Dynamically generated multi-modal application interfaces

Dissertation

to receive the academic degree Doktoringenieur (Dr.-Ing.)

submitted to the
Technical University Dresden
Faculty Computer Science

submitted by
Dipl. Ing. Stefan Kost
born on April 6, 1974 in Leipzig/Germany

supervising university lecturers
Prof. Dr. Wolfgang Wünschmann, TU Dresden
Prof. Dr. Klaus Bastian, HTWK Leipzig

Leipzig, June 18, 2006
Acknowledgment

I would like to express my thanks to all people who supported me writing this work. My very special appreciation goes to my supervisors Prof. Wolfgang Wünschmann and Prof. Klaus Bastian for their excellent guidance and efforts on my behalf.

In addition, I wish to thank the following for their contributions and assistance:

Thomas Wabner, Annett Sauer and my parents - general support
diploma students : Sebastian Paul, Steffen Ast, Michael Randel
proof readers: Eugen Weber, Frank Becker, Christian Hertel and Patricia Joliet
SourceForge.net - development resources
SMWK Dresden - funding

Daniel Veillard - libxml, libxslt
Arien Malec - check unit test library
Bruno Haible - gettext
Slava Pestov - jedit
Alan W Black - festival speech synthesis system
various authors - glib/gobject
Thomas Esser - TeTex

Finally a big thank you goes to all open source developers worldwide and to all contributors of the free dictionary at wikipedia.org.
Abstract

This work introduces a new UIMS (User Interface Management System), which aims to solve numerous problems in the field of user-interface development arising from hard-coded use of user interface toolkits.

The presented solution is a concrete system architecture based on the abstract ARCH model consisting of an interface abstraction-layer, a dialog definition language called GIML (Generalized Interface Markup Language) and pluggable interface rendering modules. These components form an interface toolkit called GITK (Generalized Interface ToolKit). With the aid of GITK (Generalized Interface ToolKit) one can build an application, without explicitly creating a concrete end-user interface. At runtime GITK can create these interfaces as needed from the abstract specification and run them. Thereby GITK is equipping one application with many interfaces, even kinds of interfaces that did not exist when the application was written. It should be noted that this work will concentrate on providing the base infrastructure for adaptive/adaptable system, and does not aim to deliver a complete solution.

This work shows that the proposed solution is a fundamental concept needed to create interfaces for everyone, which can be used everywhere and at any time.

This text further discusses the impact of such technology for users and on the various aspects of software systems and their development.

The targeted main audience of this work are software developers or people with strong interest in software development.
## Contents

**1 Introduction**

1.1 Motivation ........................................... 1
1.2 Aim of this work .................................... 4
1.3 Structure .......................................... 6

**2 Fundamentals**

2.1 Interaction model .................................... 7
2.2 Adaptation ........................................... 10
  2.2.1 Adaptation infrastructure ....................... 11
  2.2.2 Adaptation of application interfaces ............ 12
  2.2.3 Platform independence .......................... 13
  2.2.4 Internationalization ............................ 13
  2.2.5 Localization .................................. 14
  2.2.6 Culture ....................................... 14
  2.2.7 UIMS - User interface management system ...... 15
2.3 Interface ........................................... 16
  2.3.1 Aspects of an interface ......................... 16
  2.3.2 Application interface .......................... 17
  2.3.3 Interface object ................................ 17
  2.3.4 Interface toolkit ................................ 18
  2.3.5 Interface design ................................ 18
  2.3.6 Interface renderer .............................. 19
  2.3.7 Layout renderer ................................. 19
  2.3.8 Dialog ........................................ 20
  2.3.9 Dialog decoration .............................. 21
2.4 Types of applications ............................... 21
2.5 Application context ................................ 23
2.6 Interfaces for human computer interaction ............ 23
  2.6.1 GTK+ ........................................... 24
  2.6.2 Java AWT ....................................... 26
  2.6.3 JFC - Java Foundation Classes .................. 26
CONTENTS

2.6.4 MFC - Microsoft Foundation Classes .......................... 26
2.6.5 QT ................................................. 27
2.6.6 Summary .......................................... 28

2.7 Interfaces for computer computer interaction ....................... 28
2.7.1 Rexx ................................................. 29
2.7.2 WebServices ........................................ 29
2.7.3 Wsh and VisualBasic .................................... 29
2.7.4 Summary ............................................. 29

2.8 Interface presentation ........................................... 30

2.9 Interface models .............................................. 30
2.9.1 Seeheim model ........................................ 30
2.9.2 Arch model .......................................... 32
2.9.3 MVC model .......................................... 33
2.9.4 PAC model .......................................... 35
2.9.5 OAI model .......................................... 36
2.9.6 MMI framework ....................................... 36
2.9.7 Cameleon framework .................................... 37
2.9.8 Summary ............................................. 38

2.10 Standards .................................................. 38
2.10.1 Summary ............................................. 41

3 Problem description and task details ................................. 45

3.1 Examples .................................................. 47
3.1.1 User account administration ................................ 48
3.1.2 Address manager ....................................... 49

3.2 Multi-dimensional adaptation .................................... 49
3.2.1 General adaptation ..................................... 50
3.2.2 Technical adaptation .................................... 52
3.2.3 Cultural adaptation ..................................... 53
3.2.4 Personal adaptation ..................................... 54

3.3 Problems related to interface structure ............................. 54
3.3.1 Generalization versus specialization ........................ 55
3.3.2 Logical interface description ................................ 56

3.4 Problems related to interface style ................................. 58
3.4.1 Domain mapping ........................................ 58
3.4.2 Presentation mapping .................................... 60
3.4.3 Interface object mapping .................................. 62

3.5 Problems related to interface behavior ............................. 63
3.5.1 Dialog navigation ....................................... 64
3.5.2 Interaction serialization ................................... 65
3.5.3 Multiple interface instances ............................. 66

3.6 Problems related to interface content .............................. 67
3.6.1 Content selection ....................................... 67
CONTENTS

3.6.2 State persistence ........................................ 68
3.7 Technical problems ....................................... 68
  3.7.1 Programming language bindings ....................... 68
  3.7.2 Operating System integration ......................... 68
3.8 Development support ...................................... 69
3.9 Summary .................................................. 70

4 Existing approaches ......................................... 72
  4.1 Abstract toolkits ........................................ 72
    4.1.1 UIML (User Interface Markup Language) ............ 74
    4.1.2 XUL (XML User-interface Language) ................ 75
    4.1.3 XIML (eXtensible Interface Markup Language) .... 75
    4.1.4 XForms ........................................... 76
    4.1.5 AUIML (Abstract User Interface Markup Language) . 77
    4.1.6 XMLFace ........................................... 77
    4.1.7 wxWidgets ......................................... 78
  4.2 Other approaches ........................................ 78
    4.2.1 DialogML ........................................... 78
    4.2.2 Glade ............................................. 79
    4.2.3 PonG ............................................. 79
    4.2.4 HTML (HyperText Markup Language) ................. 80
    4.2.5 VoiceXML .......................................... 81
    4.2.6 WAP (Wireless Application Protocol) and WML (Wireless Markup Language) ......................... 81
  4.3 Comparison of existing approaches ...................... 82

5 The Generalized Interface ToolKit .......................... 85
  5.1 Architecture ............................................ 87
    5.1.1 Application ........................................ 90
    5.1.2 Wrapper (access-layer) ............................. 90
    5.1.3 Core (libgitk) ..................................... 91
    5.1.4 Renderer-plugins .................................. 95
    5.1.5 Interface toolkit .................................. 96
  5.2 Multi-dimensional adaptation ............................ 96
  5.3 GIML (Generalized Interface Markup Language) .......... 97
    5.3.1 Technical foundation ................................ 97
    5.3.2 Language definition and validation .................. 98
    5.3.3 Document processing ................................ 100
    5.3.4 Relation to other languages ......................... 101
  5.4 Solutions related to interface structure ............... 101
    5.4.1 Generalization versus specialization ............... 101
    5.4.2 Logical interface description ....................... 101
  5.5 Solutions related to interface style .................... 103
## CONTENTS

5.5.1 Domain mapping ........................................... 103
5.5.2 Presentation mapping .................................... 103
5.5.3 Interface object mapping ................................ 106
5.5.4 Interface object hierarchy ............................... 107
5.5.5 Interface objects in GTK+ ................................ 110
5.5.6 Interface objects in text mode ......................... 111
5.5.7 Interface objects in a 3D environment ................. 111
5.5.8 Interface objects in phone mode ....................... 111
5.6 Solutions related to interface behavior .................. 112
  5.6.1 Dialog navigation ..................................... 112
  5.6.2 Interaction serialization .............................. 115
  5.6.3 Multiple interfaces .................................. 116
5.7 Solutions related to interface content .................. 116
  5.7.1 Content selection .................................... 116
  5.7.2 State persistence .................................... 117
5.8 Technical solutions ..................................... 119
  5.8.1 Language bindings .................................... 119
  5.8.2 Operating System integration ......................... 119
5.9 Development support ..................................... 120
5.10 Reference implementation ............................... 121

6 Conclusion ...................................................... 122
  6.1 Result ................................................... 122
  6.2 Impact of the work ...................................... 124
  6.3 Comparison to other approaches ....................... 124
  6.4 Future ................................................... 125
    6.4.1 Standardisation ................................... 125
    6.4.2 Adaptation profiles ............................... 125
    6.4.3 Interface presentation ............................ 125
    6.4.4 Reference implementation ......................... 126
    6.4.5 Real life evaluation .............................. 126

7 Appendix ......................................................... 127
  7.1 GIML DTD .................................................. 127
  7.2 Simple example c-source code ............................ 128
  7.3 Simple example dialog definition ....................... 131
  7.4 Example dialog definitions in other languages ....... 132
    7.4.1 UIML (User Interface Markup Language) dialog example .................. 132
    7.4.2 XUL (XML User-interface Language) dialog example ................ 132
    7.4.3 Pong dialog example ............................... 133
    7.4.4 Glade dialog example .............................. 135
  7.5 Installation directory layout ......................... 137
  7.6 User manual ............................................. 137
## CONTENTS

7.6.1 general example command line options .................................... 137  
7.6.2 GITK command line options .................................................. 137  
7.6.3 GITK environment variables ................................................ 138  
7.6.4 GITK text renderer command line options ............................. 138  
7.6.5 GITK text renderer keyboard commands ................................. 138  
7.6.6 Included example applications ........................................... 139  

7.7 Developer manual ...................................................................... 140  
7.7.1 GITK Core API ................................................................. 140  
7.7.2 GITK Renderer API ............................................................ 140  
7.7.3 Filesystem structure for components ................................. 140  
7.7.4 Option naming scheme ....................................................... 142  

Bibliography ................................................................................. 144  

Index ............................................................................................ 151  

Glossary ....................................................................................... 156  

List of Tables ................................................................................ 161  

List of Figures ............................................................................... 162
Chapter 1

Introduction

1.1 Motivation

Today’s interface technology is insufficient for the user’s needs and modern usage requirements.

Why is this true? The current software market offers a wide variety of system platforms with a large number of different interface technologies (e.g. user interface toolkits). This variety puts strong technological demands onto software development. It needs flexible software architecture to support various current and future technologies. The situation becomes even more difficult with the diversification of the potential user base, as more and more people get access to technology or are even forced to use it. Table 1.1 gives some numbers of the latest development in Germany. It shows the percentage of households that have pc-technology, those having Internet access and mobile phones.

<table>
<thead>
<tr>
<th>Year</th>
<th>PC</th>
<th>Internet</th>
<th>Mobile phones</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>47%</td>
<td>16%</td>
<td></td>
</tr>
<tr>
<td>2002</td>
<td>36%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td>64%</td>
<td>47%</td>
<td>72%</td>
</tr>
</tbody>
</table>

Table 1.1: Information technology in german households 2004 (editorial staff, 2004)

This diversification opens a gap made of missing knowledge in technology utilisation on one side and missing adaptivity to all the usage scenarios on the other side (see figure 1.1). The eAccessibility expert working group of the European Union outlined this situation regarding disabled people as follows:

The challenge is to ensure that the principle of Design for All is applied as much and as well as possible to new products, systems and services and that new technical advances can be deployed, via standard interfaces, to existing equipment and services ... enabling their use by disabled users to enhance rather than diminish their experience.
Figure 1.1: Growing gap between users and computer applications

(eAccessibility expert working group, 2002)[page 27]

The author believes that using adaptive solutions is the best way to cope with the variety in technology and users. At present interfaces for an application are usually developed separately for each purpose and each platform. There are numerous disadvantages arising from this methodology. One can easily see that:

- The application is limited to the included interface(s).
- The application is dependent on the underlying user interface technology.
- Therefore the application is often tied to one operation system (e.g. MFC on Windows, GTK+ on Windows/UNIX), the OS (Operating System) supported by the toolkit.
- If one adds functionality to the application, changes must be made to several interfaces.
- User interfaces are often designed too specific, e.g. visually oriented, to be generally adaptable.
- Often only one view exists to certain functionality (the interface cannot be presented differently to e.g. novices and experts).
- The correctness of the interface design cannot be verified (in the meaning of appropriate use of interface objects).
- Current interfaces do not necessarily conform to platform style guidelines.

Especially for the end-user, in many situations these disadvantages result in complications, which can even make the application unusable. Example scenarios are:

- A fixation on visual (graphically-oriented) user interfaces is insufficient for blind or visually impaired people, who have difficulties working with applications that
CHAPTER 1. INTRODUCTION

use spatial (non-serial) interfaces.
The keyword is accessibility.

• Jobs where both hands are occupied and the computer has to be controlled alternatively e.g. for a surgeon or a mechanic.
  Keywords are assistive systems and multi-modal interaction.

• One needs to work with the same application (accessing the same database) across several platforms and/or from several physical locations (at home or at work).
  Keywords are mobile computing, roaming and high availability.

• The application will be used by people with different knowledge levels, who could perform better, if given personalized views.
  Keywords are knowledge awareness and user modelling.

The situation is not much better from the developer’s point of view. This too can be illustrated with a few examples:

• The developer designs a user interface although he usually is not a designer.

• Instead of leaving the choice to the end user, the developer has to choose the kind of presentation (verbosity, choice of interface objects).

• Equipping an application with a real flexible interface is very work intensive.

• The developer is bound to use programming languages and platforms that the chosen interface toolkit supports.

Another argument is that the current situation effectively blocks progress. The architectural models that applications are using today are too inflexible with regard to how the interfaces are tied to the application. This makes it difficult or even technically impossible to exchange the built-in interfaces with ones that offer new approaches and ideas; with ones that are appropriate for the user who is actually using the software and the surrounding environment. Voice output systems for example have been available for quite a long time, yet the possibility to switch to an interface offering speech technology is still not a standard application feature. Current application technology mainly relies on the low-level driver concept. That means a new technology can only be added as a subclass of existing facilities, e.g. a voice recognition system can only act as a keyboard, leaving no room for new ideas as this is not a keyboard.

This work targets two user groups. One party are the end-users. By using adaptive interface technology they undoubtedly would have a bigger chance to get the user-interface of their own choice. Especially disabled people would benefit greatly. The other party are the application developers. For this group there are numerous advantages too. The exchangeability of the interfaces would allow developers to concentrate on the application logic and the functional interface design. Thus developers would not
need to layout the interface or paint graphical buttons. Instead they would be able to concentrate on the core of their work. This also results in an economic advantage. Being able to generate interfaces on demand instead of designing them manually reduces development time while the adaptiveness of the product increases its attractiveness.

Software developers are subsequently the main audience of the concepts and ideas presented in this work, as they would have to proactively use them in their products, so as to make them available to the end-user.

1.2 Aim of this work

An adaptive interface toolkit can provide a generic solution that delivers the matching interface for the users needs and working conditions.

Figure 1.1 shows that the base problem could be solved by training users and by producing adaptive software packages. This work will look at the adaptivity part. Although several approaches and solutions already exist in this field, current technological development created the need for another solution. Chapter 4 examines such existing approaches. One of the conclusions is that many approaches try to keep the possible complexity of the applications they support high, but then face difficulties in covering a wide range of modalities. It will be shown that sacrificing flexibility for complexity is not necessary for a wide range of applications.

This work claims that a more universal concept for adaptation that still works for many applications can be provided.

Another motivation for this project is that many other projects start working towards a result without a clear up-to-date definition of what the desired result is. Therefore this work starts with a profound discussion of terminology. The field of human-computer interaction and adaptive systems is far too huge to solve everything at once. It is necessary to understand the whole workflow of creating an adaptive system first, so that it can be broken down into small tasks, which can then be addressed one by one.

Finally it can be criticized that most existing projects reinvent too much, instead of reusing technology that already exists.

This work aims to develop an architectural model based on a new interaction model. The focus of the work will be put on creating a modular design to support the complexity of the workflow. From the technical point of view, the solution should achieve a much higher degree of portability (e.g. from UI-classes) and device independence (namely input- and output-devices) of user interfaces than other solutions. The concept will be open for new technologies. In fact it will even invite their inclusion.

Regarding the end-user, this project strives to find a solution which brings more choice and flexibility without sacrificing clarity. There should be interfaces for everyone. (Stephanidis, 2001; Gimson, 2003)

This work is not just about creating a new model of how user interfaces work. Many models already have been published. It is time to actually provide a working
architecture and an implementation, so that adaptive interface technology can be evaluated in real-life situations. Thus the focus of this work regarding this aspect is the run-time of an application and not so much the design phase.

To summarize, this work aims to show that:

1. **complexity vs. universality:**
   By limiting the presentational complexity, a much greater universality can be achieved.

2. **multi-modal:**
   There is no reason for adaptive toolkits to mainly concentrate on graphical presentation.

3. **separation of concerns:**
   It is possible and also desirable to separate style and functionality in the interface definition.

4. **autonomous adaptation:**
   An interface can be generated, without the application needing to provide adaptation profiles for different targets.

5. **adaptation control:**
   Interface adaptation needs to consider not just the user, but also the usage conditions (environment).

6. **reuse of technology:**
   Such a solution can be based on many standardized and well established technologies; in fact it can bind the many specific solutions together.

7. **generally applicable:**
   Also an application can use another application and thus benefit from adaptive interfaces.

8. **dynamic adaptation:**
   Interfaces should not be 'generated' (statically); the adaptation of interface needs to be treated as a continuous process.

9. **social impact:**
   Adaptive technology is necessary for everyone and not just for minority groups (such as elderly or disabled people).

The group around Jean Vanderdonckt created the term *plasticity* to describe such a behavior of interface toolkits:

We define a context of use as the set of values of variables that characterize the computational device(s) used for interacting with the system as well
as the physical and social environment where the interaction takes place. A UI is plastic if it is able to adapt to context changes while preserving usability. (Calvary et al., 2002)

At the end of this document, the findings of the work will be summarized and reflected on these aforementioned claims.

1.3 Structure

The next chapter (see chapter 2) starts with definitions and discussion of terms, like adaptation and interface. It also briefly examines existing interaction models for user interfaces. This section is recommended reading for anyone new to the field of adaptation of interfaces.

Based on the aims of the project described in the last section, the third chapter identifies and structures the problems that one has to solve to reach these aims. Anyone who is familiar with the problems in detail might skip most the parts.

Chapter 4 introduces existing solutions and strategies and highlights their advantages and disadvantages. Among the solutions are projects such as UIML, XUL and XForms. This chapter follows the structure of chapter 3 as the approaches are measured against the catalog of problems identified before. This chapter might be skipped by anyone who already knows all the approaches discussed in it.

The analysis of the existing approaches is followed by the detailed description of the proposed solution in this work (see chapter 5). After describing the fundamentals like the architecture of the system, the chapter will mirror the structure of chapter 3. Although the chapter concentrates on the concepts, good knowledge of C, XML and XSLT is helpful for understanding some design decisions and implementation details.

The text concludes with a summary and an evaluation of the results (see chapter 6). It also includes a set of ideas for future improvement and new projects. This chapter serves as a closing bracket to the initial aims.
Chapter 2

Fundamentals

This chapter forms a theoretical foundation to build upon, by giving definitions for important terminology and describing their relevance to and interpretation in this work. Clarifying terms is especially important, as most of them have a specialized origin and thus become ambiguous when used in a new wider context.

Many definitions have been collected from current scientific publications. They are complemented by own definitions and the description of their relations. In section 2.2 this leads to the introduction of the term adaptation infrastructure and the figure 2.3 which graphically presents problem space.

This work is titled "Dynamically generated multi-modal application interfaces". The title provides a good starting point for picking the important terms. First there is the term interface. An interface is something to interact with and therefore a model for interaction is described first. This section will briefly overview problems with interaction and conclude that it is necessary for interaction partners to take care about each other, to work together, to cooperate. This motivates that interfaces should be dynamically generated and not statically built. Generating something in a fashion that is controlled by parameters is called adaptation and this will be defined more precisely along with the related terminology next.

This part is then followed by a section about the objective of this work - interfaces. It contains a classification of interface toolkits into basic and abstract toolkits. The latter group is further subdivided into abstraction classes.

That is followed by sections that classify applications and application interface.

The chapter is ended by introducing established interface models and standards related to interface technology and evaluating their relevance to the aims of this work.

2.1 Interaction model

This work deals with interfaces. Interfaces exist for the purpose of allowing interaction (see chapter 2.3).
Definition 2.1 (Interaction) Interaction is the process of two or more systems exchanging data to perform a task.

In this work the terms interaction and communication are used interchangeably. Participants of an interaction are called interaction partners or interactors. An interactor is a system with a variety of input and output channels. Each of these channels can submit data or stimuli of a certain type (like e.g. sounds, visuals or touch). These submissions are subject to interpretation by the receiving system. The sum of the bandwidth of the interaction channels define the potential interaction capabilities of the system. These potential capabilities can only be used in the optimal case, where no obstacles hinder the interaction.

![Interaction Diagram]

Figure 2.1: Optimal interaction scenario

Figure 2.1 already shows that it is possible that interaction capabilities do not match. In this case interaction can be difficult or even impossible.

Now the real world is even more complex than what is shown in figure 2.1. The potential interaction capabilities are a relative static quantity. They can change, but changes are not expected to happen instantly or often. Real communication systems can only rely on the effective interaction capabilities of the interaction partner. The environment of the interaction puts a dynamic load on the potential capabilities. This effect can be seen in figure 2.2. The remaining capabilities are the effective capabilities. It is obvious that this further lowers the chance that interaction capabilities can be successfully matched.

This leads to the realization that it is required for interaction partners to adapt to each other. Therefore they need to negotiate their capabilities to agree to a compromise
which allows them to interact. (Stary, 1996; Dix et al., 1997)

Until now interactors were treated as systems in a generalized view. In the scope of this work there are two kinds of interactors - humans and computers. As this work is a computer science project, only the following combinations are considered further:

- **HCI (Human Computer Interaction)** (human-computer interaction), **CHI (Computer Human Interaction)** (computer-human interaction). A sound definition for HCI is given by the (various contributors, 2004c):

  **Definition 2.2 (Human Computer Interaction)** The study of interaction between people (users) and computers. It is an interdisciplinary field, relating computer science, psychology, cognitive science, human factors (ergonomics), design, sociology, library and information science, artificial intelligence, and other fields. Interaction between users and computers occurs at the user interface (or simply interface), which includes both hardware (i.e. input and output devices) and software (e.g. determining which, and how, information is presented to the user on a screen).

- **CCI (Computer Computer Interaction)** (computer-computer interaction)

If one interaction partner is a computer, special restrictions are to be considered. As computers are working with digital data, the human-computer interaction is constrained beyond the cognition of the human by physical laws that apply to digital communication. Important are e.g. the sampling theorem (quantization in time domain)
and the effects digitization (quantization in the value domain). These laws limit the bandwidth of the communication channels.

### 2.2 Adaptation

Adaptation is a key concept in this work, but also in the real world. Therefore a definition for the context of this work is required.

**Definition 2.3 (Adaptation)** *Adaptation is the process of changing an object so that it complies to the given requirements. This can involve adjusting or regenerating the object or parts of it.*

One conclusion from this definition is that the object needs to be adaptable at all. It needs to offer different modes of operation that can be matched with the requirements. One can further differentiate two degrees of adaptation:

- **passive adaptation** or **adaptable system** = the object will be manually adapted by an external entity
- **active adaptation** or **adaptive system** = the object adapts itself automatically

(Fink et al., 1996)

An adaptable system is relatively static, while an adaptive system shows a highly dynamic behavior.

Adapting an object requires knowledge about what changes are necessary for a desired effect. The overall knowledge regarding adaptation can be broken down on the basis of single aspects.

**Definition 2.4 (Adaptation method)** *An adaptation regarding one single aspect of the adaptation object is the application of an adaptation method. The method describes which changes are necessary, depending on all requirements involved.*

Examples for adaptation methods are:

- **Partitioning** or **segmentation**: A method to break down big sets of data, an example can be found in the paper "Adaptation for device independent authoring" (Menkhaus and Fischmeister, 2004).

- **Color schemes** and **enlarging**: Methods to enhance the readability for visually impaired.

- **Zooming navigation**: An approach for navigation in complex hierarchical structures. (Raskin, 2001)[pages 180-197]

- **Verbosity**: Increased verbosity can help first time users.
CHAPTER 2. FUNDAMENTALS

This work talks about interfaces and therefore interfaces are the objects to be adapted. The adaptation process is controlled by parameters, the adaptation requirements. In the case of this work these requirements are adaptation profiles and consist of environment profiles and user profiles. In other words, the application adapts to the requirements of the user and the environment.

**Definition 2.5 (User profile)** The user profile consist of the following parts:

- The *interaction capabilities* of the user as a communication partner. Interaction capabilities are a compound of the sensorical and motorical capabilities.
- The *interests and preferences* of the user relating to the style of the interaction.
- The *knowledge and competence* of the user regarding the task.

Defining an individual user profile is called *user modelling* (Fink et al., 1997). This is a complete sub-problem in the field of HCI. An example of why this is so difficult is *hedonistic quality*. If a given interface does not meet the user’s taste, the user will be less efficient. Unfortunately this is hard to measure and to make it worse, for users it is difficult to express precisely what they like and do not like.

**Definition 2.6 (Environment profile)** The environment profile describes a filter that applies to the capabilities part of the user profile. It temporarily restricts or even blocks certain interaction capabilities of the user.

The w3c is working on a standard definition of such profiles (Klyne et al., 2004). Such a profile can be used to decouple profile providers (sensors) and profile consumers (adaptation methods). (Stary, 1996; Dix et al., 1997)

### 2.2.1 Adaptation infrastructure

To carry out complex adaptations (adaptations that combine several adaptation methods) an adaptation infrastructure is needed. This work suggests such a foundation layer as an interaction model for an adaptive system architecture. Figure 2.3 graphically shows the relation of the previously defined terms for the scenario of human-computer interaction. Finally a short summary for this interaction model can be given:

- The users’ potential capabilities plus the current environment form the adaptation profile.
- The adaptation profile controls the adaptation process.
- The application provides application logic (functionality) and an interface to access it.
The adaptation infrastructure provides means to read the adaptation profile and to choose and execute the respective adaptation methods.

The adaptation method performs the adaptation of one aspect of the application interface according to the requirements given by the adaptation profile.

2.2.2 Adaptation of application interfaces

Several aspects of an application interface may need to be adapted for the end user. This work defines a classification into three groups, where each group contains the adaptation subproblems.

- **Technical**: Presentation domain, number of visible items, markup attributes
- **Cultural**: Language, currency, number format, text orientation, … (see sections 2.2.5, 2.2.4)
- **Personal/individual**: Colors, font-faces and -sizes, verbosity

Thus we face a multi-dimensional adaptation problem. Furthermore the classification shows that these categories build on each other. Technical aspects need to be solved world-wide, cultural for each culture and personal for each individual. Another similar categorization scheme has been suggested by Christian Sturm (see (Sturm, 2002)) under the name TLCC (technology, language, culture, cognition). His grouping clearly shows the dependency of the later groups onto the former ones. As an example, one
does not need to think about language adaptation before the application or the product is able to use different character sets or Unicode.

Some of the subproblems listed in the categorization above (like adaptation of complexity) are still without clear and practicable solutions or ideas of how to approach them.

2.2.3 Platform independence

When talking about platform independence, we have to consider several levels of independence.

- **Programming language**: This can be achieved by supplying language wrappers (language bindings). These make an API (Application Programming Interface) available to different programming languages.

- **Interface toolkit**: This requires the use of an interface abstraction layer, like an UIMS (User Interface Management System) (see section 2.2.7) can provide.

- **Operating system**: Most classic UI systems try to keep dependencies to the underlying OS-platform small by using a set of base-libraries (e.g. GDK for GTK+). This way it is only required to port these base-libraries to make the whole system available to the new platform.

2.2.4 Internationalization

**Definition 2.7 (Internationalization)** Internationalization (sometimes shortened to “i18n”, meaning “i - eighteen letters - n”) is the process of planning and implementing products and services so that they can easily be adapted to specific local languages and cultures, a process called localization. The internationalization process is sometimes called translation or localization enablement. Enablement can include:

- Allowing space in user interfaces (for example: hardware labels, help pages and online menus) for translation into languages that require more characters

- Developing with products (such as Web editors or authoring tools) that can support international character sets (Unicode)

- Creating print or Web site graphic images so that their text labels can be translated inexpensively

- Using written examples that have global meaning

- For software, ensuring data space so that messages can be translated from languages with single-byte character codes (such as English) into languages requiring multiple-byte character codes (such as Japanese Kanji)
With other words, during the process of internationalization all locale dependent parts of an application are extracted and a generalized mechanism is built in, so that it can be localized (see section 2.2.5).

2.2.5 Localization

Definition 2.8 (Localization) Localization (sometimes shortened to "l10n", meaning "l - ten letters - n") is the process of adapting a product or service to a particular language, culture, and desired local look and feel. … This enabling process is termed internationalization. An internationalized product or service is therefore easier to localize. The process of first enabling a product to be localized and then localizing it for different national audiences is sometimes known as globalization.

In localizing a product, in addition to idiomatic language translation, such details as time zones, money, national holidays, local color sensitivities, product or service names, gender roles, and geographic examples must all be considered. A successfully localized service or product is one that appears to have been developed within the local culture. (TechTarget, 2004)

To localize a product the target locale (also region or language code) must be known. Therefore computer systems (need to) provide settings for the user to specify the preferences and an API for applications to query the active settings. Examples are the ISO C/POSIX.2 locale (see (The Open Group, 2004)) and the technologies of the Open Internationalization Initiative (see (Davis, 2003)).

2.2.6 Culture

Definition 2.9 (Culture) Culture refers to norms of behavior and shared values among a group of people. (Kotter, 1996)[page 148]

Definition 2.10 (Culture) The system of shared beliefs, values, customs, behaviors, and artifacts that the members of society use to cope with their world and with one another, and that are transmitted from generation to generation through learning. (anonymous)

The wikipedia article about culture refers to a book that collects about 200 definitions for culture.(various contributors, 2004c)

A keyword often used in this context is cross cultural adaptation. Considering the problem of giving a clear definition for what a culture is, adapting to cultural differences can be quite difficult. Furthermore cultures come in a wide variety. According to an article by Matt Rosenberg we have 193 countries on earth. It is not so easy to count cultures as this depends on which definition is used as a basis for counting. Still this number gives a good approximation (Rosenberg, 2004).
Cultural adaptation is not just a complex issue, especially when it comes to aspects like religion; adaptation needs to be handled in a sensitive way. In the case of religion, opposing points of view often exist. If several cultures have access to the product, care must be taken not to impose one point of view to all the cultures.

2.2.7 UIMS - User interface management system

**Definition 2.11 (UIMS)** UIMS represent a set of software tools for the construction and control of the interaction dialog between the user and the computational resource. (see (Thomas, 1985))

This definition neglects the adaptive capabilities a UIMS can have. Thus the following definition is suggested instead.

**Definition 2.12 (UIMS)** User interface management systems are development tools to generate and run instances of user interfaces from generic specifications adapted to the requirements of the communication partners.

Thus UIMS provide a much higher abstraction level than a user interface toolkit. In this work they will be referenced as interface management systems as they deal with interfaces in general. According to Tanner and Buxton UIMS are also known as dialog management systems and abstract interaction handlers. The same source gives a useful list of goals that UIMS implementations strive to reach:

- Isolating the dialog portion of an application from the functionality should be possible.
- The ideal tool should render all dialog styles equally accessible.
- The UI design must inevitably intertwine with its implementation, testing and evaluation.
- The application programmer is not necessarily the best person to design the UI.
- The tools will render complex interfaces more maintainable.
- The tools facilitate portability of software, programmers and end users.

(Tanner and Buxton, 1985)

It is worth noticing that it is hard to draw a line between interface toolkits and interface management systems.

Thus UIMS can be classified according to the same classification scheme that has been suggested in section 2.2.3 for platform independence.
2.3 Interface

Three definitions for the term interface are given at whatis.com (TechTarget, 2004):

Definition 2.13 (Interface) A user interface, consisting of the set of dials, knobs, operating system commands, graphical display formats, and other devices provided by a computer or a program to allow the user to communicate and use the computer or program.

A GUI (Graphical User Interface) provides its user a more or less ”picture-oriented” way to interact with technology. A GUI is usually a more satisfying or user-friendly interface to a computer system.

Definition 2.14 (Interface) A programming interface, consisting of the set of statements, functions, options, and other ways of expressing program instructions and data provided by a program or language for a programmer to use.

Definition 2.15 (Interface) The physical and logical arrangement supporting the attachment of any device to a connector or to another device.

These definitions are too specialized for the context of this work. The definition below better suits the objective of generalized ”all purpose” application interfaces.

Definition 2.16 (Interface) An interface provides well defined access to functionality of an object from outside. It appears as a layer between two parties and aids their interaction.

As long as the interface remains consistent, a client can use the object via the interface without knowing about the object’s implementation. Thus an interface could be seen as a contract between service provider and client. Later in section 2.3.5 this role is shown graphically in figure 2.5.

Although there are hardware and software interfaces, this work will only focus on the latter.

2.3.1 Aspects of an interface

The PhD thesis ”UIML: A Device-Independent User Interface Markup Language” (Phanouriou, 2000b) introduces the separation of an interface into four aspects:

- **Structure**: The organization of interface objects.
- **Content**: Resources used in the interface such as label texts and icons.
- **Style**: The presentation of interface objects.
- **Behavior**: Defines the action to be performed on interaction with the interface objects.
The application needs to provide structure, behavior and content, but the choice of the content is dependent on the modality of the interface and the user profile. So the right subset of the provided content will be chosen by the adaptation infrastructure. Finally, style is an aspect that should be provided and handled by the adaptation infrastructure. An application should not enforce an own style to the user.

This scheme helps to organize an interface description in a modular way. This allows interface toolkits to treat the aspects separately.

### 2.3.2 Application interface

**Definition 2.17 (Application interface)** *An application interface provides access to the application’s functionality, by exposing methods to access the application’s data and states, as well as to initiate actions (computations, transactions).*

This work distinguishes between two kinds of interfaces to applications. An interface for human-computer interaction, which offers the application functionality to the user, is called *user interface* (UI). In this work, these will be called *Interfaces for human computer interaction* and described in section 2.6. The other kind of interface is about interaction between two computers and is called *Interfaces for computer computer interaction*. These interfaces are discussed in section 2.7.

### 2.3.3 Interface object

*Interface objects* are components which allow the user to interact with an application. This includes the following actions:

- Receive data (listen, read, view, . . . )
- Enter, delete or edit data, set options
- Invoke actions (e.g. to start processing)

Interface objects are presented by a metaphor of the presentation domain.

Nearly each system platform seems to prefer an own name for user interface objects. The most common names are *widget*, *gadget* and *control*. The latter is ambiguous as e.g. the user cannot perform control operations on a label, still the label is part of the interface.

From now on the term *interface object* will be used to refer to such elements.

In section 5.5.4 a new canonical naming scheme for interface objects will be described. This makes toolkits together with their interface objects comparable, by providing a universal mapping.
2.3.4 Interface toolkit

An interface toolkit is an application library containing a set of interface objects, as well as functions to create, destroy, organize and interact with these objects. There are two primary kinds of toolkits. The ones which support one interface toolkit and one-to-many languages, will be called basic toolkits (see section 2.6) and the others which form an abstraction over several interface toolkits will be called abstract toolkits (see section 4.1).

Toolkits which are able to run an interface in various modalities are additionally called multi-modal interface toolkits.

2.3.5 Interface design

The task of designing an interface is best separated into two subtasks.

- **Functional design**: This is part of the core software development or coding. The task the software developer has to perform is to decide which parts of the application needs to be exposed to the outer world.

- **Presentational design**: This part deals with ergonomics and artwork.

Each task should be carried out by persons or teams, who are specialized in the respective fields.

Figure 2.4 shows how interfaces are handled these days. The interface development is not separated from the application development. Primarily this tight coupling makes development in a team difficult. Everything is handled in one component. Work on this component is delegated to several developers, who have to be careful not to touch foreign parts of the code.

A much better alternative would be dividing the design process into several tasks as shown in figure 2.5. By detaching the interface from the application we have a clear differentiation between the tasks for application developers and those for interface designers/artists. The decoupling allows both teams to work in their domain and provides a means for synchronization.

Interface design is still a crucial and much discussed topic, as there are still many open problems. With the advent of the computer-technology age, interface design had to face another challenge - to create rich but easy-to-use interfaces. This has an obvious reason - computers, like Swiss army knives, can perform a variety of tasks. Current computer applications are tools which provide multiple functions behind one interface. This way the tools can be used for several purposes. Obviously, this leads to interfaces that are more complex, as the user needs to choose what he wants to do. The difficulty is to build an interface:

1. Where the user can quickly specify for which purpose he wants to use a tool.

2. Hide unnecessary functions and options when the user applies the tool.
2.3.6 **Interface renderer**

An *interface renderer* is a module whose purpose is to dynamically create an interface for use with specific communication means, such as combination of input-technology (keyboard, mouse, etc.) and output-devices (monitors, speakers, etc.) or communication protocols. An interface renderer is usually optimized for a subset of communication means and it targets either human or machine interaction partners.

During this work the shorter term *renderer* will be used to refer to *interface rendering components*.

2.3.7 **Layout renderer**

In the past user interfaces have very often been layouted manually. This means that the position and size of interface objects were precalculated during the dialog design phase. Especially Microsoft pushed this approach with their visual IDEs where the developer places these objects onto a form at fixed positions and with fixed sizes. Such applications are very inflexible and need a lot of custom code to become adaptable. Even changing the font in use (e.g. making it bigger, to make the form easier to read for visually impaired people) or changing the language, can ruin the whole layout.

Fortunately there is an easy cure for the problem, called *layout renderer*. Instead of
using fixed positions and sizes, the developer uses layout-containers such as horizontal boxes, vertical box and grids and packs interface objects into them. Due to the fact that these containers are interface objects themselves, one can build arbitrary hierarchies of user interface objects then. The layout-renderer usually has properties which affect spacing, packing order, sizing and more. The interface objects have requirements like the minimum display size. During run time the layout renderer generates a GUI from the object tree, the properties and the constraints. The process of creating the layout is often referred to as packing, as the interface objects are packed into the available space. As positive side effects, application which are using this technique for their UIs automatically get a more consistent Look & Feel, are easier to localize (as a longer label automatically gets the required space) and can easier be shown using a different style.

All modern UI-toolkits (such as GTK+, QT and Swing) support the use of layout-renderers. An exception is the Windows world, where one still finds many dialogs that are not resizable. The missing resizeability is a good indicator for fixed layouts.

2.3.8 Dialog

At wissen.de (www.wissen.de GmbH, 2004) a general definition followed by a more specialized one is given:
Definition 2.18 (Dialog) A dialog is the conversation between two or several persons in contrast to the monologue; important form of expression used in the dramatic art. In the philosophy of the antique Sokratikes, the dialog served in the form of speech and reply for the disquisition of problems.

Definition 2.19 (Dialog) A dialog is the exchange of question and answer between the participant and the computer, led across input/output equipment.

To summarize these statements, one can say that:

Definition 2.20 (Dialog) A dialog is the interaction of at least two interaction partners through interfaces.

Such an interaction needs a medial representation. A dialog in graphical environments is usually called dialog box or dialog window. Both terms are strongly influenced in their naming by their visual appearance. Wissen.de (www.wissen.de GmbH, 2004) gives the following definition for a dialog window:

Definition 2.21 (Dialog window) A dialog window is an aid to communicate with the computer. It contains elements such as input fields, list fields, selection and control fields, with which the user makes selections or confirms messages of the computer.

In this work both the process and the representation will be called by the term dialog. A dialog may be decomposed into sub-dialogs to better handle complex sequences, aid the user in understanding the application and allow the developer to create reusable components.

2.3.9 Dialog decoration

Before one can work with a dialog, the software has to render a dialog representation. This task is usually constrained by properties of the output device (like screen size) and capabilities of the interaction partner (such as the short term memory of a user). Thus it can happen that only a part of a dialog is visible at once. In such a case the software needs to add interface objects to allow the interaction partner to reach the other parts (a scroll-bar or previous/next page buttons).

This work defines those interface objects as dialog decoration, in the same way such interface objects for window objects are called window decoration.

2.4 Types of applications

One can classify software into several categories. One classification for interactive applications which has been widely accepted divides into three categories:

- Dialog based applications
CHAPTER 2. FUNDAMENTALS

- SDI - Single document interface based applications
- MDI - Multiple document interface based applications

This leaves out a large number of programs, namely non-interactive ones (like compile jobs or daily backups). However these are irrelevant regarding this work, as they are just started and then run in the background.

Dialog based applications

This category covers settings dialogs and step-by-step guides (wizards, druids, ...). These applications are task driven. This means invoking a dialog implicitly chooses quite accurately what the user wants to do. In contrast the next two types of applications often utilize dialogs to perform specific tasks, such as setting up a printer or generating a new document from a template.

Purely dialog based applications have no pull-down-menus and no such concepts as a document to work on. The objective of such applications is to setup parameters for running a process. The task is the only context the user operates on.

SDI - Single document interface based applications

Definition 2.22 (SDI) A Single Document Interface is a way to organize graphical applications into individual windows that are handled separately by the operating system's window manager. That usually means that each window is displayed as an individual entry in the operating system's task bar or task manager, and that the window does not have a "background" or "parent" window that contains its menu or toolbar, but that each application has its own menu/toolbar. (various contributors, 2004c)

Applications that use an SDI are usually simple applications like a document viewer. Specific to this model is that with the launch of the application, the document is loaded and when exiting the application the document is closed. Therefore the application only manages one kind of menu - the menu that offers the function related to the loaded document.

MDI - Multiple document interface based applications

Definition 2.23 (MDI) Multiple Document Interface is a Microsoft Windows programming interface for creating an application that enables users to work with multiple documents at the same time. Each document is in a separate space with its own controls for scrolling. The user can see and work with different documents such as a spreadsheet, a text document, or a drawing space by simply moving the cursor from one space to another.

An MDI application is something like the Windows desktop interface since both include multiple viewing spaces. However, the MDI viewing spaces are confined to
the application’s window or client area. Within the client area, each document is displayed within a separate child window. MDI applications can be used for a variety of purposes - for example, working on one document while referring to another document, viewing different presentations of the same information, viewing multiple Web sites at the same time, and any task that requires multiple reference points and work areas at the same time. (TechTarget, 2004)

Here this term will be used in an operating system neutral fashion, as interfaces discussed here are absolutely not tied to the Microsoft Windows platform and not even to graphical presentation platforms at all.

Applications that use an MDI are more complex applications like a document editor, where one can work with several projects at once. In contrast to the SDI as described earlier in section 2.4, the application lifetime differs from the document lifetime when using the MDI. Therefore the application has to maintain a menu for the state of no loaded documents and needs to provide functionality for switching between loaded documents. (various contributors, 2004c)

2.5 Application context

An application context is the functionality which is directly accessible at a given moment. Which contexts are available depends on the state of the application (e.g. if a document has been loaded or not, if an object has been selected). Currently several techniques are established to make the functionality from those contexts available to the user (e.g. context menus, dynamic panels).

Global application context

The global context covers functions which are always accessible (right after starting the application). Available commands often do not directly contribute towards finishing the actual task. Examples are items such as exit, help and load. Items available in the global context are usually presented as a main-menu and/or a toolbar.

Local application context

The local context offers functionality which is related to an object (document, selection, ...). It is usually only available when a document has been loaded. Modern applications have context menus, which list commands that can be applied directly to a selected object.

2.6 Interfaces for human computer interaction

Definition 2.24 (Interfaces for human computer interaction) Interfaces for human computer interaction are those, where humans are the clients who interface the appli-
cation. It is desirable that such interfaces are adaptable, to suit the needs of individual clients.

An Internet search about keywords like UI and/or user interface toolkit reveals that a large number of solutions exist today. It is not the aim of this work to provide a complete list, neither is it to describe them in full detail. What follows is, based on the author’s opinion, an introduction to the major ones. It is interesting to note that all these approaches concentrate on providing graphical interfaces.

From the users point of view, most of them provide similar functionality and show the main difference in the way their interfaces look like.

Figures 2.6 and 2.7 give an overview about features and availability.

![Figure 2.6: Various toolkits with their supported language bindings](image_url)

**2.6.1 GTK+**

GTK+ (Gimp ToolKit) is a multi-platform free software GUI Toolkit, primarily designed for the X Window System. It has a C-based object-oriented architecture that allows for maximum flexibility (portability, language bindings, . . . ), and consists of the following component libraries:

- **GLib**: Provides many useful data types, macros, type conversions, string utilities and a lexical scanner.
- **GDK**: A wrapper for low-level windowing functions.
- **GTK**: An advanced widget set.

In GTK+ the look of the interface objects is factored out into a theme-engine. Style properties have been separated by API means from the other object properties. *(various contributors, 2001a)*
Figure 2.7: Various toolkits with their platform availability

Language Bindings

<table>
<thead>
<tr>
<th>Language</th>
<th>Binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>native</td>
</tr>
</tbody>
</table>

There are about 25 more bindings available (see [various contributors, 2001a][bindings.html]).

Availability

<table>
<thead>
<tr>
<th>Platform</th>
<th>Back-end</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeOS</td>
<td>GTK+ for BeOS (<a href="http://www.gtk.org/beos/">http://www.gtk.org/beos/</a>)</td>
</tr>
<tr>
<td>directfb</td>
<td>GTK+ on DirectFB (<a href="http://www.directfb.org/gtk.xml">http://www.directfb.org/gtk.xml</a>)</td>
</tr>
<tr>
<td>UNIX (X11)</td>
<td>native</td>
</tr>
</tbody>
</table>

Table 2.2: The GTK+ platform availability
CHAPTER 2. FUNDAMENTALS

2.6.2 Java AWT

The Java Abstract Windowing Toolkit (AWT) (Sun Microsystems, Inc., 1998) is part of the Java Foundation Classes (JFC) (see section 2.6.3) - the standard API for providing graphical user interfaces for Java programs.

Language Bindings
Native language (Java) bindings only.

Availability
Runs on all Java-supported platforms.

2.6.3 JFC - Java Foundation Classes

The Java Foundation Classes (Sun Microsystems, Inc., 2001) software extends the original Abstract Window Toolkit (AWT) by adding a comprehensive set of graphical user interface class libraries. Swing is the part of the Java Foundation Classes (JFC) software that implements a new set of GUI components with a pluggable look and feel. Swing is implemented entirely in the Java programming language and is based on the JDKTM 1.1 Lightweight UI Framework. The pluggable look and feel lets one design a single set of GUI components that can automatically have the look and feel of any OS platform (Microsoft Windows, Solaris, MacOS). Swing components include both 100% Pure JavaTM certified versions of the existing AWT component set (Button, Scrollbar, Label, etc.), plus a rich set of higher-level components (such as tree view, list box, and tabbed panes).

Language Bindings
Native language (Java) bindings only.

Availability
Runs on all Java-supported platforms.

2.6.4 MFC - Microsoft Foundation Classes

MFC are Microsoft’s Framework for Visual C++. It is a large and extensive C++ class hierarchy that makes Windows application development significantly easier. The classes and the associated Wizard help to quickly build a Model View Controller skeleton. (Microsoft Cooperation, 2001)
CHAPTER 2. FUNDAMENTALS

Language Bindings

<table>
<thead>
<tr>
<th>Language</th>
<th>Binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>native</td>
</tr>
<tr>
<td>Basic</td>
<td>VisualBasic</td>
</tr>
</tbody>
</table>

Table 2.3: The MFC language bindings

Availability

<table>
<thead>
<tr>
<th>Platform</th>
<th>Back-end</th>
</tr>
</thead>
<tbody>
<tr>
<td>MacOS</td>
<td>native</td>
</tr>
<tr>
<td>Win32</td>
<td>native</td>
</tr>
</tbody>
</table>

Table 2.4: The MFC platform availability

2.6.5 QT

QT is a cross-platform C++ GUI application framework. It is fully object-oriented, easily extensible, and allows true component programming. The name is spoken like the word "cute". (Trolltech AS, 2001)

Language Bindings

Native language bindings (C++) only.

Availability

<table>
<thead>
<tr>
<th>Platform</th>
<th>Back-end</th>
</tr>
</thead>
<tbody>
<tr>
<td>embedded MacOS</td>
<td>native</td>
</tr>
<tr>
<td>UNIX (X11)</td>
<td>native</td>
</tr>
<tr>
<td>Win32</td>
<td>native</td>
</tr>
</tbody>
</table>

Table 2.5: The QT platform availability
2.6.6 Summary

In the previous chapter a couple of interface toolkits for human computer interaction have been introduced. Common to them is that they all realize graphical user interfaces. Table 2.6 summarizes the results of the examination of them.

One can further notice that the set of interaction object the toolkits provide is mostly equal. Especially the basic (atomar) widgets like buttons and text-entries can be found in every toolkit. Differences start with the compound widgets. These are high level widgets (e.g. a file selection dialog) that combine several low level widgets.

### Table 2.6: Comparison of HCI interface toolkits

<table>
<thead>
<tr>
<th>Toolkit</th>
<th>Language bindings</th>
<th>Cross platform</th>
<th>Themeing</th>
<th>Accessibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gtk+</td>
<td>yes, many</td>
<td>yes, many</td>
<td>yes</td>
<td>yes, ATK</td>
</tr>
<tr>
<td>Java AWT</td>
<td>no</td>
<td>yes, many</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>JFC</td>
<td>yes, a few</td>
<td>yes, many</td>
<td>yes</td>
<td>yes, Active Accessibility</td>
</tr>
<tr>
<td>MFC</td>
<td>yes, a few</td>
<td>yes, many</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>QT</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes, QT 4</td>
</tr>
</tbody>
</table>

2.7 Interfaces for computer computer interaction

**Definition 2.25 (Interfaces for computer computer interaction)** Interfaces for computer computer interaction are those, where other application are the client which interface an application. Their application interface usually does not need to adapt to the capabilities of the client as the client itself will be tailored to meet the conditions of the interface usage.

Again there are a wide range of solutions of which the most are based on proprietary data communication protocols. Some of them are only used internally in a product or series, so that there is no full language documentation available in public. Some others are not capable to talk to multiple clients at once, so that their usage is restricted to control an application remotely (e.g. to automate processes). The underlying core technology is often some form of RPC (Remote Procedure Call), like e.g. CORBA (Common Request Broker Architecture) or COM (Component Object Modell).

Many UNIX applications have a built-in interpreter for the LISP language for such purpose (e.g. the emacs text editor) or nowadays many use a java-interpreter such as BeanShell (e.g. in the jedit text editor). Multi-client capable scripting languages can be used to tie applications together. One could invoke a script in one application and the script could talk to instances of other applications to let them perform parts of the job in cooperation. Fortunately a few common approaches do exist, which will be introduced briefly in the next sections.
2.7.1 Rexx

The Rexx scripting language provides statements for control flow (like e.g. conditions and loops). Functions are imported from libraries (like e.g. mathematical functions) or are dynamically provided by hosts. These hosts are running programs that register their libraries at run-time. As it is possible for multiple hosts to do so, Rexx can be used to combine the functionality of several applications (e.g. fetching addresses from an address book application and feeding them into a word processor to print letters with different recipients). (various contributors, 2004b)

Rexx is a standard operating system component in OS/2 and AmigaOS (ARexx).

2.7.2 WebServices

The WebServices technology introduced the term service oriented architecture. The technology basically consists of the following components and technologies:

- **WSDL**: The Web Service Description Language is an XML (eXtensible Markup Language) language that is used to specify what the service offers.

- **UDDI**: The Universal Description, Discovery, and Integration component is a service registry that stores WSDL entries and can be queried by clients.

- **SOAP**: The Simple Object Access Protocol is an XML protocol that is used for communication (e.g. publish, find, bind, and invoke operations) (Box et al., 2000).

(Gottschalk et al., 2002)

2.7.3 Wsh and VisualBasic

The WSH (Windows Scripting Host) (Microsoft Cooperation, 2003) provides a runtime environment for running scripts. It natively supports the languages VBScript (a dialect of VisualBasic) and JScript (a dialect of JavaScript), but can be extended to support other languages such as Rexx (see section 2.7.1).

The scripts can be used as a macro-language to automate steps in one application. Other applications can export functionality as COM-controls and these can be instantiated in a WSH (Windows Scripting Host) script. As an example, a script could import multiple COM-controls to link several applications into a unified workflow.

2.7.4 Summary

The approaches introduced in the previous chapters have totally different roots. Still they have one thing in common - the rationale behind them is that a component provides services, these services are somehow announced or published so that clients can use them.
One possible attribute to compare them is to look at the scope of their interaction capabilities, as shown in table 2.7.

<table>
<thead>
<tr>
<th>Toolkit</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>rexx, wsh</td>
<td>can link local applications</td>
</tr>
<tr>
<td>webservices</td>
<td>can link applications across networks</td>
</tr>
</tbody>
</table>

Table 2.7: Comparison of CCI interface toolkits

Most approach are relatively platform independent, as they have low requirements in terms of hard- and software. Technologies like wsh are still only available on the windows platform.

### 2.8 Interface presentation

Regarding to spatialness, two types of interfaces exist - serial and parallel interfaces.

- **Serial**: Menu driven, only one context at one time (one chooses a function from a list, pass parameters and execute).
- **Parallel**: Several contexts are presented at once (main menu, tool bars and context menus)

Choosing the right type for a given task is not obvious. For parallel interfaces speaks the availability of many commands and quick accessibility of them. The downside of this is a possible information overload and the tendency to shallow structured command-sets.

Interfaces using the other presentation style (serial interfaces), usually have deep structured commands-sets. This will lead to clean interfaces, but can often cause orientation problems.

### 2.9 Interface models

There is a large number of models which decompose user interface management systems into components from different points of view (technical, user-centered). Each specializes on certain aspects of the matter and uses different separation criteria.

As an insightful comparison can be found in the work of Constantinos Phanouriou, the following sections briefly introduce the models (Phanouriou, 2000b).

#### 2.9.1 Seeheim model

**Overview**

The Seeheim Conference on User Interface Technology (Seeheim, Germany. Nov 1-3, 1983.) proposed separating an application with a graphical user interface into
three distinct layers: the Presentation Layer, the Dialog Layer, and the Application Layer, as shown in the figure 2.8. The Seeheim Model has been established by X/Open Technology as a framework for a User Interface Management System (UIMS).

Description

The three layers of the model are:

- **Presentation Layer (lexical aspects):** The static, visible part of the interface built upon the X Window System and X toolkits, such as Xt Intrinsics and OSF/Motif. Of course other Toolkits can be used together with this model too. All input is entered through interface objects offered by this layer and all output is performed with the aid of the presentation layer.

- **Dialog Layer (syntactical aspects):** The dynamic portion that handles events (callbacks) and interfaces between the static screens and the application.

- **Application Layer (layer of adaptation):** The underlying application "engine" that the GUI controls or communicates with. The application can be written in
programming languages such as C, C++, or Ada or can even be an SQL-driven database.

Furthermore the model distinguishes between several modes of interaction:

- **User initiated interaction**: User invokes functions of the application through the user interface
- **System initiated interaction**: The application invokes functions in the user interface
- **Mixed interaction**: Both modes are being supported by the application

There are a few advantages arising from the separation. Designing, implementing, testing, and maintaining each of these layers individually:

- Makes the layers easier to debug and maintain
- Improves re-usability and portability
- Makes it easier to distribute tasks among interface designers and application developers
- Supports rapid prototyping, so an interface designer can see changes in the interface without having to rebuild the entire application
- Increases the interface’s consistency and adherence to company standards

(Pfaff, 1985)

**Evaluation**

Even though this model is well suited for describing architectures, it lacks a few components which are important for practical use. According to the studies of Flatscher there are no guide-lines on how to structure software packages and no concepts of modularization or object-orientation. Generally seen, a relationship with an existing software design method is not given at all. (Flatscher, 2002)

### 2.9.2 Arch model

**Overview**

The Arch model is originally based on the Seeheim model (see section 2.9.1). It separates components finer than Seeheim by adding two more of them. Additionally the abstraction level has been raised by decoupling application, dialog-system and toolkit. (various contributors, 1992)

The components and their relation are shown in figure 2.9.
Description

The five components of the model are:

- **Domain Specific Component**: The application logic.
- **Domain Adaptor Component**: The abstract interface specification.
- **Dialog Component**: The dialog run-time.
- **Presentation Component**: Target toolkit specific presentation of the interface.
- **Interaction Toolkit Component**: The target interface toolkit.

Evaluation

This model provides the essential components needed by a UIMS. Still there are no explicit guidelines about what functions belongs to which components and which information should be managed by which component. Therefore the model provides only a coarse foundation for designing highly dynamic systems, where e.g. interface toolkit components are exchanged at run-time or multimodal interaction is supported.

### 2.9.3 MVC model

**Overview**

In object-oriented development the term *model-view-controller* (MVC) refers to a methodology or design pattern for successfully and efficiently relating the user in-
interface to its underlying data models. The MVC pattern was developed using the program-
ning language Smalltalk and is now widely used in program development with
programming languages such as Java, Smalltalk, C, and C++.

The MVC model shown in figure 2.10 has been heralded by many developers as a
useful pattern for the reuse of object code and a pattern that allows them to significantly
reduce the time it takes to develop applications with user interfaces.

![MVC Model Diagram](image)

Figure 2.10: The MVC model

**Description**

The model-view-controller pattern proposes three main components or objects to be
used in software development, which are decoupled by a subscribe/notify protocol:

- **Model**: Represents the underlying, logical structure of data in a software appli-
cation and the high-level class associated with it. This object model does not
contain any information about the user interface.

- **View**: Is a collection of classes representing the elements in the user interface
(all of the things the user can see and respond to on the screen, such as buttons,
display boxes, and so forth).
CHAPTER 2. FUNDAMENTALS

• **Controller**: Represents the classes connecting the model and the view. Is used to communicate between classes in the model and view.

(TechTarget, 2004)

The main relationships in MVC are given by the Observer, Composite and Strategy design patterns (see (Gamma et al., 1995) for information about design patterns). Complex interfaces can be built by nesting views using the Composite pattern.

**Evaluation**

One advantage of the MVC model is the clear separation of data-structures, presentation and interaction. The MVC pattern is very flexible as it easily can be extended to support multiple (different) views and even nested views.

One disadvantage is the need for intelligent communication between the components. If not enough care has been taken during message design, unnecessary updates will occur, resulting in useless event flooding. (Flatscher, 2002)

### 2.9.4 PAC model

**Overview**

Another model is the presentation-abstraction-control (PAC). It too separates the architecture into three layers as shown in figure 2.11. A tuple consisting of one object from each class is often called *agent*.

**Description**

The three layers which one agent consist of are:

- **Presentation**: Handles in-/output, the visible components are typically changeable.
- **Abstraction**: Handles the semantic of the underlying application.
- **Control**: Handles cooperation and dialog of presentation and abstraction.

In an application one would build a hierarchy of agents, by connecting the control objects.

**Evaluation**

Although it looks relative similar to the MVC model (see section 2.9.3), the PAC model bears a few differences. At first it is noticeable that the latter does not differentiate input and output. Additionally the PAC model specifies a component to handle the consistency between abstraction and presentation. (Flatscher, 2002)
2.9.5 OAI model

Overview

The OAI model correlates real-world objects and actions with interface objects and actions. It is well suited for applications developed in an object oriented programming language, as it uses the object orientation paradigm itself.

In contrast to commandline applications where the user would specify commands and then objects to apply the command to, modern graphical interfaces use direct manipulation. Here the user selects objects and then the action to be applied to them. The OAI model can be used at design time to capture an interface from the object oriented point of view. (Shneiderman, 1998)

Description

The first step in building an OAI model is to gather the tasks the system should perform and the objects involved. These tasks are further refined into single steps, where each step is an action applied to an object. The object and action hierarchies in the task
domain are then mapped to the interface domain.

**Evaluation**

The model is well suited to capture object oriented aspect from the task domain. It has a strong relation to the direct manipulation interaction style. The OAI model cannot describe temporal relations. Thus batch processing or a sequence of processing steps (pipelining) cannot be modelled. *(Khella, 2002)*

### 2.9.6 MMI framework

**Overview**

The W3C Multimodal Interaction Framework (MMI model) identifies the major components for multimodal systems, where each component represents a set of related functions. The framework identifies the markup languages used to describe information required by components and for data flowing among components. The W3C Multimodal Interaction Framework describes input and output modes widely used today and can be extended to include additional modes of user input and output as they become available.

**Description**

Figure 2.12 shows the components of the MMI model. Central component is the *interaction manager*. It coordinates input and output between user and application. A multimodal implementation of this model would be able to handle different input and output modalities. *(Larson et al., 2003)* The model also includes a component called *system and environment* that enables the model to react to environmental changes and adapt to hardware capabilities.

**Evaluation**

The MMI model specification contains several examples, which show how the model relates to real world scenarios. In contrast to Seeheim and ARCH, MMI specifies the function of each component in more detail.

An important detail of this model is that it includes the hardware and the environment. Unfortunately the semantics of the connection between the *interaction manager* and the components on the right side (like the hardware and the environment) are not given.
2.9.7 Cameleon framework

Overview

The Cameleon framework specifies multiple models involved in user interface design: concepts, tasks, platforms, environments, interactors and evolution. These models are related to four layers of interface abstraction: tasks and concepts, abstract user interface, concrete user interface and final user interface. Further the context of use is introduced as a dimension. (Calvary et al., 2001)

Description

Two relations are spanned across the four layers: reification and abstraction. The first is the process of generalizing details of the interface to produce an abstract interface description. The second relation is the opposite process, which adds details to an abstract interface description.

This work suggests the separation of the layers according to their temporal behavior as shown in figure 2.13.

The context of use axis has been introduced to allow for changes in the models for different purposes. On the most abstract level tasks and concepts the models will be mostly the same. The final interfaces for different contexts of use will likely show differences though.

Evaluation

The model provides a good separation of abstraction stages an interface description can be in. It is well suited to compare adaptive interface approaches. In chapter 4
existing approaches are related to this framework and in chapter 5 the same will be done for this work.

2.9.8 Summary

A general criticism on models such as Seeheim, Arch and PAC is that they aim to model adaptive systems, but do not describe the adaptation process as such. Especially there is more than just user input that controls the adaptation process as shown in section 2.2. Only the MMI model contains a component that represents adaptation profiles (system and environment). The adaptation infrastructure model as described in section 2.2.1 goes a step further. It models the roles of the environment and adaptation profiles as part of the interaction scenario.

Another criticism is the low level of modularisation. The ARCH model provides the highest degree with five components. This is sufficient for the top level, but each of these components provides several services. This is only vaguely covered, if at all. None of the models reveal which component should encapsulate the expert knowledge for the adaptation.

Furthermore most models do not specify which parts or components should be provided by the application and which parts belong to the adaptation infrastructure.

These criticisms are shared by other people. To give one example Bastide and Palanque stating that:

Arch performs no better than Seeheim in providing indication either in the precise content of the components or in their design process.

(Bastide and Palanque, 1999)
The MMI model is an exception, as it contains descriptions how this model maps to practical applications. Likewise in this work the adaptation infrastructure model introduced in section 2.2.1 will be connected with an architectural model in section 5.1.

2.10 Standards

Several standards about basic requirements for user interface technology have been developed. These documents aim to support the application developer with the interface design task. Therefore they provide collected domain-specific know-how.

Apart from the official standards there are many unofficial standards available from platform vendors like apple, microsoft, gnome and kde. These documents provide guidance for the developer to produce a product that is compliant with the look and feel of the platform. Therefore the guidelines often contain a collection of good and bad examples for solving interface design related tasks. A product that is compliant with one of these guidelines is easier to use for an experienced user on the platform the guidelines are related to. This is so because the user will feel familiar even with a new interface. Therefore a familiar and consistent design often performs better than different designs which probably are more logical. A good example is the standard icon for saving data on the windows platform - a floppy-disk. Nowadays disks are rarely used, so one could use an icon showing a hard-disk. This would be a bad idea! Users would be confused. They are used to look out for this floppy-disk icon when wanting to save data.

The next sub chapters will briefly introduce some of the major official standards.

DIN ISO/IEC 9126 Software Product Quality

The DIN ISO/IEC 9126 norm defines product quality characteristic along with a series of subcharacteristics:

- **Functionality**: accuracy, appropriateness, interoperability, security
- **Reliability**: maturity, error tolerance, recoverability
- **Usability**: comprehensibility, lernability, usability, attractivity
- **Efficiency**: runtime performance,
- **Changeability**: analysable, modifiability, stability, verifiability
- **Transferability**: adaptability, installable, conformity, interchangeability
DIN ISO/IEC 12119 Software Packages - Quality Requirements and Testing

The DIN ISO/IEC 12119 norm specifies basic requirements of dialogs in software packages and how to test them:

- **Documentation:**
- **Understandability:**
- **Functionality:**
- **Stability:**
- **Reliability:**

(TÜV Informationstechnik GmbH, 2002)

DIN EN ISO 9241

The DIN EN ISO 9241 norm specifies guidelines for the design of computer aided work.

- **Adequacy for the task:** A dialog is adequate for a task, if it supports the user to finish a job effectively and efficiently.

- **Self descriptivity:** A dialog is self descriptive, if each dialog step is thoroughly understandable by the given feedback or can be explained to the user upon request.

- **Controllability:** A dialog is controlable, if the user is able to start the dialog and influence direction and speed until the goal has been reached.

- **Conformity to expectations:** A dialog conforms to expectations, if it is consistent and complies with the properties of the user, like knowledge, education and experience, as well as generally accepted conventions.

- **Tolerance to errors:** A dialog is tolerant to errors, if the intended result can be achieved with no or minimal corrections by the user, despite of incorrect inputs.

- **Personalisation:** A dialog is personalisable, if the dialog system admits adjustments to the demands of the task, as well as to the individual capabilities and preferences of the user.

- **Subservity of learning:** A dialog subserves learning, if it supports and instructs the user when learning the dialog system.

(Wirth, 2002; TÜV Informationstechnik GmbH, 2002)
DIN EN ISO 13407
The DIN EN ISO 13407 norm describes user-oriented strategies of how to detect and avoid usability problems in the application development phase. (TÜV Informationstechnik GmbH, 2002)

EU-Directive 90/270/EWG
The EU-Directive 90/270/EWG contains regulations about security and health care of computer aided work. It does not contain detailed guidelines of how to fulfill the requirements.

SAA - Systems Application Architecture

Definition 2.26 (SAA) IBM’s family of standard interfaces which enable software to be written independently of hardware and operating system. (various contributors, 2002)

CUA - Common User Access

Definition 2.27 (CUA) Common User Access is the user interface standard of SAA (see earlier in section 2.10). The CUA standards deal with interface appearance, programming conventions, and communications.

CUA is a development of IBM, where design rules for user interfaces have been specified for the first time. The goal was to keep user interfaces uniform for all soft- and hardware-platforms of IBM. This includes descriptive names and their translations into all business languages of IBM.

- CUA 1987: Specifies the look and basic functionality (loading files, requesting help, etc.) of graphical interfaces (MS Windows 2.1, OS/2 1.1).
- CUA 1991: Specifies functionality and components of object oriented designed graphical user interfaces (OS/2 Workplace Shell, later MS Windows versions).

(Flatscher, 2002)

WCAG (Web Content Accessibility Guidelines)
The WAI (Web Accessibility Initiative), in coordination with organizations around the world, pursues accessibility of the Web through five primary areas of work: technology, guidelines, tools, education and outreach, and research and development. The WAI technical activity includes work on technology, guidelines, and tools to increase accessibility of the Web for people with disabilities:
• The Protocols and Formats Working Group reviews all W3C technologies for accessibility.


• The Evaluation and Repair Tools Working Group develops techniques and tools for evaluating accessibility of Web sites, and for retrofitting Web sites to become more accessible.

(Brewer, 2000)

2.10.1 Summary

A major problem of those introduced standards is that they are not very generalized. They mainly focus on visual interfaces, while not capturing common interface principles. Furthermore some standards, such as CUA/SSA do not provide much more than the platform design guidelines mentioned at the begin of this chapter. Only the WCAG (Web Content Accessibility Guidelines) is including adaptivity as a means to provide better accessibility.

On the other hand norms are made to regulate something that appeared on the surface and starts to evolve into several directions. As a consequence it is unlikely that there are standards and norms for new developments.
Chapter 3

Problem description and task details

The main problem has been briefly introduced in section 1.1 and was further focused on in section 2.2. One might wonder why this topic has been brought to attention relatively late. Figure 3.1 shows the situation in the past. The application was written exactly for one computer system and the user had been trained to operate this machine as well as the software. Additionally an application was usually only available in English. In the very beginning of computer technology, personal preferences of the user were ignored completely. So this was a 1:1 situation, where no adaption was performed on the computer side. It was not needed as there was just one type of user.

Contrary to this, the ideal situation would be, if the user could choose the best suitable interface to access the desired application. That choice would depend on his capabilities, preferences and the current environment.

Nowadays the situation has changed (see figure 3.2). The right applications are
available and devices suited for all sorts of environments exists. The user can be supported by a large set of assistive technology. Many applications can be adapted to different cultural needs and most applications can be customized to the users personal needs. So this should in theory cover even the increased range of potential users.

Unfortunately the applications do not adapt to the technology, culture and personal preferences of the user in general. The missing parts that are still needed can be seen in figure 3.3. It shows that a holistic adaptation layer is required and a profile that controls the adaptation is needed. The profile makes the requirements of the user accessible to the adaptation layer (the right side of figure 3.3). The adaptation layer in turn will provide the missing connections (the left side of figure 3.3), to equip all the applications with the wanted interface. The figure also shows that the adaptation problem has a multi-dimensional nature. One has to provide adaptation in multiple categories, namely technology, culture and the individual user.

It should be noted that the application of this problem is quite universal and not limited to people with disabilities, like one could believe initially. As Vanderheiden and Henry have found out in their article (see (Vanderheiden and Henry, 2001)) adaptation is not just a matter of making software available to people with disabilities. Everyone might need to work in a situation where some senses cannot be used for communication with electronic devices. A few examples for such situations are:

- Reading a display is not advisable when driving a car. Thus speech output would make the software accessible. The same technology would make software accessible to blind people.

- It can be impossible to use a mobile phone in speech mode in a loud environment such as the subway.
Finding a solution to this multi-dimensional adaptation problem is equal to make interfaces accessible to all, everywhere and at every time.

In section 2.2 the separation of adaptation into adaptation infrastructure and adaptation methods has been introduced. This work will concentrate on designing an adaptation infrastructure. Developing adaptation methods is related to user modelling and technical issues, such as adding sensors to devices that capture the environmental profile.

This chapter contains an introduction to conceptional and technical problems related to this adaptation problem. After a general description, adaptation problems related to each aspect of the interface (structure, style, behavior and content) are discussed. This structure has been chosen in the hope that it eases the understanding of the complex adaptation problem. A section about technical problems and development support ends this chapter.

### 3.1 Examples

To illustrate the problems, two example applications will be used through the following chapters.

- **A software installation and configuration tool**: This is a classic wizard-style application. The user will be guided through an installation or configuration process in a step-by-step fashion. At each step information will be queried and further steps are determined depending on the information entered. An example for this would be a "create UNIX user account" dialog.
• **An address manager**: This in contrast is a classic data processing application. The user has objects (contact-records) and functionality to modify them (add, delete, view, edit). Further the user has a global context (the list of objects) again with functionality (e.g. search, navigate or browse).

Both examples have in common that the service the applications provide can be useful in various scenarios.

• The computer system administrator might need to unlock a user account while not sitting in front of the server. It would be helpful, if the administration software could run in a terminal window.

• The manager needs to access a contact record while being on a business trip. He would like to use his mobile phone to call the server and ask the system for the contact details.

### 3.1.1 User account administration

Figure 3.4 shows a screenshot of how such an application might look like, when presented graphically (the screenshot uses gtk+). This example is useful for demonstrating:

- Handling of complete dialog navigation (multiple steps)
- Conditional interaction flow
  - The account type can be system-account or user-account, where a system-account has no login and therefore no password
  - The Okay action gets only enabled, after all required fields (such as account name and user id) have been entered
- various presentation styles like wizard (subsequent pages) or all-at-once

### 3.1.2 Address manager

Figure 3.5 shows a screenshot of how a simple address manager might look like, when presented graphically (the screenshot uses gtk+). This example is useful for demonstrating:

- Several parallel tasks (new, lookup/browse/search), edit, delete
- Context switching, sub-dialog-grouping

The structure of the example application can be seen in figure 3.6.
3.2 Multi-dimensional adaptation

Providing a high level of adaptivity brings up a series of challenges. Many of those questions never occur during traditional interface design as the situation is much simpler.

Modern software applications already know some form of adaptation. Some applications allow the exchange of the look and feel of its interface. Adaptation to different locales in the meaning of texts and number formats is also state of the art, but still not always implemented.

As outlined at the beginning of this text, this work pursues higher goals. The next chapters will provide a detailed analysis of the problems that need to be solved to reach those goals. This analysis starts with looking at the multi-dimensional nature of the adaptation problem and strategies to cope with it.
3.2.1 General adaptation

For solving the task of adaptation one faces the general problem of doing a series of transformations $T_{S \rightarrow D}$ with minimal loss of information and maximum possible enrichment at each step. One such step can be seen as a function $T$, which gets applied to a dialog description $s$ from the source set $S$ and delivers a resulting dialog description $d$ from the destination set $D$, while using profile data $p$ for the transformation. Thus one step becomes $d = T(s, p)$. Each transformation $T$ handles one single aspect of adaptation. Those transformations can easily be chained to constructs like:
\[ d = T1(T2(...(s,p),p),p) \]. Therefore the transformation chain can be built out of those transformation steps that are required to adapt for the current user in the current situation. For the sake of simplicity the profile data will be omitted from now on.

The previous expressions describe a multi-step transformation from one source to one target. The objective of this work is an adaptable system and therefore we have multiple targets. Simply extending the transformation chain leads to the situation shown in figure 3.7.

![Figure 3.7: Adaptation for multiple targets with early fan out](image)

Source is a generalized interface. This is immediately transformed by the first column of \( T(s) \) elements into a domain specific notation. The dotted paths above and below should remind that the number of possible target representations is usually much larger as in the figure. After the initial transformation all of the intermediate dialog descriptions are transformed a couple of times by more \( T(s) \) elements until all target representations \( Dest. X \) have been produced. Please note that in this example a transformation \( T(s) \) can be anything from manual to automatic transformation.

It is likely that many of the transformations will execute common changes for each of the domains. Due to the very early separation into separate transformation chains, such a system is hard to maintain and to extend (redundant logic in some transformations). Hence, a better structure would be one like shown in figure 3.8.

In this structure the point of separation has been postponed. Some transformations have been generalized so that they can be applied prior to the separation point. This massively reduces the number of transformations (for a higher number of target domains). Another side effect of this is that the number of intermediate interface representations is reduced.

A first set of tasks one has to solve, is:

- Extract common changes into separate transformation steps
Figure 3.8: Adaptation for multiple targets with postponed fan out

- Determine the order of transformation steps
- Specify the separation point

Apart from that, one has to consider that a transformation means exchanging or converting information. If the transformation is not a bijective transformation (like it is in this case), one always loses information, as there exists no unique reverse transformation.

Along with that comes another issue one needs to pay attention to: Not every kind of information can be equally powerful represented in all domains. Even though one could e.g. describe a picture with text, this text can only give an idea of how the image really looks like. On the other hand this does not automatically make it impossible to work with an image processing application via a text console or via audio-commands.

There are tasks like format conversion and auto-correction which can be performed without seeing the result, but this is not the general case. To summarize, designing mappings is a very difficult and sometimes close to impossible task.

Finally it is important that all the adaptability does not become a burden for the developer. This work strives for a solution where the intelligence of the system adds to the result. The idea here is that it should not be required for the software-engineer to provide adaptation profiles for every possible target system along with each application. It is obvious that such a requirement would result in a high number of required profiles with a high level of redundancy. This is undesirable.

The next chapters provide a deeper analysis of the adaptation problem, structured as technical, cultural and personal adaptation.
3.2.2 Technical adaptation

Technical adaptation is required to run an application on a specific platform. In our world we have a high variety of technical "standards". Often such standards have grown out of local efforts. That means that many standards only apply for certain regions, such as continents or countries. When a product should be available across those regions, it needs to be adapted to comply to the local standards or otherwise it would not work at all. A few examples of such standards are:

- **Power supply**: Shape of plug and difference in voltage/frequency
- **Mobile network**: Different frequency bands
- **Television**: Different screen refresh rates

This work is exploring a software solution. Therefore only adaptation to different devices (device capabilities) will be covered.

3.2.3 Cultural adaptation

When the technical problems are solved, so that an application can be used in a target region, it needs to be adapted to the people that live there. This involves two steps: Internationalization (i18n) and localization (l10n) (see sections 2.2.4 and 2.2.5), where the first step prepares an application so that the second step can be carried out. For text messages step 1 means to extract all text parts from the source code to a language catalog. In the source code now the message will be referenced by an abstract identifier from the catalog. Step 2 means translating such an catalog to a new language.

Of course cultural adaptation is more that adapting language. Section 2.2.6 shows the difficulties with precisely defining the term culture. Below is a collection of (relatively obvious) things that can vary among cultures and that is relevant to computer applications.

- **Linguistics**: Language, units such as currency, date and time formats
- **Graphics**: Indicator colors, icon-metaphors, text orientation, layout, maps
- **Usage**: Shortcut-keys
- **Cognition**: Deep menus vs. shallow menus

Many cultural differences cannot be treated as easy as language catalogs. It is not possible to just setup a color table with “preferred” and “forbidden” colors for all cultures. To address this problem scientists like Hofstede have started to explore the space of differences by examining groups from different cultures. As a result Hofstede suggested these five dimensions:
• **Power-distance (PDI):** Indicates how much a high level of inequality of power (and wealth) within the society is accepted by the less powerful.

• **Collectivism vs. individualism (IDV):** Measures how much individuals are integrated in the society (collective).

• **Femininity vs. masculinity (MAS):** A high masculinity ranking is an indication that the culture experiences a high degree of gender differentiation.

• **Uncertainty avoidance (UAI):** Measures the level of tolerance for uncertainty and ambiguity. A high ranking is typical for societies with many laws and rules.

• **Long- vs. short-term orientation LTO):** Focuses on the degree the society embraces, or does not embrace, long-term devotion to traditional, forward thinking values.

(Hofstede, 1997; Hofstede, 2003)

It is still difficult to derive concrete rules for adaptation from a categorization of a culture into these dimensions.

### 3.2.4 Personal adaptation

Every human is an individual. So it becomes necessary for an application to adapt to the specific requirements of the single user.

Personal adaptation is the most difficult form of adaptation. In the last decade progress has been made in defining cultural profiles. From this ideas which information need to be part of profiles for individual users are slowly emerging. Still unsolved are the problems of capturing and providing the profile data:

• **Capturing profile data:** People can say whether they like an application or not, but have problems giving precise reasons for it. Therefore it is close to impossible to set up rules to approach the problem of personal adaptation.

• **Providing profile data:** If the problem of capturing the data ever gets solved, where can the answers be stored? The required solution is some kind of storage that can be quickly read from a machine, when a user starts a session with that machine. Further a mechanism that keeps the storage up-to-date is required. As introduced in section 2.1, the interaction capabilities that form the user profile consist of relatively static parts as well as dynamic parts. An adaptive system requires dynamic profile data to react on. If only static profile data are available an adaptable system would be sufficient (the systems behavior can be manually configured to match the user requirements).

Table 3.1 gives a small collection of issues sorted whether they are well understood (and therefore likely to be handled) or not.
CHAPTER 3. PROBLEM DESCRIPTION AND TASK DETAILS

<table>
<thead>
<tr>
<th>Adaptation is easy</th>
<th>Adaptation is difficult</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface style (font, color)</td>
<td>Knowledge level</td>
</tr>
<tr>
<td>Verbosity (amount of supportive text)</td>
<td>Determination</td>
</tr>
<tr>
<td>Behavior</td>
<td>Mood</td>
</tr>
</tbody>
</table>

Table 3.1: Personal preferences and the difficulty of adaptation

3.3 Problems related to interface structure

In section 2.3.1 the structure of an interface has been defined as the *organization of interface objects*. The range of media that should be covered by a solution influences the requirements of the structural interface description needed. So the first choice to be made is to specify the range of media that should be covered. Then one can define an interface description that is neutral to the chosen range of media.

Covering a wide range of media is called a *generalized solution* in this work, where other approaches which focus on a small range or even a single media domain are called *specialized solutions*.

3.3.1 Generalization versus specialization

When one wants to get advantages in one field, one usually has restrictions somewhere else. This is the case for the design of an adaptive solution as well. Figure 3.9 shows the situation of having toolkits which are either powerful regarding the domain they specialize in or have strong cross-media capabilities. The figure gives a few examples for toolkits in each box. It also shows one of the fundamental differences of the GITK approach in comparision to the others.

The nature of the problem that is shown in figure 3.9 is quite obvious. If a toolkit uses all the expressive power one media domain can deliver, it makes itself very dependent on that domain. A toolkit that works in the graphic domain can offer the inclusion of visual objects such as graphs, pictures or nested tables. A toolkit that alternatively can use other media would need to find equivalent representations of such objects for the target media domain. Such replacements do not necessarily exist. Even worse is that for some objects not even weak replacements exist. Table 3.2 shows an overview of some media and their representations for different senses. Two statements that can be derived from the table 3.2:

- Text can be used quite universally.
- Images and sounds are mostly bound to their native domains.

For a given set of applications that an approach targets, one has to decide on the required abstraction level. If image processing application need to be doable with the approach, a very generalized solution would not fit. If the applications are from the field of database front ends, using a specialized interface toolkit would be an unnecessary restriction.
CHAPTER 3. PROBLEM DESCRIPTION AND TASK DETAILS

55

Figure 3.9: Generalization versus specialization

<table>
<thead>
<tr>
<th>Media</th>
<th>Vision</th>
<th>Hearing</th>
<th>Touch</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>Printed text, Gestures</td>
<td>Spoken text</td>
<td>Braille dots</td>
</tr>
<tr>
<td>images</td>
<td>Native</td>
<td>Spectral sounds</td>
<td>Surface profile</td>
</tr>
<tr>
<td>sounds</td>
<td>Spectrogram, Oscilloscope</td>
<td>Native</td>
<td>Deep frequency vibrations</td>
</tr>
</tbody>
</table>

Table 3.2: Relation of media and human senses

3.3.2 Logical interface description

Interface *look and feel* must be strictly separated from the description of the functionality covered. The architectural design for the UIMS needs to include the following two decisions:

- Which is the essential information that the developer has to provide when writing the application
- Which are the variable factors the user or the style designer might want to control

(Green, 1985)

It must be absolutely clear that the application developer does not carry out the job of the interface designer regarding the decision about the resulting *look and feel*. This needs to be someone else’s concern.
The wanted framework solution needs to be a compromise between being lightweight but capable. Lightweight, so that a clear advantage over the traditional approach is given. Capable, to carry all sorts of meta-data to describe user interface without losing too much of the expressive power the target domains can provide. The enormous difficulty comes from the fact that all the sets of information which form the interface description are very tight coupled to each other. As an example for this consider the layout. This term originates from the graphical domain. When designing an interface for the graphical domain, the layout is used for several purposes. Table 3.3 shows two uses of layout and the means for these applications in the graphical and the aural domain.

<table>
<thead>
<tr>
<th>Layout is used to</th>
<th>Graphic interface</th>
<th>Aural interface</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Structure the interface objects</strong></td>
<td>Grouping, alignment and spacing</td>
<td>Pausing, delimiting sound</td>
</tr>
<tr>
<td><strong>Decorate the interface objects</strong></td>
<td>Colors, font faces and -sizes</td>
<td>Voice, volume speed</td>
</tr>
</tbody>
</table>

Table 3.3: Relation of dialog specification to presentation domain and application

The first use of layout (to structure) needs to be abstracted, so that it can be used to define the general structure of the interface. On the contrary the second application (to decorate) must be unavailable in a specification for abstract interfaces.

A problem arising from the separation of concerns is the decision which software component should provide which data.

<table>
<thead>
<tr>
<th>Information</th>
<th>Application</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Independent</td>
<td>Independent</td>
</tr>
<tr>
<td>2: List of interface objects, grouping hierarchy</td>
<td>Dependent</td>
<td>Independent</td>
</tr>
<tr>
<td>3: Style attributes</td>
<td>Independent</td>
<td>Dependent</td>
</tr>
<tr>
<td>4: Layout (multi dimensional layouts)</td>
<td>Dependent</td>
<td>Dependent</td>
</tr>
</tbody>
</table>

Table 3.4: Relation of dialog specification to presentation domain and application

Table 3.4 demonstrates that the application needs to ship the data listed in row 2. Data from row 3 on the other hand needs to come from the UIMS. Whatever can be listed in row 1 is irrelevant to this problem. Now the problematic information is everything that falls into the category on row 4. Either each application ships data with adaptations for each presentation domain, which is not feasible, as it is not known to the application which domains exist. Alternatively each presentation component needs to come with adaptations for all applications, which is an even worse situation.

Therefore the goal regarding interface structure is to design a specification tool that can be used to describe interfaces on the chosen abstraction level, without dependencies on other interface aspects.
3.4 Problems related to interface style

Interface style is an important part of the problem. The style describes how the interface ‘looks like’ or appears to us. The first choice that needs to be made is selecting the best target domain for the interface. Next it can be decided how to present the interface in general and what interaction objects to choose. Offering multiple exchangeable presentation styles for an interface is called theming or skinning.

3.4.1 Domain mapping

Interaction with software works in two ways. The software presents the interface to the user, so that it can be perceived by the users (available) senses. Then the user reacts to the data presented. This reaction triggers the next iteration in the dialog. Choosing the right domain to present the data is not always obvious. Therefore one needs to study the senses in more detail.

Perception

Humans have 7 major senses, which can be put into 2 main groups:

- **Physical**:
  - **Hearing**: Sound energy detected by the ears
  - **Vision/sight**: Light energy detected by the eyes
  - **Touch**: Pressure energy detected by the skin
  - **Temperature**: Heat energy detected by skin but with different nerve endings than for touch
  - **Balance**: Gravity energy detected by inner ear

- **Chemical**:
  - **Smell**: Chemical energy and shape of molecules detected by the nose
  - **Taste**: Chemical energy detected by the tongue

(Kurtus, 2002; FA. Brockhaus GmbH, 1999)

We need some selection criteria to choose the optimal sensorical target for the user-interface.

- **Bandwidth**: How much information can be supplied?
- **Locality**: Can the information be sent to one person directly? Does it affect others around? Can one easily switch between information sources (fast reaction time of senses)?
• **Context**: Can information apart from the current topic be presented as well?

At first each of these senses is characterized by its channel-bandwidth, which in turn depends on the amount of data that can be presented and the number of available attributes to markup the information. Additionally the bandwidth varies according to the physical capabilities of the sensor organs. Some animals can sense much more than we can (e.g. have better hearing or better night vision), but even in our own species there is a variety in what we can sense. ([Kurtus, 2000; Chudler, 2004](#)) When using the bandwidth as a selection criteria, it is apparent that only three of them, namely hearing, touch and vision are usable for reading a user interface. With the help of todays technology, the bandwidth our senses offer can be increased though, to either extend perception (infrared devices) or to correct deficiencies (glasses, hearing aid).

From the chosen three, vision is suited best as it allows the perception of multi-dimensional data (space and time), where hearing and touch can only receive sequential data. With vision we are able to glance over the information and then concentrate on the desired parts. This way a program can offer several contexts at once and the user can just ignore parts which are temporarily unimportant.

In the articles by Lumsden, Brewster, et. al. one can read about how audio can be used for interfaces. The authors conclude that it is very worth to explore the possibilities of enhancing interfaces with audio. Depending on the task, the additional amount of information which can be supplied through audio can be quite substantial. ([Lumsden et al., 2002; Lumsden and Brewster, 2001b; Lumsden and Brewster, 2001a; Lumsden et al., 2001b; Lumsden et al., 2001a](#))

The senses taste and smell are not trained well enough and probably even not fine-grained enough to perceive complex information (in terms of transitions speed and resolution). Hearing and smell have the disadvantage of the need to ensure that the sensorical output does not affect others e.g. wearing headphones for local hearing. In contrast to vision it is not so easy to switch between sources, e.g. one can quickly switch the focus between several monitors, but would need technical assistance to switch between several audio-sources. Still most UI-toolkits are limited to mainly visual output. Touch and hearing are definitely suitable for low-bandwidth interfaces.

The disadvantage of touch is that is usually needs training of the user, before good performance is achieved.

**Action**

Furthermore the user needs to be able to react by sending information (commands and data) to the software. Currently there are many different technologies available for data input (character input, pointing and selecting), which usually utilize hands, the human voice or head/eye-motion.

- Head-tracking (see [http://www.naturalpoint.com/](http://www.naturalpoint.com/))
- Joystick, pointing stick
CHAPTER 3. PROBLEM DESCRIPTION AND TASK DETAILS

- Keyboard (incl. keyboards with special function keys such as Braille keyboards)
- Microphone together with voice recognition software
- Mouse, trackball
- Graphic tablet, touch pad, touchscreen

There have been several attempts to classify input devices. One such taxonomy for devices used for locating objects on the screen can be found in the work of Buxton (Buxton, 1993).

It is desirable to choose input devices depending on the habits, the work environment and the capabilities of the user.

Summary

Several tasks need to be tackled in order to equally support alternative domains:

- Choose an appropriate domain for interaction, depending on the user, the task and the environment.
- Provide mappings of the interface content to representations in alternative domains, such as text and audio. Offer matching input methods to the user, such as speech or keyboard input.

3.4.2 Presentation mapping

An interface can be presented in various styles. The term presentation mapping refers to the process of producing a certain kind of presentation style from an abstract interface description. While choosing the right rendering plugin is important to adapt to the physical properties of the user, choosing the right presentation mapping can take care of the user’s knowledge level. There are many factors involved which one has to consider to choose the right mapping for the right user (for details see section 2.2 and the definition of user profile). Example: Depending on the knowledge of the end-user, the dialog might be presented differently. Many command-line tools e.g. have a switch called ’verbose’. Supplying this switch when invoking the command, instructs the tool to output more status information and helpful hints during execution. Graphical user-interfaces often provide such extra information in a status bar or in the form of tooltips. The latter are usually presented as small boxes that appear close to the mouse-cursor, when pausing it over an interface object.

Regarding the user account administration example (see section 3.1.1) first time users would probably like to see the guiding help, but experience administrator will not.

Generally speaking there are two major tasks to be performed in this layer:
1. **Layouting interface objects**: Distribute the interface objects over the available space.

2. **Decorating interface objects**: Apply styles to the presentations.

These two tasks impact the whole appearance of the dialogs. Every renderer has a certain level of freedom in choosing interface objects and how to present them (layout and visual attributes). Most GUI toolkits come with guidelines that suggest which choice is appropriate in which situation. For some toolkits these style guides have changed over time. Unfortunately the guidelines are not separated as computable rules of some form, they often come as a part of the documentation. The developer then interprets them and usually hard-codes them into the user-interface. When the guidelines change, the whole user-interfaces has to be redesigned. To cure the situation, these style guidelines should be grouped together as a preset (e.g. "Windows2000 styles guides") and the software should apply these rules to the user interface design. This way the users can choose the style they are comfortable with. Figure 3.10 shows two example dialog layout presets.

![Figure 3.10: Two possible dialog layout styles](image)

**Layouting interface objects**

This task consists of two subtasks. The major part is to distribute the interface objects over the available space, while considering the available dimensions. Of note is that time is treated as just another dimension in space. This is necessary to model interfaces where the user is only communicating with the application by audio.

Advanced graphical user interfaces have one or more layout algorithms, where the interface designer packs the interface objects into containers and adjusts constraints of the layouting process. Good toolkits automatically add meaningful constraints, such as that a single line text field does not benefit from additional vertical space, but that a list might. On the basis of the available space and the constraints, the interface objects are positioned and sized by the layout algorithm.
In the scope of this work such a layout algorithm needs to be extended. Layout algorithms for graphical user interfaces are working with two dimensions. They distribute the objects vertically and horizontally over the available screen space. The requirement in this work is to design algorithms for arbitrary dimensions. The problem here is to describe the layout in a way independent from the dimensionality while producing usable layouts in any of the dimensions.

Such a layouting process might divide the interface into visible and not-yet visible parts (e.g. due to space restrictions). In this case it is necessary to add interface objects to allow the end-user to navigate between the parts and to commit or abort the dialog (see section 3.5.1).

The dialog layout itself might be constrained even further by applying a general presentation style, e.g. display all-at-once or focus-oriented in a wizard style.

**Decorating interface objects**

One subtask here is to transfer semantic markup (such as default action, or important message) to the target domain. It is clear that this is not as simple as it sounds. One reason for that is that different interface technologies require different metaphors to be used. Marking something as important may use volume in a speech interface, but bold text or color in a text-based interface.

Additionally this task is the right place to apply style-guidelines (operating system style guides) and user preferences (bigger fonts, colors with more contrast). See section 3.4.3 for a list of usable decoration attributes for each possible target domain.

Applying the styles requires a careful organization of the transformation order, so that no collisions of style settings occur.

**Summary**

To summarize, the presentation mapping defines the overall layout and presentation of the dialog for a specific presentation domain.

### 3.4.3 Interface object mapping

In every sensual domain one has a large number of interface objects to choose from when building a dialog representation. Obviously no simple 1 : 1 mapping exists here. Instead one faces the problem that in some domains several alternative representations for one interface object exist and that the mapping is often a matter of corporate/personal preference. The same applies to the range of available attributes to decorate the presentation. Table 3.5 gives an overview about the available attributes for each of the major human senses. (Kurtus, 2000)

A useful concept for interaction is the use of metaphors. Our daily life is full of graphical metaphors (icons). Using different domains for presentation is initially more difficult to handle as it requires the user to learn new metaphors. For the interface
CHAPTER 3. PROBLEM DESCRIPTION AND TASK DETAILS

<table>
<thead>
<tr>
<th>Sense</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hearing</td>
<td>Emphasis/intonation, loudness, voice, speed</td>
</tr>
<tr>
<td>Smell</td>
<td>Odor, intensity</td>
</tr>
<tr>
<td>Taste</td>
<td>Flavor, intensity</td>
</tr>
<tr>
<td>Touch</td>
<td>Roughness, temperature, hard/softness, structure, resistance</td>
</tr>
<tr>
<td>Vision</td>
<td>Color, font-faces, -styles and -sizes, graphical items and -styles (shaded, highlighted), animation (movement, blinking)</td>
</tr>
</tbody>
</table>

Table 3.5: Attributes of human senses

developer it is a challenge to design easy to understand metaphors for alternative presentation domains. Again the use of the attributes listed in table 3.5 as well as the choice of metaphors should be externally controllable, e.g. by presets or style guidelines. This sub-problem is closely related to the problem of decorating interface objects (see section 3.4.2).

In summary the goal of this step is to design an algorithm which can map the interface objects found in the abstract dialog definition to the target domain, on the basis of exchangable style guides. In contrast to the previous step, which is operating on the dialog level, this step is working on the widget level.

When the presentation has been chosen, the task that follows is to look at the interaction objects.

3.5 Problems related to interface behavior

Problems related to the 'look' of an interface have been discussed in the previous section. This section introduces the problems related to the 'feel' of an interface. The 'feel' of an interface deals with properties such as interaction styles and navigation. In contrast to the style of an interface the possible variety regarding behavior is not obvious. But a research on the topic of interaction styles reveals that this is a broad field. In the book ”Designing the User Interface” (Shneiderman, 1998)[page 72] one can read more about the five primary interaction styles:

- Direct manipulation,
- Menu selection,
- Form fill in,
- Command language and
- Natural language.
These various styles are more or less suited for each target domain. As an example, *command language* would be a better choice than *direct manipulation* for a speech interface.

The styles are also linked to the experience and the knowledge level of the user. In the case of the user account example (see section 3.1.1) an experienced administrator who knows the defaults, would prefer to use a command-interface to enter `user-add --new jsmith "John Smith"` instead of launching a graphical account editor.

Another aspect of behavior is how navigation is handled. Navigation needs to ensure that users always know:

- Where they are,
- What they can do and
- How to get back.

The next two chapters deal with inter and intra dialog navigation.

### 3.5.1 Dialog navigation

Providing navigation on the dialog level (inter dialog navigation) means adding action-widgets like *previous* and *next* to navigate around in the dialog space. These interface objects should be clearly separated from the dialog contents as they are not needed for the primary purpose of the dialog: to enter data. The number of interface objects needed for dialog navigation depends on the underlying dialog structure and the style of presentation:

- **Page network**: Linked pages
  - **Linear**: Previous, next
  - **Hierarchical**: Previous, next, up

- **2d graphics**: Scrollbars for x- and y-axis

- **3d graphics**: Scrollbars for x- and y-axis, transparency to look through

The list above shows that using three dimensions not only gives more space to place interface objects, but also makes navigation more difficult. In such an environment it is easier to get lost and overlook things.

The address manager example introduced earlier (see section address manager) has multiple tasks (enter, search, browse). In a one dimensional interface, one would start with a menu letting the user pick a task, then executing this task and afterwards returning to the menu. In a two or more dimensional interface views for multiple tasks can be shown simultaneously.

Several schemes exist to control the commission of changes:
• Okay
• Okay, cancel
• Apply, okay, cancel
• Try, revert, okay, cancel

These actions have the following interpretation:

• Cancel: Ignore the changes (revert) and end the dialog.
• Okay: Acknowledge some information or apply the changes and end the dialog.
• Try, apply: Apply the changes without leaving the dialog.
• Revert: Restore settings as they were before opening the dialog.

Furthermore there might be actions such as help, as dialogs usually have no menu bar. Again it is clearly apparent that there is no single solution for all applications and all modalities. An okay command to leave the dialog is the minimum requirement. If the dialog is for querying parameters about a task that is to be started, okay and cancel commands are needed. If the purpose of the dialog is to manipulate properties of an object, try and revert commands are useful. The okay command would only serve to finish the dialog in this case.

3.5.2 Interaction serialization

Like the navigation between dialogs, the navigation inside the dialog (intra dialog navigation) needs to be organized. Intra dialog navigation deals with the focus transitions from one interface object to another interface object.

When using a serial interface (e.g. speech) interaction needs to be serialized. A simple example is a dialog with many options. These options have to be presented in a step-by-step fashion to the user when using a serial context. When using a multi-dimensional interface (e.g. using a virtual world), an adaptive system should be able to make an advantage of the extra dimension and present the interface in 3 dimensions. This also means that there will be no sequential interaction like in the first case. Figure 3.11 shows one problem with serializing interaction for the user account administration example (see section 3.1.1). When the interaction path splits out, there is no simple choice which interaction object to activate next.

Another problem related to the order, are backward references. Figure 3.12 gives an example for this case. Imagine setting the "account type" to "system account" instead of "user account". In the example this will cause the password entry to become disabled as a system account cannot be used to login and thus needs no password. Such a case is confusing as the user would initially enter a password, chose the type and then the entered data item would be removed as it is not needed. In a serial environment this
is even worse, as the affected item is not shown at that moment and will not be visited again. Interaction is already difficult enough, so it should not be further obscured with unnecessary interaction steps.

Along with that, the dialog navigation needs to be adapted appropriately (see section 3.5.1), so that the user can easily walk through the application.

### 3.5.3 Multiple interface instances

When multiple persons are working together with one application and one set of data, this is known as CSCW (computer supported cooperative work). This raises a series of problems which are:

- Who controls the dialog transitions (some dialog are even sequences of transitions).
- How to manage the focus and cooperatively transfer it from one worker to another.

When the interface engine in use supports multiple media, then this topic gets another important use - cooperative work of people where some of them might be handicapped.
3.6 Problems related to interface content

Content is the data that is processed by the application, either entered by the user or displayed as a result.

3.6.1 Content selection

The difficulty about providing a clean separation of content from other interface aspects in the context of cross-media interfaces is to define what the content actually is. The value of a text-input field clearly belongs to the ‘content’ category. The text of a label is interface content as well. But the font that is used to render it is not. This belongs to the presentation style. It gets more difficult when applying this separation to interface objects, such as toolbar widgets. These widgets can use an image to represent their function. Such an image could be drawn differently, but still represent one and the same function. Therefore the image metaphor is content, but the way it is drawn is style.

To summarize, text-items such as labels, titles, tooltips and help-texts should be internationalized using language catalogs. The catalog technique can be used for other content elements as well. Unfortunately it is not so straightforward to set up catalogs with a different set of image metaphors for each culture.
3.6.2 State persistence

With the advent of ubiquitous computing, people will use techniques such as roaming. That means the application will be used under varying environmental conditions and with changing modalities. These conditions might change which the application is in use.

That puts an additional load onto the interface management system as it needs to provide a mechanism to remember the state of the interface (focus, already entered values) when re-adapting (changing style, changing interface modality).

3.7 Technical problems

3.7.1 Programming language bindings

If the programming languages available to the developer to use with the solution should not be restricted, bridges between the native language of the solution and other languages are needed. This requirement raises complexity if totally different technologies have to be bridged (e.g. C which is mainly procedural and Java which is mainly object oriented). The modules that provide such functionality are called language bindings or wrappers.

Having bindings to scripting languages (like Perl and Python) would facilitate rapid prototyping, while bindings for object oriented languages (like C++ and Java) would aid development of production systems.

There are two paradigm classes a set of language bindings needs to bridge:

bullet compiled versus interpreted
bullet procedural versus object oriented

3.7.2 Operating System integration

The various media an adaptive solution aims to support differ from each other in many ways. An interface solution needs to consider two levels at which those media are to be supported. Primarily, applications need to be extended to handle interfaces in a media-neutral way. This is what many solutions focus on. A second level that seems to be unconsidered is the infrastructure outside applications. Many devices we interact with come with multiple applications installed. The operating system of a device is there to provide services outside of applications. Unfortunately common desktop operating systems such as UNIX (Linux), Windows and MacOS mainly provide support for textual and graphical operation. A complete adaptive solution needs to provide extensions to the operating system services that work with the new media.

The next chapters discuss the services user authentication and application launcher under that point of view.
CHAPTER 3. PROBLEM DESCRIPTION AND TASK DETAILS

User authentication

Before a user can start to work on a computer system, it is often required to log in first. This refers to a mechanism where users prove their identity, before being able to use their resources. Today’s operating systems usually come with a text based login prompt as well as a graphical login manager.

A system that offers multiple modalities needs to support those modalities for the user authentication as well, so that a consistent interface is realized throughout the whole session.

As an example, imagine an administrator who wants to remotely unlock a user account (see section 3.1). The administrator will use a mobile phone to open a session on the server that locked the user. Then the server’s operating system would need to use the modality of the phone connection to run the dialogs of the user account administration software.

Application launcher

A related problem is the facility to launch an application. That facility is often called a shell. Operating systems usually come with several launching mechanisms. The common way is to provide a graphical, menu-based application launcher. In e.g. Windows this is called ”Start-Menu”. Another mechanism is to start applications from a shell prompt, by entering the name of the program to run.

The problem can be illustrated with the admin-tool scenario (see section 3.1). When administrators sit in front of their workstation, they might prefer a graphical user interface, but when using the application over a dial-up network, they need to use a text-based user interface. The application launcher in use might be a kind of ”Start-Menu” or especially in the second case a command shell. In both cases the launcher needs to take the environment into account. When the user is using a remote network connection, probably no graphical environment is available. In other situations the user might not be able to look at a screen at all.

3.8 Development support

As stated in the introduction, developers have to cope with very complex tasks. New approaches in technology related to software development should therefore support the developer. What does that mean? Which are the fields in software development that are affected?

First is the workflow. Using a new approach often changes the order and number of tasks that need to be carried out to reach a goal. A new solution should come with good and well explained reasons for changing the workflow.

Another field is the tool support. Figure 3.13 shows related fields graphically.

Development today is close to impossible without good tool support. Common tools are flexible text-editors or an IDE (Integrated Development Environment). These
Figure 3.13: Needed tool support for developers

are often extended by plugins that add special functionality such as support for programing languages, database connectivity, interface design, project building, debugging and others. If a new approach to interfaces development changes the development workflow, it is likely that this requires new plugins to maintain an effective workflow.

To allow for a smooth transition from an old workflow to the new workflow, migration tools are needed. Such tools help to convert data used with tools from the old workflow into the format required by the new tools. In the context of interface toolkits, this means that the tools support the developer to port older interface designs over to the new toolkit. Such support can range from assistance to automatic conversion.

A final aspect that gains more and more importance is support for rapid application development. This requires functionality to quickly setup and try interface designs.

3.9 Summary

A variety of problems have been described in the previous sections. Each one addresses a specific part of the general adaptation problem. Therefore a possible solution should probably be modularized into a series of components, where each component addresses such a sub problem. This leads to a two phase categorization of the problems:

1. **Construction**: Choose components and determine the best order for the application of the components.

2. **Execution**: Run the adaptation chain and let each component perform one specific adaptation step.

Table 3.6 summarizes the individual sub problems that have been described in the last chapters.

The problems related to structure have been put at the beginning intentionally. The related design decisions must be made prior other decisions, as the other problems build upon them. Additionally to that, the table illustrates that there is adaptation on the dialog and on the widget level for interface style and interface behavior.
<table>
<thead>
<tr>
<th>Aspect</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface</td>
<td>- Choose level of abstraction</td>
</tr>
<tr>
<td>structure</td>
<td>- Design of specification language</td>
</tr>
<tr>
<td>Interface</td>
<td>- Choosing presentation and action space</td>
</tr>
<tr>
<td>style</td>
<td>- Choosing dialog style</td>
</tr>
<tr>
<td></td>
<td>- Choosing widget style</td>
</tr>
<tr>
<td>Interface</td>
<td>- Interaction on the dialog level</td>
</tr>
<tr>
<td>behavior</td>
<td>- Interaction on the widget level</td>
</tr>
<tr>
<td></td>
<td>- Focus management (multiple interfaces)</td>
</tr>
<tr>
<td>Interface</td>
<td>- Choice of proper content</td>
</tr>
<tr>
<td>content</td>
<td>- Managing dialog state across presentations</td>
</tr>
</tbody>
</table>

Table 3.6: Summary of the problems grouped by interface aspects
Chapter 4

Existing approaches

The multi-dimensional adaptation problem has been introduced in the previous chapter (see chapter 3). A research on the Internet and in the literature reveals many solutions for specific adaptations. The topic of this chapter is the introduction of the most important technologies, approaches and ideas.

Most currently existing approaches try to achieve independence from programming language like Glade (see section 4.2.2) and/or the underlying UI-toolkit like wxWidgets (see section 4.1.7 and UIML (see section 4.1.1). Only a few solutions offer automatic domain independence. This is a capability of the approach to adapt to a target system without the need of specific guidelines from the application. Such a system could equip an application with an interface type that was not available to or was not considered by the application developer.

Basic toolkits have been introduced in section 2.6 and abstract toolkits are discussed in section 4.1. Furthermore a number of software packages exist which provide solutions for specific problems in a fashion that could be used for more general problems. These concepts are introduced in section 4.2.

This chapter is closed with a comparison of the introduced approaches.

4.1 Abstract toolkits

To achieve independence from an interface platform, one has to generalize features and map them to the respective target system. For that two techniques can be utilized:

- **Least set of features**: Only include features available on all target platforms. Therefore features are limited to the features of the weakest supported system (see figure 4.1). This approach is easier to implement, but can easily lack essential features.

- **Widest set of features**: Include all imaginable features. When a certain feature is not available on a target platform, it will be ignored or emulated (see figure 4.2). This approach can be richer in features, but is more difficult to implement.
Most of the technologies presented in the next chapters try to emulate missing features. Furthermore they have in common that they can all produce various interfaces from (more or less) one specification. Such a specification is often based on XML technology. Robin Cover introduces many of these approaches on his website (Cover, 2004).

One can find many more approaches in the literature, where some of them seem to became dormant or never developed beyond a concept (Avanti, Homer UIMS) and some never became widely well known (TeleUSE).
4.1.1 UIML (User Interface Markup Language)

Overview

The creators of UIML describe their approach as follows:

UIML stands for User Interface Markup Language. UIML is an XML language for defining user interfaces. It is used for defining the actual interface elements. This means the buttons, menus, lists and other controls that allow a program to function in a graphical interface like Windows or Motif. UIML is used to define the location, and design of controls. It also defines actions to take when certain events take place. Users create events when they interact with the interface by typing a key on the keyboard or moving and clicking the mouse. (Phanouriou, 2000a)

Section 7.4.1 in the appendix contains a sample dialog description.

Architecture

UIML uses rendering modules that dynamically or statically generate a user interface from an interface description. This general approach sounds sensible. The whole system is based on Java technology and even the processing of the abstract dialog description towards the dialog representation is all done in Java logic. UIML uses XML based dialog descriptions. Version 1 was using a format similar to CSS (Cascading Style Sheets) for style files and content databases. Version 2 holds all information about the dialog in one XML file. (Phanouriou, 2000b)

Evaluation

It is important to note that this approach is commercially backed by a company called Harmonia (see (various contributors, 2001b)). The software itself is currently not available for free (only a trial version which times out after a while is available for download) which is a major drawback. It is important that these systems should be as cheap as possible or even freely available, especially in the context of systems that should help social minorities. Otherwise these communities might not be able to afford them and the developers might not use them.

The major points one could argue regarding the architecture are:

1. The need for adaptation profiles for each target system,
2. The separation of content, style, structure and behavior is not enforced,
3. Bad reuse of existing technology (i18n, xsl, xinclude),
4. Adaptation is carried out as one time conversion and not as a continous process and
5. Too much of the processing is “hard-coded” in Java logic.
The latter makes it inflexible and hard to adapt to future requirements. One would need to change the system itself and cannot change adaptation rules in external files.

4.1.2 XUL (XML User-interface Language)

Overview

XUL stands for XML user interface language. According to the definition given by the OpenXULAlliance, this includes a wide variety of approaches starting from Mozilla XUL to Microsoft’s XAML (Bauer, 2004). These approaches are called XUL motors.

The Mozilla XUL Project describes (their) XUL as follows:

The XML User Interface Language (XUL) is a markup language for creating rich dynamic user interfaces. It is a part of the Mozilla browser and related applications and is available as part of Gecko (the browser rendering engine). It is designed to be portable and is available on all versions of Windows, Macintosh as well as Linux and other Unix flavors. With XUL and other Gecko components, you can create sophisticated applications without special tools. (Andersen and Deakin, 2004)

Evaluation

As the name XUL covers multiple solutions, it is hard to draw a conclusion here.

Pro:

- Most approaches can create good looking interfaces.
- Most approaches provide a good level of separation of content and style from structure and behavior.

Contra:

- Not standardized. Nearly every approach uses its own language and own set of tools.
- Often only graphically oriented. Language defines attributes such as "image".
- XML technology is often only used for platform independence.

(Puerta and Eisenstein, 2001)

4.1.3 XIML (eXtensible Interface Markup Language)

The little information available about this technology proposes a solution that can adapt interfaces to multiple different devices. (Puerta and Eisenstein, 2001) Unfortunately no technical details are available. As the available information dates back to the year 2001 one can assume that the project has been discontinued.
4.1.4 XForms

Overview

XForms (The Next Generation of Web Forms) is an XML application that represents a new approach for forms on the Web. It splits traditional XHTML (Extensible HyperText Markup Language) forms into three parts: XForms model, instance data, and user interface. This separates presentation from content, allows reuse and gives strong typing. Consequentially, this reduces the number of round-trips to the server, offers device independence and a reduced need for scripting. (Dubinko et al., 2003)

An implementation is required for both - the server and the client. The specification mostly addresses the client side, where the XForms processor

- Produces the form presentation
- Handles events
- Performs validation

The first W3C recommendation for the XForms standard was published during the development of this project.

Evaluation

The W3C mainly targets web-applications with the XForms standard. It is in fact designed to be the successor of HTML (HyperText Markup Language) forms.

Some characteristic properties of this approach are:

- Network communication is tightly bound to the request-response scheme of the HTTP (HyperText Transfer Protocol) protocol
- No specification for local (non-network) interaction
- The only interaction with the application is the submission of data
- No relation between interface objects (such as of a label for an input field)

A big part of the XForms standard is the data validation specification. Data validation should avoid round-trips to the server by pre-checking data on the client. It was motivated by that fact that especially these days communication with the server still causes noticeable delays. It can be argued whether or not client side validation is the cure for the delay problem. Client side validation effectively doubles the data-validation code. Data that is sent to the server needs to be checked again, as the server cannot rely on the XForms client validation capabilities.

Furthermore this can cause confusion for the user, as parts of the data is validated locally and other parts remotely. This leads to different types of error message displays (local error are reported in message box and remote errors are shown on the resulting
Finally, local data validation is always limited to the local data set. For huge database relations, this would require sending a huge dataset to the client. The goal of the validation specification was to speed up form handling, but sending a large amount of data can cause worse delays than an additional data gathering step.

4.1.5 AUIML (Abstract User Interface Markup Language)

Overview

The AUIML (Abstract User Interface Markup Language) Toolkit designed at the IBM Labs, provides software development tools that allow to write an application once and run it using Java Swing or HTML without any changes.

AUIML is an XML dialect that is a platform and a technology-neutral representation of panels, wizards, property sheets and more. AUIML captures relative positioning information of user interface components and delegates their display to a platform-specific renderer. Depending on the platform or device being used, the renderer decides the best way to present the user interface to the user and receive user input.

Evaluation

For a long time no direct online resources were available. In the year 2004 some resources have been published at the IBM website (CSA Tooling team, 2004). The AUIML focuses on visual interfaces, namely HTML and Java Swing. It comes with tool support in form of a plugin for the Eclipse IDE.

4.1.6 XMLFace

Overview

The XMLFace toolkit is a library that allows a developer to quickly and easily build GUI based applications through the use of XML description files as opposed to using Java code. By changing just one setting, the user can choose between using Java Swing or the SWT. (XEsoft GmbH, 2004)

Evaluation

This solution is wrapping the different API of two graphical toolkits. Both toolkits have a comparable functionality. Thus the mapping is relatively straightforward and can concentrate on technical aspects.

The XMLFace toolkit uses an own XML language to describe the interface and Java property files for the content. The XML language is tied to the graphical UI domain.
4.1.7 wxWidgets

Overview

The wxWidgets toolkit provides a single, easy-to-use API for writing GUI applications on multiple platforms. It was formerly known under the name wxWindows. On top of powerful GUI functionality, wxWindows gives the developer online help, network programming, streams, clipboard, drag and drop, multi threading, image loading and saving in a variety of popular formats, database support, HTML viewing and printing, and much more. (Smart, 2001)

Architecture

The toolkit provides a uniform C++ class library. Platform details are hidden from the developer and user. Relating platform specific features, the wxWidgets toolkit does not follow the lowest common denominator approach. The application developer needs to ensure that the application run on all targeted platforms.

Evaluation

The focus of the wxWindows software is on platform independence, but not on the abstraction of interfaces. Neither does the toolkit aim to provide adaptation at runtime. The platform is chosen at compile time of the application and cannot be changed later on, except by recompilation.

4.2 Other approaches

There are numerous other approaches available that contain good ideas for particular problems and provide good detail solutions. They all do not aim to provide a generic adaptive interface solution. A few of those projects are discussed next.

Several of these are based on XML technology, which underlines the practicability of such an approach.

4.2.1 DialogML

Overview

DialogML is an XML markup language for simple, flexible and efficient development of setup like applications (software installation and configuration programs). It provides a small set of elements for the definition of interactive interfaces and event driven command calls. The XML language is specified by a DTD (Document Type Definition). It uses CSS to apply layout and styles.

Localization of dialogs is part of the specification, although based on its own framework.
The design integrates the SEML (Simple Event Management Language) technology from the same author. This language is used to specify interaction rules. (Heuer, 2001)

Evaluation

This approach currently consists just of design documents. There is no reference implementation available and the public CVS repository is empty. As the latest sign of life was from mid of year 2000, the project seems to have been discontinued.

4.2.2 Glade

Overview

Glade is a GUI builder which uses an own XML for describing interfaces. The generated interfaces are programming language independent. There are two ways to use the interfaces:

1. **Code generation**: Modules for code generation exist for languages such as C, C++ and Perl. The generated code can easily be integrated into existing projects or can even be taken as the template for a new project.

2. **Run-time loading**: Glade comes with a library that can load Glade-XML specifications and construct dialog instances at run-time. Again, several language bindings for this library exist.

Final interfaces runs on the gtk+ widget library. (Chaplin, 2001) Section 7.4.4 in the appendix contains a full length example.

Evaluation

The used XML language is a 1 : 1 mapping of the functionality the gtk+ widget library supports. It only abstracts the target programming language to use. That fits exactly with the purpose of Glade as a interface design tool. Apart from that, it serves well as a prototype generator.

4.2.3 PonG

Overview

PonG is a library and a GUI tool for creating configuration dialogs. It uses an XML file which describes the configuration dialog and the gconf keys (gnome configuration registry) that should be used. Adding a configuration dialog to an application just requires to add a few of lines of code. (Lebl, 2001) Section 7.4.3 in the appendix contains a full length example.
CHAPTER 4. EXISTING APPROACHES

Evaluation

This solution had its final update in summer 2001 and has been discontinued due to time constraints. It is clearly specialized in the generation of settings dialogs. In the existing state it has a strong dependency towards gtk and the gconf registry. PonG supports localized dialogs by integrating itself into the gettext workflow. One specialty of the approach is the support of various knowledge levels. A dialog can have a different appearance depending on whether the user is classified as: Novice, User or Hacker.

The approach is suited quite well for the task it has been designed for. The implementation of adapting for different kinds of users is handled in a quite simple fashion, but nonetheless it was implemented.

4.2.4 HTML (HyperText Markup Language)

Overview

HTML (HyperText Markup Language) is originally based on SGML. The recent version XHTML is based on XML.

(X)HTML uses a mixed content model (structure, content, style, behavior). Separation of style is possible via CSS.

When (X)HTML documents are used in conjunction with CGI (Common Gateway Interface) or JavaScript, interactive applications become possible (separation of behavior). CGI request are sent to an application on the server, which processes the request and generates the response (usually as a new (X)HTML page). Such an application can be written in any programming language. Thus such HTML based interfaces are programming language independent. If JavaScript is used processing is performed on the client. Due to incompatibilities between browser implementations of JavaScript it is risky to rely on it.

Finally an HTML browser is needed to generate a presentation of the document. As these are available on a wide range of devices, (X)HTML provides interfaces which are device independent as well. (W3C HTML Working Group, 2002)

Evaluation

HTML was not invented for device independent markup. The evolution of the standard up to the current XML reimplementation (XHTML) nevertheless formed a path towards achieving device independence. HTML 4.0 and XHTML have an improved structural model (elements such as tabindex, summary, caption, ...). By using CSS, all graphical layout information can be separated from the XHTML document.

To execute logic on the client side, HTML provides the script tag, that allows the use of languages such as JavaScript and ECMAScript.

A document is still best presented visually, as the form objects are still visually oriented. And finally HTML applications are restricted to the request response model for communication with the application logic on the server.
4.2.5 VoiceXML

Overview

VoiceXML (Voice Extensible Markup Language) is designed for creating audio dialogs that feature synthesized speech, digitized audio, recognition of spoken and DTMF key input, recording of spoken input, telephony and mixed-initiative conversations. Its major goal is to share the advantages of web-based development and content delivery with interactive voice response applications. (Boyer et al., 2000)

The interface is specified in an own XML called VoiceXML. A document is interpreted by the VoiceXML interpreter and executed by the FIA (form interpretation algorithm). This algorithm determines which form-field to visit next, prompts the user and collects input.

Evaluation

The VoiceXML approach is oriented on the request response mechanism used for web applications. It can be integrated in a web application, if it is flexible enough to answer with VoiceXML instead of HTML. Unfortunately there is no support for local use of such applications.

The VoiceXML language is simplistic when compared to languages like HTML. It distinguishes between dialog content and dialog navigation by providing two kinds of dialogs: forms and menus. The language itself could have been made more foolproof. It e.g. uses tags like \(<\text{goto next}="\text{abc}"\>\) to reference \(<\text{form id}="\text{abc}"\>\), instead of \(<\text{goto form id}="\text{abc}"\>\). Further it often provides multiple ways to express things, where often this serves as shortcut to reduce typing. An example is that prompts without markup can omit the enclosing \(<\text{prompt}\>\) tags. Considering that writing (in the meaning of keying in) dialogs specification is not a major time consuming part of development, such optimizations are contra productive as they also can cause confusion.

A VoiceXML document may contain ECMAScript expressions. These expressions can e.g. control dialog transition or validate input. This leads to the same problem web-applications have - one needs to carefully decide which logic is executed in the application and which in the interface (if there should be any logic at all).

Adapting to the user in terms of locale or personal habits is left to the application developer and not supported by the language.

4.2.6 WAP (Wireless Application Protocol) and WML (Wireless Markup Language)

Overview

This technology consist of two parts:

1. **WAP (Wireless Application Protocol)**: The *wireless application protocol* is used for communication and data transfer.
2. **WML (Wireless Markup Language)**: The *wireless markup language* is used to create pages that can be delivered using WAP.

WML is designed for low-bandwidth, small-display devices. The language is based on XML. As part of this design, the concept of a deck of cards was utilized. A single WML document (i.e. the elements contained within the `<wml>` document element) is known as a deck. A single interaction between a user agent and a user is known as a card. Using WMLScript, user selections or entries can be handled and routed to already loaded cards. (ThinkBurst Media, Inc, 2004)

The WAP is a multi-layered protocol stack. Its layers are:

- Wireless Application Environment (WAE)
- Wireless Session Layer (WSL)
- Wireless Transport Layer Security (WTLS)
- Wireless Transport Layer (WTP)

WAP was conceived by four companies: Ericsson, Motorola, Nokia, and Unwired Planet (now Phone.com). (Open Mobile Alliance Ltd., 2002)

**Evaluation**

WAP has been designed for client server based communications. There is no support for local use.

WML is specialized in describing interfaces for client devices with small screens. The language is concentrating on the interface structure. It contains a scripting language based on ECMAScript to e.g. do form validation before submitting data.

The interface presentation is left to the client and not part of the specification.

### 4.3 Comparison of existing approaches

Table 4.1 summarizes the level of adaptiveness of the previously introduced abstract toolkits. According to the classification that has been suggested in section 2.2.3 most approaches abstract a series of underlying toolkits. Those which are purely based on SGML (Standard Generalized Markup Language) or XML can be used with about any programming language too.

Apart from that, one can see that none of the approaches delivers true automatic adaptation over all three aspects of adaptation. All focus on some aspects of the adaptation problem.

Related to this is the evaluation of which changes are needed in the toolkit to support new modalities (new i/o hardware, new communication channels). Table 4.2 lists those changes.
CHAPTER 4. EXISTING APPROACHES

<table>
<thead>
<tr>
<th>Solution</th>
<th>Programming language</th>
<th>Interface toolkit</th>
<th>Devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>UIML</td>
<td>Java</td>
<td>manually to e.g. awt/swing, html, wap voicexml</td>
<td>manually to pc, palm, mobile</td>
</tr>
<tr>
<td>XUL</td>
<td>?</td>
<td>swing, xp toolkit</td>
<td>only pc</td>
</tr>
<tr>
<td>XIML</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>AUIML</td>
<td>Java</td>
<td>swing, html</td>
<td>pc</td>
</tr>
<tr>
<td>XMLFace</td>
<td>Java</td>
<td>swing, swt</td>
<td>pc</td>
</tr>
<tr>
<td>HTML</td>
<td>any</td>
<td>html browser based</td>
<td>automatic to all devices that have a browser</td>
</tr>
<tr>
<td>WAP, WML</td>
<td>any</td>
<td>wml browser based</td>
<td>automatic to all (mobile) devices that have a browser</td>
</tr>
<tr>
<td>wxWindows</td>
<td>C++</td>
<td>automatic to e.g. gtk+, mfc</td>
<td>pc</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of approaches with respect to aspects of adaptation

<table>
<thead>
<tr>
<th>Solution</th>
<th>Components</th>
</tr>
</thead>
<tbody>
<tr>
<td>UIML</td>
<td>new output renderer, change of all applications</td>
</tr>
<tr>
<td>XUL</td>
<td>new XUL renderer (no reuse)</td>
</tr>
<tr>
<td>XIML</td>
<td>?</td>
</tr>
<tr>
<td>AUIML</td>
<td>internal extension</td>
</tr>
<tr>
<td>XMLFace</td>
<td>internal extension</td>
</tr>
<tr>
<td>XForms</td>
<td>new form renderer (no reuse)</td>
</tr>
<tr>
<td>HTML</td>
<td>language and browser</td>
</tr>
<tr>
<td>WAP</td>
<td>language and browser</td>
</tr>
<tr>
<td>wxWindows</td>
<td>new class library</td>
</tr>
</tbody>
</table>

Table 4.2: Comparison of approaches with respect to what changes are needed to support another modality

The term no reuse means that the approach does not provide a common foundation code layer for renderers to build on. A first conclusion from this table indicates that most approaches are not designed to support future modalities. Only UIML and wxWindows have been designed with such a requirement in mind. Still, the current solutions do not separate adaptation methods from adaptation infrastructure clearly enough to approach the problem classes one by one. This becomes apparent when e.g. for UIML the choice of modality is still linked all the way through the toolkit onto the application side. It therefore can be concluded that none of the described approaches fulfills the requirements for a holistic approach to the adaptation that have been made in the beginning of this work.

Another comparison can be made regarding the level of abstraction the approaches
cover. In section 2.9.7 the Cameleon framework has been introduced. This defines 4 levels of abstraction for user interfaces. Table 4.3 gives a comparison of the approaches in respect to the levels of abstraction they cover.

<table>
<thead>
<tr>
<th>Solution</th>
<th>Concepts and tasks</th>
<th>Abstract interfaces</th>
<th>Concrete interfaces</th>
<th>Final interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>UIML</td>
<td>-</td>
<td>o</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>XUL</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>XIML</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>AUIML</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>XMLFace</td>
<td>-</td>
<td>-</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

\(\text{\textquoteleft-\textquoteleft} = \text{unsupported}, \text{\textquoteleft o\textquoteleft} = \text{partly supported}, \text{\textquoteleft x\textquoteleft} = \text{fully supported}\)

Table 4.3: Comparison of approaches regarding the levels of abstraction

This table shows that most approaches start with the concrete interface which is already related to the target domain (e.g. graphics, speech). It is important that solutions which aim to provide adaptive interfaces also need to cover the level of abstract interfaces. This is mandatory for a flexible adaptation at run-time, as it has been introduced in section 2.9.7. Although the level of concepts and tasks can be covered by tool-integration into the workflow, it is not part of the run-time levels. Therefore the decisions supporting adaptation at this level are of static nature. In other words the decisions are made at design time.

Among the previously presented approaches is a group of very specialized projects, such as DialogML, VoiceXML, WAP/WML and others. These projects do not have a universal adaptation of general applications as a goal, they just cover special areas. This highly reduces the complexity of the adaptation problem. Common to these solutions is the choice of the XML language as a means to describe the domain-independent interface.

Some approaches (XForms, VoiceXML, WAP/WML) integrate facilities to do form checking before submitting data. The time consuming submission of form data to the server is given as justification for this step. This can be argued, as sending the form checking logic together with the verification data to the client instead, can be time-consuming as well. In the case of VoiceXML and WAP/WML it even led to the addition of a script-language to the markup language specifications to enable the execution of logic on the client. This can be problematic, as it leads to scattering of logic and more complex client applications.
Chapter 5

The Generalized Interface ToolKit

Several problems have been analyzed in chapter 3. There is a common agreement that the user should not be the one to adapt. Therefore the user will be put in the center of the design, so that the application adapts to the user (and the environment). The first is also known as user centered design or design for all. The solutions introduced in chapter 4 only partially address the problems that are to be solved in this work. A major deficiency is the lack of automatic adaptation to e.g. different media in any of the approaches.

To successfully meet all criteria that have been listed in the introduction, it is necessary to develop applications with such goals in mind. In his book "user interfaces for all" Stephanidis has called this a proactive approach (Stephanidis, 2001). The paradigm which dominated the last decades was to fix the applications which were available, by trying to add usability enhancing features afterwards. These techniques can often do this only with very limited success. The reason for the low success rate is that applications are not written to be interfaced in that way. The enhancer tools do not get enough meta information (e.g. interface object relations) about the GUI to adapt it to new requirements.

In consideration of this GITK is a toolkit that has to be explicitly utilized by an application. The difference to regular toolkits such as GTK+ and QT (see section 2.6) is that GITK is a meta-toolkit, a toolkit that maps to other toolkits. In other words - GITK provides abstraction on the dialog level. The usage of GITK itself does not differ much from that of other toolkits. The developer segments the application into tasks. For each task that needs user interaction a dialog is started and an event-callback mechanism is used to react onto user input. The difference lies in the way the dialog is specified. The interface specification used by the GITK approach provides rich structural meta data about the dialog and is media independent.

XML based markup languages are a common tool to formulate and process specifications. Other approaches use XML languages too. As has been shown in the conclusion of the last chapter, these projects do not try to achieve the same level of abstraction as the GITK project. To provide a language for domain independent, media neutral description, it was necessary to design another markup language - the GIML (Gener-
The GITK can be seen as a black box that receives the dialog description and handles all details to produce and run an adapted instance of the dialog. The application is totally decoupled from the resulting interface. Thus the GITK architecture is an implementation of the adaptation infrastructure model introduced in section 2.2.1. This black box can transform an abstract dialog description into an adapted representation. This in turn will be the source for the final interface.

It is important to understand that the transformation of the dialog specification is designed and handled as a dynamic, repetitive process and not as a one way conversion. The requirements to the adaptation can and will change at run-time. Therefore the resulting interface must be readapted. This also includes a change in the modality (graphical interface to audio interface). As such a change can happen in the middle of working with a dialog, the state of the interface needs to be captured continuously. GITK accomplishes this by keeping the whole state of the interactive session inside the interface specification, also and especially at run-time.

To summarize, the GITK is a black box for the application and the final interface. This also is a logical consequence of figures 3.1, 3.2 and 3.3. When the adaptation problem was introduced, it has been suggested that a solution needs to sit between the application and the variable parts that need to be adapted universally. Figure 5.1 shows how this approach will look. It is apparent that the adaptation problem is greatly simplified contrary to the situation shown in figure 3.3.

![Figure 5.1: Role of GITK in the adaptation problem](image-url)

The detailed description of the GITK approach is the focus of this chapter. First, the component architecture that forms the black box will be described. This is followed by an overview of the GIML (Generalized Interface Markup Language), the interface
object naming scheme and the canonical interface object hierarchy. Similar to the structure of chapter 3 the description of the solutions are grouped according to their relation to the aspect of an interface: structure, style, behavior and content. For each problem class the descriptions contain details regarding the architecture and the GIML interface description. The solutions to the technical problems and development support end this chapter. Two example applications have been introduced in section 3.1. When illustrating the own solutions these examples will referred back to.

5.1 Architecture

All previously introduced models for adaptive systems require a certain level of modularization. This is necessary for separating responsibilities. Therefore this project uses a multi-layered architecture with several components on each layer. Figure 5.2 shows the modules of the GITK architecture. As one can see, the GITK approach uses an application architecture with 5 separate layers. The processing of the dialog description can be seen as a pipeline that runs across the layers.

- **Application**: Contains the end-user-application objects (data objects and methods to access and process them); supplies abstract dialog descriptions.

- **Wrapper (access-layer)**: Programming language wrappers (not required for C)

![Figure 5.2: The layers of the GITK architecture](image-url)
• **Core** (**libgitk**): The master library to be used by the application; manages all the plugins and the whole transformation process.

• **Renderer-plugins** (**libgitk-renderer- such as text or gtk**): Uses the transformed dialog descriptions to render and run the final interface; these components form the end of the processing pipeline

• **Interface-toolkits**: Platform specific interface toolkits, such as gtk+, html or curses

Figure 5.3 shows the same components along with their relation, including external dependencies.

In the past decades interface architectures have been refined into smaller and smaller components with well defined responsibilities. In section 2.9 several such interface models have been introduced. So how does GITK relate to these models? In the Seeheim model (see section 2.9.1), GITK would take the roles of the presentation and the dialog layer.

When using the ARCH model as the basis (see section 2.9.2), the GITK components can be mapped more exactly to the components of the model:
• **Domain Specific Component**: The application using the GITK
• **Domain Adapter Component**: The GIML dialog description
• **Dialog Component**: The dialog rendering plugin
• **Presentation Component**: The transformation stylesheets
• **Interaction Toolkit Component**: The interface toolkit, used by the renderer

This mapping demonstrates that the GITK architecture provides the 3 components in the middle of the arch and defines the objects that are passed between the components. As an extension GITK also defines a run-time behavior for the ARCH model by using exchangable parts for the inner components. The original ARCH model only misses the notion of reacting to changes in the requirements of the user and the application. Furthermore it does not handle the dynamic influence of the environment.

The position of GITK in the MVC model is not clearly visible (see section 2.9.3). GITK is a system layer between the user and the application domain. Thus both view and controller would have unique interfaces to work with, independent of the devices in use.

In the Cameleon framework (see section 2.9.7), the GITK approach covers the layers from abstract interface to final interface. Models from the concepts and tasks level need to be imported into the GITK system.

A general model for adaptation has been introduced in section 2.2. In this model the gitk-core library provides the adaptation infrastructure, the transformer and rendering plugins provide the adaptation methods and the style-sheets deliver the adaptation profile.

Figure 5.4 illustrates that the initial interface description runs through several stages, where it gets transformed in every step. Each one of these transformation steps handles one special aspect of adaptation. It is important to understand that the transformation is not a one way conversion. Initially the output documents will be generated by forward transformation. At run time changes to both, the input and the output documents can be propagated to the other one. In other words, the content of the input and the output document will be kept in sync. Thus the transformation pipeline needs to be understood as a bidirectional system.

This pipeline model allows an easy extensibility and invites researchers to start further experiments, by simply reconfiguring the pipeline to try different kinds of adaptation. Using one and the same dialog description to generate multiple interface presentations (which is also called single authoring) adds to the experience of GITK as a platform for testing.

Finally a user profile completes the architecture. This profile controls which interfaces the user wants to launch along with the application, as well as all options influencing the adaptation process. This includes the choice of stylesheets to use. Of course it is desirable to have a mechanism that builds most parts of the user profile dynamically. Only this way can the system react to dynamic changes in the profile.
5.1.1 Application

Instead of the usual approach, this part only contains the application logic and the interface callbacks. The source code will be much less cluttered, algorithms are not obstructed by interleaved interface code and the developer can better concentrate on writing the code.

Even though GITK does not require modularization on this level, it is generally suggested to separate the data (the model) from the callbacks (the controller).

5.1.2 Wrapper (access-layer)

The application accesses the provided functionality through a library (shared or static - depending on the platform) with a very lightweight API. A big advantage of an API with only few entry points is that it can easily be ported to other platforms. Languages other than C can easily be supported by providing language bindings (wrapper, see section 5.8.1).

Section 7.7.1 presents an overview of the module API. Detailed developer documentation is part of the reference implementation (see section 5.10).
5.1.3 Core (libgitk)

The main purpose of the core library is to provide one single component to the application developers that their application communicate with. The library uses the delegate design pattern to forward certain tasks to specific plugins (see section 5.1.4).

![Figure 5.5: The GITK core architecture](image)

Figure 5.5 illustrates the structure of the core together with the relation to the plugin modules. The Application programming interface (API) and the Plugin programming interface (PPI) provide the core functionality to each component group. Furthermore the core library provides several sub-systems that handle all internals of the GITK system:

- **Renderer plugin management**: Selection, load/unload and control of rendering plugins
- **Transformation control**: Set up the transformation pipeline, data synchronization and running the transformations
- **Profile management**: Manage a set of properties, which are set by sensors and consumed by adaptation methods
- **Context management**: Maintain a list of dialogs including their modality state
- **Event handling**: Consistent event handlers across toolkits and media
Error handling: Catching errors in dialog documents, handling wrong API usage

Logging and introspection: Help the application developer to track what is happening during application run-time and offer a mechanism for simulating adaptation scenarios

Introspection system

The behavior and functionality of the core library can be controlled at installation time (via configure) and at run-time (via command line options and environment variables). To aid application development, the library comes with an optional feature that allows looking into the system while it runs. For that it spawns a new thread at application startup. This thread runs an embedded web server (see (Hughes, 2002)) that provides a management console for the running application. Figure 5.6 shows what it looks like.

![Browser based management console](image)

Figure 5.6: Browser based management console

In this scenario one can use a web-browser and point it to the local web-server (it listens to a port that can be given as a startup-option). Requests to this server return
pages giving the running status of the application and offer several options to control. With these one can test the adaptation capabilities of the system. Among the options are language, style and the renderer itself. Furthermore the management console allows a look at the XML dialog data at both ends of the transformation pipeline. When one changes something in the application (moving the cursor or editing data), these changes can be verified in the XML. Figure 5.7 gives an example screenshot for this feature.
Figure 5.7: Management console showing current XML of a dialog
Transformer-plugins

A transformer plugin runs adaptation methods that are independent from the selected target domain. The adaptation methods are implemented as stylesheet transformations with XSLT (eXtensible Stylesheet Language Transformation). An alternative would be to operate on the DOM (Document Object Model) tree in the core-library. This option has not been chosen, as the adaptation logic would be hard to be modified or to be exchanged. Furthermore XSLT offers a powerful transformation language, that in the C programming language does not exists. Several of those transformations can be applied to one dialog description consecutively. The domain- and style-transformation steps will usually introduce domain dependent nodes into the dialog description object tree.

- **I18n**: Localize texts, which means to exchange placeholders with their language dependent expansions from a locale catalog. All text strings inside the system are in the Unicode UTF-8 encoding. (Unicode, Inc, 2003)
  
  GITK uses GNU (GNU is Not UNIX) gettext as the technical infrastructure. Each component - core, renderer and application - can provide independent language catalogs for various locales. (Haible, 2001)

- **Defaults**: Add default values (some of them such as text-orientation are not statically available). This makes sure that following transformers can expect certain elements to be there (such as disabled, required).

- **Domain**: Transform into domain specific GIML. This includes e.g. handling of domain specific dialog navigation.

- **Style**: Apply domain specific style transformations, such as colors, fonts, voices and pitch.

5.1.4 Renderer-plugins

In section 2.3.6 a description of an interface renderer was given. Rendering plugins are components as such. They perform all domain-specific adaptation methods to build a specific interface from abstract specification. Each renderer specializes in working with a few input and output devices (see table 5.1 for a few examples), as well as an interface toolkit that works in the chosen domain. Thus a renderer is a component specialized to produce an interface using the device capabilities of the platform it runs on (see section 3.2.2). The mapping operations to derive the specific interface from the abstract one are described in detail in later sections (see sections 5.5.3, 5.5.2 and 5.6.2).

To define these mappings, the renderer needs knowledge from the following fields:

- **Psychology and cognitive science**: What constrains the communication in a domain.
CHAPTER 5. THE GENERALIZED INTERFACE TOOLKIT

<table>
<thead>
<tr>
<th>Renderer</th>
<th>Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>text</td>
<td>Braille support, screen reader, hardware cursor, speech-output</td>
</tr>
<tr>
<td>gtk</td>
<td>shortcuts (toolbar, keyboard accelerators), themes</td>
</tr>
<tr>
<td>html</td>
<td>limitation to request-response model, themes</td>
</tr>
<tr>
<td>phone</td>
<td>navigation via touch tones, speech-output, display for video-phones</td>
</tr>
<tr>
<td>opengl</td>
<td>layout in 3D, visibility of objects</td>
</tr>
<tr>
<td>web-service</td>
<td>machine usable</td>
</tr>
</tbody>
</table>

Table 5.1: Examples of renderers and technological aspects they cover

- **Technical skills**: How to implement the mapping to the underlying toolkit.

For well understood domains like graphics and text, the standards mentioned in section 2.10 can give valuable recommendations for the renderer implementation. Hence one renderer encapsulates all the knowledge about adaptation in one target domain.

Each renderer is implemented as a shared object. This allows the core component to dynamically load and unload them at run-time. Additionally, this concept allows plugin development independently from application development. As a consequence, experts of the target domain can be included in renderer development teams. There they can contribute their domain knowledge to the design of adaptation methods. Application development teams usually face the problem of finding out which target domains to consider and what to take care of. Now they can concentrate on the logic part of the application and leave the real interface to the usability experts.

Section 7.7.2 presents an overview of the renderer-module API. Detailed developer documentation is part of the reference implementation (see section 5.10).

5.1.5 Interface toolkit

In sections 2.6 and 2.7 several interface toolkits have been introduced. Toolkits like these can be used by renderer plugins (see section 5.1.4) as a backend. That means that the renderer translates the dialog specification into a series of method calls to the backend toolkit library to construct the interface. At dialog run-time it also intercepts all events between the GITK system and the toolkit to abstract the toolkit specific event handling from the application.

5.2 Multi-dimensional adaptation

The problem of the multi-dimensional adaptation has been addressed in GITK by adding an extra layer to the application architecture (see figure 5.1) and the flexible combination of processing steps in there. Domain independent adaptations are carried out earlier in the processing pipeline by the transformer plugins (see section 5.1.3). Then a transition to the target domain follows and further processing is handed over to
the renderer plugin (see section 5.1.4), which finally produces (or renders) the interface.

Instead of using a limited set of hard-coded adaptations, in this project the decision has been made to add heuristics to the renderer plugins. Additional functions allow the software-engineer to provide hints for guiding the adaptation process. Each renderer exposes some presentation aspects to user profiles and provides defaults for them. Examples are colors and fonts for graphical interfaces. On the basis of the built-in heuristics and the optionally provided hints, the renderer will cast its decisions about remaining presentation aspects.

An interactive application has an event-loop. This one captures incoming events (mouse clicks, key presses, etc.) and provides them to the application so that it can react. When the adaptation profile changes at run-time, the event-loop needs to be paused. Then the change can be applied and the event-loop can be restarted. Depending on the type of change, different exit and re-entry points in the renderer are required. As an example, exchanging the renderer means to shut down the old renderer (with its main-loop) and initialise a new renderer. If only the language of the dialog changes, it is sufficient to change the text labels and adapt the layout. Generally it can be concluded that the adaptation feels more natural, the higher the granularity of the exit and re-entry points is. The downside of this is that it is very challenging from the technical point of view to achieve a high granularity. This is illustrated in figure 5.8.

![Diagram of event-loop granularity](image)

Figure 5.8: Granularity of the event-loop

### 5.3 GIML (Generalized Interface Markup Language)

The GIML language has been designed to support an abstract interface definition on one side as well as a continuous reification process towards the final user interface.

#### 5.3.1 Technical foundation

As already pointed out earlier, this solution utilizes a markup language to describe dialogs. Of course other representations of dialogs such as grammars are possible as
well (Antona et al., 2001).

The GIML is based on XML, so all features available to XML and related technologies are also available to GIML. This provides access to a vast amount of tools and techniques to work with GIML files. One such technique is XSL (eXtensible Stylesheet Language) using XSLT and XPath (XML Path Language). With the aid of those technologies one can address nodes and transform documents in a very flexible way. (Reuters, 2000; Thompson, 2003; Clark, 1999; Clark and DeRose, 1999)

Another advantage is the possibility to insert other XML based languages like SVG (Scalable Vector Graphics) and XHTML by using XML namespaces (Bray et al., 2002). GIML makes use of this possibility by e.g. using:

- The Dublin core standard to embed dialog meta-data in a reusable fashion (various contributors, 2003) and

- The i18n namespace for internationalization (Piroumian, 2002)

Further it is very helpful that dialog descriptions formulated in an XML language can easily be transformed into a different format with the techniques listed above. This makes it possible to model a dialog using UML (Unified Modelling Language), export this as XMI (XML metadata interchange) and use XSLT to generate a GIML dialog description from it. (Object Management Group, Inc, 2003a; Object Management Group, Inc, 2003b) The GIML file can be previewed with one of the supplied example applications (see section 7.6.6). Similar to this example of a reification import (import an abstract specification), an abstraction import can be done as well. This would import final user interface specification (e.g. XUL or Glade definitions) and generalize the domain specific elements.

Finally it is an advantage that XML conforms to the Unicode standard (see (Unicode, Inc, 2003)). This is crucial for supporting the various locales (special characters).

### 5.3.2 Language definition and validation

GIML is defined by a DTD (see section 7.1 for description of the elements). The reference implementation installs entries for the XML catalog resolver, so that GIML files can be validated locally (e.g. when designing them). If the DTD is not available locally it will be automatically fetched from the Internet if possible. DTD validation is used by the example applications (see section 7.6.6).

An initial XML schema definition is available as well. This needs better support in the used XML library (libxml2) before it can fully replace the DTD.

During the initial phase of the GITK project, other XML based languages have been reviewed. On the basis of this review a list of requirements and desired properties of the GIML has been collected.

1. **Low number of tags**: The language design is currently demand driven. The whole system is developed in an iterative process. Therefore the language has
only tags that are in use. It should not have tags that are rarely needed nor should it have multiple ways to express the same thing.

2. **Consistent naming**: Although this sounds like something obvious, it has not been taken care of in most of the examined XML languages. The GIML follows some simple rules:
   - Objects names are nouns
   - All language elements are lowercase
   - Referencing attributes are named in the form `objectname_referenced attribute` (e.g. `widget_id` refers to the `widget` with the given `id`)
   - Attributes should be easily readable (has_focus="true" instead of just focus="focus")

When designing a new language, one does not need to consider backwards compatibility. Therefore a relatively clean design can be achieved. It is unfortunate to see languages like UIML using inconsistent naming.

3. **Use of generalized tags with attributes**: When designing an XML language, one needs to balance between number of tags and their attributes. This balancing can be compared to the one in object-oriented design, where XML-tags map to classes and XML-attributes to class-members. The GIML will use generic tag names together with an attribute to name sub-classes, e.g. `<widget type="action">` and `<widget type="option-choice">` instead of `<action-widget>` and `<option-choice-widget>`.

4. **Separation of concerns**: A GIML document can hold all the various aspects of the interface: Structure, style, content and behavior. It is desirable to reflect the separation of interface aspects in the language design. XML offers namespaces to accommodate such need.

When the GITK project was started, the DTD technology was used to validate XML documents. Unfortunately DTD standards were designed in parallel, but independently from the XML namespace standards. Recently XML schema technology started to replace DTDs for validation purposes. That will allow the GIML to use namespaces in the future. (Obasanjo, 2002) provides good guidelines for designing a schema definition.

Other languages such as UIML separate the interface aspects by putting them into different XML sections. Unfortunately this design decision obscures the relation of the different aspects, as one needs to follow levels of indirections (id-references) to find peering elements in the document.

5. **Model logical relation of interface objects**: An XML document is inherently well suited to model relations through the hierarchical nature of the language. Therefore the hierarchy should be used to model containment-relations such as
label belongs-to widget and widgets are contained in a group. These relations are important for two tasks: the layout generation process and for providing a navigation scheme.

6. **No data validation constraints**: Some technologies such as XForms include form-validation in their specifications. Their rationale is clear - in a request-response environment with bandwidth restrictions, every saved transmission cycle is a gain. On the other hand formal validation is not always possible (validating against database contents). Further it is critical to rely on client validation. Therefore the application needs to revalidate anyway. For GITK the decision was made not to include validation constraints into the language yet. There are just a few that are considered. One is the maximum text length. This is helpful to calculate a sufficient width for a text entry box. Another example are min/max values for ranges. When a graphical widget should be used, the range must be known to construct the widget.

Further general guidelines about designing an XML based language can be found in the RFC 3470 (Hollenbeck et al., 2003).

### 5.3.3 Document processing

XSL stylesheets are used to transform GIML through several modular steps into a render specific IML (Interface Markup Language). This render specific IML contains a subset of GIML as well as domain specific markup (see figure 5.9). Thus we have an own markup language $IML_G \in XML$ and domain specific markup languages $IML_S \in XML$ where $IML_G \cap IML_S \neq \emptyset$. One piece of data that certainly belongs to the common part of all IML is the interface structure. Thus with the $IML_G$ one needs to be able to describe the interface structure.

![Figure 5.9: GIML and target IML](image-url)
5.3.4 Relation to other languages

In the past, other XML based markup languages for the description of user interfaces have been designed. Those known at the start of this project have in common that they are designed for a more specialized purpose. In the cases of XUL (see (Hyatt, 2000)), the language is tightly coupled to the graphical domain. UIML (see (Phanouriou, 2000a)) as another alternative relies on domain specific data to carry out the adaptation. All examined languages target user interfaces and not just interfaces.

As GIML describes an interface from the functional point of view, it is imaginable to produce final interfaces for any domain from it. This includes interfaces for non human users (an application that interacts with the GITK application through e.g. a generated webservice interface.

5.4 Solutions related to interface structure

5.4.1 Generalization versus specialization

In section 3.3.1 two general approaches have been introduced. This work aims to provide a generalized solution. That means the generated interfaces should run on a wide range of media. Therefore the interface with all the aspects needs to be transformed into various presentations. In chapter 3.3.1 the difficulties with providing alternative representation for media outside their native domains has been shown.

At this time the GITK approach concentrates on textual content. A possible future extension for the GITK implementation is to support annotations for media objects. Such descriptions could be used to give some hints in non-native environments. This approach is followed by e.g. the HTML language with the alt and title attributes for some tags. Some user interface toolkits have accessibility extensions, such as atk for gtk+, which allows annotation of interface objects.

5.4.2 Logical interface description

In the last chapter GIML has been introduced as the tool used in this solution to describe dialogs in a media-neutral fashion. To achieve the chosen goal it is crucial not to have tags (physical markup) for renderer specific attributes such as font faces and color, but instead semantic markup like emphasize or additional. This quality of the description is called media-neutral. It is the task of the interface generation process to add properties related to style and behavior. The error of mixing physical and semantic markup has been made in e.g. HTML. This language allows the writer to mark a part of the text with the tags <em> to express emphasis. This is usually rendered in italic letters. Alternatively one can mark up the same text directly with <i> for italic font style. The visual result will be the same. However in the latter case, a software cannot find out the reason for the text fragment being highlighted that way.
It is important to specify which components belong to one dialog, regardless whether they will be presented as one unit or split into sub-dialogs later. Beyond this the language needs to provide means to model component relations such as containment, equality and dependencies. The latter can be transformed into transitions between sub dialogs. As often demonstrated in the past, a RTN (Recursive Transition Network) is a good working model to describe the state transitions in an application. Every time one can group nodes in the RTN together and extract them as a subgraph, an independent dialog has been discovered. This set of nodes along with the transitions can be seen as an application within an application and will be handled as such in GITK.

Another important idea is the separation of dialog contents and dialog navigation. This allows the support of various dialog layouts as well as transparent addition of domain specific dialog navigation (for e.g. sub-dialog navigation). In the GITK system the text-renderer makes use of this by providing two action widgets to step back and forth through sub-pages of big widget-groups.

The GIML language has been designed with these requirements in mind. A full example GIML dialog definition can be found in figure 5.10.

Figure 5.10: The main structure of a GITK dialog description

One can see the hierarchical structure which is build by nesting \texttt{<widgetgroup>} tags and the \texttt{<widget>} elements inside. Tags like these can be found in many other interface markup languages like \texttt{<part>} in UIML. A widget group can be seen as a \textit{presentation unit}. This structural information is e.g. used to create boxes in the gtk-renderer, pages in the text-renderer and sections in the phone-renderer. Another aspect shown in this example is the separation of dialog and widget level. There are \texttt{<dialogwidgets>} describing a property of the dialog and below the \texttt{<widgetgroup>} tags describing what is inside the dialog.
CHAPTER 5. THE GENERALIZED INTERFACE TOOLKIT

5.5 Solutions related to interface style

5.5.1 Domain mapping

As shown in section 5.1, GITK uses separate rendering plugins to generate and handle the requested interface. Each plugin is specialized to produce adequate results for a specific combination of input and output devices.

But how is the desired plugin selected? At first every application needs to be started. This is done by an application launcher (see section 5.8.2). The launcher itself knows the environment it runs in. It could pass this knowledge to the application, so that this will use the same modality. The GITK launcher, which will be described later, provides this behavior. Furthermore the platform in use often limits the available plugins. As an example a telephony based platform would only support one renderer for exactly this environment. The situation is more difficult in a normal computer system. Such systems usually support various modalities. Until now this was not a problem, as interfaces were not using alternative modalities. As an example audio-output was only used to play back music and not to operate a program. In contrast GITK enables the available modalities to be used for interfaces. Therefore GITK needs a mechanism to select the desired modality. In the reference implementation the following sequence is used for that:

- Command line arguments (see section 7.6.2)
- Environment variable (see section 7.6.3)
- Default renderer: marked in the filesystem by a symbolic link
- Available renderers and environment: a DISPLAY environment variable shows that a graphical environment is available

Another question is how the dialog data is being transformed to be interpreted by the chosen renderer? Certain aspects for each domain can be automatically adapted (e.g. dialog navigation as described in section 3.5.1). For the transformation to the media domain of the renderer, each renderer installs an own stylesheet (for details please refer to section 7.7.3). This stylesheet is the last one in the transformation pipeline that is applied to the dialog data.

5.5.2 Presentation mapping

When presenting an interface to the interaction partner, GITK has several degrees of freedom to carry this out. Possible variations when mapping between several types of presentation are:

- Overall style: Display as dialog or wizard (assistant)
- Verbosity: Display additional information along with labels or just as tool tips
Apart from this, generating a presentation of an interface has a deeper meaning. The dialog layout needs to be set up, dialog navigation must be added and the contained interface objects have to be decorated.

Again GITK uses XSL stylesheets to perform the mappings. The user and environment profile need to supply data to allow to select a mapping. This choice needs to be passed to the stylesheet.

Regarding the user account administration example (see section 3.1.1) a different XSL stylesheet would be used for first-time users for the inter-dialog layout to show form data in groups together with guiding help.

**Layouting interface objects**

The basic concept is to traverse the source-tree and pack the interface objects into boxes/containers on alternating axis/dimensions (depending on the number of dimensions the renderer supports). This basic algorithm is the same for each renderer, besides that every renderer comes with its own set of constraints (supported dimensions, available space) to control the outcome. It is quite obvious that this algorithm is not capable of generating every imaginable layout. One limitation is that it is not possible to control which dimensional axis is chosen next.

- **1 dimension:**
  - Text: If using paging to split the interface presentation, where to insert page breaks (if possible on group boundaries). An alternative is to use scrolling.
  - Audio: Same paging problem as in the text case. Furthermore the size of a presented page needs to be kept small. This is required to ensure that the user can remember the items it contains!

- **2 dimensions:**
  - Graphics: This case is like existing UI-toolkits. When there are too many entries to fit the screen, scrolling is usually used. Additionally techniques like paging (by using notebook tab pages) can be applied.

- **3 dimensions:**
- **Graphics**: This dimensionality may need to deal with the effect that interface objects hide others. When there are too many items to fit the screen, 3 dimensional interfaces can use zooming.

- **4 and more dimensions**:
  - **Animated graphics**: While interfaces as such can be generated, the author could not come up with useful scenarios for them.

Figure 5.11 shows a stylesheet that just flattens the whole widgetgroup hierarchy for the serial case.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0" ...>
  <xsl:template match="/giml:widgetgroup">
    <!-- skip empty groups -->
    <xsl:if test="count(/giml:widget)=0">
      <!-- copy whole group -->
      <widgetgroup>
        <xsl:copy-of select="@*"/>
        <xsl:copy-of select="/giml:label"/>
        <xsl:copy-of select="/giml:widget"/>
        <xsl:apply-templates select="/giml:label[0]"/>
      </widgetgroup>
    </xsl:if>
    </xsl:template>
</xsl:stylesheet>
```

**Decorating interface objects**

Apart from distributing the interface object over the presentation space, the elements need to be decorated (see section 3.4.2). Like in all other processing steps, this involves a XSL stylesheet to add domain specific markup to the document tree. The additional data contains presentational elements and attributes. This can be as simple as setting a presentation theme, which is offered by the underlying interface-toolkit (see figure 5.12). In other cases the same effect can be achieved by directly adding attributes with presentation parameters to the interface objects (see figure 5.13).

The current version of the styles only allows the renderer to add styling information. If it turns out that this is insufficient, the system can be easily extended. As an example, profiles that are included by the stylesheets would allow the user and the application to guide the styling behavior. In the case of the user, the system would store profiles for each renderer, remembering the user preferences. Currently the order of stylesheet application is relatively simple:
• **Global from renderer**: Each renderer can support theming for all applications

• **User specific for renderer**: Here the user can override style settings for each renderer

Adding application dependent styling would complicate the scenario, as an application ideally should not need to do anything specific to a given renderer. The effect that following such an approach has, can be seen in difficulties the UIML (see section 4.1.1) project has with mapping to different modalities. One way to relax the complexity would be to make styling preferences that are provided by the application dependent upon the dimensionality of the target renderer. This is based on the fact that there will be multiple renderers for each dimensionality (adaptation for telephone usage has similar constraints as adaptation for blind people and usage with a Braille-keyboard).

### 5.5.3 Interface object mapping

GIML uses a super-set of interface objects, which form a generalized dictionary of domain-independent interface objects. Their naming is strictly decoupled from their presentation. Table 5.2 provides a few examples of commonly used names and names that GIML uses. One can see that the common names are often inspired by their graphical representation. They are not meaningful anymore when used in e.g. a pure audio based context. Details about the domain independent naming scheme are discussed in the next chapter.

<table>
<thead>
<tr>
<th>common name</th>
<th>canonical name</th>
</tr>
</thead>
<tbody>
<tr>
<td>push-button</td>
<td>action</td>
</tr>
<tr>
<td>scroll-bar</td>
<td>value choice/range/limited</td>
</tr>
<tr>
<td>check-box</td>
<td>option choice/single/boolean/compact</td>
</tr>
<tr>
<td>radio-button</td>
<td>option choice/single/multiple/expanded</td>
</tr>
</tbody>
</table>

Table 5.2: Examples for canonical interface object naming
After the interface object types have been mapped, another set of stylesheets will be executed to add domain-specific default nodes and to apply user-definable attributes. A simple example of such a stylesheet is shown in figure 5.14.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0">
  <xsl:output method="xml" doctype-system="http://gitk.sourceforge.net/giml.dtd"/>
  <xsl:template match="giml:meta">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:if test="count(dc:title)=0">
        <dc:title>unnamed</dc:title>
      </xsl:if>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

Figure 5.14: Part of a stylesheet that adds a default title if missing

### 5.5.4 Interface object hierarchy

One of the preparing tasks for this project was to compare the interface object sets (see section 2.3.3) provided by interface toolkits like those listed in section 2.6. Among the results of this investigation was the finding that the interface objects which the toolkits have in common can be put into a hierarchy (see figure 5.15). The depth of this tree captures the interface object specialization. Important is the fact that the taxonomy is about the main purpose of the objects. In other words it defines a major purpose for each interaction object and uses this for classification. One justification for this approach is that using such an object in an uncommon way is confusing for the user. Imagine a real life light switch that looks like a toggle, but needs to be pressed.

The definition of the main purpose of a widget can be illustrated with the categorisation of a list widget. The reason that a list might have an embedded scroll-bar is due to constraints of visualization (having more entries than it can show). Although that does not affect the placement of the list-widget in the hierarchy. The scrollbar is only technique to presents the widget, it does not change the major purpose of it.

The taxonomy starts with a subdivision of all interface objects into three main categories. This classification step is based on the main purpose of the interface object: to display information, to enter or edit information and to invoke actions. The groups with edit and display interface objects has been split further. Table 5.3 gives the semantics of the hierarchy node names.
Some organizational attributes such as editable/non-editable appear more than once in the hierarchy, so that other similar hierarchies are possible.

Generally this solution follows the concept of including the widest set of features, which means that renderers have to emulate or ignore missing features (by reducing specialization). They will walk up the tree until they have an implementation for the interface object. With this approach further refinement of the tree and more complete coverage of the tree by renderers can lead to more appropriate interfaces. The contrary approach of only using a common set of features is not suitable for the future and inappropriate for a solution with an unrestricted number of target implementations (which GITK is). Adding a new widget would require the updating of all existing renderers.

While working on the hierarchy, several questions and problems appeared:

1. A first problem is to decide if something is an attribute of an object or if the object should be split in sub-categories. As an example, the length of a text-entry clearly is an attribute of the text-entry, but the maximum number of options \( n \) for an option-choice is not. In the case of \( n = 2 \) the option-choice would become a boolean choice like a checkbox, and in the case of \( n > 2 \) it should become something like a list or a set of radio-buttons.

2. A related problem is how to handle compound interface objects. A scrollable list is an option-choice (pick one entry from the list), but has a value-choice interface
object attached that allows the picking of a numerical value (the list offset). In this case the scrolling could be regarded as presentation specific (if the screen would be big enough, scrolling would not be needed), but there might be other cases.

3. The next problem is which level a renderer is required to implement. For very
simple interfaces it seems to be sufficient to implement the second level (action, data entry and data display). This produces a working interface, whose drawbacks include not being comfortable to use and not being error safe (e.g. option-choices must be entered numerically and cannot be selected from a list).

4. Finally, structured interface objects (like trees, and tables) are problematic to handle. Their purpose is to present structured data. They use low-level interface objects (labels, text entries) to display the data itself. A suggested solution is to handle them by using nested `<widgetgroup>` tags together with special roles hinting how the widget groups should be presented.

So how is this hierarchy used? When a dialog is defined, every interface object is assigned a type (`<widget id="Message" type="label"/>`). This type points to one of the nodes in the hierarchy. Later during concrete interface generation, each renderer needs to map the abstract component type to an available component. Several scenarios exists for this case:

- **One-to-one**: The action type can often be mapped directly to e.g. a button.
- **Complex-to-simple**: A spin-button (number entry with increment and decrement buttons) is mapped to a simple number entry widget (e.g. for voice input).
- **Complex-to-compound**: A list type is mapped to a menu with navigation widgets (e.g. for serialized text or voice output).

Often several possible mappings are available, therefore the mapping to use needs to be made part of the adaptation profile. For that reason these preferences are embedded into the various transforming stylesheets.

XSLT templates are again used to transform a widget node into an GIML-block with low-level interface objects the renderer knows about. The decision regarding which presentation is preferred has been left to the user, or at least to the style-sheet configurator.

As a concluding note, this hierarchy is still work in progress. Many other projects use widget type mapping tables or similar approaches for adaptation. But often these mappings are not handled as an important component of the overall system.

### 5.5.5 Interface objects in GTK+

The GTK+ includes a rich set of widgets. Currently the GITK renderer uses only a few of them. This is mostly because the toolkits behind other renderers such as the one used by the text-mode renderer are currently not very capable (see section 5.5.6). After all the aim of the reference implementation is to provide some demonstrations that work well with several renderers.

The object oriented nature and the rich introspection capabilities of the GObject framework make it easy to add mappings for new widgets.
5.5.6 Interface objects in text mode

The text mode renderer uses a commandline shell to produce a non-graphical text and speech based interface. It is based on its own (incomplete) toolkit, as the author was not able to find a usable toolkit. This toolkit utilizes the Curses library for terminal output. It currently only implements a subset of the interaction objects classified in the hierarchy introduced in section 5.5.4. Available are actions, various character entries, boolean option-choices and $1:n$ option-choices.

To extend this limited set of features, it would be sensible to make a separate package out of the toolkit first. After that the renderer can use this toolkit.

5.5.7 Interface objects in a 3D environment

This renderer is based on the qb (pronounce: cube) interface library written by Sebastian Paul. (Paul, 2004) The qb toolkit uses the Coin3D library which is an open source implementation, compatible to the SGI Open Inventor 2.1 standard. (Systems in Motion AS, 2004)

The provided interface objects are similar to those from 2d interface toolkits like GTK+, but interaction objects are easier to recognize as they have real depth. Currently the library provides about the same amount of widgets as the text-mode renderer. The toolkit has a modular design. It uses external open inventor models for the appearance of the interaction objects. Therefore a totally different look can be achieved by exchanging a whole set of models. This can be done at run-time by choosing a different style.

5.5.8 Interface objects in phone mode

The phone renderer uses the phone interface library by Steffen Ast. (Ast, 2004) This toolkit too is currently restricted to simple interface objects such as text-input and option-choices. Similar to the text-mode renderer, this library should be extracted from the renderer and then extended with more interface object implementations in order to become really useful.

The specialty of this renderer is that the toolkit normally uses the CAPI (Common ISDN Application Programming Interface) for input-output operations. Therefore one can operate an application by listening to the speech output from the phone speaker and enter data by using the telephone keypad.

The renderer has a try-out-mode, which uses the sound-output from the computer and a subset of the computer keyboard keys. This mode simulates the very restricted interface capabilities, without the hardware requirements to run the CAPI.
5.6 Solutions related to interface behavior

In the previous section the various ways of generating a presentation of an interface have been described. Likewise behavior can be made *themeable*. That means a renderer could offer several interaction styles (see section 3.5 for a list of interaction styles). At run-time an applicable style that fits the requirements would be chosen. A renderer will usually restrict the selection of interaction styles, as each renderer is specialized in a domain and this makes some styles inappropriate (e.g. direct manipulation in a phone-interface).

A first design decision in GITK was to separate interface content from dialog navigation. The dialog navigation is not directly related to the task the user performs. It supports the user and handles limitations of the presentation in use. A similar design can be found in VoiceXML, which has two dialog types *forms* and *menus*. Menus handle navigation and forms data input and result presentation. This separation allows the generalization of the navigational behavior for both, inter and intra dialog navigation.

Another factor that influences the behavior is the application logic. The GITK uses an event callback model to link application logic to the interaction. The application can register handlers on events that can occur for parts of the interface. Parts of the interface are dialogs and widgets. Table 5.4 lists the currently implemented events.

<table>
<thead>
<tr>
<th>Component</th>
<th>Events</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dialog</td>
<td>On-close</td>
<td>When a dialog gets finished</td>
</tr>
<tr>
<td></td>
<td>On-show</td>
<td>When entering a dialog</td>
</tr>
<tr>
<td></td>
<td>On-hide</td>
<td>When leaving a dialog</td>
</tr>
<tr>
<td>Widget</td>
<td>On-invoked</td>
<td>When a action widget has been activated</td>
</tr>
<tr>
<td></td>
<td>On-changed</td>
<td>When an entry widget has been edited</td>
</tr>
</tbody>
</table>

Table 5.4: Supported events for interface components

5.6.1 Dialog navigation

Depending on the dimensionality of the target domain, one can think of several possible algorithms to solve the dialog navigation problem. A GITK application simply lists the requested widgets for committing changes in a separate section. This section can only appear once in every dialog. An example specification fragment can be seen in figure 5.16. When the dialog gets layouted, the renderer adds these widgets plus extra widgets if they are needed for navigation. Possible dialog-widgets an application can request are:

- **Cancel**: Ignore the changes (revert) and end the dialog
- **Okay**: Apply the changes and end the dialog
- **Try,Apply**: Apply the changes
CHAPTER 5. THE GENERALIZED INTERFACE TOOLKIT

Figure 5.16: Example of specifying dialog widgets

- **Revert**: Restore settings as of before opening the dialog

The placement and decoration of these elements are free choices for the renderer. That allows the implementation of multiple dialog commission layouts and the choice of one at run-time, depending on the users preference or even on the operating system version (to match the vendors style guides). Earlier in section 3.4.2 figure 3.10 was illustrating two example dialog layout presets. Each preset is using a different placement of dialog navigation widgets.

What follows are details regarding navigation in interfaces with one-, two- and three dimensional presentation.

**One-dimensional**

In section 3.5.1 two variants have been introduced:

- **Linear**: Using commands such as previous and next
- **Hierarchical**: Using commands such as previous, next and up

The text and phone renderers use the linear scheme. Its domain.xsl flattens the widget-group hierarchy to a sequence. An example of this process is shown in figure 5.17. The implementation recursively processes the interface object hierarchy and moves child groups behind their parent groups, so that they are on the same level.

Figure 5.17: Example of flattened dialog hierarchy
Two-dimensions

Renderers for the graphical domain can use the following partitioning techniques to handle dialogs that will not fit on the screen or that should be segmented for usability reasons:

- **Sequential**: Using dialog pages like a notebook with tabs or a wizard with previous and next buttons
- **Spacious**: Using a view port with scrollbars

Figure 5.18 gives two examples for sequential segmentation.

![Sequential Segmentation Examples](image)

Three-dimensions

Using three dimensions for interfaces provides a lot of space for the interface. This makes it quite challenging to provide good, logical orientation. Figure 5.19 gives two examples for 3d user interfaces, looking glass (various contributors, 2004a) and open croquet (Kay et al., 2004). Both use 3d objects for navigation and both can embed existing 2d applications into their environment. (Paul, 2004) uses the additional third dimension in the interface like it is used in the real world, i.e. interface objects for interaction have depth. The user can easily see what is just a decoration and what is there for interaction.

An alternative approach for 3D navigation is Zoomworld (see (Raskin, 2001)[section 6.2]). The idea behind Zoomworld is to zoom into the menu levels. In the beginning it gives a good overview. Starting there users select the region they are interested in. This region is enlarged and shows more detail. A GITK renderer could use this technique to zoom into the widget-group hierarchy.
5.6.2 Interaction serialization

This problem is highly related to the layout. In a usable interface the interaction sequence should match the layout of the interactions objects. It would be highly confusing if the focus of interaction would jump back and forth within the dialog.

The layout of a dialog differs a lot depending on the dimensionality of the interface presentation. So does the interaction.

One-dimension

In a one dimensional interface, interaction is strictly serialized into a sequence in time. Backward changes in the interface as discussed in section 3.5.2 need to be avoided. This can be done by rearranging the order of interface objects.

To serialize the interface the GITK system paginates a dialog into presentation units. This technique aids browsing through the dialog and choosing the data item to edit. Presentation units are small parts of the whole dialog that fit the presentation space (screen space or amount of items that can be remembered by the user). A first pagination hint are the widgetgroup tags. Further subdivisions are done on the basis of device capabilities (screen space) and mental contraints of the user (items to remember).

By segmenting groups into a set of pages, the interaction path problem described in section 3.5.2 can be addressed. The user can pick the item to activate from the group and then proceed with the dialog.

Two and more dimensions

In multi-dimensional interfaces several interface objects can be presented side by side. This for example allows the renderer to flatten the interface hierarchy graphically. Where in sequential interfaces one has to provide a strictly linear interaction path, one can present alternatives in parallel when using multi-dimensional interfaces. Thus
the user decides which path to take. An example is shown in figure 5.20, where the user can either enter the path name with the keyboard into the text-entry box or choose it from a directory browser that is invoked by the "choose" button.

![Figure 5.20: Two dimensional presentation of alternative interaction paths](image)

Renderers that offer such interfaces (e.g. gtk+ and opengl) should make the general interaction sequence follow the reading order (e.g. left-to-right and top-to-bottom for Latin language cultures).

### 5.6.3 Multiple interfaces

At the current state of the project this issue cannot be specifically addressed yet. Therefore what follows is merely a concept that still needs an implementation.

The primary decision that needs to be made is about how the instances communicate with each other. Possible solutions are a peer-to-peer or a server based approach. The peer-to-peer approach has the advantage that still no server is needed, but requires some rendezvous-mechanism. Besides an easy lookup of the partners, using a server provides additional benefits such as keeping a session open. Apart from that, mixed mode architectures are possible, like using a directory server for looking up sessions and then using direct communication. A solution in this field should definitely reuse the experiences that have been gathered in the field of web services and Internet messaging.

On top of that a focus interchange mechanism can be built. One way to implement this would be to use an exclusive write locking algorithm (like using g_mutex from glib). GITK already uses such locks to secure dialog access in multi threaded environments. In the case of multiple interface instances, communication partners would raise their hands to gain the right to interact. Such a request would be queued and whenever the one who currently has the interaction rights releases it, the partner who is first in the queue will receive it. This model is similar to a group discussion scenario. An alternative would be using a human moderator who could reschedule the queue.

### 5.7 Solutions related to interface content

#### 5.7.1 Content selection

In chapter 3.6.1, the difficulty of a clean definition of content regarding interfaces has been brought up. Textual information was identified as content that can be easily
exchanged in difficult presentations of an interface. The GITK system utilizes unified message catalogs that are based on GNU gettext for this purpose. Here the application translator does not need to work with separate catalogs for text messages used in the application logic and in the interface part.

An overview of the files and tools that take part in the gettext build cycle are shown in figure 5.21. (Haible, 2001) Figure 5.22 shows the separation into 2 steps, build time and compile time, and an example snippet for an XML dialog description. One can see that it works for XML files similar to e.g. c-sources.

The build-time step is covered by the gettext tool chain. The run-time processing is handled by custom XSLT functions that the core library of the GITK system registers with the XSL processor. This results in a uniform integration of the translation step into the general XSLT based processing architecture of the GITK approach.

### 5.7.2 State persistence

The state of the dialog is captured in the core-library, not in the renderer, nor in the application. This is an important design decision as only this way the requirements of e.g. ubiquitous computing can be met. This design allows the exchange of the rendering plugin at run-time, when radical environmental changes require switching to a different modality.

This feature is realized by enforcing these rules:
• Always write the data the user has entered back to the model.
• Always record focus state changes.

The update and synchronisation behavior is illustrated in figure 5.23. The techni-
pipeline. When the application changes something in the interface, forward propagation makes this change available to the current renderer. Analogous to that backward propagation is used to synchronise the source document with the target document, when the renderer applies changes such as user input. Following these guidelines, one can safely change styles of an interface or even exchange rendering plugins while the application runs as shown in figure 5.8. As the dialog transformation pipeline is bidirectional, both the original, media neutral dialog representation and the domain dependent dialog descriptions exist and are kept synchronized during dialog run-time. Therefore switching the renderer means,

1. Shutting down the old renderer
2. Constructing a new pipeline for the new renderer
3. Activating the new renderer, and
4. Continuing the dialog session

5.8 Technical solutions

5.8.1 Language bindings

To create language bindings (or wrappers), GITK uses swig (simplified wrapper and interface generator). (et al., 2004) The GITK package comes with a specially formatted API definition. From this file the swig-tool basically auto-generates the language bindings for several possible programming languages. The biggest difficulty with this approach is that the core developer needs to supply type mappings between the C language types and target language types.

5.8.2 Operating System integration

GITK provides its services in a system neutral way. This makes it easier to write portable applications. On the other hand GITK relies on operating systems e.g. to access input and output devices. As today’s operating systems are not fully aware of multi-modal interfaces, GITK needs to provide services which are missing on supported platforms.

While designing and implementing the reference implementation, the services GITK depends on have been carefully chosen, so that as many as possible are operating system independent (e.g. XML handling, xsl processing and i18n).

User authentication

As GITK allows the use of applications over all kinds of media, it is possible to use a system without e.g. visual outputs, but with audio outputs instead. As described in
most systems requires a proof of identity before a user can start to work on the system. Unfortunately those native user authentication mechanisms of common operation systems do not support other media. Therefore the GITK package should supply a respective login mechanism, e.g. in the form of a pluggable authentication module (PAM). This has not been done yet due to time constraints.

**Application launcher**

The same problem as with running the user login dialog applies to launching an application. Unfortunately the native application launching mechanisms of common operation systems do not support other media. Therefore the GITK package already comes with an enhanced, media aware application launcher (see section 7.6.6).

This launcher gains its flexibility from using GITK for its interface as well. On systems using it, the tool will permanently run and therefore replace or complement the original launcher. Another benefit is that the GITK library comes with logic for choosing the appropriate modality based on what is applicable and available (see section 5.5.1) on the platform at run-time.

### 5.9 Development support

Chapter 3.8 lists fields related to software development that should be supported by a new approach. Good tool support is mandatory for developers to work efficiently.

New technology often introduces new programming languages and file-formats. The GITK approach does not. Developers choose from the supported programming languages one they are most comfortable with. The dialog descriptions are formulated in an XML language. Most editors and IDEs have XML support built in or available in the form of a plugin. Transformations use XSL and again most editors do support editing and executing XSL transformations. Finally the i18n adaptation method of GITK is using gettext and therefore does not introduce new files or workflow steps here.

When software does not work as expected, it needs to be debugged. Therefore developers need help to allow them to look into the application at run-time. GITK provides a built-in web server, which can be used to do that. The developer can change parameters controlling the adaptation process and immediately sense the results. Apart from that the toolkit does configurable logging, which helps to spot problems.

GITK offers connectivity to other XML based solutions via XSL stylesheets, which could convert dialog descriptions into the GIML language. There are two kinds of conversion scripts:

- **Abstract interface specifications**: conversion by using reification
- **Final interface specifications**: conversion by using abstraction

These scripts can serve two purposes:
1. **Support the developer with migration**: In this case the tools try to convert as much as possible from the old interface description into the new GIML based description.

2. **Use for rapid prototyping**: If an interface design tool exists for a supported toolkit, then this tool can be used for prototypes and the resulting interface description can be converted into a GIML based interface description.

More support for **rapid application development** comes in the form of the GITK dialog viewer, which can run a dialog in all supported modalities (see section 7.6.6).

### 5.10 Reference implementation

The concepts and ideas previously described have been implemented in an open-source project. The sources of the GITK reference implementation can be obtained from [http://gitk.sourceforge.net](http://gitk.sourceforge.net) for free. Apart from the file-download, the site offers other useful resources, such as news, mailing lists, issue trackers and more. GITK is free software and distributed under the **GNU GPL** (GNU Public License). Other licences are available on demand. (Kost, 2005)

The site is especially recommended for those who like to stay up-to-date with the ongoing development of the reference implementation.

The provided software demonstrates the solution described in the work and invites to experiment. Further it aims to provide a well engineered foundation by using modern software development techniques, such as unit-tests (**check**), code inspections (**splint** and **flawfinder**) and run-time inspection/profiling (**valgrind**).

The software has been developed and tested on Solaris, Linux and Windows. It should also run on most UNIX platforms though. Developers can find more information on technical details, API overview and more in section 7.7.
Chapter 6

Conclusion

In the beginning of this work a number of aims for this project (see section 1.2) had been set. Here they are again in a summarized form:

1. Universality by concentration on less complex interfaces
2. No fixation to graphical user interfaces, real domain independence
3. Clean separation of interface aspects through the architectural model
4. Automatic adaptation for many kinds of applications
5. Flexible architecture, providing continuous adaptation
6. Reuse of existing technology
7. Provide adaptivity to human and machine users
8. Dynamically generate the interface as the adaptation requirements change
9. Adaptivity is needed by everyone. It does not deal with minorities only.
10. Openness for new developments

So which of these have been reached with the development of the GITK architecture?

6.1 Result

The chapter 2 began with a definition of terms and structuring of knowledge in relation to this work. Special attention was given to terms like adaptation and interface. Based on the definitions a new adaptation infrastructure model has been created. Important about this model is that it reveals the complexity of the adaption process. Not only the user has influence on the outcome of the adaptation, but the environment and the hard- and software the application runs on also influences the result. The comparision to
other models in the same chapter showed that this aspect had been ignored for a long
time in models describing interface architectures. To summarize, this chapter gave the
outline of what is needed to provide "Dynamically generated multi-modal application
interfaces”.

The next chapter focused on the problem details. It showed the multi-dimensionality
of the adaptation problem. Further sections described details of mapping problems
that occur when mediating between the various modalities of an interface. The chapter
showed that the problems can be grouped by the aspects of an interface they are related
to (structure, style, behavior and content). It also introduced a separation of adapta-
tion on the dialog and on the interaction object level. Finally the problem analysis
suggested an order in which to approach and solve the problems.

Chapter 4 introduced and evaluated already known approaches and solutions. The
chapter concludes that it is required to separate into solutions that provide adapta-
tion methods and those which provide an adaptation infrastructure. The evaluation
showed that none of the abstract toolkits is general enough for the purpose of serv-
ing as an adaptation infrastructure. However many of the projects examined describe
interesting techniques that can be used as adaptation methods.

On the basis of the previous findings, chapter 5 introduces and describes the GITK
approach to the multi-dimensional adaptation problem. Therewith this work bridges
the gap between the adaptation infrastructure interaction model and the concrete ar-
chitectural model the GITK uses. The example applications included in the software
package developed during this work give a good overview of capabilities of the GITK
reference implementation. They prove that the GITK infrastructure provides an adapt-
able system and in a few aspects even an adaptive one. The variety of already possi-
ble adaptations shows that GITK approach is usable for a wide range of applications.
These application targets have in common that they do not require the presentation
of complex data. On the contrary, other application domains need further work, such
as ones that are very tightly coupled to one or more types of media (e.g. a graphics-
editor needs visual output). Some of these applications might be enabled for GITK if
additional adaptation methods are made available to the GITK infrastructure.

Inspecting the introduced architecture one sees that the components of the adap-
tation process have been clearly separated and that the design of the transformation
pipeline provides the wanted mechanism for continuous adaptation. To summarize,
one can see that the goals regarding the design of a generalized adaptation infrastruc-
ture (aims 1 to 3) have been successfully reached.

The GITK solution is based on established technologies (e.g. thoroughly use of
XML technologies, integration with a standard i18n framework). Therefore aim num-
ber 4 has been met as well.

Finally aim number 5 has been met as GITK is published as open source and invites
participation through its open architecture.

As motivated in the beginning of this work, there is a big need for a solution such
as the one developed in this work. The importance of such a solution for the future,
suggests that an accepted technology can quickly become widely used. Although the
software currently more or less has the state of a proof-of-concept, care has been taken that the software complies with a high coding quality, by using unit-tests and source-code analysis tools. Development of the software will be continued beyond this work.

6.2 Impact of the work

A change of interface technology causes an impact on users and interface developers. In the current state the project will not yet have a noticeable impact on the user. Right now the project primarily supplies the adaptation infrastructure and not adaptation methods. However adaptation methods are the components which perform the effective changes of the interface. The included adaptation methods are only a starting point and need contributions by experts in the respective field. Therefore a usability study would not be the right approach to measure the results of this project.

In contrast to the user, software development is impacted by this work. The separation of the interface aspects (see section 2.3.1) allows specialists to perform the respective parts of design, implementation and testing.

6.3 Comparison to other approaches

In relation to the chosen range of target applications, the described approach is not too simplistic and not overcomplicated either. Below one can see a few examples of what can be considered as disadvantages of other approaches regarding the aims of this work.

- **Too specific:**
  - **XForms**: Concentrates on web-applications
  - **XUL**: Focuses on graphical presentations
  - **VoiceXML**: Focuses on audio presentations

- **Too complicated:**
  - **UIML**, **XForms**: Need application specific adaptation profiles for each target domain

- **Redundant solutions:**
  - **UIML**: Own techniques for i18n, profile application and document modularisation

One point where most approaches are superior to the GITK implementation is the quality and completeness of the presentation layer. The wide range of supported modalities in the GITK project is quite challenging to the amount of implementation effort needed
CHAPTER 6. CONCLUSION

to compensate for that. However once this work has been done, the efforts are paid back by better reusability.

Further advantages of GITK are:

- Transparency, as it is open source
- Active development

6.4 Future

Development of this concept and the GITK-software has not been finished. The project is open source and will be continued. Everyone is invited to participate. The home-page at http://gitk.sourceforge.net shows the current status and is the source for future results. (Kost, 2005)

The next sub-sections list fields recommended for future work. Integrating the technologies introduced there into the GITK infrastructure would enable it for more kinds of applications.

6.4.1 Standardisation

Chapter 4 has shown that several projects contribute to the field of adaptive interfaces. The personal opinion of the author is that these projects need to collaborate more. If techniques emerging from these projects are to be accepted by users and software developers, they need standardisation. One step towards standardisation would be to adjust the interface description language the projects use. If the projects would use a common interface object naming scheme and a common interface object hierarchy (see section 5.5.4), interoperability would be easier to achieve.

6.4.2 Adaptation profiles

To turn adaptable applications into adaptive systems, user profiles and environment profiles are needed. User modelling and the ongoing technical development contribute to that. Further on future devices will have more sensors to read from the environment. Integration of profiles could be done of the basis of CC/PP (Composite Capability/Preference Profiles) (Klyne et al., 2004). Questions about the storage location of the static parts of these profiles are still open.

6.4.3 Interface presentation

Although the current version of the GITK solution can already produce a variety of interfaces, the quality of them needs improvement. This is important for the solution to be used in real life.
One way to improve the quality of the interface presentation is to implement more and better adaptation methods into the rendering components. This needs the cooperation with experts in the respective media domains.

Another option to explore is to look into presentation patterns. The idea here is to give a role to a group of interface objects that describes their purpose (e.g. "media control"). If these roles come from a defined vocabulary, rendering components could apply domain specific presentational transformations to produce interfaces that look and feel as expected. (Nichols et al., 2004)

Finally it would be interesting to implement more renderers. Implementations have been sketched out for e.g. a HTML-renderer that would provide interfaces that can be used remotely through a web-browser.

### 6.4.4 Reference implementation

The current reference implementation is able to show that the concepts developed in this work are not just ideas; the functionality has been implemented and can be demonstrated. While working on this project, new requirements have been found and many ideas have been gathered. Putting all of this into the existing software will be difficult.

Therefore some refactoring would be a good candidate for further work on the software. Now that the code has already undergone a few revisions and therewith the desired structure has become more clear, the refactoring should make the code more object oriented (e.g. by using GObject as a paradigm for object oriented development in C).

Another target of refactoring would be to extract the toolkits from most of the renderers. Only the GTK+ based renderer connects to an existing toolkit. For the other renderers the project ships its own toolkits. Extracting these from the renderer would allow the development of the toolkits independently from the renderer.

### 6.4.5 Real life evaluation

To demonstrate current and future capabilities, more and especially complexer examples are needed. An integration into an existing system would be interesting to allow real life testing. A good candidate could be a PIM (personal information management) application.
This chapter contains a collection of resources for reference. It starts with the GIML language definition and an example GITK application along with the used GIML dialog description.

That is followed by some examples for different dialog descriptions. These are referenced from the chapters where the respecting language has been discussed.

Next comes information about the GITK software in general and information for the user and developers.

7.1 GIML DTD

```xml
<!DOCTYPE giml SYSTEM "http://gitk.sourceforge.net/giml.dtd">
```

The GIML language design follows some simple rules:

- # when another element is referenced by an attribute, the attribute is named tagname-attribute, e.g. widget-id
- all tag/attribute names are lowercase
- element tag names are nouns
- tag/attribute names and contents should be readable (has_focus=true, is_disabled=false)
- @ingroup gitkcore

```xml
<!DOCTYPE % widgettype action>
```
### 7.2 Simple example c-source code

```c
/* Std: gitkHelloUser c.v 1.49 2005/01/04 16:24:28 ensonic Exp $*/

/* @file  gitkHelloUser.c */

/* @author Stefan Kost <ensonic@users.sf.net> */
CHAPTER 7. APPENDIX

/* @date Fri Jun 29 16:43:27 2001 */
/* @brief first simple test for gitk. */
/* */
/* This is a simple dialog application. It queries a few information from the */
/* user and responds with a message box. */
/* @nd no add another field 'age' (number), when age <18 greet informal, otherwise formal */
/* @ingroup gitk examples */
/* */
/** @def group gitk examples libgitk examples */
/* This package contains a set of example application to demonstrate the */
/* capabilities of gitk. The gitkLayout tool leads and runs GML dialog */
/* descriptions. The gitHelloUser is a simple dialog application. */
/* */
#include "config.h"
#include <stdio.h>
#include <libgitk/gb BOOL g global
#include <libgitk/gitk.h>*/
/* */
/* int tool--extract --type=gittext/xml gitHelloUser_main.xml in */
/* */
/** @brief dialog field: user name */
char * user_name=NULL;
/** @brief dialog field: sex */
char *sex=NULL;
/** @brief dialog field: age */
char *age=NULL;
/** @brief application termination status */
gboolean ret=TRUE;

//== global event handler ================
/** @brief handle global close events like pressing ctrl-c or closing the dialog window */
+@nd we should add a dialog here asking "Are you sure?" */
gboolean global_on_close(gpointer dialog, gchar *widget_id, GitkEventPtr event, gpointer user_data) {
    gitk_log("global_on_close()");
    ret=FALSE;
    return(TRUE);
}

/** @brief gets invoked after a dialog has received the focus */
gboolean global_on_dialog_show(gpointer dialog, gchar *widget_id, GitkEventPtr event, gpointer user_data) {
    gitk_log("global_on_dialog_show()");
    return(TRUE);
}

/** @brief gets invoked before a dialog will lose the focus */
gboolean global_on_dialog_hide(gpointer dialog, gchar *widget_id, GitkEventPtr event, gpointer user_data) {
    gitk_log("global_on_dialog_hide()");
    return(TRUE);
}

//== main dialog ================
/** @brief gets invoked when a action widget has been invoked */
gboolean main_on_invoked(gpointer dialog, gchar *widget_id, GitkEventPtr event, gpointer user_data) {
    gitk_log("main_on_invoked("%s ")", widget_id);
    if (!strcmp(widget_id, "Okay")) {  
        gitk_event_loop_end();
        ret=TRUE;
        return(TRUE);
    }  
    else if (!strcmp(widget_id, "Cancel")) {  
        gitk_event_loop_end();
        ret=FALSE;
        return(TRUE);
    }  
    return(FALSE);
}

/** @brief gets invoked when a widgets value has been changed */
gboolean main_on_changed(gpointer dialog, gchar *widget_id, GitkEventPtr event, gpointer user_data) {  
    char * value=gitk_widget_get_value(dialog, widget_id, "value");
    gitk_log2("main_on_changed("%s ")", widget_id, gitk_save_get_string(value));
    if (!strcmp(widget_id, "UserName") {  
        return(FALSE);
    }  
    return(TRUE);
}
CHAPTER 7. APPENDIX

```c
if ((gchar **)user_data) g_free((gchar **)user_data);
else if (!strcmp(widget_id, "Sex") || 
    gint num_value = atoi(value) || 
    switch(num_value) { /* undo use translated versions here */ 
    case 0: +(gchar **)user_data = g_strdup("(Mc)"); break;
    case 1: +(gchar **)user_data = g_strdup("(Ms)"); break;
    default: +(gchar **)user_data = g_strdup("(Mc/Ms)"); break;
} 
    g_free(value);
} 
return(FALSE);
}
#endif

boolean main_show(void) {
    gpointer const dialog = gitk_dialog_new_from_file(GIMLE_PATH "gitkHelloUser_main.xml");
    if (dialog) {
        if (gitk_dialog_show(dialog, FALSE) == FALSE) {
            // --- main callbacks
            gitk_log("dialog parsed and shown");
            // --- start event loop
            gitk_log("run");
            gitk_event_loop_start();
            gitk_log("hiding");
            gitk_dialog_hide(dialog);
            gitk_log("dialog done");
            result = TRUE;
        } else {
            // --- couldn't happen
            gitk_dialog_free(dialog);
        }
    }
    else {
        gitk_printf("error parsing dialog => %d : %s\n", gitk_get_error_code(), gitk_get_error_str());
    }
    return(result);
}
```

```
//== info message

void info_show(void) {
    gchar *message;
    gchar *new_message = g_strdup("Hello %s \n");
    new_message = g_strlcat(message, new_message, strlen(user_name) + 4);
    sprite = g_strdup("Hello %s \n", sex, user_name);
    gitk_logf("opening message dialog \"%s\", message");
    gtk_template_message_with_title(message, "Greetings");
    g_free(message);
```

```c
//== main

```
7.3 Simple example dialog definition

```xml
<DOCTYPE gimp SYSTEM "http://gimp.sourceforge.net/DTD/gimp.dtd">
<!-- Std: gitHelloUser_main.xml in v 1.8 2004/11/10 08:55:07 ensonic Exp 5
   @file gitHelloUser_main.xml
   @author Stefan Kost ensonic@users.sf.net
   @date Thu Jan 17 11:22:38 2002
   *
   * @brief main dialog for gitHelloUser.c

<xml version="1.0" encoding="UTF-8"/>
```

7.4 Example dialog definitions in other languages

This section gives some examples for other XML languages for comparison purpose.

7.4.1 UIML dialog example

The sample dialog shown below will render to a frame with a label.

```
<xml version="1.0" encoding="ISO-8859-1"/>
<xml public="" id="UML2.dtd"/>
<xml>
  <head>
    <meta name="Purpose" content="UML document example"/>
  </head>
  <interface name="myInterface">
    <structure>
      <part name="TopHello" class="Frame">
        <style>
          <property name="rendering">Congratulations ...</property>
          <property name="title">Congratulations ...</property>
          <property name="resizable">true</property>
          <property name="foreground">white</property>
          <property name="size">500,100</property>
          <property name="location">100,100</property>
        </style>
      </part>
      <part name="Label">
        <style>
          <property name="text">Sample label</property>
        </style>
      </part>
    </structure>
  </interface>
</xml>
```

7.4.2 XUL dialog example

This example shows a frame containing a label, an image and a button.

```
<xml version="1.0" encoding="ISO-8859-1"/>
<xml public="" id="UML2.dtd"/>
<xml>
  <head/>
  <interface name="myInterface">
    <structure>
      <part name="TopHello" class="Frame">
        <style>
          <property name="rendering">Congratulations ...</property>
          <property name="title">Congratulations ...</property>
          <property name="resizable">true</property>
          <property name="foreground">white</property>
          <property name="size">500,100</property>
          <property name="location">100,100</property>
        </style>
      </part>
      <part name="Label">
        <style>
          <property name="text">Sample label</property>
        </style>
      </part>
    </structure>
  </interface>
</xml>
```
7.4.3 Pong dialog example

The example shows the various supported data types in a settings dialog.

```xml
<?xml version="1.0"?>
<PongElements>
<BaseConfPath>apps/testpong/foo</BaseConfPath>
<DialogTitle>Cesky property a takovy</DialogTitle>
<DialogTitle>Cesky property a takovy</DialogTitle>
<Help>
<Name>testpong</Name>
<Path>index.html</Path>
</Help>
<Level>
<Name>Novice</Name>
</Level>
<Level>
<Name>Novice</Name>
</Level>
<Level>
<Name>User</Name>
</Level>
<Level>
<Name>Hacker</Name>
</Level>
<Level>
<Name>Hacker</Name>
</Level>
<Level>
<Name>Foo</Name>
</Level>
<Level>
<Name>Int Test</Name>
</Level>
<Level>
<Name>Hacker</Name>
</Level>
<Widget>
<Name>testint1</Name>
</Widget>
<Label>Test Int</Label>
<Type>Pong:Radio:Group</Type>
<Tooltip>This is just a testing of an integer, nothing really interesting, but try putting the setting above 23 or below 23 and watch the sensitivity of the Foo radio buttons toggle</Tooltip>
<Argument>
<Name>lower</Name>
<Value>-10</Value>
<Translateable>False</Translateable>
</Argument>
<Argument>
<Name>upper</Name>
<Value>2000</Value>
</Argument>
<Argument>
<Name>onmouseover</Name>
<Value>True</Value>
</Argument>
<Widget>
<Name>testint2</Name>
</Widget>
<Label>Int Test</Label>
<Type>Pong:Radio:Group</Type>
<Argument>
<Name>horizontal</Name>
<Value>True</Value>
</Argument>
</PongElements>
```
7.4.4 Glade dialog example

This example shows a spinbutton with a label and a text entry widget in a table layout.
<"xml version="1.0" standalone="no" \> \<!-- mode: xml --\>
<DOCTYPE glade-interface SYSTEM "http://glade.gnome.org/glade-2.0.dtd">
<glade-interface>
<widget class="GWindow" id="window1">
<property name="title" translatable="yes">GWindow</property>
<property name="type">GTK_WINDOW_TOPLEVEL</property>
<property name="window-position">GTK_WIN_POS_NORMAL</property>
<property name="modal">False</property>
<property name="default-width">3440</property>
<property name="default-height">2590</property>
<property name="allow-grow">True</property>
<property name="visible">yes</property>
<signal name="destroy" handler="gtk_main_quit" />
</child>
<widget class="GtkTable" id="table1">
<property name="n-rows">3</property>
<property name="n-columns">1</property>
<property name="homogeneous">False</property>
<property name="row-spacing">0</property>
<property name="column-spacing">0</property>
<property name="border-width">2</property>
<property name="visible">yes</property>
</child>
<widget class="GtkLabel" id="label1">
<property name="label" translatable="yes">I am a label</property>
<property name="justify">GTK_JUSTIFY_LEFT</property>
<property name="wrap">True</property>
<property name="valign">0.5</property>
<property name="vpad">0</property>
<property name="visible">yes</property>
</widget>
<packing>
<property name="left-attach">0</property>
<property name="right-attach">1</property>
<property name="top-attach">0</property>
<property name="bottom-attach">1</property>
</packing>
</child>
<widget class="GtkSpinButton" id="spinbutton1">
<property name="climb-rate">0</property>
<property name="digits">2</property>
<property name="numeric">False</property>
<property name="update-policy">GTK_UPDATE_ALWAYS</property>
<property name="snap-to-ticks">False</property>
<property name="wrap">False</property>
<property name="value">0</property>
<property name="adjustment" />
<property name="visible">yes</property>
</widget>
<packing>
<property name="left-attach">1</property>
<property name="right-attach">2</property>
<property name="top-attach">0</property>
<property name="bottom-attach">1</property>
</packing>
</child>
<widget class="GtkEntry" id="entry1">
<property name="editable">True</property>
<property name="visibility">True</property>
<property name="max-length">10</property>
<property name="has-focus">yes</property>
<property name="visible">yes</property>
</widget>
<packing>
<property name="left-attach">2</property>
<property name="right-attach">3</property>
<property name="top-attach">0</property>
<property name="bottom-attach">1</property>
</packing>
</child>
</glade-interface>
7.5 Installation directory layout

Figure 7.1 shows the filesystem structure of the installed software packages.

7.6 User manual

The following sections should serve as a quick user manual for the supplied example applications. Their behavior can be controlled by using command-line options (for temporary settings) and setting environment variables (for permanent choices).

7.6.1 general example command line options

When starting the examples from the command-line, it is possible to supply a few parameters. These parameters are available in all GITK-based tools and do follow the GNU Coding Standards (see chapter ”4.6 Standards for Command Line Interfaces” in (Stallman and volunteers, 2003)). See table 7.1 for a brief description.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h, –help</td>
<td>show help screen</td>
</tr>
<tr>
<td></td>
<td>(display usage and list supported command-line options)</td>
</tr>
<tr>
<td>--version</td>
<td>show version information</td>
</tr>
</tbody>
</table>

Table 7.1: General example command line options

7.6.2 GITK command line options

All programs based on the libgitk can handle a few switches, which are briefly described in table 7.2. Below are a few example invocations:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--gitk-renderer</td>
<td>which rendering module to use (e.g. text, gtk)</td>
</tr>
<tr>
<td>--gitk-style</td>
<td>which style to use</td>
</tr>
<tr>
<td>--gitk-server-port</td>
<td>which port to use for the internal HTTP server</td>
</tr>
</tbody>
</table>

Table 7.2: GITK command line options

./bin/gitkHelloUser --gitk-renderer=gtk
./bin/gitkHelloUser --gitk-renderer=text --gitk-style=color
### 7.6.3 GITK environment variables

All programs based on the libgitk support a few environment variables. See table 7.3 for details. Apart from that, there are more environment variables that affect the behavior of GITK. See 7.4 for details. To select a specific locale when starting an application under UNIX one could invoke the application as shown below:

```bash
env LC_ALL=de ./bin/gitkHelloUser
env LC_ALL=en ./bin/gitkHelloUser
```

### 7.6.4 GITK text renderer command line options

The options shown in table 7.5 can be used in addition to the general options (see sections 7.6.1 and 7.6.2).

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>–gitkr-text-speech</td>
<td>which speech module to use (festival)</td>
</tr>
</tbody>
</table>

### 7.6.5 GITK text renderer keyboard commands

Table 7.6 shows all keyboard commands that can be used in the text renderer for dialog navigation. The chosen keys are partly restricted by the fact that the renderer works on a console, which also can be on a remote terminal. There not all key-actions can be queried by the application. When a widget is active more key-bindings might be available.
7.6.6 Included example applications

GITK provides a separate package with a few examples. These can demonstrate the systems capabilities and aid development of new applications. Table 7.7 lists the examples and gives a short description of their purpose.

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gitkHelloUser</td>
<td>simple dialog with confirm message</td>
</tr>
<tr>
<td>gitkLauncher</td>
<td>enhanced application launcher (start menu)</td>
</tr>
<tr>
<td>gitkLayout</td>
<td>dialog viewer (for GIML files)</td>
</tr>
</tbody>
</table>

Table 7.7: Included example applications

**gitkHelloUser**

gitkHelloUser is the main GITK demo-application. It has no real purpose other than demonstrating features of GITK.

**gitkLauncher**

The launcher is a tool for choosing applications to start from a menu (see section 5.8.2). This application is not yet finished, as the text and phone renderers have not yet implemented lists. For the menu structure definition XML has been chosen. Further development should consider the standard proposal in (Bastian et al., 2004) for the XML file syntax.

**gitkLayout**

This is a useful tool for testing GIML files. It presents the supplied GIML dialog definition. The dialog itself is not fully functional, although entering data work.
The tools should be invoked as below.

```bash
./bin/gitkLayout --gitk-renderer=text
    gitk-examples/gitkHelloUser_main.xml
```

### 7.7 Developer manual

GITK comes with just two APIs - one for the main library and one for the renderer modules. The APIs are briefly introduced in the next sections. The project comes with detailed API references, e.g. in the form of hyper-linked html pages. The “modules” link in the top navigation bar is the recommended entry point. Further naming and structuring schemes for the GITK components are described.

GITK orients itself on the filesystem hierarchy of UNIX systems (see filesystem hierarchy standard ([Russell and Quinlan, 2001](#)) for more details). As the software is packaged according to respective GNU guidelines and with the help of the GNU Autotools, most details of the build process are handled automatically (e.g. installing language catalogs, running the test suite).

#### 7.7.1 GITK Core API

The core API contains classes to handle e.g. dialogs and widgets, as well as the whole internal system (rendering plugin manager, sensors and much more). Although the library is written in C, it is designed object oriented. Instances are represented by internal handles and methods that work on data objects receive the object as the first parameter.

This API is interesting for both application developers and plugin writers.

#### 7.7.2 GITK Renderer API

The renderer plugin API defines the entry points a plugin writer needs to implement, so that the plugin can be used in the GITK system. This API is only interesting for rendering plugin writers.

The core component uses the exported entry points to interact with the plugin. Currently there are only 8 such entry points that need to be implemented. Three of them deal with construction, destruction and introspection of a rendering plugin. The remaining ones control the event-loop and the dialog transitions.

#### 7.7.3 Filesystem structure for components

GITK consist of several components. This was shown in figure 5.3. The components are shared libraries, XML- and XSL-files, language catalogs and many more. Below
one finds all information about the naming schemes to use for the file and directory
names, as well as the file-system structure to use.

In the next chapters "$prefix" will denote the directory one has chosen when
building and installing GITK. As it has been pointed out earlier, most files (binaries,
libraries, man-pages, shell-scripts and language catalogs) are already handled by the
GNU Autotools. GITK comes with many data files (XML and XSL files) as well. Ac-
cording to the FHS (Filesystem Hierachy Standard) these belong to a directory called
"share/\<package\>" (see (Russell and Quinlan, 2001)). In this work \<package\>
expands to "gitk". Below this entry a further structure has been defined, which is
introduced next. The name of each package plays a central role in the naming of sub-
directories.

**XSL stylesheets**

GITK uses style-sheets in many places, but mainly for two purposes - for domain
adaptation and for dialog decoration (see section 5.1.3 for more details). All style-
sheets should be installed below the "$prefix/share/gitk/xsl" directory. Each
package should create its own directory (named like the package).

The stylesheet for the domain transformation must be called "domain.xsl". If
such a stylesheet does not exists, a symbolic link should be created to the file "$prefix-
/share/gitk/xsl/gitk-core/xsl/noop.xsl".

All stylesheets which apply decorations (presentation mapping) belong to the style
subdirectory of the package. Further a symbolic link to the default style should be
created as "\<package\>/style/default.xsl". This can be a link to the "$prefix-
/share/gitk/xsl/gitk-core/xsl/noop.xsl"

**language catalogs**

All language catalogs ("*.mo" files containing text translations for various languages)
are installed in "$prefix/share/locale/\<language>/LC_MESSAGES/". Each lan-
guage catalog is named like the package it belongs to, followed by the extension
".mo".

**GIML dialog definitions**

All GIML dialog definitions (/*.xml files) are installed below the "$prefix/share/-
gitk/giml" directory. Each package must put the files into an own directory. Files
from the core component (e.g. dialog templates) are in the sub directory "gitk-core"
(this is the name of the core package). Renderers will place their files in sub directories
named "gitk-renderer-" followed by their name (e.g. "gitk-renderer-text").
Applications should use the name of the package they belong to as well.
7.7.4 Option naming scheme

Options can be specified on the command line or via environment variables. The names of all package specific option should contain the package name in their name to avoid naming conflicts. Therefore all options related to the core-library start with "--gitk-" and those for e.g. the text-mode renderer start with "--gitk-text-". Each package should handle standard options such as "--help" and "--version". (Stallman and volunteers, 2003)
Figure 7.1: The filesystem structure of a GITK installation
Bibliography


FA. Brockhaus GmbH (1999). Der Brockhaus in fünfzehn Bänden. 57


Khella, A. (2002). Objects-actions interface model. *University of Maryland, Department of Computer Science Homepage*. 37


Raskin, J. (2001). *Das intelligente Interface.* Addison-Wesley Verlag. 10, 113


various contributors (1995-2003). Dublin core metadata initiative. dublin core www site. 97

various contributors (1999-2002). Free on-line dictionary of computing. foldoc dictionary www site. 42


W3C HTML Working Group (1 August 2002). Xhtml 1.0 the extensible hypertext markup language (second edition). http://www.w3.org/TR/xhtml1/. 79


Index

3D, 111
abstract interaction handlers, 15
abstract interface, 84
abstract toolkits, 18, 72
Abstract User Interface Markup Language, 77
Abstract Windowing Toolkit, 26
accessibility profile, 125
active adaptation, 10
Ada, 25, 32
adaptable system, 10, 125
adaptation, 10, 67
adaptation infrastructure, 7, 11, 38, 47, 82, 86, 89
adaptation method, 10, 95
adaptation methods, 47, 82, 89, 96, 125
adaptation profile, 11, 52, 89, 97, 110, 125
adaptation profiles, 38
adaptive system, 10, 125
AmigaOS, 29, 68
application context, 23
application interface, 17
application launcher, 69, 120
application logic, 3, 11, 80, 112
arch model, 32, 88
architectural model, 3, 4
architectural modell, 38
architecture, 87
AUIML, 77
Avanti, 73
backward references, 65, 115
Basic, 27
basic toolkits, 18, 72
behavior, 63, 112
BeOS, 25
bidirectional transformation, 89, 118
C, 25, 32, 33, 68
C++, 25, 27, 32, 33, 68
cameleon framework, 37
CCI, 9
CGI, 80
Check, 121
CHI, 9
Coin3D, 111
color schemes, 10
command line options, 103, 137, 138
Common User Access, 41
communication protocols, 19
components, 88
cross cultural adaptation, 14
computer supported cooperative work, 66
cross cultural adaptation, 14
computer-human interaction, 9
cross cultural adaptation, 14
cross cultural adaptation, 14
concepts and tasks, 84
core, 88, 91
content, 67, 116
cross cultural adaptation, 14
cross cultural adaptation, 14
content selection, 67, 116
cross cultural adaptation, 14
core, 88, 91
cross cultural adaptation, 14
culture, 14, 53
CSCW, 66
cultural adaptation, 53
CSS, 80
curses, 88, 111
INDEX

DCMI, 98
decorating interface objects, 62, 105
design for all, 1, 85
Developer manual, 140
development support, 69, 120
device independence, 4
dialog, 20
dialog decoration, 21
dialog management systems, 15
dialog navigation, 64, 65, 103, 112
DialogML, 78
dimension-independent, 61
DIN EN ISO 13407, 40
DIN EN ISO 9241, 40
DirectFB, 25
directory layout, 137
document type definition, 98, 127
DOM, 95
domain transformation, 141
domain-mapping, 58, 103
Druid, 22
DTD, 98, 127
dublin core, 98
dynamic adaptation, 97
ECMAScript, 80–82
effective interaction capabilities, 8
enlarging, 10
environment profile, 11, 47
environment variables, 103, 138
EU-Directive 90/270/EWG, 40
event-loop, 97
example, 48, 49, 128, 131, 132, 139
example applications, 139
example source code, 128
example XML dialog definition, 131
eXtensible Interface Markup Language, 75
filesystem structure, 140
FlawFinder, 121
Generalized Interface Markup Language, 97
gettext, 67, 95
GIML, 88, 95, 97, 106, 127, 141
GIML DTD, 127
GIML viewer, 139
GITK, 88, 95, 119, 121, 137, 138, 140
GITK Core API, 140
GITK renderer, 138
GITK Renderer API, 140
gitkHelloUser, 139
gitkLauncher, 139
glade, 79, 135
global application context, 23
GNU, 95, 121, 137, 140
goals of GITK, 4
GPL, 121
GTK+, 2, 13, 20, 24, 88, 110
guidelines, 40, 61–63, 105
HCI, 9
HDML, 81
Homer UIMS, 73
HTML, 80, 88
human-computer interaction, 9
HyperText Markup Language, 80
i18n, 13, 67, 98
IML, 100
input devices, 19, 60, 103
installation, 137
interaction, 7, 21, 32, 65, 115
interaction capabilities, 8, 11
interaction channels, 8
interaction model, 4, 7, 11
interaction modes, 32
interaction partner, 8, 103
interaction partners, 8
interaction serialization, 65, 115
interaction style, 63
interactor, 8
interface, 16
abstract, 84
concrete, 84, 110
interface aspects, 16
interface design, 18
interface models, 30
interface object, 17, 106
interface object hierarchy, 17, 107
interface object mapping, 62, 106
interface renderer, 19
interface toolkit, 18, 24, 88, 96
interface toolkits, 1, 107
Interfaces for human computer interaction, 72
internationalization, 13, 98
intra dialog navigation, 65
introspection system, 92
Java, 25, 26, 33, 68
Java AWT, 26
Java Foundation Classes, 26
JavaScript, 29, 80
JFC, 26
JScript, 29
l10n, 14
language binding, 13, 24–27, 68, 87, 90, 119
layout algorithm, 61
layout renderer, 19
layouting interface objects, 61, 104
ldml, 14
Linux, 68, 121
local application context, 23
locale, 14
localization, 14
logical interface description, 56, 101
look and feel, 14, 26, 38, 49, 56
MacOS, 25–27, 68, 78
MDI, 22
media-neutral, 101
meta data, 98
MFC, 2, 26, 111
Microsoft Foundation Classes, 26
migration, 69, 120
MMI, 36
model
arch, 32, 88
MVC, 33, 35, 89
PAC, 35
Seeheim, 30, 32, 88
model-view-controller, 33
Motif, 31, 74
Multi-dimensional adaptation, 49, 96
multi-dimensional adaptation, 12
multi-dimensional adaptation problem, 46
multi-modal interaction framework, 36
multi-modal interface toolkits, 18
multiple document interface, 22
multiple interface instances, 66
multiple interfaces, 116
MVC model, 33, 35, 89
OAI, 36
object actions interface, 36
Open Inventor, 111
operating system integration, 68, 119
option naming scheme, 142
OS/2, 29, 41
output devices, 19, 21, 95, 103
PAC model, 35
pagination, 65, 115
partitioning, 10, 65, 115
passive adaptation, 10
Perl, 25, 68
persistance, 68, 117
personal adaptation, 54
platform independence, 13, 15
pong, 79, 133
potential interaction capabilities, 8
presentation mapping, 60, 103, 141
presentation patterns, 126
presentation unit, 102
presentation units, 115
presentation-abstraction-control, 35
profile
accessibility, 125
adaptation, 11, 52, 89, 97, 110, 125
environment, 11, 47
user, 11, 47, 89, 97
profile, user, 125
Python, 25, 68

QT, 20, 27
rad, 69, 120
rapid application development, 69, 120
real life evaluation, 126
recursive transition network, 101
reference implementation, 121, 126
renderer-plugin, 88, 95
rexx, 29
roaming, 68
RPC, 28
RTN, 101

SAA, 40
SDI, 22
Seeheim model, 30, 32, 88
segmentation, 10
SGML, 80
shell, 69
simple object access protocol, 29
single authoring, 89
single document interface, 22
Smalltalk, 33
SOAP, 29
Solaris, 26, 121
Splint, 121
SQL, 32
state persistence, 68, 117
structure, 54, 101
style, 58, 103
style guidelines, 62
stylesheet, 89, 95, 100, 103–106, 110, 141
swig, 119
Swing, 20, 26
system architecture, 87
Systems Application Architecture, 40
technical, 68, 119
technical adaptation, 52

TeleUSE, 73
themeable, 24, 58, 96, 105, 112
tool support, 69, 120
toolkit, 13, 15
transformer-plugin, 95
transition network, 101

ubiquitous computing, 68
UI-toolkit, 13, 15, 18, 20, 59, 72, 104
UIML, 74, 101, 124
uiml, 132
UIMS, 13, 15
UML, 98
Unicode, 12, 13, 95, 98
unified modelling language, 98
UNIX, 2, 25, 27–29, 47, 68, 78, 121, 138, 140
user authentication, 69, 119
user centered design, 85
user interface, 17
user interface management system, 15
User Interface Markup Language, 74
User manual, 137
user modelling, 11, 47, 125
user profile, 11, 17, 47, 60, 89, 97, 125
UTF-8, 95

Valgrind, 121
VBScript, 29
verbosity, 10
VisualBasic, 29
VoiceXML, 81, 112, 124

WAP, 81
WCAG, 41
Web Content Accessibility Guidelines, 41
WebServices, 29
Win32, 25, 27
window decoration, 21
Windows, 2, 22, 24, 26, 27, 29, 41, 61, 68, 74, 78, 121
windows scripting host, 29
Wizard, 22
WML, 81
workflow, 69, 120
wrapper, 68, 87, 90
wsh, 29
wxWidgets, 78
wxWindows, 78
X11, 25
XAML, 75
XForms, 76, 124
XHTML, 80
XIML, 75
XMI, 98
XML, 80, 97, 140
XML metadata interchange, 98
XML user-interface language, 75
XPath, 97
XSL, 97, 104, 105, 140, 141
XSLT, 95, 97, 100, 104–106
XUL, 75, 101, 124
xul, 132
XUL motors, 75
XWindows, 24, 27, 31

zooming
  enlarging, 10
  navigation, 10
Glossary

accessibility  Accessibility is a term for describing features and capabilities that enhance electronic systems to be usable for people with disabilities such as blindness, low vision, and motor impairments. [http://lists.w3.org/Archives/Public/w3c-wai-eo/2003JanMar/0032.html](http://lists.w3.org/Archives/Public/w3c-wai-eo/2003JanMar/0032.html)

API  *Application Programming Interface*  The public part of a programming library.

CAPI  *Common ISDN Application Programming Interface*  An interface layer between ISDN communication hardware and the operating system.

CC/PP  *Composite Capability/Preference Profiles*  A CC/PP profile is a description of device capabilities and user preferences. This is often referred to as a device’s delivery context and can be used to guide the adaptation of content presented by that device.

CCI  *Computer Computer Interaction*  Interaction scenario, where contrary to HCI two computer are the communication partners.

CGI  *Common Gateway Interface*  An interface standard which provides a method of executing a server-side program (script) from a website to generate a webpage with dynamic content. Scripts conforming to this standard may be written in any programming language that produces an executable file, but are most often written in Perl, Python, C, C++, or TCL.

CHI  *Computer Human Interaction*  see HCI

CSS  *Cascading Style Sheets*  A simple mechanism for adding style (e.g. fonts, colors, spacing) to Web documents written in the HTML language.

DOM  *Document Object Model*  An application programming interface (API) for well-formed XML documents (and HTML). It defines the logical structure of documents and the way a document is accessed and manipulated.

DTD  *Document Type Definition*  The purpose of a DTD is to define the legal building blocks of a XML document. It defines the document structure with a list of legal elements.
DTMF  *Dual-Tone Multi-Frequency*  A method to transmit key-presses on a telephone keypad over the telephone line. It uses a four by four matrix of frequencies to encode up to 16 key-codes.

GIML  *Generalized Interface Markup Language*  A XML based markup language for GITK dialog descriptions.

GITK  *Generalized Interface ToolKit*  The multi-modal interface package described in this work.

GNU  *GNU is Not UNIX*  The GNU Project was launched in 1984 to develop a complete UNIX-like operating system which is free software: the GNU system.

GPL  *GNU Public License*  The licence applied to most software from the Free Software Foundation and the GNU project and other authors who choose to use it.

GTK+  *Gimp ToolKit*  A multi-platform toolkit for creating graphical user interfaces. Offering a complete set of widgets, GTK+ is suitable for projects ranging from small one-off projects to complete application suites.

GUI  *Graphical User Interface*  Is a method of interacting with a computer through a metaphor of direct manipulation of graphical images and widgets in addition to text.

HCI  *Human Computer Interaction*  The study of interaction between people (users) and computers. It is an interdisciplinary field, relating computer science, psychology, cognitive science, human factors (ergonomics), design, sociology, library and information science, artificial intelligence, and other fields. Interaction between users and computers occurs at the user interface (or simply interface), which includes both hardware (i.e. input and output devices) and software (e.g. determining which, and how, information is presented to the user on a screen).

HTML  *HyperText Markup Language*  A language for publishing hypertext on the World Wide Web. It is a non-proprietary format based upon SGML, and can be created and processed by a wide range of tools, from simple plain text editors to sophisticated WYSIWYG authoring tools.

HTTP  *HyperText Transfer Protocol*  The primary method used to communicate information on the World Wide Web. HTTP is a sessionless request/response protocol between clients and servers.

IDE  *Integrated Development Environment*  A software application that has a text editor as a core and that comes with a lot of extensions that support software development. Such extensions range from programming language support (syntax
parsing and highlighting), debugging, special editors e.g. for user interfaces and so on.

**Linux** Linux is a free Unix-type operating system originally created by Linus Torvalds with the assistance of developers around the world. Developed under the GNU General Public License (GPL), the source code for Linux is freely available to everyone.

**OS** Operating System An operating system is the system software responsible for the direct control and management of hardware and basic system operations, as well as running application software.

**RPC** Remote Procedure Call A protocol that allows a computer program running on one host to cause code to be executed on another host without the programmer needing to explicitly code for this. When the code in question is written using object-oriented principles, RPC is sometimes referred to as remote invocation.

**RTN** Recursive Transition Network State transition graph which can contain subgraphs that can be used recursively. There is a standard graphical notation to present such networks. An application of RTNs is to model automata.

**service** A service is a collection of functionality. Applications provide services to end-users (humans or other applications) through interfaces.

**session** A session begins when the user starts to interact with an application and ends when requested by the user or the application. An application uses sessions to keep track of the application state for an individual user. If the application supports it, a session can be made persistent and can be continued later.

**SGML** Standard Generalized Markup Language The Standard Generalized Markup Language (SGML, ISO 8879) is a metalanguage in which one can define markup languages for documents.

**SOAP** Simple Object Access Protocol A lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. Its messaging framework defines, using XML technologies, an extensible messaging framework containing a message construct that can be exchanged over a variety of underlying protocols.

**SVG** Scalable Vector Graphics Is the description of an image as an application of the Extensible Markup Language XML. Different from a raster graphics image (which consists of a bit pattern description), a vector graphics image is expressed by mathematical statements. The scalable part of the term emphasizes that vector graphic images can easily be enlarged or shrunk (whereas an image specified in raster graphics is a fixed-size bitmap). Thus, the SVG format enables the
viewing of an image on a computer display of any size and resolution, whether a tiny LCD screen in a cell phone or a large CRT display in a workstation. In addition to ease of size reduction and enlargement, SVG allows text within images to be recognized as such, so that the text can be located by a search engine and easily translated into other languages.

**UIML** *User Interface Markup Language*  A XML based language for defining user interfaces. It is used for defining the actual interface elements.

**UIMS** *User Interface Management System*  A system supporting the development and execution of user interfaces, usually on top of windowing systems.

**UNIX**  An operating system which in the past dominated the server market and has now spread to desktop systems as well.

**WAP** *Wireless Application Protocol*  A specification for a set of communication protocols that standardizes the manner in which wireless devices (such as mobile phones, pagers, two-way radios, radio transceivers, smartphones, and communicators) are used for Internet access.

**WCAG** *Web Content Accessibility Guidelines*  These guidelines explain how to make Web content accessible to people with disabilities.

**WML** *Wireless Markup Language*  A XML based language that allows the text portions of Web pages to be presented on cellular telephones and personal digital assistants (PDAs) via wireless access. It was formerly known as HDML (Handheld Devices Markup Languages). WML is part of the Wireless Application Protocol (WAP).

**WSH** *Windows Scripting Host*  The Windows Script Host is an interpreter available for the Windows platform, which allows automation of computer operations. For example, it can allow programs to talk to each other and accomplish tasks without human intervention.

**XForms** *The Next Generation of Web Forms*  A specification of Web forms that can be used with a wide variety of platforms including desktop computers, hand helds, information appliances, and even paper. XForms started life as a subgroup of the HTML Working Group, but has now been spun off as an independent activity.

**XHTML** *Extensible HyperText Markup Language*  A family of current and future document types and modules that reproduce, subset, and extend HTML, reformulated in XML. XHTML Family document types are all XML-based, and ultimately are designed to work in conjunction with XML-based user agents. XHTML is the successor of HTML, and a series of specifications has been developed for XHTML.
XML  *eXtensible Markup Language*  Extensible Markup Language (XML) is a simple, very flexible format for structured text derived from SGML (ISO 8879).

XSL  *eXtensible Stylesheet Language*  XSL is a family of recommendations for defining XML document transformation and presentation.

XSLT  *eXtensible Stylesheet Language Tranformation*  XSL Transformations is a language for transforming XML.

XUL  *XML User-interface Language*  The XML User Interface Language is a markup language for describing user interfaces. With XUL you can create rich, sophisticated cross-platform web applications easily.
List of Tables

1.1 Information technology in german households 2004 (editorial staff, 2004) ................................................................. 1
2.1 The GTK+ language bindings ........................................ 25
2.2 The GTK+ platform availability ...................................... 25
2.3 The MFC language bindings .......................................... 27
2.4 The MFC platform availability ....................................... 27
2.5 The QT platform availability ......................................... 27
2.6 Comparison of HCI interface toolkits .............................. 28
2.7 Comparison of CCI interface toolkits ............................... 30
3.1 Personal preferences and the difficulty of adaptation .......... 55
3.2 Relation of media and human senses ............................... 56
3.3 Relation of dialog specification to presentation domain and application ................................................................. 57
3.4 Relation of dialog specification to presentation domain and application ................................................................. 57
3.5 Attributes of human senses ........................................... 63
3.6 Summary of the problems grouped by interface aspects ....... 71
4.1 Comparison of approaches with respect to aspects of adaptation ................................................................. 83
4.2 Comparison of approaches with respect to what changes are needed to support another modality ................................................................. 83
4.3 Comparison of approaches regarding the levels of abstraction ................................................................. 84
5.1 Examples of renderers and technological aspects they cover ................................................................. 96
5.2 Examples for canonical interface object naming ................ 106
5.3 Semantics of interface object hierarchy node names ............ 108
5.4 Supported events for interface components ....................... 112
7.1 General example command line options ......................... 137
7.2 GITK command line options ........................................... 137
7.3 GITK environment variables .......................................... 138
7.4 Other environment variables ......................................... 138
7.5 Text renderer command line options ............................... 138
7.6 Text renderer keyboard commands ................................. 139
7.7 Included example applications \hspace{4cm} 139
List of Figures

1.1 Growing gap between users and computer applications ........... 2
2.1 Optimal interaction scenario ...................................... 8
2.2 Interaction scenario with environmental influence .................. 9
2.3 Schematics of adaptation and involved components .................. 12
2.4 Interfaces in the past ............................................. 19
2.5 Interfaces in the future ............................................. 20
2.6 Various toolkits with their supported language bindings ........... 24
2.7 Various toolkits with their platform availability .................... 25
2.8 The Seeheim model ................................................. 31
2.9 The ARCH model .................................................... 33
2.10 The MVC model ..................................................... 34
2.11 The PAC model ..................................................... 43
2.12 The MMI framework ............................................... 44
2.13 The Cameleon framework .......................................... 44
3.1 Relation between application and user in the past ................... 45
3.2 Relation between application and user in the present ............... 46
3.3 Relation between application and user in the future ................. 46
3.4 Example screenshot of the account admin application ............... 48
3.5 Example screenshot of the address book application ................. 49
3.6 State-transition-network for the address book example ............... 50
3.7 Adaptation for multiple targets with early fan out ................. 51
3.8 Adaptation for multiple targets with postponed fan out ............. 52
3.9 Generalization versus specialization ................................ 55
3.10 Two possible dialog layout styles ................................ 61
3.11 Problem of serializing interaction order when having alternative routes ...... 66
3.12 Problem of back references and interaction serialization .......... 67
3.13 Needed tool support for developers ................................ 70
4.1 Feature space for the least set of features ........................... 73
4.2 Feature space for the widest set of features ......................... 73
5.1 Role of GITK in the adaptation problem ............................ 86
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td>The layers of the GITK architecture</td>
<td>87</td>
</tr>
<tr>
<td>5.3</td>
<td>The GITK components</td>
<td>88</td>
</tr>
<tr>
<td>5.4</td>
<td>The GITK processing pipeline</td>
<td>90</td>
</tr>
<tr>
<td>5.5</td>
<td>The GITK core architecture</td>
<td>91</td>
</tr>
<tr>
<td>5.6</td>
<td>Browser based management console</td>
<td>93</td>
</tr>
<tr>
<td>5.7</td>
<td>Management console showing current XML of a dialog</td>
<td>94</td>
</tr>
<tr>
<td>5.8</td>
<td>Granularity of the event-loop</td>
<td>97</td>
</tr>
<tr>
<td>5.9</td>
<td>GIML and target IML</td>
<td>100</td>
</tr>
<tr>
<td>5.10</td>
<td>The main structure of a GITK dialog description</td>
<td>102</td>
</tr>
<tr>
<td>5.11</td>
<td>Part of a stylesheet that flattens widgetgroups</td>
<td>105</td>
</tr>
<tr>
<td>5.12</td>
<td>Example of adding decoration by using style-themes</td>
<td>106</td>
</tr>
<tr>
<td>5.13</td>
<td>Example of adding decoration by directly adding style attributes</td>
<td>106</td>
</tr>
<tr>
<td>5.14</td>
<td>Part of a stylesheet that adds a default title if missing</td>
<td>107</td>
</tr>
<tr>
<td>5.15</td>
<td>The hierarchy of user interface objects</td>
<td>109</td>
</tr>
<tr>
<td>5.16</td>
<td>Example of specifying dialog widgets</td>
<td>113</td>
</tr>
<tr>
<td>5.17</td>
<td>Example of flattened dialog hierarchy</td>
<td>113</td>
</tr>
<tr>
<td>5.18</td>
<td>Two possible segmentation styles</td>
<td>114</td>
</tr>
<tr>
<td>5.19</td>
<td>Two examples for 3d user interfaces</td>
<td>115</td>
</tr>
<tr>
<td>5.20</td>
<td>Two dimensional presentation of alternative interaction paths</td>
<td>116</td>
</tr>
<tr>
<td>5.21</td>
<td>Overview of the gettext tool chain</td>
<td>117</td>
</tr>
<tr>
<td>5.22</td>
<td>The i18n support in GITK</td>
<td>118</td>
</tr>
<tr>
<td>5.23</td>
<td>Updates and synchronisation in the GITK processing pipeline</td>
<td>118</td>
</tr>
<tr>
<td>7.1</td>
<td>The filesystem structure of a GITK installation</td>
<td>143</td>
</tr>
</tbody>
</table>