XHTML 15
4.2 XSL-FO example, formatting text 17
4.3 A header element defined in XML. 18
4.4 Styling of header text using XSLT. 18
4.5 Drop-down list implemented in XUL. 19
4.6 Drop-down list coded in OGDL 23
4.7 Customer data encoded as XML 24
4.8 Customer data encoded as YAML 24
5.1 Python program which uses the pyVisage framework. 30
B.1 (Appendix) XML Schema document 48
C.1 (Appendix) Java's BorderLayout implemented in XUL. 50
C.2 (Appendix) A simple GUI specification. 51
C.3 (Appendix) Auto generated generic GUI specification using XSLT. . . 51
C.4 (Appendix) XSLT document describing transformation rules for the code in Listing C.2 51

List of Tables

5.1 Overview of desirable framework features and goals and how well they are handled by pyVisage. 35

I have seen a strong correlation between people who have some music in their background and programming skills. I have no idea why, but I suspect that some of the areas of the brain that make someone musical also make them good at software development.

Dave Thomas, author of “Pragmatic Programmer”,
in reply to questionnaire for stifflog[2] [Rzeszótko, 2006]

Chapter 1

Introduction

Obstacles are those frightful things you see when you take your eyes off your goal.

Henry Ford

This chapter will give a brief introduction to the project, the context, and its motivations and goals.

1.1 About the project

In the past recent years there has been a trend towards using XML in numerous parts of software systems to *describe* aspects of the system instead of explicitly programming how these aspects should work. The argument is that, where appropriate, using declarative descriptions is easier to comprehend and maintain, and it mostly requires less lines of code than traditional programming of the system's desired behaviour.

This project aims to investigate the ongoing trend towards using XML as means for creating rich and complex graphical user interfaces (GUIs). More specifically it addresses how this discipline is applicable to the artistic and niché software ImproSculpt which demands the ability to adapt its GUI according to what context it is used in.

1.2 Motivation

The author of this report has experience with and acknowledges the profit of using XML for various purposes in software systems. The discipline is coherent with his engineering education on component- and object-orientation as well as the motivation to reduce the coupling between modules of a software system [Hawryszkiewycz, 2001, Bass *et al.* , 2003]. However, while this makes sense from an engineer's point of view, it still requires the knowledge of an additional technology and language, and it often needs considerable time and effort to learn how to apply the language to optimal use.

The author wants to investigate if it is possible to apply this discipline for a programmer with limited or no education in programming. Creating computer programs is in no way an activity exclusive to computer engineers, but the formalism, methods and best practices are often what separates the engineer from the self-taught. This "border-line" has already been expressed by the electronic media artist Paul De Marinis describing his first visit to the Xerox PARC in 1976.

> In the ensuing discussion Kay seemed bent on dissuading me from my own technological-musical interests. He seemed to say that artists should not try to create technology. "Be content with what we make for you." [...] I left PARC that day basically disappointed. It seemed to exclude me and to pooh-pooh my interests in creating technology on my own. It seemed to declare that brilliant engineers knew what was best for everybody, including artists. [Harris, 1999]

The attitudes described in the above quote is what the author of this report believes is wrong, but is present in many engineer and science environments.

Since the author started to experiment with programming several years before actually practicing it in an education context, and also being a hobbyist musician, he feels he can somehow relate to how Brandtsegg as an artist may approach the development of a software system. The problems related to applying engineering complexities to art catches the interest of the author, since he feels that he in many ways he has a foot in both domains.

1.3 Goals

The goal for this project is to develop a proof-of-concept framework which enables the algorithmic composition software ImproSculpt to have its graphical user

interface set up using a description language with XML syntax. This effort also need to be evaluated in cooperation with the artist and end-user of the proposed language to see if it is believed to improve the development and use of ImproSculpt. As for descriptions and info on ImproSculpt, I will refer to the following resources: [Brandtsegg, 2005, Collet *et al.* , 2006, Brandtsegg, 2004 (?), Semb & Småge, 2005, 2006]

I have divided this task into the following goals:

G1 Review the current state-of-the-art of XML languages for configuration of GUIs, with an emphasis on desktop applications.

G2 Design a specification of a language for describing a desktop application's GUI. The language must defined in such matter so it is easily human readable and easy to edit by hand.

G3 Implement said framework

G4 Evaluate the results in cooperation with Brandtsegg.

The overall research questions for the project are:

R1 How appropriate is using a declarative language as XML for GUI configuration of the ImproSculpt application?

R2 How is the framework accepted by the artist as a viable method to create the various GUIs of his application?

1.3.1 Contribution

The author regards this project as a contribution to the research activities on art and software and must be seen in context to work already done on the topic. In addition, this project deals with a discipline embraced by general modern computer science, so it should be "[...]contributing to a general advance in knowledge and the state of the art which will have extensive applications ranging far from the specific problem initially tackled." [Sedelow, 1970]

1.4 Chapter guide

This report starts with presenting the methods used for this project in chapter 2 (page 5).

Chapter 3 (page 10) briefly explains the process carried out during the project.

Chapter 4 (page 14) then presents a review of the current state if the art of GUI configuration using XML.

Chapter 5 (page 25) presents the author's proposal for a framework for GUI configuration implemented in Python. The chapter also elaborates on the various choices that has been made and discusses any trade-offs associated with the use of the framework.

Chapter 6 (page 36) presents conclusions as well as suggestions on further work on this topic.

The appendix section contains the requirements specification for the generated artifacts of this project, XML Schemes of the developed languages, and code examples.

1.5 The project context

This project is part of the author's Master's degree diploma in systems engineering at the Norwegian University of Science and Technology (NTNU). Strictly speaking it is a depth study taking place the semester before the actual diploma thesis, and not part of the diploma per se. Still, it acts as a preparation for the last semester's diploma, first and foremost for the author to learn to conduct a research project.

Chapter 2

Research Method

If we knew what it was we were doing, it would not be called research, would it?

Albert Einstein

This chapter describes how the research has been conducted in order to meet the goals of the project.

2.1 Limitations

"Each method and approach brings with it a certain overhead in terms of understanding and competence. In the limited time available, and with your limited resources, it is usually better to do one thing well, rather than two things badly" [Cornford & Smithson, 2006] Regardless of which research framework one finds appropriate, for a half year student project one must do some adaptations to the methods to be able to conclude within the period. Also, for the author, this project also serves as a learning facility to be able to conduct a successful research project and write his master thesis in the spring 2007.

2.2 Design science research

As this project involves actual development of an artifact (an language for describing GUIs), the design science research approach seemed appropriate to follow.

2.2.1 About the method

The author has used the article *Guidelines for Design Science in Information Systems Research* [Hevner *et al.* , 2004] to plan the activities during the project. Hevner *et al.* argue that design science and the more traditional behavioral science complements each other in the fields of information systems and computer science. While behavioral science has its roots in natural studies, design science is rooted in engineering, and therefore seems like a suitable approach for the author with his limited research experience, to undertake.

2.2.2 Purpose of the method

Design science deals mainly with two activities:

- creating
- and evaluating

> The design process is a sequence of expert activities that produces an innovative product (i.e. the design artifact). The evaluation of the artifact then provides feedback information and a better understanding of the problem in order to improve both the quality of the product and the design process. This build-and-evaluate loop is typically iterated a number of times before the final design artifact is generated. [Hevner *et al.* , 2004]

This fact that design science is a cyclic activity makes it slightly resemble the action research method discussed later as a considered alternative in section 2.3.

The artifacts created from design science activities “are rarely full-grown information systems that are used in practice. Instead, artifacts are innovations that define the ideas, practices, technical capabilities, and products through which the analysis, design, implementation, and use of information systems can be effectively and efficiently accomplished” [Hevner *et al.* , 2004].

2.2.3 Differentiating routine design from research design

What differs design science from routine engineering has already been mentioned as the innovation and evaluation elements of the process. It is innovative in that the design and implementation must provide a solution that is not yet acknowledged as

common practice. And since it is not considered common practice, the design must be evaluated to see if whether it represents a positive contribution to the intended area of research.

2.2.4 Guidelines

There are proposed 7 guidelines in [Hevner *et al.* , 2004] to assist researchers in how to conduct a successful design science research project. However, they also state that none of the guidelines are mandatory and needs to be considered how, to what extent, and if at all they should be applied to a specific project.

Below are descriptions on how each guideline is applied to this project. For a general description of each guideline see the already mentioned article [Hevner *et al.* , 2004].

1: Design as an artifact The resulting artifact of this project is a proof-of-concept framework enabling the isolation of graphical user interface specification of a desktop application. The realization of this framework emphasizes on being applicable to the ImproSculpt application.

2: Problem relevance The artifacts created should solve or increase the efficiency of a phenomena. The phenomena in question here is the use of ImproSculpt in various context demanding customized GUIs. A rich and complex application as ImproSculpt needs an efficient way to switch between user interfaces tailored to a specific use in a given context. A GUI specification that is separated from the application code and read at run-time will be a better solution to this phenomena since the different GUIs then get pluggable and easily exchangeable.

3: Design evaluation There are several quality attributes that an IT artifact can be evaluated upon. In accordance with the research goals, for this project I have chosen to evaluate how well the framework *fits with the intended domain*, i.e. how well it adheres to the requirements of being easy to use, fulfills its intended function, and uses a language which is easy readable and comprehensible.

4: Research contributions The contributions of this project is, in addition to the produced artifacts itself, a further understanding of how engineering is applicable to artistry.

5: Research rigour The author relies mainly on qualitative methods for analyzing the activities and results of this project. The results are tested and decided to

either work, partly work, or not work on various quality and functional goals, which are stated later in the report.

6: Design as a search process In general, the author is searching how to utilize available means to reach desired ends while satisfying laws existing in the environment [Hevner *et al.* , 2004]. The point about satisfying laws existing in the environment is especially important for this project as one of the goals is to investigate how a particular computer engineering discipline is endorsed by an artist. (See section 1.3 and especially research goal R2)

7: Communication of research This paper will present an evaluation and conclusions on the findings concerning my research goals, as well as technical details on the implemented framework and designed language.

2.2.5 Evaluation

Evaluation is a crucial element of a research process. According to [Hevner *et al.* , 2004], primarily mathematical and computational methods are used to evaluate the results of design science research, however, empirical techniques may also be used.

The proposed solution and its employment are evaluated mainly by the author himself. It has been desirable to include the actual end-user, Brandtsegg, in the evaluation process, but this has sadly not been possible due to a blown time schedule.

The evaluation will investigate on the following specific issues, and provide a discussion on to what extent each issue is fulfilled.

- Framework manages to set up
  - labels,
  - buttons,
  - textfields.
- The mentioned widgets can be connected to the application code.
- The language is easy to comprehend and write.
- The solution is easily portable to arbitrary GUI toolkits.
- The language is expressive, it does not impose excessive constraints on the GUI designer.

2.3 A considered alternative

The author has also considered the action research approach, as this is a very popular method of research in computer science.

As an action researcher *forsakes their traditional role as an observer of events and takes part with the subjects in the problem situation* [Cornford & Smithson, 2006] this research approach resembles design science. However, due to its cyclic repetitive process model which is more extensive than the design science one, and the fact that action research is not a recommended approach for one person with a limited time-frame, the author has chosen to not use this method of research.

Chapter 3

Process

Look at a day when you are supremely satisfied at the end. It's not a day when you lounge around doing nothing; it's when you've had everything to do, and you've done it.

Margaret Thatcher

This chapter briefly explains and evaluates the process of the project realization.

3.1 Choosing of project

The general context of the project had already been chosen by the author before the roughly two months spring break between the eight and ninth semester. The project was given by Professor Letizia Jaccheri at the Department of Computer Science and Information Science, NTNU.

The author had one meeting with his main supervisor Jaccheri which gave a brief presentation of the project domain. Also, on the 8th of June the author and Jaccheri met with Øyvind Brandtsegg, a Research Fellow in the Arts [KHiB, 2006] at the Department of Music at NTNU and the developer of the algorithmic composition software ImproSculpt [Brandtsegg, 2005].

Before taking his spring break from studies, the author had mainly been presented the practical domain of the project; to improve the GUI of Brandtsegg's software.

3.2 Project beginning

The first meeting after spring break was conducted the 24th of August. It quickly became evident that I must focus on defining the research goals for the project. The research goals are the results of an evolutionary process, requiring me to get an insight in both the art and GUI domain, as well as the current state-of-the-art of art and software and aesthetic software [Fishwick, 2006, Harris, 1999, Sedelow, 1970, (examples)]. I also had to start studying the Python programming language.

The author being a non-experienced researcher, Brandtsegg gave a good explanation which elegantly summarizes the research goal problem:

> You decide what someone else should gain from our work. You won't manage to tell everything, thus you choose one specific matter.[1]

3.3 The search for project goals and research questions

The author was prepared that finding and defining practical and research goals would be a lengthy process. In addition, after several weeks of what seemed like "walking around in the dark", a meeting with Associate Professor Harald Trætteberg at NTNU in late September the author finally managed to define the overall practical goal of the project; to create a GUI solution involving the use of XML as description language. From then, the goal was to design a language to describe/program the GUI of Brandtsegg's ImproSculpt application. However, this goal was revised on a meeting with Letizia in early November to instead do a small proof-of-concept implementation of a framework which uses descriptions in XML to set up a GUI for Python applications. The outcome of this implementation is further discussed in chapter 5.

The research questions was to some extent already defined and has only had minor revisions during the process. It has not been easy to come up with reasonable research questions in a domain so unfamiliar for the author, which is the case of *software and art*. In the beginning it was also not clear what the ImproSculpt software really did, as it seemed almost impossible to get it to run on the author's computer. After lots of mail correspondence, another meeting with Brandtsegg, and several adaptions of the software it finally managed to start and make some noise. But it was not before I attended the concert "Kanon" at the club Blæst in

[1] Translation from Brandtsegg's originally Norwegian quote: "Man bestemmer hva 'de andre' skal ha nytte av vårt arbeid. Man makter ikke å fortelle alt så man velger ut én ting."

Trondheim the 31st of August I realized what the software was capable of. In addition, Brandtsegg held an interesting workshop the day after where he explained his software, how it fits into a research context on algorithmic composition, and how he uses it for live performances or as a continuous sound and music generator for various art and sound installations.[2]

What the author understood from this was that while the GUI could benefit from a visual and aesthetic upgrade, the real challenge is how it efficiently solves the need for having several different GUIs which is tailored for specific purposes, it being a concert or a control panel for a sound installation. In fact, Brandtsegg doesn't really use the GUI at all during a real-time perfomance, but various external hardware controllers which sends control messages to the program.

3.4 Tasks

The work done mainly involves an in-depth study of some of today's most known standards and technologies for using XML for information presentation and graphical user interfaces. I chose to focus on a few technologies instead of a broader review since I needed to gain an extensive understanding of the concepts to be able to employ them to my own implementation. For a broader review of existing XML languages I will refer to [Frogner & Arulanantam, 2003]. My review can be found in chapter 4.

Initiated as late as november, the implementation of a proof-of-concept GUI framework (chapter 5) has also involved learning the Python language, and in particular its possibilities for class reflection and use of meta-classes which enables dynamic instantiation and adaption of objects. This implementation has in a way been a process of trial and error, and at many points the task seemed actually quite hopeless. Luckily, due to the programming experience and training already possessed by the author, once the implementation became an actual running program, the further development was a breeze, and a proof-of-concept framework which enabled the concluding discussion of the efforts was not far away.

[2]Visit the web page `http://www.flyndresang.no/en` to read about the sound installation "Flounder" at Inderøy in Norway and listen to the sound which is produced by ImproSculpt and emitted through the sculpture.

3.5 Miscellaneous learnings

An IT project usually involves the learning and/or use of technologies which doesn't fit into the scope of the project context and research domain. I will still mention some of the notable technologies and other knowledge I have used in order to realize the project results.

Unit testing Unit tests has the last years become a very natural method for me to use when I do implementations. With previous experience with the JUnit framework for Java, I searched at an early point to find an equivalent solution for Python to do automatic unit testing. The pyUnit library is part of the current Python distribution, and has been used for unit testing relevant parts of the implementation.

State of the art regarding art and software My supervisor has provided an extensive literature selection on the field of art and software. The reading and study of this in conjunction with the TDT69 Artistic Software: Processes and Products course [Jaccheri, 2006] has made the author gain an understanding of this area of research, which was unfamiliar before starting the project.

Various GUI toolkits for Python In the process of defining my project goals, the author did a small investigation of the available GUI toolkits. As these libraries are mostly Python interfaces to libraries written in other programming languages, mainly C or C++, the author has also became familiar with the concept of the automatic generation of these "language bindings" using tools like SWIG, Py++ or similar [Py++, 2006, SWIG, 2006].

Scientific writing using LaTeX The realization of this report has required the use of several LaTeXpackages.

Chapter 4

Existing solutions

Get your facts first, and then you can distort them as much as you please.

Mark Twain

This chapter aims to present a review of what solutions currently exist for XML configuration of user interfaces. It will also discusses strengths, trade-offs, and any other issues that needs to be addressed on this matter.

4.1 XHTML

Probably the most well-known language to create user interfaces is the HTML language. HTML is what is known as a markup language; one uses special symbols called tags to describe how to display text and graphics. The HTML language has been maintained and developed by the World Wide Web Consortium (W3C) since 1996.

In 2000 W3C released the specification for XHTML 1.0, which is basically a reformulation of the latest HTML specification with some minor additional constraints, making it conform to the XML syntax [W3C, 1994-2006].

The main purpose of XHTML is mainly to specify the structure of static textual and graphic elements in a web document, but it also gives the ability to design simple means for user interaction by the use of forms. A form can be a part of a document and provides the user with a basic set of interactive widgets, such as

- check-boxes and radio-buttons
- text fields and text areas
- drop-down lists
- buttons

4.1.1 User interfaces

A form is realized using simple nesting of the elements to indicate the structural composition of the GUI. For instance, a drop-down list can be made in XHTML as shown in Listing 4.1. The `form` element declares the top-level container for the GUI. It contains one drop-down list, declared by the `select` element, which in turn contains three possible options, declared by the `option` elements. The “Oranges” option is also explicitly specified to be `selected` using an additional attribute in one of the `option` elements.

```
1 <form>
2    <select>
3       <option>Apples</option>
4       <option>Pears</option>
5       <option selected>Oranges</option>
6    </select>
7 </form>
```

Listing 4.1: Drop-down list implemented in XHTML. This example shows how structural composition is made easy with a markup-language using nesting of elements.

XHTML is an important contribution to enable the non-computer engineer to create the presentation layer of information and applications. Today almost anyone can, with a minimum of learning required, quickly create simple and workable user interfaces for use on web pages.

4.1.2 The layout problem

A user interface is not only about presenting information and interactivity controls. It should also be presented in an aesthetic, eye-pleasing, and most importantly user-friendly manner. The interface must provide an efficient and immediately logic way for the user to do his/her work.

Although still possible for compatibility reasons, it is not a good strategy to use the design and layout features of XHTML. One may specify layout properties, as for instance font size, colours, borders, and even placement of elements when writing XHTML, but this obviously does not make an easily modifiable document. If one at a later time decides to change the style of any part of the design, it is then required to make this change to every element implementing the specific style.

4.1.2.1 Cascading Style Sheets

To cope with this, XHTML utilizes a description language called Cascading Style Sheets (CSS) [W3C CSS, 1994-2006] to define the styles and page layout of an XHTML document. This effectively gives full controll on how any element is presented to the user. However, CSS imposes yet another syntax and language for the developer to learn and master. The syntax is however very simple, using a flat structure of `property: value` pairs, but because of this a style sheet can very quickly grow into a large uncomprehensible file which is difficult to maintain.

CSS is widely supported by web browsers and considered the de-facto standard for formatting documents for the web.

4.2 Extensible Stylesheet Language

The Extensible Stylesheet Language (XSL) provides, as CSS, means for defining the styles and layout of XML documents. XSL uses XML syntax, and is thus effectively solving one of the problems with CSS in that the styling is done using the same syntax as the user interface composition in XML or XHTML. XSL is maintained by the [W3C, 1994-2006].

Styling in XSL is achieved using 2 of the 3 members of the XSL family of languages; XSL Formatting Objects (XSL-FO) and XSL Transformations (XSLT)

4.2.1 XSL Formatting Objects

XSL-FO is an XML-language that describes both data and how it is presented on screen, on paper, or other media.

This format has a separate section for describing page setup and behaviour, but the layout and styles are defined together with the actual data structuring, so it sort of resembles XHTML (section 4.1) *without* the use of CSS.

> The XSL FO markup is quite complex. It is also verbose; virtually

> the only practical way to produce an XSL FO file is to use XSLT to produce a source document. Finally, once you have this XSL FO file, you need some way to render it to an output medium. [Eisenberg, 2001]

As Eisenberg puts it, XML-FO is a format that is not meant to be humanly edited or read, but more as a "last point before press". XML-FO is a result of a parsing process, and represents the actual explicit styling and data output that is ready for a renderer component to put out to the screen, while it is still in a exchangeable XML format. The XML-FO document should be both machine generated and then afterwards further machine parsed for presentation to the user.

```
1 <fo:block font-size="14pt"
2           font-family="verdana" color="red"
3           space-before="5mm" space-after="5mm">
4    This a heading.
5 </fo:block>
```

Listing 4.2: XSL-FO example on how text is formatted in XSL-FO. It demonstrates the verbosity of XSL-FO.

But what is the point then with this additional parsing step? The XML-FO is closer to a 1:1 map from the configuration to the output rendering, and thus making the implementation of the rendering component easier. But this is of little use since XML-FO itself is a complex format to "hand-write". This is where XSLT comes to use (subsection 4.2.2).

Listing 4.2 shows how a heading can be formatted using XSL-FO.

4.2.2 XSL Transformations

XSL Transformations (XSLT) is generally a language for transforming one XML format into another, and one of it's intended use is to transform a custom made XML-format into the generic XSL-FO format. In this way we can implement rendering components in our software that only needs to deal with one standardized language. This loosens the coupling between the development of the actual language and the component which utilizes the language.

XSLT can be viewed as styling of the various elements of an XML document, and that there is actually a transformation going on from the XML document to another XML document (in XSL-FO format) which in turn is processed by the software is happening "behind the scenes". This enables the custom designed XML format to be evolve more independently of the parsing components.

```xml
<heading>
    This is a heading
</heading>
```

Listing 4.3: XML

```xml
<xsl:template match="heading">
  <fo:block font-size="14pt"
          font-family="verdana"
          color="red"
          space-before="5mm"
          space-after="5mm">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>
```

Listing 4.4: XSLT

Listings 4.3 and 4.4: The combination of XML (data) and XSLT (styling).

The Listings 4.3 and 4.4 show how XSLT can be used to “style” an arbitrary XML format. In Listing 4.3 we have our own XML format which enables us to mark headings with the `heading` element. The XSLT definition in figure 4.4 specifies that every occurrence of the `heading` element will be transformed to an XSL-FO `block` element with the additional specified styling attributes. The `<xsl:apply-templates/>` element specifies where the contents of the `heading` element is placed. (Notice that the code in figure 4.2 is exactly the same as the lines 2-8 in figure 4.4, where in the latter “`This is a heading.`” has been replaced by “`<xsl:apply-templates/>`”.)

4.2.3 About complexity

The combination of XSL-FO and XSLT gives us the possibility to apply styles and layout to an arbitrary XML document, much in the same manner as using CSS to style an XHTML document with the added advantage of using the same syntax for both structure and styling. However, pushing the concept of generality to this extent comes at the obvious price of using more layers, and one must really consider the cost vs. benefit if such an approach is planned.

4.3 XUL

XUL is an acronym for XML User interface Language. This is a language developed and mainly used by the [Mozilla Organization, 1998-2006b], but the generic nature of the language name is maybe indicating a bigger ambition. The most well-known applications by the Mozilla Organization, the web browser Firefox and email client

Thunderbird, have their graphical user interfaces fully implemented using XUL.

XUL features several means for providing a user interface:

- As a standalone application, either bundling the Mozilla platform with the application, or requiring a Mozilla browser to be installed on the client machine.
- As a software extension. Extensions can provide additional functionality to for instance Firefox or Thunderbird. XUL enables a GUI to be merged with an existing one, making it a potential language of choice for plug-in architectures.
- As a remote application providing a interface in a web browser.

This and the following sections about XUL is mostly written using the extensive tutorial by [Deakin, 2006].

4.3.1 XUL organization

The XUL files for an application are commonly divided into 3 packages to separate content from the styling and also to enable the use of locale specific files. A package is simply a semantic concept describing the separation, and is mostly realized using a directory for each package.

4.3.1.1 Content package

This package contains the declarations of windows with the containing user interface elements, and these are the actual XUL files. The interface gets its behaviour from JavaScript [JavaScript, 2006] files which is also part of the Content package (see subsection 4.3.3).

Again using the now familiar drop-down list from Listing 4.1, the same widget can be implemented in XUL as shown in Listing 4.5.

```
1 <menulist>
2   <menupopup>
3     <menuitem label="Apples"/>
4     <menuitem label="Pears"/>
5     <menuitem label="Oranges" selected="true"/>
6   </menupopup>
7 </menulist>
```

Listing 4.5: Drop-down list implemented in XUL.

4.3.1.2 Skin package

The styling of the user interface is separated from the content declarations, and as with XHTML (section 4.1) this is done using Cascading Style Sheets. If any custom graphics are necessary, those files are also found in this package.

4.3.1.3 Locale package

This is an optional package and is used to include locale specific files for the interface. Typically, this package contains different translations of text entities present in the user interface.

4.3.2 The XUL box model

Every container in XUL defines a rectangular area in which it will place its containing components. By default components are aligned next to each other horizontally, but vertical alignment can also be specified. Since a container may also contain other containers, it is possible to specify complex layouts.

Container boxes are specified using `<hbox>` and `<vbox>` to create respectively horizontal and vertical aligning boxes. Using a combination of nested boxes we can easily reconstruct for instance Java's BorderLayout [Sun Microsystems, 2004, 1995-2006] as shown in figure 4.3.2.

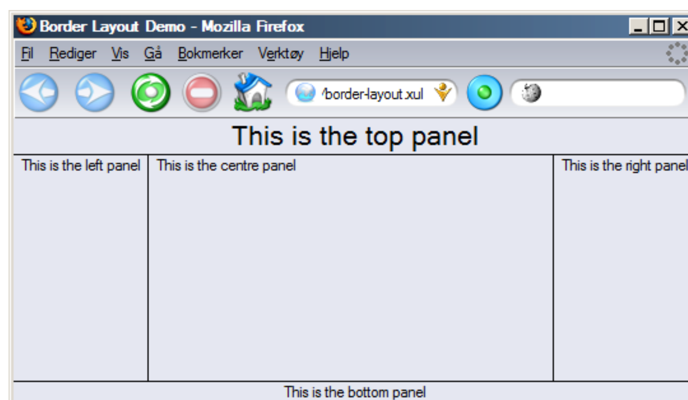

Figure 4.1: Java's BorderLayout implemented in XUL. The BorderLayout is an often used layout with four panels surrounding a flexible centre panel. The top and bottom panels have fixed height, and the left and right panels have fixed width. The complete implementation is in appendix C.1.

4.3.3 Behaviour

XUL also features ways to specify actions in consequence of GUI interaction and also GUI updates happening as consequence of changes in the application state. These features can in many ways be regarded as light-weight object-oriented design patterns.

4.3.3.1 Commands

Commands are used to specify operations that is possible to carry out while interacting with the interface. These commands can the be linked with one or more GUI components. A command declaration can also specify attributes that will be inherited by the components linking to it. This slightly resembles the Command design pattern in the way as it separates the actions performed from the GUI code even though it is specified in the same file. [Cooper, 1998]

4.3.3.2 Broadcasters and Observers

XUL uses the concepts of broadcasters and observers to implement the observer design pattern which is commonly used in GUI implementation design. The Observer pattern streamlines the updating of often several GUI components as a consequence of change in the application state/data without requiring any action from the user. [Cooper, 1998]

The Observer pattern uses objects called *observers* which most commonly refers to GUI components that monitors (or observes) some data which is due to change. The data which is observed is most commonly refered to as the *subject*.

In XUL, a component can be declared as an *observer*, and the subject is refered to as a *broadcaster*. When such links between observers and a broadcaster are established, it is only necessary to update the broadcaster data to have any relevant observer components to update their displays.

4.3.4 XBL

XBL, Extensible Bindings Language, is a related language to XUL and a proposed W3C standard [W3C, 1994-2006]. This language enables the binding of actions to XUL GUI components, but also enables the creation of composite reusable XUL elements, which is in a way analogous to the already discussed XSLT (subsection 4.2.2). When using XSLT a new XML document is created, but with XBL, this transformation is done “on-the-fly”. Also, the linking from a XUL

document to a XBL specification is done via a CSS document, which to the author seems a bit cumbersome.

4.4 Non-XML languages of interest

While researching, the author has come across several approaches other than using XML for describing and configuring components of a software system. It is beyond the scope of this project to conduct any further studying of these, but some are mentioned here for others to maybe pursue the possible appliance of them in other projects.

For an extensive list of alternatives to XML, see [Tchistopolskii, 2006].

4.4.1 OGDL

OGDL, Ordered Graph Data Language, is one of numerous efforts to get rid of the verbose XML syntax. As the name implies, its purpose is to store graphs in a parseable textual form.

A graph consists if a set of *nodes* and a set of *arcs* or *links* connecting pairs of nodes. Both nodes and arcs can be labeled to indicate semantics of both entities and relations. The graph theory was invented by the Swiss mathematician Leonard Euler. [Luger, 2005]

Recalling figure 4.1, a drop down list can be modeled as a graph as shown in Figure 4.2. The figure clearly shows a drawback of the OGDL language; it does not support labeled arcs, only nodes can be labeled, and as such the OGDL code in Listing 4.6 is not capable to fully represent the graph in Figure 4.2, at least not in a clean and logical way. One possibility is to let arcs going from the *select* node to implicitly lead to *option* nodes, but there would still be the problem of indicating which option that should be *selected*. In addition, we are limiting the descendant nodes to be *only* option nodes.

In general, assigning attributes to graph nodes is problematic, since it only has means for defining a label. There would perhaps be possible to encode attributes into the text string using a predefined syntax, but then the readability of OGDL will probably be lost.

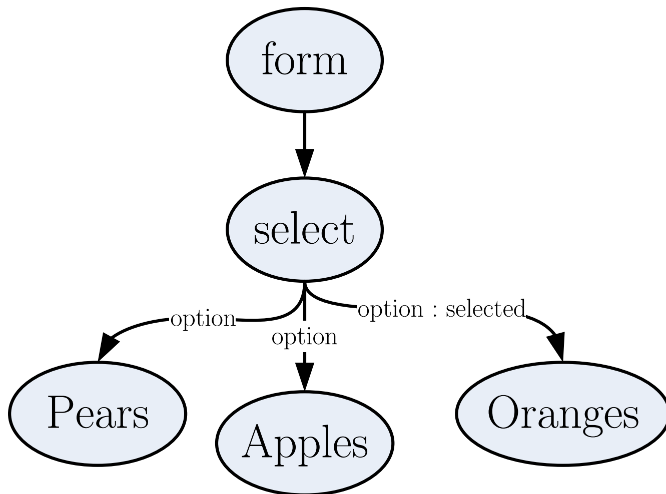

Figure 4.2: Graph model

```
1 form
2     select
3         Pears
4         Apples
5         Oranges
```

Listing 4.6: OGDL

4.4.2 YAML

The recursive acronym[1] YAML stands for YAML Ain't Markup Language and resembles OGDL in that they both uses space and indentation to structure text.

One of the main characteristics that drew the author's attention to it is that YAML is *designed for human readability* [Ingerson *et al.* , 2001-2006]. It uses a syntax that resembles Python [Python, 1990-2006] to indicate structure, using indentation and various characters to separate keys from values. The Listings 4.7 and 4.8 compares how simple customer data can be encoded in XML and YAML.

[1]A recursive acronym is referring to itself in its definition and is a humorous way to construct names often used in computer technology. In the case of YAML, the 'Y' stands for the acronym itself, whereas 'A', 'M' and 'L' stands for the real words Ain't Markup Language.

```
1  <customer>
2    <name>
3      Rune Flobakk
4    </name>
5    <address>
6      Easy Street 3
7    </address>
8    <postalCode>
9      7030
10   </postalCode>
11   <phoneNumbers>
12     <mobile>
13       12345678
14     </mobile>
15     <work time="9,16">
16       23456789
17     </work>
18   </phoneNumber>
19   <rebates>
20     <vip/>
21     <loyalty>5</loyalty>
22   </rebates>
23 </customer>
```

Listing 4.7: XML

```
1  customer:
2    name: Rune Flobakk
3    address: Easy Street 3
4    postalCode: 7030
5    phoneNumbers:
6      - mobile: 12345678
7      - work: 23456789
8        time: 9 16
9    rebates:
10     - vip
11     - loyalty: 5
```

Listing 4.8: YAML

Listings 4.7 and 4.8: Comparison between XML and YAML syntax for encoding customer data.

Chapter 5

Solution proposal

The limits of my language mean the limits of my world

Ludwig Wittgenstein, Australian philosopher

This chapter presents the results of my research activities; the *design artifact*.

5.1 Name

As for most artifacts in the world of programming libraries and frameworks, the artifact presented in this chapter has been given a name. Since it really is a prototype design it can be considered more a working title. The artifact has been named *pyVisage* by the author, following the name convention used by most Python libraries with a preceding "py". The pyVisage name is according to the author's knowledge not already in use by any publicly available library.

5.2 Common motivation

What is the motivation for introducing another "paradigm" into the world of programming, and in this particular case, GUI programming? As already briefly mentioned in the introduction chapter, two arguments for using a declarative approach to GUI programming are:

- It is (arguably) easier to comprehend and maintain
- It requires less lines of code

The following sections will elaborate more on this matter.

5.2.1 GUIs are declarative by nature

A large part of creating user interfaces for software systems is declarative even though traditionally implemented using a sequence of statements in a programming language. One chooses a pre-made GUI widget to use, customizes it for a particular use usually by setting various properties, specifies one or sometimes several actions associated with this widget, and specifies it's compositional relation to another widget.

For instance, to create a button which purpose is to acknowledge some options made in a dialogue window, this is usually implemented as the following sequence of pseudo-statements:

- instantiate a button component (choice of pre-made GUI widget)
- specify the button's text to be "Ok", "Save changes" or similar (setting a property)
- specify what action to take or function or method to call, when the user clicks this button
- add the button to the dialogue window component (specification of the button's compositional relation to the dialogue window, i.e. the dialogue window contains this button)

5.2.2 Actions - a clarification

The author uses the term action as a generalization of the procedures that a GUI component may trigger when interacted upon. What is important is that the action itself is not part of the declarative model.

An action can be viewed as a sequence of executable statements. When the statements have been successfully executed, the action has been performed. This encapsulation concept of executable statements to form an referable entity makes it possible to declaratively bind actions to specific GUI components. It is the actual binding that is part of the declarative model, not the resulting action.

5.2.3 Separation of GUI from the remaining application

Having numerous such statements as in the list in subsection 5.2.1 may clutter up the code and make it hard to read and comprehend. If we manage to separate the configuration of the GUI widgets and their composition from the application code, not only will we gain shorter and more readable code, but it also forces loose coupling between the user interface and the other parts of a software system. This in turn also effectively enables easy shuffling between various GUIs.

5.2.3.1 Aesthetic computing

In the research context of aesthetic computing, there is acknowledged that there is a long history of artists applying mathematics and computer technology to their work. However, a reoccurring question is why there is not any extensive practice of the converse situation, i.e. artistic practices applied to computing. Fishwick's hypothesis is we are only beginning to see the effect of art on computing, as one does now not normally see the same level of artistic theory and practice applied to mathematics and computing.

The effect of separating the GUI from the remaining system and as well ensure efficient means to juggle between different GUIs can be regarded as appliance of one of the three *aesthetic groups*, as stated by Fishwick, the *modality*.

Modality expresses the various ways one interacts, senses and interfaces with objects, which very much is an analogy to the human-computer interaction (HCI) discipline of computer science. As a piece of art can employ pluralism, interaction, dynamism, and materiality to enhance modality, a computer program can employ various user interfaces in order to interact with the program. This has also become more cost efficient by using "[...] technologies such as XML (e.g. with its pronounced content-presentation capability) and mass customization are making it possible to *apply* multiple styles and representations to computing. As the subjectivist hallmark of the arts becomes less expensive, representations in computing will change." [Fishwick, 2006]

5.2.4 Architecture and GUI

Software architecture is a relatively new field of IT research, and its definition has evolved some from the early 70s. In 1975 Fred Brooks defined architecture as "the complete and detailed specification of the user interface". Brooks further writes that "Where architecture tells *what* happens, implementation tells *how* it is made to happen." [Bass *et al.* , 2003] Today we consider the word architecture in a software

context to involve the structure of an entire system, but Brook's distinction is still valid and fits perfectly with the author's argument that GUI development are declarative by nature (subsection 5.2.1). Since GUI development specifies *what* a GUI looks like and *what* actions are triggered by user interaction, it should be separated from the implementation details which describes *how* actions are executed.

5.3 The framework

This section describes the implementation and use of the proposed prototype as a foundation for further development of a GUI component for ImproSculpt which uses XML for setting the user interface.

5.3.1 Handling XML in Python

Python v2.4 has some bundled libraries for basic XML processing and handling, though nothing for applying stylesheets in the form of XSL. The author has chosen to use the 4Suite library [4suite.org, 2006] since it includes means for XSLT processing. It also features an easy installer for the Windows OS which ensures proper installation and makes it ready to use.

5.3.2 Chosen strategy

The author has chosen a strategy inspired by the XSLT/XSL-FO approach already explained in section 4.2.

5.3.2.1 Evolution of the idea

While investigating existing XML technologies within the GUI and information presentation domain, the author was particularly intrigued by the use of Extensible Stylesheet Language, more specifically the XSL Transformations technology which is used to transform an XML document into another format.

During the parallel development of the framework and language it became more and more evident that the quality requirements stating that the XML language must emphasize *readability* and being *easy to edit and maintain* will have great impact on how complicated the parser implementation needs to be. The easiest XML format to parse would be a format employing a set of very few general concepts, and where

the nesting of the elements closely resembles the structure of the generated object hierarchy for the GUI inside the software.

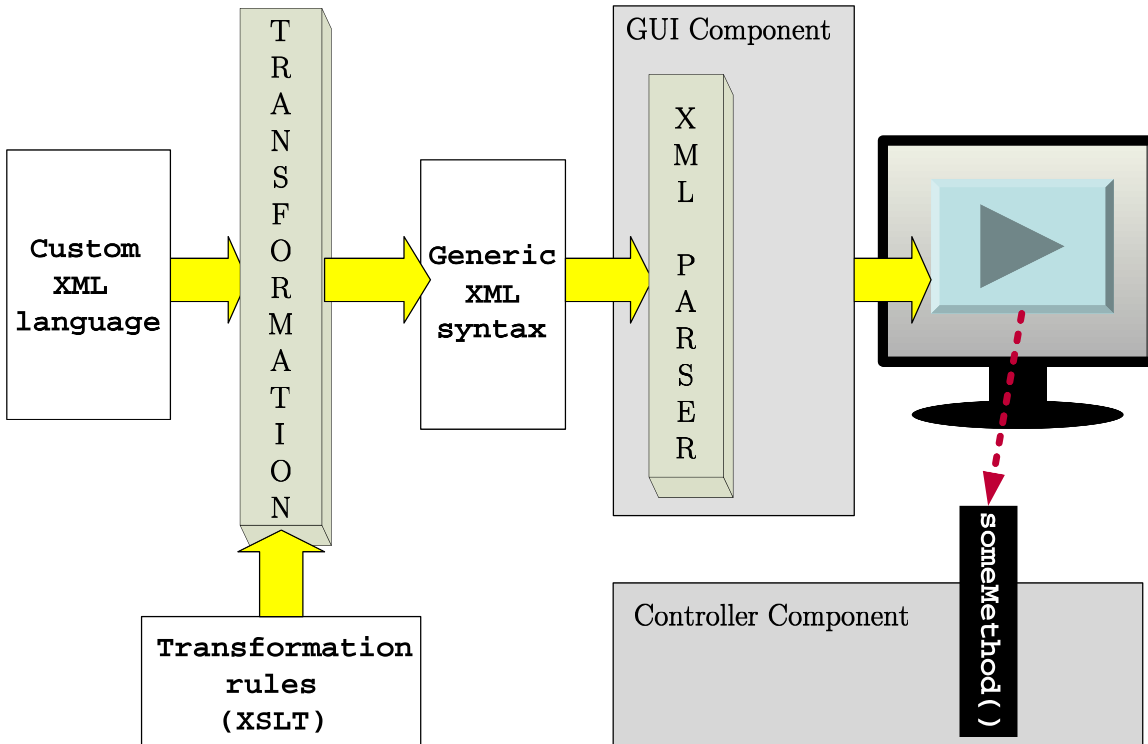

Figure 5.1: Conceptual diagram showing the 2-step processing of XML data to finally be able to set up a graphical user interface. For an animated version of this diagram, visit the web address `http://folk.ntnu.no/flobakk/fordypning/xslt-demo`.

5.3.2.2 Process description

Figure 5.3.2.1 outlines the transformation → parsing → GUI creation concept. (Remark: the figure does not show the actual architecture of the developed framework.)

The *Transformation* component receives the user defined GUI specification together with an XSLT stylesheet. Using the stylesheet, the component can then transform the GUI specification to a more generic format which lies more close to the object model employed by the software, and this greatly reduces the required complexity of the *parser* component. What is an important point is that the transformation part is convergently a no-cost element in terms of development time, as it is only a

matter of utilizing existing functionality of an XSLT-enabled XML processing library (see subsection 5.3.1).

5.3.3 Using the framework

The general use of the framework is very simple, and does mainly involve 2 tasks to be done from the program code:

- Create an instance of the `Visage` class
- Showing this instance's GUI where appropriate

Appendix C contains examples of input files for the framework. Both C.2.1 and C.2.2 shows specifications of the same GUI, while the former also requires the presence of an XSLT specification.

Say the specification in appendix C.2.1 and XSLT specification in appendix C.2.3 are respectively placed in the files `simplegui.xml` and `wxpython.xslt.xml`. Listing 5.1 is the Python code required to build and show the GUI:

```
1 from wxPython.wx import wxPySimpleApp
2 from framework import Visage
3
4 application = wxPySimpleApp()
5
6 gui = Visage( "simplegui.xml", "wxpython.xslt.xml" )
7 gui.getRoot().Show(True)
8
9 application.MainLoop()
```

Listing 5.1: A simple Python program that will start the framework and show the produced GUI shown in Figure 5.2.

We have accomplished a complete separation of the GUI specification from the application code.

5.3.4 Using the XML Schema

Provided in the appendix of this report is also am XML Schema document which describes the grammar of the generic XML language which is used by the parser (i.e. not the XSL transformer component). In addition to its function as a valid

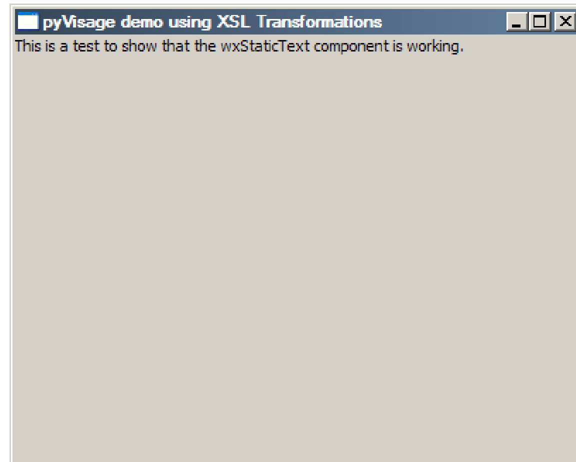

Figure 5.2: Example of a GUI set up using the pyVisage framework.

formal grammar description, it can also be used to provide writing assistance in certain XML editors.

Figure 5.3 shows how this can look like in an XML editor with this kind of functionality.

5.4 XML language

This section provides an informal explanation of the proposed language (XML formats) that the framework uses for input. The author refers to Appendix B for a formal grammar specification.

5.4.1 The generic language

The generic language can be viewed as a map from declarative XML syntax to executable program statements. Its structure closely resembles the object model which the parser builds to provide a GUI to the application. This approach greatly reduces the need to maintain a complex parser implementation, but it also does not provide the developer with a particularly usable XML language. However, the developer is not required to use, or even know about, this language as it is automatically created and handled internally by the framework based on a much simpler and human readable language.

Still, being something that is supposed to be handled transparently inside a framework, it is necessary to explain the concepts of the language. The framework

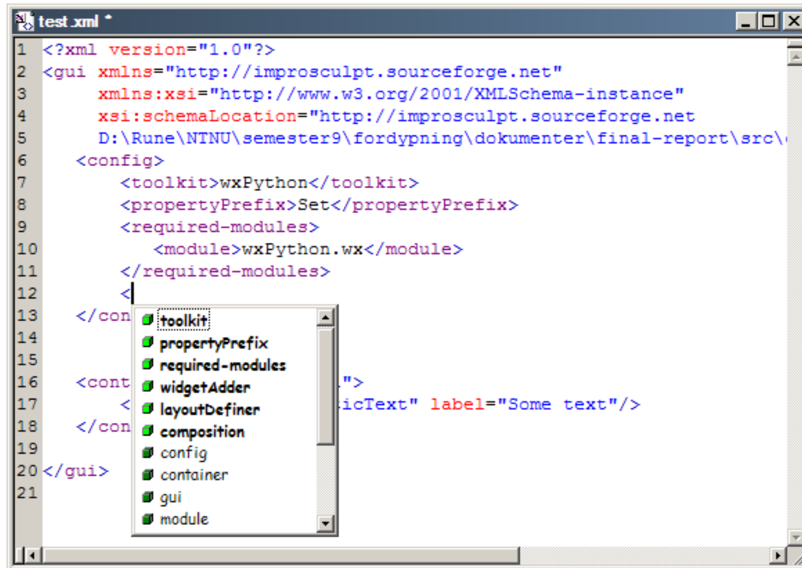

Figure 5.3: Writing a GUI description with assistance from the XML Schema in Appendix B. The selectable pop-up list shows every available word, and the boldfaced ones are valid for the current context.

is merely a proof-of-concept prototype in accordance with Hevner *et al.* 's statements on design science research discussed in subsection 2.2.2, and needs to be further developed. Luckily, the language concept is rather simple since it has focus on generality instead of being easily readable.

5.4.1.1 The root element: `gui`

Every XML document needs exactly one root element in which all other elements reside. The element's name is simply `gui` and really needs no more discussion. Just consider the example below:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<gui>
    ...
    ...
</gui>
```

5.4.1.2 The configuration section: `config`

The first child element inside the `gui` element is expected to be the `config` element, which contains several configuration parameters for the parser to know how it should handle the parsed elements. See the code in subsection C.2.2 for which elements which currently are allowed inside the `config` element.

At this point most of the elements from the mentioned code example are read by the parser, but not everything affects how the parser acts. The elements are present to give an idea of which aspects that must be considered to further develop the framework to support arbitrary GUI toolkits.

The parser will at this point load the modules indicated with the `required-modules` element. It will also use the string specified by `propertyPrefix` to be able to construct method names to set component properties. If this string is set to "Set" and a property named "label" should be set, the parser will call the method `SetLabel()` to inject the property's value.

5.4.1.3 The component elements: `container` and `widget`

The various GUI components are represented by the XML language as `container` and `widget` elements. At this point there is not really any difference in how these two elements are handled by the parser, but the idea is that a `container` is a GUI component which holds other components, i.e. windows, panels, etc, while a `widget` is a top-most (or down-most depending on how the GUI component tree is visualized) component not capable of holding other components. It is thus not legal to put components inside a `widget` element.

The parser currently treats both `container` and `widget` elements equally.

All the required data on how to instantiate components are provided using *attributes* for each element. The parser will recognize a set of specialized attribute names and handled accordingly.

`type` This attribute specifies which class that will instantiate the component. The module containing it must be specified in the configuration section, and the attribute value must be written as a fully qualifiable name, i.e. as `package.module.Class`. There is however one exception for convenience. If one of the modules in the configuration section is specified with the attribute `default="true"`, classes belonging to this module may be specified using the class name only.

`layoutManager` This attribute is recognized by the parser but not handled in any particular way at this point as further discussed in section 5.5 section 5.5. A

layout manager is used by components to be able to place its containing components in an appropriate manner. This attribute indicate a class in the same manner as the `type` attribute.

Any attribute not recognized by the parser is treated as a "property" and will be injected into the component using the method already discussed in subsubsection 5.4.1.2.

5.5 Evaluation

This section presents an evaluation of the pyVisage framework. Table 5.5 summarizes how the goals have been met.

5.5.1 Discussion on language readability

As stated in section 5.5, the readability of the GUI specification language has a potential to be excellent, and is only limited to what effort the developer makes to extend the standard XSLT document. This enables the developer to reduce common compositional constructs into simple elements. One example can be to reduce a question dialog with OK and Cancel buttons into a simple

```
<okCancelDialog>Are you sure?</okCancelDialog>
```

statement.

One idea for a in the future complete framework is to provide several XSLT documents for a set of common GUI toolkits [wxpython.org, 2006, pyFLTK, 2006, JUCE, 2006] which will enable the parser logic to stay static, and also provide a common starting point language which features the most common GUI concepts available in most toolkits.

Recall the question dialog already mentioned in this section. One GUI toolkit may have this as an already pre-made component which only needs to be instantiated, while another might not feature common dialogs at all. The XSLT document can then handle this in a completely transparent manner for the developer, which only needs to know where to put `<okCancelDialog>`. Depending on the GUI toolkit in use, the XSLT document will then either simply create the code for instantiating an appropriate dialog component, or it will instantiate the necessary components and compose them to "emulate" a pre-made dialog component.

Feature	Implemented			Remarks
	Yes	No	Partly	
The framework must be able to set up the screen layout of a chosen set of widgets currently utilized by ImproSculpt.			•	The lack of layout management accordingly prevents the widgets to be placed correctly.
The framework must set up how each of the chosen widgets are connected to the rest of the application logic.		•		This concept is not handled by the framework.
The framework must be able to set up the compositional relationships between widgets.	•			This is fully covered by the use of references to parent and children components. The instantiated widgets are also correctly "added" to their container components.
The framework must be able to set up the compositional relationships between widgets.	•			This is fully covered by the use of references to parent and children components. The instantiated widgets are also properly "added" to their container components.
The framework must have limited support for relevant layout managers.			•	This is not implemented though the language contains constructs for setting it up. They are however at this point ignored by the parser.
Widget support	•			The widgets stated by the requirements specification (Appendix A) are fully supported. In addition, due to the general nature of the language and parser, other simple widgets which are instantiated should also be supported.
GUI specification language must feature great readability	•			This is achieved in a very successful manner using the XSL strategy explained in subsection 5.3.2.

Table 5.1: Overview of desirable framework features and goals and how well they are handled by pyVisage.

Chapter 6

Conclusion and future work

Apparently three out of four people make up 75% of the population

David Letterman

The review of the current state of the art goes into the depths of some of the current buzz-words regarding XML layout definition languages, spanning from the most "mainstream" techniques as XHTML, to an attempt to see into the future on what that might be the superseding XML. XML has a clear disadvantage of being verbose, though it gains advantage from the vast amount of available processing tools (parsers, validators) and techniques (e.g. transformations), as well as being a widely embraced practice.

The outcome of the implementation task shows an interesting approach to GUI design which would be interesting to further develop. The use of XSL Transformations looses the coupling between the parser logic and GUI specification language, and greatly enhances the freedom to design a readable and comprehensible language without worrying about how it will affect the complexities of the parser. Still, introducing XSLT involves the study and learning of yet another technology, and it is still a question whether Brandtsegg wants to investigate the time required to employ this or not.

Indeed, the author feels that this project has maybe provided more to the knowledge base of the GUI development domain than the research field of aesthetic computing and artistic software. Still, he believes that the results are applicable to artistic software development.

Not every planned aspects of the project has been carried out, but the author believes it has still been a success. Building a comprehensive GUI framework is a daunting task, and has probably affected the scope of the project as well as the quality of the implementation. The current framework is in no state to be used in practice, but is hopefully a valid foundation for further development.

6.1 Future work

This section presents suggestions on possibilities for further work and research.

6.1.1 Missing functionality

The pyVisage framework must get some of the still missing functionality implemented before it can be considered to be put to actual use. The two most essential features are support for layout managers, and being able to connect actions to GUI components.

6.1.2 GUI toolkit support

The current concepts of the generic XML language (section 5.4) needs to be investigated in conjunction with an extensive review of the various available GUI toolkits and how they build up GUIs. What parts of the language must be extended to accommodate differences between the toolkits? Is the suggested strategy of a static parser implementation which reads a static generic XML language, and then providing several XSLT documents in order to ensure cross-toolkit independence a viable solution?

6.1.3 Cooperation with an artist

The further work can maybe benefit the research on aesthetic computing and artistic software if it is done in close cooperation with an artist. The project employs the engineer's desire to generalize, and it could be interesting to bring in an artist to maybe provide a counterbalance to these approaches.

6.1.4 Investigating new scripting features of Java 6

On December 11, 2006, Sun Microsystems released the new Java SE 6 platform [Sun Microsystems, 2006], which among various new features includes a new framework and API for scripting languages. This enables scripting environments, as Python, to access Java objects, and vica versa, and this may be interesting to investigate possible ways to extend pyVisage to include support for building GUIs using various Java GUI toolkits as Swing and SWT.

References

4SUITE.ORG. 2006. *4Suite: an open-source platform for XML and RDF processing.* Website: http://4suite.org. Last checked: December 1, 2006.

BASS, LEN, CLEMENTS, PAUL, & KAZMAN, RICK. 2003. *Software architecture in practice.* 2nd edn. Addison-Wesley.

BRANDTSEGG, ØYVIND. 2004 (?). *Oeyvind Brandtsegg Music and Software.* Website: http://oeyvind.teks.no. Last checked: December 20, 2006.

BRANDTSEGG, ØYVIND. 2005. *Nye kunstneriske muligheter ved improvisatorisk bruk av komposisjonsteknikker.* Project description for Research Fellowship in the Arts at the Department of Music, NTNU. (Norwegian document). Available on the Web at http://sofie.khib.no/stipend/prosjekt/brandtsegg%20web.pdf.

COLLET, THIBAULT, ROBLES, MAXIMO RAMIREZ, & BRANDTSEGG, ØYVIND. 2006. *Improsculpt sourceforge website.* Website: http://improsculpt.sourceforge.net. Last checked: December 20, 2006.

COOPER, JAMES W. 1998. *The Design Patterns Java Companion.* Addison-Wesley. Pages 177–185.

CORNFORD, TONY, & SMITHSON, STEVE. 2006. *Project research in information systems.* 2nd edn. Palgrave MacMillian.

DEAKIN, NEIL. 2006 (February). *XULPlanet: XUL Tutorial.* Website: http://www.xulplanet.com/tutorials/xultu. Last checked: November 26, 2006.

EISENBERG, J. DAVID. 2001 (January). *Using XSL Formatting Objects.* O'Reilly Media, Inc. Website: http://www.xml.com/pub/a/2001/01/17/xsl-fo/. Last checked: November 17, 2006.

FISHWICK, PAUL. 2006. *Aesthetic Computing.* Leonardo Books. Chap. 1, An Introduction to Aesthetic Computing, pages 3–27.

FROGNER, MORTEN, & ARULANANTAM, VIMALRAJ. 2003. *Bruk av XML-språk til å beskrive brukergrensesnitt i skrivebordsapplikasjoner.* Project work, TDT4735 Software Engineering, Depth Study, Norwegian University of Science and Technology (NTNU), Norwegian document.

HARRIS, CRAIG. 1999. *Art and Innovation.* Leonardo Books. Chap. 9, The Xerox PARC Artist-in-Residence Program, Paul De Marinis, pages 164–184.

HAWRYSZKIEWYCZ, IGOR. 2001. *Introduction to systems analysis & design.* 5th edn. Prentice Hall.

HETLAND, MAGNUS LIE. 2005. *Beginning Python: From Novice to Professional.* Apress.

HEVNER, ALAN R., MARCH, SALVATORE T., PARK, JINSOO, & RAM, SUDHA. 2004. Design science in information systems research. *MIS Quarterly*, **28**(1), 75–105.

INGERSON, BRIAN, EVANS, CLARK, & BEN-KIKI, OREN. 2001-2006. *YAML Aint'n Markup Language.* Website: `http://yaml.org`. Last checked: November 15, 2006.

JACCHERI, MARIA LETIZIA. 2006. *TDT69 Artistic Software: Processes and Products course website.* Website: `http://www.idi.ntnu.no/~letizia/tdt69`. Last checked: December 21.

JAVASCRIPT. 2006. *Mozilla Developer Center - About JavaScript.* Website: `http://developer.mozilla.org/en/docs/About_JavaScript`. Last checked: November 26, 2006.

JUCE. 2006. *JUCE (Jules' Utility Class Extensions).* Websites: `http://www.rawmaterialsoftware.com/juce/` and `http://pyjuce.tuxfamily.org/templeet/pyjuce/`. Last checked: November 16th.

KHIB, BERGEN NATIONAL ACADEMY OF THE ARTS. 2006. *Programme for Research Fellowships in the Arts.* Website: `http://www.kunststipendiat.no`. Last checked: December 18, 2006. For English description, click 'Summary in English' on the website menu.

LUGER, GEORGE F. 2005. *Artificial intelligence - structures and strategies for complex problem solving.* Pearson Education Limited. Chap. 2, pages 80–84.

MOZILLA ORGANIZATION. 1998-2006a. *Mozilla.org - Home of the Mozilla Project.* Website: `http://www.mozilla.org`. Last checked: November 21, 2006.

Mozilla Organization. 1998-2006b. *XML User Interface Language (XUL)*. Webpage: http://www.mozilla.org/projects/xul. Last checked: November 7, 2006.

Py++. 2006. *The py++ package*. Website: http://language-binding.net/pyplusplus/pyplusplus.html. Last checked: November 4, 2006.

pyFLTK. 2006. *pyFLTK - Python wrapper for the Fast Light Took Kit*. Website: http://pyfltk.sourceforge.net. Last checked: November 16th.

Python. 1990-2006. *Python Programming Language*. Website: http://python.org. Last checked: November 21, 2006.

Rzeszótko, Jaroslaw. 2006 (July). *stifflog^2 - Stiff asks, great programmers answer*. Webpage: http://sztywny.titaniumhosting.com/2006/07/23/stiff-asks-great-programmers-answers. Last checked: October 15, 2006.

Sedelow, Sally Yeates. 1970. The Computer in the Humanities and Fine Arts. *Computing surveys*, **2**(2).

Semb, Thor Arne Gald, & Småge, Audun. 2005. *Artistic software*. Project work, TDT4735 Software Engineering, Depth Study, Norwegian University of Science and Technology (NTNU).

Semb, Thor Arne Gald, & Småge, Audun. 2006. *Software architecture of the algorithmic music system improsculpt*. M.Phil. thesis, Norwegian University of Science and Technology (NTNU).

Sun Microsystems. 1995-2006. *The Java^{TM} Tutorials: How to Use BorderLayout*. Website: http://java.sun.com/docs/books/tutorial/uiswing/layout/border.html. Last checked: November 26, 2006.

Sun Microsystems. 2004. *Api documentation for BorderLayout*. Website: http://java.sun.com/j2se/1.5.0/docs/api/java/awt/BorderLayout.html. Last checked: November 26, 2006.

Sun Microsystems. 2006 (December). *Feature Story: Java Gets Better. Faster. Easier.* Website: http://www.sun.com/2006-1211/feature/index.jsp. Last checked: December 21.

SWIG. 2006. *Simplified Wrapper and Interface Generator*. Website: http://www.swig.org. Last checked: November 4, 2006.

TCHISTOPOLSKII, PAUL. 2006. *XML Alternatives*. Website: http://www.pault.com/pault/pxml/xmlalternatives.html. Very extensive list of the available alternatives to XML. Last checked: November 15, 2006.

W3C. 1994-2006. *World wide web consortium*. Website: http://www.w3.org. Last checked: November 11, 2006.

W3C CSS. 1994-2006. *Cascading Style Sheets Homepage*. Website: http://www.w3.org/Style/CSS. Last checked: November 11, 2006.

W3C XML SCHEMA. 2004 (October). *World Wide Web Consortium XML Schema*. Website: http://www.w3.org/XML/Schema. Last checked: October 29, 2006.

WXPYTHON.ORG. 2006. *wxPython*. Website: http://wxpython.org. Last checked: November 16th.

Index

action, 26
action research, 6, 9
aesthetic computing, 27, 36
architecture, 27
artist, 37
artistic, 27
attribute, 34
attributes, 33

behavioral science, 6
best practice, 2
box model, 20

Cascading Style Sheets, *see* CSS
composition, 27
computational, 8
config, 33
container, 33
CSS, 16
cyclic, 6, 9

declarative, 3, 26, 28, 31
design, 6
design artifact, 6, 25
design science, 5–9

empirical, 8
encapsulation, 26
evaluation, 6, 8
executable, 26
Extensible Stylesheet Language, *see* XSL

framework, 8, 25, 28, 30, 37

goals, 2, 5, 34
grammar, 30
graph, 22
graphical user interface, *see* GUI
GUI, 3, 15, 27, 28
guidelines, 7

HCI, 27
HTML, 14
human-computer interaction, *see* HCI

implementation, 28, 36

layoutManager, 33
loose coupling, 36

markup language, 14
mathematical, 8
modality, 27

object model, 29

parser, 28–30, 33, 34
property, 33, 34
propertyPrefix, 33
prototype, 25, 28
pyVisage, 34, 37

readability, 28, 34
required-modules, 33
research, 5
  framework, 5
  questions, 3
routine engineering, 6

separation, 27, 30
stylesheet, 28, 29

tags, 14
transformation, 29
type, 33

web pages, 15
widget, 26, 33

XBL, 21
Xerox PARC, 2
XHTML, 14–16, 36
XML, 1–3, 14, 27, 28, 36
XML Schema, 30
XSL, 16, 28
    Formatting Objects, *see* XSL-FO
    Transformations, *see* XSLT
XSL-FO, 16
XSLT, 16, 28, 34
XUL, 21

Appendix A

Requirements specification

A.1 Artefacts

The requirements specification describes the artifacts that this project will output:

A.2 A proof-of-concept framework implemented in Python that will set up a subset of the existing GUI of ImproSculpt (from now refered to as “the framework”).

A.3 Language specification of the XML language(s) used by the framework.

A.2 Proof-of-concept framework

This artifact is an initial proof-of-concept implementation of a framework which enables the developer to specify the GUI of ImproSculpt using XML. The sections below describes various aspects and constraints this framework must adhere to.

A.2.1 Expressiveness

This section describes specifically what the framework must be able to achieve.

- The framework must be able to set up the screen layout of a chosen set of widgets currently utilized by ImproSculpt. See A.2.2 for which widgets that must be supported.
- The framework must be able to set up how each of the chosen widgets are connected to the rest of the application logic.

- The framework must be able to set up the compositional relationships between widgets, i.e. to properly set up widgets belonging to containers.
- the framework must have limited support for relevant layout managers. Layout managers are GUI components that handles, often dynamic, positioning, of widgets rather than using absolute positioning.

A.2.2 Widgets

This is a list of the widgets the framework must support:

- text label
- text field
- button

In addition, the framework must be able to express the composition of GUI elements and containers belonging to other containers.

A.3 Language specification

The language must be described in a non-ambiguous way. To achieve this, XML Schemes [W3C XML Schema, 2004] of the XML language(s) used by the framework must be produces. This works both as a formal grammar description, and enables certain more advanced XML editors to use the XML Schemes to provide auto-completion and other XML authoring aids to the developer. This in turn gains the requirement of easy manual editing which is discussed in a below section.

In addition, there must be created an informal textual description of the language, how it is used, and simple examples of how it is used.

A.3.1 Language design

The language design must emphasize

- human readability
- ease of manual editing

as well as being powerful enough to create fairly complex user interfaces. The verbosity of the language must balance these two goals. Readability will suffer from a too verbose syntax, but at the same time one should not be excessively constrained by using this language instead of traditional programming.

While the design should focus on generality, it must not be on cost of the readability. The language should contain several different keywords instead of a few general ones to ensure readability.

A.3.2 Independence

The language should be *independent of the chosen GUI toolkit*, and the various language elements should not be directly linked to the constructs of a particular toolkit, but instead reflect common similarities between the various available toolkits.

Appendix B

XML Schema

```
1  <?xml version="1.0"?>
2  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
3              targetNamespace="http://improsculpt.sourceforge.net"
4              xmlns="http://improsculpt.sourceforge.net"
5              elementFormDefault="qualified">
6
7     <xsd:element name="gui">
8         <xsd:complexType>
9            <xsd:all maxOccurs="1" minOccurs="1">
10              <xsd:element name="config" type="config"/>
11              <xsd:element name="container" type="container"/>
12           </xsd:all>
13        </xsd:complexType>
14    </xsd:element>
15
16    <xsd:complexType name="config">
17       <xsd:all maxOccurs="1">
18           <xsd:element name="toolkit"/>
19           <xsd:element name="propertyPrefix"/>
20           <xsd:element name="required-modules" type="modules"/>
21           <xsd:element name="widgetAdder"/>
22           <xsd:element name="layoutDefiner"/>
23           <xsd:element name="composition">
24              <xsd:complexType>
25                 <xsd:attribute name="method" type="xsd:string"/>
26              </xsd:complexType>
27           </xsd:element>
28       </xsd:all>
29    </xsd:complexType>
30
31    <xsd:complexType name="modules">
32           <xsd:all maxOccurs="1">
33               <xsd:element name="module">
34                   <xsd:complexType mixed="true">
35                    <xsd:attribute default="true" name="default" type="xsd:boolean
                          "/>
36                   </xsd:complexType>
37               </xsd:element>
38           </xsd:all>
39    </xsd:complexType>
40
```

```
41     <xsd:complexType name="container" mixed="true">
42        <xsd:choice maxOccurs="unbounded">
43           <xsd:element name="container" type="container"/>
44           <xsd:element name="widget">
45              <xsd:complexType>
46                 <xsd:attributeGroup ref="componentRequirements"/>
47              </xsd:complexType>
48           </xsd:element>
49        </xsd:choice>
50        <xsd:attributeGroup ref="componentRequirements"/>
51        <xsd:attribute name="layoutManager"/>
52     </xsd:complexType>
53
54     <xsd:attributeGroup name="componentRequirements">
55        <xsd:attribute name="type" use="required"/>
56        <xsd:anyAttribute/>
57     </xsd:attributeGroup>
58
59  </xsd:schema>
```

Listing B.1: XML Schema document for the generic GUI description language demonstrated in subsection C.2.2

Appendix C

Code examples

This appendix contains a selection of complete code examples too extensive to include in the report text.

C.1 “BorderLayout” in XUL

```
<?xml version="1.0"?>
<?xml-stylesheet href="chrome://global/skin/" type="text/css"?>
<window id="border-layout" title="Border Layout Demo"
        xmlns="http://www.mozilla.org/keymaster/gatekeeper/there.is.only.xul">

<vbox flex="1" align="stretch">
   <hbox pack="center" style="border-bottom: 1px black solid;">
      <label style="font-size: 16pt">
         This is the top panel
      </label>
   </hbox>

   <hbox align="stretch" flex="1">
      <vbox style="border-right: 1px black solid;">
         <label>This is the left panel</label>
      </vbox>
      <hbox flex="1">
         <label>This is the centre panel</label>
      </hbox>
      <vbox style="border-left: 1px black solid;">
         <label>This is the right panel</label>
      </vbox>
   </hbox>

   <hbox pack="center" style="border-top: 1px black solid;">
      <label>This is the bottom panel</label>
   </hbox>
</vbox>
</window>
```

Listing C.1: Java’s BorderLayout implemented in XUL.

C.2 Examples on XML data utilized by pyVisage

C.2.1 A simple GUI specification

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<gui>
  <window width="400" height="300">
    <title>pyVisage demo using XSL Transformations</title>
    <label>This is a test to show that the wxStaticText component is working.</
        label>
  </window>
</gui>
```

Listing C.2: A simple GUI specification.

C.2.2 Auto generated generic GUI specification

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<gui>
  <config>
    <toolkit>wxPython</toolkit>
    <required-modules>
      <module default="true">wxPython.wx</module>
      <module>wxPython.gizmos</module>
    </required-modules>
    <composition method="constructor"/>
    <widgetAdder>Add</widgetAdder>
    <layoutDefiner>setSizer</layoutDefiner>
    <propertyPrefix>Set</propertyPrefix>
  </config>

  <container type="wxFrame" clientSizeWH="400,300"
             title="pyVisage demo using 'raw' XML input">
    <container type="wxPanel" layoutManager="wxGridBagSizer">
      <widget type="wxStaticText"
              label="This is a test to show that the wxStaticText component is
                  working."/>
    </container>
  </container>
</gui>
```

Listing C.3: Auto generated generic GUI specification using XSLT.

C.2.3 XSLT document

This XSLT document describes how to transform the specification in appendix C.2.1 into a generic XML format showed in appendix section C.2.2.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <gui>
```

```
7         <!-- Common configuration for the wxWidgets GUI toolkit -->
8         <config>
9           <toolkit>wxPython</toolkit>
10          <required-modules>
11            <module default="true">wxPython.wx</module>
12            <module>wxPython.gizmos</module>
13          </required-modules>
14          <composition method="constructor"/>
15          <widgetAdder>Add</widgetAdder>
16          <layoutDefiner>setSizer</layoutDefiner>
17          <propertyPrefix>Set</propertyPrefix>
18        </config>
19
20        <xsl:apply-templates/>
21
22      </gui>
23    </xsl:template>
24
25    <xsl:template match="window">
26      <!--
27        This template will create a wxFrame with a containing wxPanel
28        which uses the wxGridBagSizer as layout manager
29      -->
30      <xsl:element name="container">
31        <xsl:attribute name="type">wxFrame</xsl:attribute>
32        <xsl:attribute name="clientSizeWH">
33          <xsl:value-of select="@width"/>,<xsl:value-of select="@height"/>
34        </xsl:attribute>
35        <xsl:attribute name="title"><xsl:value-of select="title"/></xsl:attribute>
36        <container type="wxPanel" layoutManager="wxGridBagSizer">
37          <xsl:apply-templates/>
38        </container>
39      </xsl:element>
40    </xsl:template>
41
42    <xsl:template match="label">
43      <!--
44        This template will create a wxStaticText component
45      -->
46      <xsl:element name="widget">
47        <xsl:attribute name="type">wxStaticText</xsl:attribute>
48        <xsl:attribute name="label"><xsl:value-of select="."/></xsl:attribute>
49      </xsl:element>
50    </xsl:template>
51
52  </xsl:stylesheet>
```

Listing C.4: XSLT document describing transformation rules for the code in Listing C.2