Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Electrical Engineering (ESAT)

Equivalent Representations of Multi-Modal User Interfaces through Parallel Rendering

Kris Van Hees

Dissertation presented in partial fulfilment of the requirements for the degree of Doctor in Engineering

June 2012

Equivalent Representations of Multi-Modal User Interfaces through Parallel Rendering

Kris Van Hees

Jury:
Prof. em. dr. Adhemar Bultheel, chair
Prof. em. dr. ir. Jan Engelen, promotor
Prof. dr. Gerhard Weber, co-promotor
(TU Dresden)
Prof. dr. ir. Bart De Moor
Prof. dr. ir. Erik Duval
Dr. Alistair D. N. Edwards
(University of York)
Prof. dr. Jean Vanderdonckt
(Université catholique de Louvain)

Dissertation presented in partial fulfilment of the requirements for the degree of Doctor in Engineering

June 2012

© Katholieke Universiteit Leuven – Faculty of Engineering
Kasteelpark Arenberg 10 Box 2442, B-3001 Heverlee (Belgium)

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotocopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm or any other means without written permission from the publisher.

D/2012/7515/40
ISBN 978-94-6018-504-5

Acknowledgements

"The word yellow wandered through his mind in search of something to connect with. Fifteen seconds later he was out of the house and lying in front of a big yellow bulldozer that was advancing up his garden path."
(Douglas Adams, "Hitchhiker's Guide to the Galaxy", 1979)

Ever since graduating in July 1995, the idea to continue research within the area of computer science wandered through my mind. However, I never settled on what exactly I wanted to focus my research on. When I met some friends who were quite proficient with computers despite being totally blind, I learnt about the apparent limitations that existing assistive technology imposed upon them, especially when interacting with windowing environments. The research interest found something to connect with. And one might say that I did indeed lie down in front of a big yellow bulldozer named "Doctorate".

During my search for someone who might be interested in the accessibility of computer systems, Herman Van Uytven (ICTS KU Leuven) kindly referred me to prof. Jan Engelen as an interested party. I emailed him on Feb 15th, 2002, and he expressed immediate interest in supervising a doctorate. We met on May 24th, 2002, during my vacation trip to Belgium, and we discussed my proposal extensively. I was fortunate to also meet with prof. Joos Vandewalle, who convinced me to conduct my doctorate within the Faculty of Engineering. The administrative aspects of commencing the doctorate were somewhat complicated by the fact that I live in the USA, but finally, on Jan 8th, 2004, the official permission was granted. I am very grateful to the members of the Doctoral Committee of the Faculty of Engineering for having allowed me this wonderful opportunity.

Conducting a doctorate at KU Leuven while living and working in the USA certainly has had its challenges. When I started this phase in my academic development, I never anticipated it would take 8.5 years to bring it to completion. Throughout all this time, and despite many unforeseen events that serve to remind us about the reality of life, prof. Engelen believed in my work, and in me.

Prof. Gerhard Weber (Technische Universität Dresden) provided much insight and expertise in the field of accessibility. Prof. Bart De Moor and prof. Erik Duval kindly agreed to be assessors, and I am thankful for their continuing support throughout the course of this doctorate.

It is an honour that prof. Adhemar Bultheel, dr. Alistair D. N. Edwards, and prof. Jean Vanderdonckt agreed to join my supervisory committee as members of the examination committee. I certainly appreciate the extensive constructive commentary and discussions during the preliminary defence, and strongly feel that the quality of this work has improved significantly because of the jury's questions and comments.

I also wish to express my gratitude to dr. Gottfried Zimmermann, dr. Stefan Kost, prof. Constantine Stephanidis, and prof. Jean Vanderdonckt for providing me with research materials. Christophe Strobbe kindly agreed to present my paper at the 1st International ÆGIS conference in my absence. Prof. Cindy Reedy (Arcadia University) gave me the opportunity to present a guest lecture to her graduate students on Inclusive Education and Assistive Technology. Dr. David Smith provided invaluable insights from his work on the original GUI concept at Xerox PARC.

My parents gave me the opportunity and guidance to grow into the person that I am today, and I am thankful for everything they have done for me. They were always encouraged by my thirst for knowledge, aside from the *occasional* influence it had on attending classes during my earlier years at KU Leuven. After I commenced my doctoral work, almost every overseas phone call that we shared included a short enquiry about its status. My father always expressed an unwavering trust in my ability to bring the doctorate to a successful completion, and he spoke often about looking forward to attending my public defence. Sadly, he was taken from this earth prematurely, and he is not able to witness the completion of this adventure in person.

The time spent on doctoral research and related activities has been substantial, and I am forever grateful to my wife Kathy for her love, patience, understanding, and support. I must also thank our children Daniel, Alyanna, and Nikolas, who all came into our life unaware that their father had embarked on this lengthy endeavour. Collectively, they made sure that I was not able to spend as much time on my doctoral research and the writing of this dissertation as I envisioned, but I can honestly say that I do not regret a single second of those distractions.

I dedicate this work to Kathy, my friend, my love, . . . my wife.

Kris Van Hees
June 4th, 2012

Abstract

Even though the Graphical User Interface (GUI) has been in existence since 1974, and available for commercial and home use since 1984, blind users still face many obstacles when using computer systems with a GUI. Over the past few years, our daily life has become more and more infused with devices that feature this type of user interface (UI). This continuing trend increasingly impacts blind users primarily due to the implied visual interaction model. Furthermore, the general availability of more flexible windowing systems such as the X Window System has increased the degree of complexity by providing software developers with a variety of graphical toolkits to use for their applications.

Alternatives to the graphical user interface are not exclusively beneficial to the blind. Daily life offers us various opportunities where presenting the UI in a different modality may be a benefit. After all, a disability is a condition that imposes constraints on daily life, and often those same constraints are imposed by environmental influences.

Current approaches to providing alternate representations of a user interface tend to obtain information from the default (typically visual) representation, utilising a combination of data capture, graphical toolkit hooks, queries to the application, and scripting. Other research explores the use of adapted user interface development or context-based runtime UI adaptation based on user and environment models. All suffer from inherent limitations due to the fact that they provide alternate representations as a derivative of the default representation, either as an external observer or as an adapted UI.

Based on the original design principles for graphical user interfaces, this work shows that the original design can be generalised where a GUI is essentially the visualisation of a much broader concept: the Metaphorical User Interface (MUI). Expanding upon this MUI, a new definition is provided for “Graphical User Interface”.

The well-known paradigm to provide access to GUIs rather than graphical screens has been very influential to the development of assistive technology solutions for

computer systems. Validation for this paradigm is presented here, and based on the MUI concept, the focus of accessibility is shifted to the conceptual model, showing that access should be provided to the underlying MUI rather than the visual representation.

Building further on the MUI concept, and past and current research in Human-Computer Interaction (HCI) and multimodal interface, a novel approach to providing multimodal representations of the user interface is presented where alternative renderings are provided in parallel with the visual rendering rather than as a derivative thereof: Parallel User Interface Rendering (PUIR). By leveraging an abstract user interface (AUI) description, both visual and non-visual renderings are provided as representations of the same UI. This approach ensures that all information about UI elements (including semantic information and functionality) is available to all rendering agents, eliminating problems such as requiring heuristics to link labels and input fields, or seemingly undetectable elements.

With the PUIR framework, user interaction semantics are defined at the abstract level, thereby ensuring consistency across input modalities. Input devices may be tightly coupled to specific renderings (e.g. a pointer device in a bitmap rendering), but all user interaction by means of such a device maps to abstract semantic events that are processed independent from any rendering.

The novel approach presented in this work offers an extensible framework where support for new interaction objects can be included dynamically, avoiding the all too common frustration that results from needing to wait for assistive technology updates that *might* incorporate support for the new objects.

The PUIR approach can contribute to the fields of HCI and accessibility well beyond the immediate goal of providing non-visual representations of GUIs. By providing a framework where UI rendering and user interaction are abstracted, additional rendering agents and support for additional input modalities can be provided to accommodate the needs of other disability groups. The use of an underlying AUI-based processing engine also ensures that a diverse group of users can collaborate using a similar mental interaction model regardless of the rendering they use. The PUIR framework is also capable of supporting (accessible) remote access to applications, and the presented work may benefit automated application testing methodologies as well by providing a means to interact with an application programmatically.

Beknopte samenvatting

Hoewel het GUI concept reeds sinds 1974 bestaat en beschikbaar is voor commerciëel en persoonlijk gebruik sinds 1984, worden blinde gebruikers nog steeds geconfronteerd met frequente obstakels wanneer ze computersystemen met een GUI gebruiken. Tijdens de afgelopen jaren zijn apparaten met dit soort UI meer gebruikelijk geworden in ons dagelijks leven. De beperkende invloed op blinde gebruikers blijft groeien ten gevolge van het impliciet visuele interactie model. Bovendien heeft de algemene beschikbaarheid van meer flexibele systemen zoals het X Windows System de complexiteit nog verhoogd doordat softwareontwikkelaars een ruimere keuze hebben tussen verscheidene grafische toolkits voor hun toepassingen.

Alternatieven voor de grafische gebruikersomgeving zijn niet exclusief nuttig voor blinden. Het dagelijkse leven geeft ons situaties waar weergave van de UI in een andere modaliteit nuttig kan zijn. Een handicap is nu eenmaal een conditie die beperkingen oplegt aan het dagelijkse leven, en diezelfde beperkingen kunnen dikwijls opgelegd worden door omgevingsinvloeden.

Huidige methodes om alternatieve weergaven voor de gebruikersomgeving te kunnen verstrekken gebruiken meestal informatie van de standaard (meestal visuele) weergave, via onderschepte informatie, haken in het grafische toolkit, oproepen naar de toepassing en scripting. Ander onderzoek verkent het gebruik van aangepaste gebruikersomgevingen of aanpassende UI's op basis van gebruiker en omgeving modellen. Al deze methodes hebben beperkingen ten gevolge van het feit dat ze alternatieve weergaven als een afgeleide van de visuele weergave verstrekken, hetzij als een externe waarnemer of als een aangepaste UI.

Op basis van de originele GUI ontwerp principes toont dit werk aan dat de GUI gegeneraliseerd kan worden zodat het eigenlijk een visualisatie is van een meer algemeen concept: Metaphorical User Interface. Op basis van de MUI kan een nieuwe definitie voor “Graphical User Interface” gegeven worden.

Het welbekende principe om toegang te verlenen voor GUIs in plaats van

grafische schermen heeft een sterke invloed gehad op de ontwikkelingen van ondersteunende technologie oplossingen voor computer systemen. Validatie voor dit principe wordt hier gegeven, en op basis van het MUI concept wordt de focus voor toegankelijkheid verlegd naar het conceptuele model. Dit toont aan dat toegang moet verleend worden aan de MUI in plaats van de visuele weergave.

Verder bouwend op het MUI concept, en huidig en verleden onderzoek in HCI en multimodale omgevingen wordt een nieuwe methode voorgesteld om multimodale versies voor de UI te verstrekken via alternatieve weergaven in parallel met de visuele weergave in plaats van als afgeleide: Parallel User Interface Rendering. Op basis van een AUI beschrijving kunnen zowel visuele als niet-visuele versies aangemaakt worden als weergaven van éénzelfde UI. Deze aanpak zorgt ervoor dat alle informatie betreffende UI elementen (inclusief semantische betrekkingen) beschikbaar is voor alle weergave-agenten, waardoor problemen zoals de noodzaak aan heuristieken om verbanden tussen labels en velden te vinden of schijnbaar onzichtbare elementen vermeden kunnen worden.

In het PUIR raamwerk wordt de semantiek van de gebruikersinteractie op het abstracte niveau gedefiniëerd waardoor consistentie tussen input modaliteiten gegarandeerd kan worden. Invoerapparaten kunnen sterk gekoppeld zijn aan specifieke weergave agenten (bijvoorbeeld een muis in een bitmap weergave). Alle interactie van de gebruiker via een dergelijk apparaat wordt naar abstracte semantische interacties vertaald zodat ze onafhankelijk van de weergave geïnterpreteerd kunnen worden.

De vernieuwende aanpak die in dit proefschrift voorgesteld wordt is een uitbreidbaar raamwerk waarin ondersteuning voor nieuwe UI elementen dynamisch kan worden opgenomen. Dit vermijdt de al te frequente frustratie ten gevolge van het feit dat nieuwe elementen dikwijls pas in een nieuwere versie van een toegankelijkheidstechnologie beschikbaar zijn.

De PUIR aanpak kan bijdragen leveren tot de gebieden van HCI en toegankelijkheid voor meer dan het onmiddellijke doel om niet-visuele weergaven voor GUIs te verstrekken. Doordat een raamwerk beschikbaar wordt gemaakt waarin UI weergave en gebruikersinteractie geabstraheerd worden, is het mogelijk om extra weergave agenten en ondersteuning voor extra invoermodaliteiten te verstrekken om de behoeften van andere gebruikersgroepen te ondersteunen. Het gebruik van een onderliggende AUI-gebaseerde verwerkingseenheid zorgt er ook voor dat diverse gebruikersgroepen kunnen samenwerken op basis van een soortgelijk mentaal interactie model, ongeacht de weergave die ze gebruiken. Het PUIR raamwerk kan ook (toegankelijke) interactie met programmatuur van op afstand mogelijk maken. Dit werk kan verder ook van toepassing zijn in het kader van het geautomatiseerd testen van toepassingen doordat het programmatische interactie met een toepassing mogelijk maakt.

Contents

Acknowledgements **i**

Abstract **iii**

Beknopte samenvatting **v**

Contents **vii**

List of Figures **xv**

List of Tables **xix**

List of Acronyms **xxi**

1 Introduction **1**

1.1 History 4

1.1.1 Consoles, teletypes, and terminals 5

1.1.2 The Graphical User Interface 5

1.1.3 UNIX-type systems 6

1.1.4 Application programming interfaces 7

1.1.5 Analysis 8

1.2 Core terminology 10

1.2.1 Usability 10

1.2.2 Blindness and visual impairment 11
1.2.3 Accessibility 11
1.2.4 Multimodality 12
1.3 Thesis 13
1.3.1 Motivation 13
1.3.2 Scope of the work 15
1.4 Contributions 17
1.5 Chapter by chapter overview 19

2 Graphical User Interface, Human-Computer Interaction, and Universal Access 21

2.1 Introduction 21
2.2 Graphical User Interface 22
2.2.1 GUIs and blind users 23
2.2.2 GUI accessibility vs screen accessibility 25
2.2.3 GUI design principles 25
2.2.4 Towards a new definition for GUI 29
2.3 Non-visual access to GUIs: HCI concerns 31
2.3.1 Coherence between visual and non-visual interfaces 32
2.3.2 Exploration in a non-visual interface 32
2.3.3 Conveying semantic information in a non-visual interface . 32
2.3.4 Interaction in a non-visual interface 33
2.3.5 Ease of learning 33
2.4 Universal Access 33
2.4.1 Universal Access: a new perspective on HCI 34
2.4.2 High level requirements for universal usability 34
2.4.3 GUI design principles vs UA requirements 37
2.5 Abstract User Interface descriptions 40

2.5.1 Technical requirements for AUI description languages . . . 41
2.6 Conclusions 43

3 Target user survey **45**
3.1 Participation 46
3.1.1 Demographics 47
3.2 Concepts of the Graphical User Interface 48
3.3 UI elements: perceptual or conceptual 50
3.4 Mental models 52
3.5 Assistance from sighted peers 56
3.6 Troublesome UI elements 58
3.6.1 Trigger-happy UI elements 58
3.6.2 Unidentified UI elements 59
3.6.3 Semantic relations between UI elements 59
3.7 Conclusions 60

4 State of the art **65**
4.1 Introduction 65
4.1.1 The four layers of user interface design 66
4.1.2 Unified Reference Framework 67
4.1.3 The CARE properties 70
4.2 Evaluation criteria for related work 72
4.2.1 Classification of related work 73
4.3 Related works 74
4.3.1 Accessibility 75
4.3.2 Abstract user interface descriptions 75
4.3.3 WAI-ARIA 78
4.4 Abstraction of a Final User Interface 80

4.4.1 Archimedes a.k.a. Total Access System 81
4.4.2 TIDE/VISA 85
4.4.3 GUIB 89
4.4.4 GNOME Accessibility Architecture (Orca) 94
4.5 Adaptation of an Abstract User Interface 98
4.5.1 MetaWidgets 99
4.5.2 Fruit 102
4.5.3 HOMER UIMS 106
4.5.4 Ubiquitous Interactors 112
4.5.5 GITK 116
4.6 Conclusions 120
4.6.1 Shortcomings 120
4.6.2 Requirements 124

5 Parallel User Interface Rendering 127

5.1 Introduction 127
5.2 Design principles 130
5.2.1 A consistent conceptual model with familiar manipulatives as basis for all representations 130
5.2.2 Concurrent use of multiple toolkits at the perceptual level . 133
5.2.3 Collaboration between sighted and blind users 135
5.2.4 Multiple coherent concurrently accessible representations . 138
5.3 Design 142
5.3.1 Conceptual model 145
5.3.2 AUI descriptions 147
5.3.3 AUI engine 152
5.3.4 Rendering agents 159
5.4 Comparison to Model-View-Controller 162

5.5 Conclusions 163

6 Context-based interaction 165

6.1 Introduction 165

6.2 Devices and events 167

6.2.1 Events 170

6.3 Synchronising user interaction 176

6.4 Modality-dependent user interaction events 177

6.5 Conclusions 183

7 Implementation 185

7.1 Introduction 185

7.2 Events 188

7.2.1 UI creation events 191

7.2.2 User interaction events 192

7.2.3 Reuse and reduce 195

7.3 AUI engine 195

7.3.1 PUIR UI Description Language 197

7.3.2 AUI engine API 198

7.3.3 Unified UI widgets 199

7.3.4 Focus management 201

7.3.5 Keyboard-based user interaction 201

7.4 Rendering agents 202

7.4.1 Reification 204

7.4.2 Abstraction 205

7.4.3 In-process rendering agents 205

7.4.4 External rendering agents 206

7.5 Conclusions 207

8 Evaluations **209**

8.1 Introduction 209

8.2 Internal validation 210

8.2.1 Assessment of the design against requirements 210

8.2.2 Evaluation of the work against state of the art criteria 212

8.3 External validation 217

8.3.1 Test scenario 1: Basic functionality 217

8.3.2 Test scenario 2: Basic comparison 220

8.3.3 Test scenario 3: Accessibility API 221

8.4 Conclusions 223

9 Conclusion **225**

9.1 Contributions 225

9.2 Future work 228

A Target user survey **231**

A.1 Questionnaire 231

A.2 Scoring 232

A.2.1 User interface elements/concepts: Window, Button, and Desktop 233

A.2.2 The importance of knowing the UI layout 234

A.2.3 The significance of a mental model 234

A.2.4 The usefulness of descriptions by a sighted peer 235

A.2.5 What type of mental model is used? 235

B AUI Widgets **239**

B.1 Containers 239

B.2 Components 242

C PUIR UI Description Language DTD **249**

D Example: Programmatic UI creation 259

E Example: AUI description for the PUIR framework 267

Bibliography 271

Curriculum vitae 287

List of Publications 289

List of Figures

1.1 Kenbak-1, the world's first personal computer 5

2.1 WYSIWYG: Star User Interface 27

2.2 Two layers of metaphor; two mappings 30

2.3 Relating GUI to UA and HCI 38

3.1 UI elements: Perceptual vs conceptual 51

3.2 Importance of layout, mental model, and verbal descriptions 54

3.3 The complexity of entry field label placement 60

3.4 Impact of age 61

4.1 Four distinct layers of user interface design 66

4.2 Design layers in the Unified Reference Framework 68

4.3 Unified Reference Framework 69

4.4 Abstraction of a Final User Interface 80

4.5 Total Access System with VTAP 81

4.6 The VISA-Comp System 86

4.7 GUIDE 90

4.8 The GNOME Accessibility Architecture 94

4.9 URF Diagram for the GNOME Accessibility Architecture 95

4.10 Adaptation of a Abstract User Interface 98

4.11 URF Diagram for MetaWidgets 101
4.12 The ”Fruit” project 102
4.13 URF diagram for Fruit 104
4.14 HOMER UIMS 107
4.15 The ”Rooms” metaphor 109
4.16 URF diagram for HOMER UIMS 110
4.17 The Ubiquitous Interactor 113
4.18 GITK 116

5.1 Screen reader using the visual UI as information source. 128
5.2 The role of the conceptual model 131
5.3 X11 session with multiple graphical toolkits 134
5.4 Example of a visual layout that can confuse screen readers. 138
5.5 Example of the effects of a viewport on text visualisation. 138
5.6 Multiple perceptual representations of the same user interface . . 140
5.7 Schematic overview of Parallel User Interface Rendering 143
5.8 Logical flow from interaction to presentation 144
5.9 Web forms bear a striking resemblance to UI data entry screens. . 148
5.10 Sample hierarchical AUI object model 154
5.11 Sample hierarchical CUI object model for the AUI in Figure 5.10 . . 159

6.1 Simultaneous device interaction 166
6.2 Example AWT event sequence 170
6.3 Schematic design for synchronised event handling 178
6.4 Transforming concrete events into abstract events – 1 179
6.5 Transforming concrete events into abstract events – 2 181

7.1 Schematic overview of the experimentalconcept implementation . 187
7.2 GUI representation of the example UI 188

7.3 Class inheritance tree for U2I widgets 200
7.4 Event abstraction vs presentation reification 203
7.5 Schematic overview of remote rendering agent support 207

8.1 URF diagram for PUIR 213
8.2 User interface of the PUIRDemo application 219

D.1 GUI representation of the example UI using Java/Swing 260

E.1 Visual representation of the example UI using PUIR 268

List of Tables

1.1 Driving a car as a complex interaction constraint 3
1.2 Time line of Accessible Technologies 8

3.1 Demographic information of survey participants 46
3.2 Importance of layout vs usefulness of descriptions 56

4.1 Comparison of related works – 1 of 2 121
4.2 Comparison of related works – 2 of 2 122

5.1 Examples of controls and objects in the conceptual model 146
5.2 Widgets provided by the AUI layer, by container 153
5.3 Operations supported by abstract widgets 158

6.1 Levels on which events operate 172
6.2 Distribution of AWT events by event level 174

7.1 AWT/Swing events during UI creation 190
7.2 AWT/Swing events during user interaction 193

8.1 Comparison of PUIR against related works 218

A.2 Survey results – 1 of 2 236
A.3 Survey results – 2 of 2 237

List of Acronyms

2D 2-Dimensional

3D 3-Dimensional

API Application Programming Interface

ARIA Accessible Rich Internet Applications

AT Assistive Technology

AT-SPI Assistive Technology Service Provider Interface

AUI Abstract User Interface

AWT Abstract Widget Toolkit

CSCW Computer Supported Cooperative Work

CSM Current Screen Model

CUI Character-based User Interface
also: Conceptual User Interface
also: Concrete User Interface

DfA Design for All

DTD Document Type Definition

FIFO First-In, First-Out

FUI Final User Interface

GUI Graphical User Interface

HCI Human-Computer Interaction

HTML HyperText Markup Language

JFC	Java Foundation Classes
MUI	Metaphorical User Interface
MVC	Model-View-Controller
NVUI	Non-Visual User Interface
OCR	Optical Character Recognition
OS	Operating System
OSM	Off-Screen Model
PUDL	PUIR UI Description Language
PUIR	Parallel User Interface Rendering
RBI	Reality-Based Interaction
RUI	Remote User Interface
SMIL	Synchronised Multimedia Integration Language
T&C	Tasks and Concepts
U2I	Unified User Interface widget toolkit
UA	Universal Access
UI	User Interface
UIDL	User Interface Description Language
UIMS	User Interface Management System
URF	Unifying Reference Framework
VISC	Virtual Screen Copy
WAI	Web Accessibility Initiative
WIMP	Window, Icon, Menu, Pointing device
WWW	World Wide Web
WYSIWYG	What-You-See-Is-What-You-Get
X11	X Window System, version 11
XML	Extensible Markup Language

Chapter 1

Introduction

"Let us not then speak ill of our generation, it is not any unhappier than its predecessors.
Let us not speak well of it either.
Let us not speak of it at all."
(Samuel Beckett, "Waiting for Godot", 1954)

Over the past few years, our world has become more and more infused with devices that feature a graphical user interface, ranging from home appliances with LCD displays to mobile phones with touch screens and voice control. Although GUIs have been in existence for nearly 30 years, blind[1] people still encounter significant obstacles when faced with this type of user interface. On UNIX-type systems where mixing graphical visualisation toolkits is common, non-visual access is even more problematic. The popularity of this group of systems keeps growing, while advances in accessibility technology in support of blind users remain quite limited.

Alternatives to the graphical user interface are not an exclusive need for the blind. Daily life offers us various opportunities where presenting the UI in a different modality could be a benefit. Sometimes the problem at hand is as simple as sunlight glare on the screen of an Automated Teller machine (ATM); other times one might be faced with the dangers of using a cellular phone while operating a vehicle [83]. In addition, consider the use of computer displays in operating rooms where a surgeon certainly would prefer not turning away from their patient in order to access some information on the screen. All these situations are very similar to the needs of a blind individual trying to access a computer system.

[1] This work use the term "blind" as opposed to "visually impaired" to indicate that an individual has no usable sight at all. See section 1.2.2 for an explananation of terminology as used in this dissertation.

Alan Newell expressed that by embracing the needs of extra-ordinary people, we do not limit the applicability of our work. Instead, we will discover and refine techniques that will benefit the overall user community [102]. This is illustrated by an example from his article:

> "A simple example of the value of "design for disability" was a Norwegian telephone, with a large keypad specifically designed to assist people with physical disabilities; this was found to be invaluable in outdoor kiosks where climatic conditions meant that able-bodied users wore very thick gloves."

The presence of higher level technologies in environments where conditions may be variable contributes to the applicability of multimodal interfaces. Obrenović, et al. present accessibility as a multimodal design issue that can be described in terms of interaction constraints [108]. Analysis of the constraints imposed by disabilities, and the effect of environmental constraints on activities in daily life shows a remarkable overlap. Table 1.1 presents the effect of some constraints on one's ability to operate a moving vehicle. Being able to present information through synthetic speech, and providing multiple data entry methods allows for safer operation of in-car interfaces [35].

Looking towards contemporary and future trends in computing, Weiser coined the term "Ubiquitous Computing" [162] to refer to a form of computing where machines fit the human environment rather than forcing humans to adapt to theirs. In the context of user interfaces, ubiquitous computing involves some level of I/O processing that aims to mimic human communication channels such as speaking, hearing, gesturing, reading, writing, . . . , which obviously relates to multimodal interaction because in this context graphical user interfaces are no longer appropriate [166]. York also notes that based on an extensive literature review on HCI research topics, multimodal interaction and I/O techniques in general are of paramount importance in view of ubiquitous computing. This clearly shows that work in this field has application well beyond the realm of accessibility for people with disabilities.

Aside from the obvious advantages of multimodal user interfaces, and the ability to provide the most appropriate representation based on the context of use, another very important aspect of daily life ought to be considered: people working together. This can take the form of collaborating on a work project, teaching or training, or even performing the same job functions as co-workers. Barriers to collaboration can lead to segregation between the user groups based on their abilities and needs [125]. In extremis, failure to make collaboration possible might be considered a form of discrimination.

Not only is the ability to collaborate an aspect of equal participation in society and work environments, it also influences people's self-esteem. One of the

Constraint	Situation	Influence
Traffic situation	Car stopped	No specific reductions.
	Normal traffic	It is not convenient to require the user to move or use hands. Also, user's central field of vision is directed towards the road.
	Traffic jam	In addition to the normal traffic situation, additional limitation in handling attention requests, so the user is more focused and stressed.
Noise level	Low	No specific reductions.
	Normal	A user's audio perception, audio 3D cues, and speech can be used provided that they are of significant intensity.
	High	All audio effects are significantly reduced.
Visual conditions	Day	No specific reductions.
	Night or fog	Driving conditions are tougher; user is more focused and stressed.
Weather conditions	Dry	No specific reductions.
	Rain or snow	Driving conditions are tougher; user is more focused and stressed.
Emotional state	Relaxed	No specific reductions.
	Stressed	Limited ability to handle attention requests and use complex interaction modalities.
Passengers	No	No specific reductions.
	Yes	Other users can use interfaces. Can affect noise level and linguistic effects.

(Table based on [108].)

Table 1.1: Driving a car as a complex interaction constraint

respondents to the survey presented in chapter 3 phrased this very eloquently [85]:

> "I would like to add that one reason why this topic is important to me is that I would like to be able to help sighted persons with their computers. I'm fairly knowledgeable about computers, but I can't translate for a sighted user. My dad is 84, and struggling to learn how to use email, etc. I can only be of limited help to him, because I can't tell him where to look for certain buttons, etc. Some of my dad's difficult experiences with learning his computer – bad instructors, impatient tech support, or my niece or nephew who have difficulty understanding his frustrations, for instance – has gotten me interested in wanting to help older adults learn computing. But, I am unable to do this without a solid working knowledge of what things look like."

Leveraging the well established paradigm of separation between presentation and application logic [112], advances in multimodal UI development, and existing research on abstract user interface descriptions, this dissertation presents a novel approach to providing equivalent representations of multimodal user interfaces, using a parallel rendering technique to support simultaneous presentation and operation of the UI across different modalities.

The accessibility of computer systems has received quite a lot of attention throughout the past 50 years, both in terms of research and commercial applications. It is therefore beneficial to first introduce this field of research in its historical context. This can be found in section 1.1, followed by a definition of important terms in section 1.2. The thesis statement is presented in section 1.3, along with motivations and explanations of the working hypotheses and a declaration of scope. An enumeration of the main contributions of the conducted research to the field of Human-Computer Interaction and Accessibility follows in section 1.4. The chapter concludes with an outline of the subsequent chapters in this dissertation in section 1.5.

1.1 History

According to the Computer History Museum, the world's first personal computer entered the market in 1971: the Kenbak-1 (Figure 1.1) [95]. Input was provided by means of buttons and switches, and output consisted of a series of lights. The computer was more educational than functional, but it would have been quite easy to make it accessible to blind users. Sadly, at the time, accessibility of computer systems was primarily focused on professional use.

(Image courtesy of Computer History Museum.)

Figure 1.1: Kenbak-1, the world's first personal computer

1.1.1 Consoles, teletypes, and terminals

Seven years earlier, in 1964, staff of the Medical Computing Center at the University of Cincinnati College of Medicine published a groundbreaking paper outlining techniques and aids in support of professional computer work for the blind [136]. The authors recognised that proficiency in operating computers would be an important asset for blind individuals and proudly wrote: "[...] in work with computers the blind may operate without handicap (for the first time in history) [...]". Within a few years, time-shared computing presented an obstacle by introducing remote terminal units. As early as 1968, blind programmers were able to overcome this problem using specialised braille terminal devices [2, 119], and similarly an observation was made that: "With a device of this type a blind programmer has virtually no disadvantages compared with a sighted programmer."

Circa 1974 research at Xerox PARC led to the development of the graphical user interface. It remained an internal project without successful commercial application for roughly 10 years. Within that same time frame, terminals with synthetic speech output were introduced as an alternative to the quite cumbersome teletype-based braille terminals [117, 82]. This time in history also marked the general availability of personal computers as a commercial product [96, 97]. These systems provided the user with a text-only character-based user interface. Within 2–3 years, various solutions to provide access for the blind emerged, first as self-voiced applications, and shortly after as generic screen readers [47].

1.1.2 The Graphical User Interface

While Xerox PARC was not successful in marketing the graphical user interface as a competitive advantages, companies within the personal computer market

segment succeeded. Well known examples are Apple Computer, Inc. with the Macintosh (1984) and Microsoft Corporation with Microsoft Windows 1.0 for the IBM PC (1985). This development presented significant obstacles to the blind. Whereas before non-visual access was primarily a matter of reading the screen content as characters in system memory, now assistive technology (AT) was faced with pixel graphics and visual metaphors [16]. The introduction of the GUI was perceived as a true crisis because [22] “[. . .], some design decisions aimed at improving the interface for the non-disabled user are making those same systems less accessible to those that are handicapped.”

Bowe (quoted by Edwards [38]) phrased the implications of this development quite pointedly:

> “When Drexel University required every freshman to buy a Macintosh. . . it was sending a message that ’No blind person need apply here’.”

In 1989, five years after the introduction of the Macintosh personal computer, Berkeley Systems, Inc. released outSPOKEN, a GUI screen reader with a novel approach to determining the screen content: the Off-Screen Model (OSM) [126]. In 1992, Microsoft Windows became accessible to the blind with the release of SLIMWARE Window Bridge. In subsequent years various other popular screen readers for personal computers emerged [47], along with revisions of existing ones. Microsoft Windows gained significant popularity in the workplace and at home, and a lockstep progression ensued between operating system (OS) releases and screen reader upgrades. IBM released Screen Reader/2 for their OS/2 systems in 1994 [140], introducing scripting in the screen reader in support of applications with graphical user interfaces.

1.1.3 UNIX-type systems

UNIX-type systems saw a quite different development in terms of accessibility for the blind. Development started at Bell Labs in 1969, with first availability in 1975, and a public release in 1982 [55]. Blind users were able to benefit from existing remote access facilities (such as terminals with synthetic speech output).

Staff at the Massachusetts Institute of Technology started work on a GUI environment for UNIX in 1984, and the X Window System (X11) was released in 1987. At this point in time, it was common for blind users to access UNIX systems from a personal computer with a Microsoft Windows-based screen reader. As the OS expanded from servers to workstations, the emerging X11 GUI environment posed the same complications as seen earlier with non-UNIX based personal computers [16]. Circa 1992 the Graphics, Visualisation, and Usability Center at

the Georgia Institute of Technology commenced the Mercator project to [99] “[. . .] provide transparent access to X11 applications for computer users who are blind or severely visually-impaired.” A significant contribution from this project was the introduction of the Remote Access Protocol (RAP) in X11 [44], which could be considered to be the first accessibility application programming interface (API).

When a free UNIX-type OS (Linux) became available, the general public gained a powerful alternative to the primarily Microsoft Windows-based world of personal computers. At the same time, workstations with commercial flavours of the OS were gaining in popularity. With the added power of numerous enthusiastic programmers embracing the open source development model, X11-based desktop usage increased, and with it the need for accessibility. The GNOME Accessibility Project [137] embraced this challenge with the help of commercial entities and the formation of an accessibility working group within the Free Standards Group [53]. In 2005, Sun Microsystems released their Solaris 10 operating system with various accessibility features, including the Gnopernicus screen reader [76]. That same year, development started on a successor: Orca [21]. Both are specific to the GNOME desktop environment, rather than being generic X11 screen readers. Developers at IBM also worked on a screen reader for GNOME: LSR (Linux Screen Reader) [111]. The project started circa 2006 but was abandoned roughly a year later as a result of changes in corporate strategy.

Apple Computer, Inc. moved to a UNIX-type OS in 2001 with the release of Mac OS X. The company recognised the strategic importance of accessibility and a full-fledged screen reader was included with the Mac OS X 10.4 release in 2005: VoiceOver.

1.1.4 Application programming interfaces

Prior to the introduction of GUI environments, screen readers were able to access the content of the screen as characters in system memory. When that was no longer possible, developers were forced to find alternative techniques to retrieve information that in essence was not meant to be accessible to other applications such as the screen reader [40].

The Mercator project introduced the concept of promoting changes or additions to the underlying system in support of AT solutions. It was instrumental in adding the Remote Access Protocol (RAP) in X11 along with various hooks in the toolkit, enabling notifications about UI elements to be provided to external listeners [43].

The introduction of an accessibility API soon became a fundamental feature of the OS. Microsoft Windows was first augmented with Microsoft Active Accessibility (MSAA) in 1997, and in later releases with the UI Automation API (2005). Likewise, Apple Computer, Inc. added an accessibility API to Mac OS X with its

Technology	Year	Assistive Technology
IBM 1401 Mainframe	1961	
	1964	Punch cards, console probe
	1968	Braille terminal
	1974	Terminal with synthetic speech
IBM PC	1981	
	1984	DOS screen reader
	1988	Screen Reader (IBM)
MS Windows	1985	
	1992	GUI screen reader (WinVision)
	1997	MSAA (API)
	2005	UI Automation (API)
	2011	ISO/IEC 13066-2 Standard
Apple Macintosh	1984	
	1989	GUI screen reader (outSPOKEN)
MacOS X	2001	
	2002	Apple Accessibility API
	2005	VoiceOver (MacOS X)
	2011+	ISO/IEC 13066-5 Standard
X Window System	1987	
SunOS 4.1.1 w/ X11	1990	
Solaris 2.1 w/ X11	1992	
Linux w/ X11	1992	
	1994	Remote Access Protocol (API)
	1994	Mercator
GNOME	1999	
	2001	AT-SPI (API)
	2005	Gnopernicus
	2006	Orca
	2011+	ISO/IEC 13066-4 Standard

Table 1.2: Time line of Accessible Technologies

10.2 release in 2002. On the UNIX-type side, the GNOME Accessibility Project introduced the AT-SPI API in 2001, and it has been submitted as a candidate for standardisation with the Free Standards Group.

1.1.5 Analysis

Table 1.2 provides an overview of various influential developments towards providing blind users access to computer systems. The advances made towards

accessibility of mainframe systems is quite significant. By 1974, braille terminal devices and terminals with synthetic voice had been developed. While many improvements were made to both forms of AT in more recent years, the fundamentals of the technologies were established several years prior to the introduction of the personal computer. It is therefore reasonable to conclude that the majority of effort put into developing screen readers post-1981 was spent on implementing mechanisms to access screen content and/or building off-screen models rather than presentation in an accessible format.

The overview in Table 1.2 shows that commercial systems with a GUI environment (aside from systems with X11) on average became accessible to the blind for the first time within five years. Commercial screen readers have kept up to date with system updates, albeit at times with minor delays (several months).

Computer systems with GUI environments based on X11 experienced a very different history, with experimental developments that did not quite reach the level of their commercial counterparts on non-X11 based systems. Development on the Mercator project (and its spinoffs) ceased, and Gnopernicus was abandoned in favour of Orca. Reliance on the AT-SPI API and a CORBA-based infrastructure limits Orca to the GNOME desktop environment, although standardisation efforts may result in a wider adoption of the API. Therefore, 24 years after the first release of the X Window System we must conclude that there is still no generic screen reader available for X11-based systems.

Contrary to other GUI environments, X11 does not impose the use of any specific graphical toolkit or desktop environment on the user. The level of flexibility offered by this design poses a significant complication for screen reader developers, and it is one of the main reasons why progress towards accessibility for the blind has been atypical in comparison with other systems.

Altogether, it is clear that accessibility is not generally a consideration when new technology is designed. In fact, it is often added in response to pressure from the user population or to satisfy regulatory requirements, years after the initial release of the technology [19]. Adding support for accessibility features to an existing product is usually quite costly, and it is bound to face limitations imposed by the original design[2].

An important development that is likely to influence accessibility of computer systems in years to come is the work towards standardisation of accessibility APIs. The ISO/IEC 13066-1 base standard was approved and published on May 5th, 2011 [68], while additional parts (as listed in Table 1.2) for specific APIs are at different stages on the path towards formal approval.

[2] It can be argued that any such limitation can be overcome given sufficient resources, but the economic considerations for doing so are typically prohibitive.

1.2 Core terminology

The thesis statement presented in section 1.3 depends on some important core terminology from the field of Human-Computer Interaction. Many concepts in this field lack a clear consistent definition for which consensus has been reached. Multimodality is a good example, because at a minimum it can be interpreted both from a system-centric context and from a user-centric point of view. The remainder of this section provides the definitions that are applicable for the presented work.

1.2.1 Usability

Literature has defined "usability" in many different ways, often due to differences in view point and context. Various international standards have provided definitions that are not quite consistent with one another:

> "The ease with which a user can learn to operate, prepare inputs for, and interpret outputs of a system or component." (IEEE Std.610.12 [65])

> "A set of attributes that bear on the effort needed for use and on the individual assessment of such use, by a stated or implied set of users." (ISO/IEC 9126 [66])

Yet, in the context of multimodal user interfaces, the following definition from ISO 9241-11 is most on target [67]:

Definition 1.1. *"The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use."*

It specifically defines three areas of concern that are to be evaluated towards qualifying usability: effectiveness, efficiency, and satisfaction. When considering multiple target user groups (e.g. groups with differing abilities and needs), it is important to be able to measure usability as a success criterion. Achieving equivalent levels of usability (measured by a common standard) is a long-term goal for the approach presented in this dissertation.

1.2.2 Blindness and visual impairment

Various terms relating to blindness and visual impairment are used in research literature. By its very nature, visual impairment covers a very broad area ranging from no light perception at all to blurred vision, and every gradation in between. Light perception relates to the ability to e.g. determine through vision whether one is in a dark or bright location. In addition, the field of view may be restricted or it could include so-called blind spots.

The term “legally blind” is used to indicate that someone has met a specific set of criteria based on either low acuity or a restricted field of vision. The criteria differ from country to country; in the United States of America, legal blindness is defined as having a visual acuity of 20/200 or less in the better eye, with the use of a correcting lens, or a field of vision where the widest diameter subtends an angular distance of 20 degrees or less in the better eye [30].

The term “low vision” is used to describe “individuals who have a serious visual impairment, but nevertheless still have some useful vision” [64].

The term “blind” is often used in a restrictive sense to indicate that someone’s vision is limited to light perception or less. Individuals who are deemed “blind” do not have any usable vision.

The term “visually impaired” is used for any individual who is deemed legally blind [87].

Of all visually impaired people, the blind constitute a well defined group. The remainder of the community of legally blind individuals cover a whole range of impairments and their associated needs. Accomodating one could exclude another: using enlarging of text may be an obvious accomodation for partially sighted people, but makes matters worse for a user with a restricted field of vision because even less will be visible to them [39].

1.2.3 Accessibility

Bergman and Johnson define “accessibility” as follows [7]:

Definition 1.2. *Providing accessibility means removing barriers that prevent people with disabilities from participating in substantial life activities, including the use of services, products, and information.*

This definition is unfortunately not specific enough for the work presented in this dissertation.

As illustrated in section 1.1, the accessibility and usability of systems and devices with a graphical user interface has been a concern for over 25 years. The overall goal was very accurately phrased in 1964, albeit at that time with an overly optimistic observation that [136] "[. . .] in work with computers the blind may [now] operate without handicap [. . .]."

Mynatt recognised that an important aspect of accessibility is often overlooked or taken for granted [98]: "An implicit requirement [is to] facilitate collaboration among sighted and blind colleagues. [. . .] Therefore it is imperative that [they] be able to communicate about their use of application interfaces."[3]

The barrier to collaboration between sighted and blind users is indeed often overlooked when considering the accessibility and usability of computer systems, which is a sad irony in view of the current proliferation of distributed work environments where workers are no longer in close proximity to one another. In addition, the aforementioned definition of accessibility does not quite make the requirement for "usability" explicit [159]. While it is certainly implied (". . . the use of services, products, and information"), the need for usability as defined in section 1.2.1 in order for a system to be truly accessible is of such great importance that an explicit inclusion of the requirement is certainly warranted.

Specific to the context of computer systems, a more refined definition of "accessibility" can therefore be formulated:

Definition 1.3. *A computer system is fully accessible when (a) any user can access and use all functionality independently*[4]*, (b) when that user can engage in meaningful collaboration about the system with peers, regardless of individual needs, and (c) when all users are provided with an equivalent level of usability.*

1.2.4 Multimodality

Within the context of this work, multimodality is more a characteristic of the actual system rather than an aspect of user interaction. Nigay and Coutaz provide a definition for multimodality from a system centric point of view [103]:

Definition 1.4. *Multimodality is the capacity of a system to communicate with a user along different types of communication channels and to extract and convey meaning automatically.*

This definition clearly indicates that multimodality applies to both input and output modalities. Then what sets multimodality apart from another common term:

[3]When the collaboration involves multiple users accesses the same data using the same instance of an application, this is also referred to as "Computer Supported Cooperative Work" (CSCW).

[4]Either through direct manipulation ("direct access") or indirectly ("assisted access") by means of some form of assistive technology solution.

multimedia? Coutaz defines a multimedia system [32] as a system that allows the acquisition, storage, and distribution of data across multiple media[5]. The relation between the two terms can then be established as presented by Stanciulescu [133]:

Definition 1.5. *A multimodal system is a system with multimedia capabilities that enables semantic data handling.*

1.3 Thesis

This dissertation provides the defence for the following thesis:

> The ability to provide *multiple equivalent representations* of a user interface *in parallel* across *multiple modalities* promotes *accessibility*.

The remainder of this section will provide the motivation for the thesis, working hypotheses, and a declaration of scope.

1.3.1 Motivation

Various aspects of the approach described in this work have been the topic of research in past years. This section will discuss the underlying concepts of the thesis statement, providing motivation for each component.

Multiple representations

The vast majority of user interfaces encountered in daily life are graphical in nature. The appeal of a visually pleasing interface is strong, yet it does restrict user interaction primarily to the visual channel. It also poses a significant hurdle for users with visual impairments as discussed at the beginning of this chapter. One of the influential responses to these concerns is the multimodal UI, a paradigm shift away from conventional Window-Icon-Menu-Pointer (WIMP [28]) interfaces towards providing greater expressive power, naturalness, flexibility, and portability [110].

Multimodality can be leveraged at two different levels:

[5]"Media" refers to physical devices used for input or output, or a physical object used to store data.

- A user interface representations can benefit from supporting interaction across multiple modalities. The various modes of communication complement one another.
- Multiple representations, each supporting interaction across one or more modalities. Users can choose to interact with the system by means of a specific representation. Any such representation in and of itself may be multimodal as well.

With the prevalence of cross-platform applications, the large variety of interaction devices (smart phones, netbooks, laptops, desktop computers, PDAs, . . .), and in support of individual user needs (e.g. visual impairment [5]), support for multiple representations has become an important feature in user interface systems.

Parallel presentation

Collaboration between users is an important aspect of user interaction, be it in the form of assistance, or in the form of working together on a project. In view of the aforementioned large variety of contexts of use, it is obvious that situations will and do occur where collaboration between users with different needs is desirable. A sighted instructor teaching a blind user how to use a computer system is a very common and obvious example [161]. While one can expect the instructor to be very well versed in operating the UI from the context of use of a blind user, it is certainly not the most natural mode of user interaction for the instructor.

Christian wrote on the topic of multimodal user interfaces and accessibility for the blind [29]:

> "Simply giving the blind efficient access to computers should not be the goal of these types of interfaces. These interfaces should enhance the ability of blind users to integrate into the larger community of users. This would enable, for example, allowing blind workers to collaborate with sighted coworkers on projects at the office. The interface should give the blind user a clear 'picture' of what a sighted partner is doing with the system. To this end, the interface should attempt to convey to the blind users a mental model of the system similar to that of sighted users."

Aside from training, users commonly have questions about the operation of an application or system. From an accessibility perspective, it is quite unreasonable to not offer support for collaboration between users with quite different needs. It is also unreasonable to expect all users to be proficient with every modality that can

be used to interact with a system. How can one then ensure that collaboration is not hindered by differences in interaction modality?

The solution that is proposed in this work is to allow multiple representations to be rendered in parallel, allowing e.g. a sighted user to observe visually all interaction that a blind user is performing by means of a UI representation that is more appropriate for the blind user. In reverse, the blind user should be able to observe all interactions performed in a GUI by a sighted user, and that observation should be able to take place through a UI representation that is non-visual.

Functionally equivalent representations

The notion of collaboration expressed in the previous topic clearly depends on both representations (and in general, any number of representations) to be "in sync". This means that if multiple users are observing the system through different UI representations[6], each user must be presented with the same semantic UI state, rendered in each respective representation as needed. In other words, all UI representations should be independent from the actual functional operation of the system.

1.3.2 Scope of the work

The work presented here is limited by the following considerations:

- All systems and applications are assumed to be interactive. This means that all interaction with a system or application is provided by a user rather than an automated system. It is also assumed that user interaction with the system or application will yield output that is easily observed by a user.

  The aim of this work is to provide a solution that is based on describing the user interface from a functional point of view, and while it is not within the scope of this work, it would be possible to generate final UIs for any context of use, including automated system interaction facilities (e.g. automated testing frameworks).

- Multimodal interaction can cover a large variety of modalities, making use of all five major human senses[7]. The research presented here is primarily limited to visual and auditory interactions, and tactile (through Braille) to a lesser extent.

[6] UI representations could differ in terms of the modalities used for user interaction, or merely in how the same set of modalities is used in different ways.

[7] The main human senses are: sight, hearing, smell, taste, and touch.

- The graphical user interface is the standard interface on most computer systems. Most users are familiar with this type of interface, and it is available on almost every computing platform. While a GUI environment is very common, it still often offers a significant degree of complexity by virtue of supporting visualisation by means of multiple graphical toolkits.

  In this work, all representations are essentially equivalent and thus no preference should be given to any. However, for the purpose of comparison, it is assumed that the *natural* representation is visual (GUI). Applications will therefore be introduced with only a visual interface, and the outcome of the work will support multiple equivalent representations.

- For the purpose of the work presented in this dissertation, the overall scope of supported representations has been limited to a visual GUI representation and a non-visual representation. This choice was made for three important reasons:
  - Providing both a visual and a non-visual representation, while ensuring that equivalence between them is maintained, is one of the most extreme scenarios.
  - As described in section 2.2.1, enabling blind users to access a GUI-based computer system is of great importance. The presented work aims to provide the foundation for an approach that may accomplish this important goal in future work.
  - As defined in section 1.2.2, the blind represent a well define group, with well understood needs. It is obvious that the visual channel is not usable for a blind individual, and therefore an alternative communication channel must be used. Also, the vast majority of research that has been conducted on the topic of HCI for people with visual impairments has been focused on the blind.

- Although “facilitation” is hardly a scientific concept, it is quite relevant to the realm of accessibility and usability. During the design and development of user interfaces, choices are made that may have significant impact on the accessibility and usability of a system, and success is often dependent (in part) on how well the system facilitates designing interfaces that will be accessible. Still, in extremis, a developer could e.g. opt to render the entire UI at the level of the application, merely using a canvas widget to display the UI as an image. While this could still be done using the mechanisms presented in this work, it is a violation in spirit of the underlying concepts. This dissertation does not address the complications that arise from such “creative programming.” It is recognised that no system is foolproof in view of the flexibility offered to designers and developers, and that such flexibility is needed to ensure that no arbitrary limits are imposed upon the creative talents of UI designers.

- The primary goal (presented in section 1.3) is to provide a design for a user interface system. It is recognised that performance is not taken into consideration in this work.
- The target audience for this dissertation is the community of HCI researchers and professionals, especially anyone with a focus on accessibility. Professionals involved in advocacy activities related to accessibility of computer systems may be interested in the presented approach in terms of accessibility as a feature versus an add-on.

  Ultimately, the end user population as a whole (regardless of any needs a user might have) is the intended beneficiary of this work.

1.4 Contributions

The main objective of the research presented in this doctoral dissertation has been presented in section 1.3. The research conducted throughout the course of the doctorate program provides the following main contributions:

- **A re-interpretation of the Graphical User Interface concept**
  When researchers developed the original concept of the GUI (also known as the WIMP interface) the full impact of their creation may not have been known. The research presented in chapter 2 shows that the graphical user interface can be re-interpreted as a visualisation of a broader fundamental concept, introduced here as the Metaphorical User Interface. The MUI is specified at the conceptual level, whereas the GUI operates at the perceptual level. In view of recent developments with novel user interaction technologies, a deeper underlying form can be identified as the Conceptual User Interface, capturing the original intent of the GUI design.
- **Validation of the “access to GUIs – not graphical screens” argument**
  Edwards, Mynatt, and Stockton discussed this argument for a paradigm shift in the provision of accessibility for GUI-based computer systems in their 1994 paper [45]. The authors did not provide research-based validation for their claim . In chapter 2 this argument is revisited, and the presented research provides the missing validation.
- **Providing access to the underlying conceptual user interface, not its visual representation**
  This contribution is a derivative of the previous two contributions. The decomposition of the GUI concept into a conceptual component and a perceptual component makes it possible to further refine the “access to GUIs – not graphical screens” paradigm shift. The presented research and

analysis in chapter 2 shows that the accessibility problem should shift from the perceptual layer to the conceptual layer.

- **Survey to determine familiarity of WIMP-based user interface elements and mental models for blind users**
  Various research in the past has been based on the assumption that the underlying metaphor (and derived user interface elements) for the original GUI design is not appropriate for blind users. The survey presented in chapter 3 shows that the blind are in fact quite familiar with these elements and the metaphor they are based upon. However, the survey also shows that the accuracy of the mental model used by blind users is very dependent on the quality of information that the user can access to create the model. Limitations in assistive technology present a significant complication, often augmented by the fact that (at times in relation to the AT limitations) sighted peers do not usually provide good verbal descriptions of the UI, even in response to directed questions.

- **Concurrent representations of the same conceptual model in different modalities**
  The Parallel User Interface Rendering approach presented in chapter 5 introduces a novel approach to providing multimodal user interfaces by breaking through the limitations imposed by derivative models and their representations. The PUIR framework provides each user with an appropriate and semantically equivalent representation of the underlying conceptual UI. The GUI becomes one of potentially many representations of the same UI.

- **Unified processing of user interaction at a semantic level**
  Alternative representations of a user interface cannot be provided as first generation renderings if user interaction semantics are dependent on a specific modality. This has not only been a problem with the accessibility and usability of systems for people with special needs, but also in terms of e.g. ensuring that user interaction with a system can be done both by keyboard and by mouse. The techniques presented in chapter 6 make it possible for all user interaction semantics to be defined in a modality-independent way at the abstract UI level, while supporting multiple representations across modalities.

- **Experimental implementation to validate the designs**
  A fairly minimal version of the PUIR framework has been implemented as an experimental system. The purpose of the implementation is to validate the design of the PUIR approach, and to gain knowledge concerning the complexity of implementing the PUIR system in an existing GUI environment.

- **Validation of the approach**
  Based on the established requirements for this work, derived from the state of the art discussion, the approach presented in this work is to be validated. Internal validation comprises assessment against the requirements, together with an evaluation based on the same criteria used in the state of the art analysis. External validation outlines a test plan approach to perform real-world testing of the approach, and provides information on performing comparison testing also.

1.5 Chapter by chapter overview

The remainder of this dissertation is organised in 8 chapters, providing an in-depth discussion of the problem areas in terms of challenges and requirements, and presenting the contributions of this work.

- Chapter 2 discusses the design principles that form the basis for the graphical user interface concept, and relates them to providing non-visual access to GUIs and the associated HCI issues. The concept of Universal Access (UA) is also discussed as a new perspective on HCI. Relating the GUI concept with UA and HCI issues, the presented research discusses the use of AUI descriptions in the specification of user interfaces.

  The concept of using AUI descriptions as underlying technology for the provision of non-visual access to GUIs was presented (with analysis) in conference papers at HCI International 2005 [145], AAATE 2005 [146], ICCHP 2006 [147], and UIDL 2011 [151].

- Chapter 3 presents a survey of the target user population (individuals who are totally blind). The purpose of the survey is to gain information about how blind users perceive graphical user interfaces, how they construct mental models for UIs, and what specific UI elements present themselves as significant obstacles to user interaction for the blind.

- Chapter 4 provides a discussion and analysis of related works. Based on presented evaluation criteria and identified shortcomings, further requirements for the approach presented in this dissertation are formulated.

  A subset of the content of this chapter has been published in conference papers at ICCHP 2006 [147], ICCHP 2010 [150], and UIDL 2011 [151]. Part of the content is under review for publication in an upcoming issue of the Universal Access in the Information Society journal [152].

- Chapter 5 introduces the Parallel User Interface Rendering approach. First, the design principles that the PUIR approach is based on are introduced. The remainder of the chapter presents the actual design of the framework.

  A significant portion of the content of this chapter was presented in conference papers at HCI International 2005 [145], AAATE 2005 [146], ICCHP 2006 [147], ICCHP 2010 [150], and UIDL 2011 [151].

- Chapter 6 introduces the concept of a single event queue where all user interaction events are posted (from any and all devices) for handling by an abstract UI engine. To make this possible, events must be transformed from the context of the input device to the context of the AUI model.

  The handling of events from different input modalities in the PUIR framework was first discussed in an unpublished paper in 2007 [148], and addressed further in conference papers at ICCHP 2010 [150], and at the 1st International ÆGIS Conference [149].

- Chapter 7 presents the experimental implementation of the PUIR framework in Java. Through a discussion of the various components, this chapter illustrates the practical application of the design discussed in chapter 5, and the handling of the two problems presented in chapter 6. The implementation is validated with a sample application, developed both as a regular Swing application and as a PUIR-based application.

  The experimental implementation of the PUIR framework has previously been presented in conference papers at ICCHP 2010 [150], and at the 1st International ÆGIS Conference [149].

- Chapter 8 provides a discussion about the evaluation and validation of this work. Both internal and external evaluation is presented. Internal validation involves an evaluation of the presented work in view of the requirements, making a determination for each one whether it was addressed effectively and efficiently. External validation involves the development and execution of an evaluation plan. This chapter provides such a plan, although the actual execution of the plan as external validation remains to be completed as future work.

- Chapter 9 presents a summary of the presented research, with some discussion and insights on future work.

Chapter 2

Graphical User Interface, Human-Computer Interaction, and Universal Access

"Excuse me; but as a geologist, you would rather study a book,
some special work on the subject and not a drawing."
"The drawing shows me at one glance
what might be spread over ten pages in a book."
(Ivan Turgenev, "Fathers and Sons", 1961)

This chapter discusses the important relationships between the Graphical User Interface concept, the field of Human Computer Interaction, and the concept of Universal Access.

2.1 Introduction

For many years, the accessibility of GUIs has been approached from the user's perspective. How can one enable a blind individual to operate a graphical user interface? This chapter shows that this has been found to be the wrong way to approach the problem, and instead the question to ask is: "How can one design a user interface such that it can be presented to a sighted user in a visual way, and to a blind user in a non-visual way, while maintaining the same interaction semantics?" The team that invented the GUI did so based on design principles that turn out to be more influential than previously envisioned, enabling

a restatement of the definition of “Graphical User Interface”. This forms the foundation for the remainder of this dissertation. Along the way through this chapter, an important yet unsubstantiated argument by Edwards, Mynatt, and Stockton [45] is shown to be correct based on established research. Surprisingly, their argument for providing GUI accessibility rather than screen accessibility remained an accepted assumption without real backing for over 15 years.

Section 2.2 starts with the dictionary definition of *GUI*, and formulates a more accurate definition within the context of accessibility, drawing on the original GUI design principles. Section 2.3 provides a description of HCI issues that relate to non-visual access of GUIs. Section 2.4 further explores how accommodations can be made for a wider variety of needs, and relates the Universal Access concept to GUIs and HCI. Section 2.5 discusses abstract user interface descriptions as a step towards UA. In closing, the conclusions of this introductory chapter can be found in section 2.6.

2.2 Graphical User Interface

The American Heritage® Dictionary of the English Language, Fourth Edition, defines a Graphical User Interface as:

Definition 2.1. *GUI: An interface for issuing commands to a computer utilising a pointing device, such as a mouse, that manipulates and activates graphical images on a monitor.*

From the perspective of blind users, this very definition contains two elements that tend to raise concerns:

- “a pointing device”: While the mouse is definitely the most common pointer device in use, various alternatives do exist, such as touchpads, touch screens, tablets, haptic input devices, . . . The significant common feature is that they support the “Seeing and Pointing” principle (see page 26).
  Problem: How can a blind user point at something he or she cannot see?
- “graphical images”: This is generally considered to refer to the use of windows, icons, and menus. Albeit not explicitly mentioned in the definition, any text in the UI is also rendered as a graphical image, because computers with a GUI use a bit-mapped display.
  Problem: How can a blind user interpret inherently graphical information and visual metaphors?

Together, these two elements constitute what is known as the WIMP [28] interface. It should be noted that the GUI paradigm is not limited to WIMP interfaces;

advances in technology and understanding of the psychological principles of HCI have led to a new form of interaction: Reality-Based Interaction (RBI) [70]. This work primarily discusses GUI accessibility in view of WIMP interaction, although no explicit limitations are imposed on the original contributions. Section 9.2 provides further information on how the novel approach presented in this work can be expanded upon within the context of alternative interaction models.

2.2.1 GUIs and blind users

The intuitive concern about the (in)accessibility of GUIs, especially in view of the aforementioned definition, is exemplified in a 1989 article in PC/Computing by Herb Brody [18]:

> "[...] the Macintosh uses a graphical user interface that is essentially inaccessible to the blind: the Mac's screen memory contains only raw information about each pixel, and the screen readers can't make sense of the bit-mapped display. The more graphical the interface, the less translatable it is into speech. A screen full of icons, pictures, and overlapping windows becomes gibberish to a screen-reading program seeking clean ASCII code. To the extent that DOS and OS/2 emulate the Mac's graphical interface, the blind will soon be locked out of PCs as well."

This assessment of the impact of the GUI paradigm on blind users' access to mainstream computer technology has been invalidated by history as presented in Section 1.1.2. However, in the late 1980s it was a view shared by many, primarily because of the believe [16] "that blind people cannot make sense of what appears on a graphical computer screen, that they cannot interact with a computer using a mouse and pointing system, and that graphical computers are so different in basic design that even standard text cannot be understood." Why would many subscribe to this notion of inherent inaccessibility, when it is widely accepted that although the world around us is essentially a 3D visual entity, blind people are quite capable navigating it with the use of basic accommodations such as a white cane?

A large contributing factor has been the misconception that a GUI is essentially a bit-mapped display of windows, icons, menus, dialog boxes, ... that the user interacts with by means of a pointing device. Given that the earlier character-based user interface (CUI) could be made accessible by providing a solution that would literally read the content of the screen and present it to the blind user in an alternative format (a screen reader), research and development primarily focused on the problem of deciphering and interpreting the content of the graphical screen.

Boyd, Boyd, and Vanderheiden enumerate the major differences between the CUI and GUI paradigms in their influential paper "The Graphical User Interface Crisis: Danger and Opportunity" [16]:

- *Pixels as the technical foundation for the Graphical User Interface*
  This refers to the technology used to present the user interface to the user: a bit-mapped display. The paper refers to this difference as the most technical and fundamental.

- *The use of visual metaphors: icons, windows, menus, dialog boxes, and control buttons*
  This is listed as the most radical difference because a GUI employs visual imagery instead of textual commands. The images are carefully chosen as visual metaphors.

- *Locational and contextual (formatting) information*
  The spatial placement of visual objects onto the screen often carries meaning, although the sighted user may not realize this. A sequence of images often conveys a sense of order or priority, while the placement of text in a particular location relative to a text input element may imply that the text is a label for that element.

- *Mouse controlled interaction, screen navigation, and random access*
  This refers to the pointer device that is mentioned in the definition of a GUI.

- *The standardised interface*
  While the paper states that the introduction of GUIs facilitated the ideal of one interface for all applications, reality is quite different, especially on UNIX-type systems. The availability of multiple distinctly different graphical toolkits limits the scope of any standardisation effort. Still, for each toolkit a standard API has been made available that application developers utilise.

It is important to observe that these differences between CUI and GUI are primarily stated from the user's perspective, i.e. from the perspective of a person who is confronted with the presentation of the UI as a GUI. Screen readers therefore were designed to try to make sense of graphical information as it appeared on the screen while effectively ignoring the existence of the pointer device. Research efforts took place to provide useful alternatives for blind users such as tactile mouse devices [16] and tactile tablets [158] but without mainstream adoption.

2.2.2 GUI accessibility vs screen accessibility

When the assistive technology solution for non-visual access to a GUI involves interpreting the graphical screen content, the blind user is effectively forced to learn the visual metaphors that are used and to map the visual presentation onto the non-visual mental model of the user interface. This approach is not only counter-intuitive; it is also more limited. Edwards, Mynatt, and Stockton make a strong case for providing access to the GUI instead of providing access to the screen content [45]: "[...] the power of graphical user interfaces lies not in their visual presentation, but in their ability to provide symbolic representations of objects which the user can manipulate in interesting ways." Although this observation seems beyond doubt, the authors did not provide references to support their argument. It is therefore prudent to validate their claim through analysis of past research, commencing with the original design principles for the graphical user interface paradigm.

2.2.3 GUI design principles

The historical overview in section 1.1.2 lists Xerox PARC as the birthplace of what is now known as the *Graphical User Interface*. The design team that pioneered this novel way to interact with computer systems employed a very scientific methodology, approaching the problem using four fundamental principles based on cognitive psychology [8]:

- Users should be presented with an explicit model of the system, that is both familiar and consistent. It should build upon known objects and activities that users already employ in daily life.
- Based on a well-known psychological principle ("recognition is generally easier than recall" [3]), locating a visual entity and pointing at it is a powerful form of interaction, because it is much easier than recalling a name and typing it.
- Users should be presented with commands that are uniform across multiple contexts that have corresponding actions. Therefore, a delete command could delete a character in a word, delete a word in a text, delete a line from a document, delete a line from a drawing, or even delete a file from a directory.
- The screen should always present the current state of the object that the user is manipulating. This concept has been named: "What-You-See-Is-What-You-Get" (WYSIWYG).

These fundamental principles drove the formulation of eight design goals for the Star User Interface at Xerox PARC [130]. The following sections provide an explanation of each goal.

Familiar user's conceptual model

A conceptual model is to be defined, capturing concepts that the user is already familiar with. This mental model enables the user to understand and interact with the system. Employing concepts that are based on analogies or metaphors of the physical world has the distinct advantage that the user can relate to them in a non-technical manner.

The physical metaphor for sending mail could be implemented as moving a document to an "outbox" icon. The document contains the recipient's address. The physical world would require us to duplicate the document if we wanted to send it to multiple people, whereas the electronic version allows us to simply list additional recipients in the document.

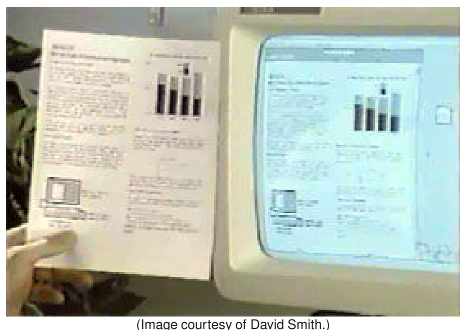
Although the system does not impose the limitations of the physical world, the metaphor remains valid as a familiar concept to the user.

Seeing and pointing

As mentioned earlier in this chapter, cognitive psychology teaches us that it is easier to see something and point at it than it is to recall a word and typing it. Research has also shown that conscious thought depends heavily on concepts in the short-term memory, and that the capacity of short-term memory is limited. The visual presentation of the user interface serves as a "cache" for the short-term memory, decreasing the load significantly [4, 93]. It also reinforces the analogy with the physical world, allowing the user to see the display as "reality". The user interface as it is seen by the user matches the mental model: objects are defined by their visual properties, and actions by their effect on the visualisation.

What-You-See-Is-What-You-Get (WYSIWYG)

When manipulating documents meant to be printed, there is a significant advantage to being able to see an accurate rendition of the printed page. A WYSIWYG-based system accomplishes this by means of bit-mapped displays. Manipulations to a document are represented immediately, so that the user can examine the appearance of a page on the computer screen, and continue making



(Image courtesy of David Smith.)

Figure 2.1: WYSIWYG: Star User Interface

changes until it is satisfactory. The printed page will be virtually identical to the displayed image (see Figure 2.1).

In general, being able to visualise[1] the result of user interaction immediately strengthens the user's perception of the conceptual model.

Universal commands

David Smith, et al. stated [130]: "Just as progress in science derives from simple, clear theories, progress in the usability of computers is coming to depend on simple, clear user interfaces." One important way to accomplish this is to define a generic set of fundamental commands or operations that are widely applicable. Not only does this reduce the overall size of the command set, but it also provides an important level of consistency to the user. Once a concept is known, it can be applied everywhere, thereby making it easier for users to develop a mental model of the system.

It should be noted that whereas Star [131] provided a special key on the keyboard for each generic command, current systems no longer utilise such custom hardware. This universal commands design goal is however still applicable within the context of interaction metaphors.

Consistency

Consistency is an important goal for any design, yet it is not always easy to achieve, especially in an environment that is created as an analogy with the physical world. It is also not always desirable from a user friendliness perspective.

[1] Or more generically: observe.

Consistency requires that objects in the system behave in the same way wherever they are encountered.

One consideration must be model dominance. The analogy with the physical world is the foundation for the user's understanding of the system, and for learning new ways to apply known operations.

Another important consideration is pragmatic in nature: it is a bad idea to implement functionality that generally performs operations that users do not expect, especially if it may result is loss of work items. When a user sends a couple of pages to a colleague by postal mail, there is a chance that the mail item will not arrive. While this is fortunately a rare event, people have learnt that the possibility exists. When a user sends an electronic document by e-mail to a colleague, there is a practical assumption that the document will not be erased from the sender's system at the time of sending the e-mail, even though the analogy with the physical world would imply that.

Simplicity

The best designs tend to be more simple in nature. Within the context of the GUI paradigm and the variety of potential users, simplicity is not easy to define. Novice users may see it as a focus on ensuring that it is easy to perform most operations. Expert users are more likely to see simplicity as a means to increasing efficiency. The two viewpoints are not quite compatible.

Simplicity can be achieved in a way that is likely to satisfy all users by enforcing consistency, and thereby simplifying the mental model that the user must understand in order to operate the GUI.

Another means to simplicity is minimising redundancy. If there are multiple ways to accomplish the same task, the overall user interaction becomes more complex without real benefit.

Modeless interaction

David Smith, et al. quote a colleague (Larry Tesler) for the definition of a UI mode [130]:

Definition 2.2. *A mode of an interactive computer system is a state of the user interface that lasts for a period of time, is not associated with any particular object, and has no role other than to place an interpretation on operator input.*

Modes are typically used to address a need for overloading functionality onto a single operation, and should therefore be introduced with caution. Common examples are found with text editors that have explicit text input and text-based command invocation modes, and with mode-driven focus management where text input is typically directed to the UI element that has focus *unless* a specific mode is active, in which case the text input is redirected to a specific element other than the one that holds focus[2].

It is not always possible to avoid modes, and neither should they be considered a design flaw by default. Some systems allow for a user assistance exploration mode where UI elements can be selected in order to access their documentation. This is a very helpful use of modes.

User tailorability

The individual needs of users are impossible to predict, regardless of the complexity of a system. User tailorability can be implemented by allowing the user to extend the system with custom features, and it can also be provided in part by means of customisation options.

2.2.4 Towards a new definition for GUI

The design principles behind the original concept of a graphical user interface enumerated in the previous section identify the conceptual model that the GUI is based on as the most fundamental. Several other principles are formulated to further enhance the mental image that the user develops. Even the most specifically visual aspects of the GUI are used to strengthen the perception of the conceptual model. The established metaphor for the GUI paradigm was (and still is) defined as a physical office, with the top surface of an office desk (the “desktop”) as one of its most prominent features.

With this renewed focus on the importance of the conceptual model as underlying design principle of the GUI paradigm, a more accurate definition can be formulated:

Definition 2.3. *GUI: A user interface based on a conceptual model using familiar concepts, presented to the user using simple, consistent visual metaphors rendered in real-time on a bit-mapped display.*

The visual representation provides the user with visual metaphors and a way to interact with these metaphors (the pointer device). It is however the underlying

[2]This is often used for so-called modal dialogs.

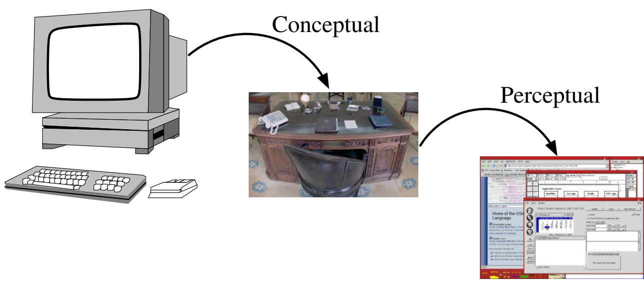


Figure 2.2: Two layers of metaphor; two mappings

model that defines the semantic interpretation of the UI. As such, there are two layers of metaphor involved in the design and implementation of a graphical user interface as described by Gaver [50]:

- Conceptual: This layer consists of metaphors that relate concepts of the computer environment (hardware, software, ...) to the model world that exists in the mind of the users. Figure 2.2 illustrates how entities within the computer effectively can be thought of as representative of a physical desk top. Technical concepts are mapped onto objects that the user is familiar with in the physical world, and thereby the metaphors allow the user to apply generalised knowledge about objects (especially in terms of how they can be used) to the less familiar world of computing.
- Perceptual: The model world that the user envisions is not tangible. Further mapping of this world onto the computer display by means of visual metaphors results in a reality that the user can interact with. Figure 2.2 shows the mapping from the (imagined) physical desk top to the visual presentation that is commonly known as the GUI desktop.

In terms of accessibility, it is important to observe that only the perceptual mapping involves visual metaphors. The model world that is created as a result of the conceptual mapping is defined in terms of manipulatives. Edwards, Mynatt, and Stockton stated [45]: “At the highest level, we can describe an interface in terms of the operations it allows us to perform in an application. [...] It is the operations which the on-screen objects allow us to perform, not the objects themselves, which are important. This level is the semantic interpretation of the interface.” The Mercator project however still operated based on the programmatic implementation of the GUI by means of the X11 graphical toolkit.

The model world that the conceptual metaphors relate to is effectively an abstraction of the physical world. This observation allows for a further specification of the definition of a graphical user interface:

Definition 2.4. *GUI: A user interface presentation that utilises visual metaphors to render and interact with an abstract user interface that conceptualises the operational characteristics of an application.*

This definition satisfies the requirements that are expressed in the original design concepts for a graphical user interface. It is also consistent with the notion that one should provide access to GUIs, not the content of graphical screens [45]. The analysis presented in this section makes it possible to validate the Edwards/Mynatt/Stockton argument. The focus of providing accessibility should therefore be on the interaction between the user and the computer rather than the visual representation.

2.3 Non-visual access to GUIs: HCI concerns

The field of Human-Computer Interaction is the study of the interaction between people and computers. It specifically addresses all aspects of the user interface, be it in hardware, in software, or at a semantic or logical level. It also considers variables on the side of the user, such as limitations, accommodations, and abilities. Blind users accessing applications or systems with a graphical user interface is therefore a problem context that certainly belongs in the HCI field.

Given the prevalence of graphical user interfaces in daily life, blind users do face a common lack of accessibility. Interpreting Definition 1.3 presented in section 1.2.3 in the context of blind users and a GUI system, the problem can be described as:

> A computer system with a graphical user interface is fully accessible when (a) a blind user can access and use all functionality independently, (b) when that user can engage in meaningful collaboration about the system with peers, regardless of individual needs, and (c) when the user is provided with a level of usability that is equivalent to that provided to sighted users that use the graphical user interface directly.

Mynatt, Weber, and Gunzenhäuser formulate five important HCI concerns that need to be addressed in order for an approach towards providing non-visual access to a GUI to be deemed a viable solution [100, 57], based on the works of Edwards and Vanderheiden, et al. [37, 156]. These concerns can certainly be taken as necessary requirements under the accessibility problem description given above, yet it is recognised that they do not constitute a sufficient set of requirements.

2.3.1 Coherence between visual and non-visual interfaces

Collaboration between sighted and blind users requires coherence between the visual and non-visual presentations of the user interface. The mental model of how to interact with an application must be sufficiently similar for both user groups to allow clear communication about how to accomplish tasks within the context of an application. Any user should be able to observe the actions of another, regardless of what interface is being used by either user.

Weber further specified coherence in two different forms [159]:

- *Static coherence*: A mapping between all visual and non-visual objects, which lets users identify an object in each modality. This form of coherence is most commonly the primary focus for screen readers.
- *Dynamic coherence*: A mapping that defines for each step in interaction within the visual modality one or several corresponding steps within the non-visual modality. This form of coherence satisfies the “Equivalence” CARE property presented in section 4.1.3. This form of coherence is favoured in non-visual toolkits for longer periods of interaction.

2.3.2 Exploration in a non-visual interface

Non-visual modalities (auditory and tactile) are limited in their ability to provide information to the user in part due to their predominantly serial nature, whereas a visual user interface often can provide information in parallel in a very efficient way. A screen reader implementation must provide specific mechanisms to explore the non-visual interface. Given that the GUI is capable of providing information by means of spatial properties of UI elements (often beyond the scope of a single application), non-visual alternatives must also be provided.

2.3.3 Conveying semantic information in a non-visual interface

The inherently graphical nature of GUIs commonly leads to presenting semantic information in a strictly visual way: icons, object attributes, appearance, . . . A non-visual presentation must be able to convey relevant[3] aspects of that information in an alternative format, because it is effectively part of the semantics of the application.

[3] In this context, “relevant” means that the information carries meaning at the conceptual level.

Mynatt and Weber phrased this concern as “conveying graphical information in a non-visual interface” because their work was focused on providing non-visual access to existing graphical user interfaces. Therefore, their approach had no choice but to capture and interpret graphical information for the purpose of presenting it in an alternative format to the user. Advances in the HCI field support the more generic phrasing of the concept presented here, and it will be used throughout this work as well.

2.3.4 Interaction in a non-visual interface

Interaction within a GUI is often based on visual idioms (clicking buttons, moving sliders, dragging objects, ...) whereas a blind user requires specific non-visual forms of interaction.

2.3.5 Ease of learning

The introduction of non-visual access to GUIs should not be a major obstacle for blind users. The success of the GUI concept depends in part on the intuitive nature of that environment, and on the fact that users can share knowledge easily and learn from one another. Ease of learning can be accomplished by ensuring that the non-visual UI is sufficiently intuitive to its target group, and that sighted and blind users can share a sufficiently similar mental model of interaction semantics.

2.4 Universal Access

Stephanidis provides a description for Universal Access [134]:

> “[...] Universal Access refers to the global requirement of coping with diversity in; (i) the target user population (including people with disabilities) and their individual and cultural differences; (ii) the scope and nature of tasks; and (iii) the technological platforms and the effects of their proliferation into business and social endeavours. [...] Though there have been efforts in the direction of specifying the attributes of Universal Access, we are still far from an operational definition of the term. [...] As a result, Universal Access remains an abstract goal, rather than a well-articulated engineering target.”

A popular alternative description is “interaction by anyone, anywhere, and at any time.” UA is not defined within a specific context but rather applies to all aspects of life which is one of the reason why an operational definition is still lacking. The underlying concepts have long been recognised, and to some extent aspects of UA have been codified through legislative procedures, such as the Communications Act of 1934, Section 508 of the 1986 Federal Rehabilitation Act, and the 1992 Americans with Disabilities Act (ADA).

2.4.1 Universal Access: a new perspective on HCI

In the development of software, the user interface component handles the direct interaction with the user and it is here where UA introduces new perspectives. The all-inclusive principles that drive this paradigm shift broaden the context of HCI. Whereas in the past direct access or access through add-on (assistive) technologies have been deemed adequate to address the needs of users, Universal Access implies that accessibility should be an integral part of the design rather than merely a concern. From the onset of user interface design, the broadest interpretation of user population is to be taken into consideration [155, 7, 135, 143]. This aspect is often called *Design for All* (or *Universal Design*).

While the all encompassing nature of UA has drawn significant criticism in view of perceived complexity, impracticality, and increased cost, it is important to note that there is no expectation that a single solution is appropriate for everyone [7, 135]. Instead, UA promotes a user-centric approach. People also often do not realize that many products in everyday life were originally designed as an assistive technology for people with special needs. The telephone originated from research on hearing aids. As an alternative to much more bulky braille transcribed books, the audio cassette was invented as an alternative format for the blind. People who could not use a fountain pen due to dexterity limitations were helped by the invention of the ballpoint pen. These are three representative examples where the general public certainly benefited from the research and development concerning assistive technology [42].

2.4.2 High level requirements for universal usability

As discussed in section 2.4.1, Universal Access has triggered an important paradigm shift in HCI by putting a strong focus on user-centric UI design and development. Where previously users would accommodate the limitations of technology in products and systems with a rigid UI, advances in HCI drove the introduction of user preferences as a way to provide a limited level of accommodations for users’ wants. UA has changed the perspective towards the

technology accommodating the limitations of the the user population. HCI aims to provide support for a wide range of human abilities levels across the modalities of interaction while maintaining a high degree of flexibility towards user preferences. Therefore, focus is both on users' needs and wants.

Trewin, Zimmermann, and Vanderheiden formulated high level requirements for universal usability [143]. In their terminology, they refer to programs and devices as *targets*, devices that are used to access and interact with targets are *controllers*, and the combination of the user, the environment, and the controller is referred to as the *delivery context*. The UI that the user interacts with is considered a *concrete user interface*.

The remainder of this section presents the requirements, and how each can be interpreted within the context of this dissertation.

Applicability to any target

Within the context of user interfaces for computer systems, Universal Access implies that the UI can be represented on any relevant target, such as home and office computers, appliances, information kiosks, ... There is no clarity on whether this requirement implies that any user should be able to access a specific UI representation on any relevant target, especially given that the target is not considered a component of the delivery context.

While the ideal of UA would likely include a scenario where any user should be able to access any target, practical considerations often limit the ability to accommodate all combinations. Section 2.4.1 stresses the fact that UA does not imply that a single solution is appropriate for everyone. Within that context, this requirement supports the notion of specialised targets to accommodate specific needs, and it puts an expectation on both the UI design and the targets to support being able to present the UI an any relevant target.

A typical example can be found with electronic book readers. A sighted user might use a very book-alike device using E-Ink technology, whereas a blind user may opt for an audio-only device. If the two devices use the same software, with a graphical user interface for one and a non-visual audio-based interface for the other, this requirement would still be met if at a semantic level the UIs are equivalent, aside from presentation.

Applicability to any delivery context

A user interface that satisfies the principles of UA takes into consideration all aspects of the delivery context:

- User: age, gender, culture, education, health, abilities, expertise, ...
  Note that there can be considerable overlap between these facets of the user profile.
- Environment: noise level, lighting, population, level of privacy, local vs remote, ...
- Controller: direct access, assistive technology, recognition-based interfaces, ...

It is important to note that within the context of this dissertation, "user interface" does not refer to a specific presentation and interaction model, but rather to the underlying conceptual model of user interaction. This requirement therefore refers to the ability to provide a concrete user interface for any applicable delivery context. Various existing approaches to accomplish this are discussed in chapter 4.

Personalisation

Usability requires that the user is able to personalise various aspects of the user interface to accommodate specific individual needs. Often this is part of satisfying the previous requirement to support any delivery context. Some approaches to UA provide for automated personalisation, usually with support for the user to make further adjustments.

Trewin, Zimmermann, and Vanderheiden do not explicitly address user preferences, and in fact the examples given for the personalisation requirement are all aspects of the delivery context and therefore can be addressed as such. It seems prudent to associate user preferences with this requirement. Personalisation is a form of accommodation for user's "wants" while the delivery context can be interpreted as a reflection of the "needs".

Flexibility

This requirement seems to be unnecessary within the context of Universal Access because it is covered already by the applicability to any delivery context and personalisation. The examples provided in [143] can all be interpreted within the scope of the delivery context. For this reason, this dissertation does not consider flexibility as a separate requirement for UA.

Extensibility

Given the wide variety of possible targets, contexts, etc... a mechanism should exist to allow target providers and third parties to provide specific extensions to address (and resolve) missing functionality and to augment existing implementations. This should be done in a manner where extending a specific concrete user interface does not negatively impact support for other targets and/or delivery contexts.

Extensibility should also cover support for altering or augmenting the overall underlying user interface in a way where the changes are available on all targets and in every delivery context. Any such change should consider the variety of targets it can affects. Being able to provide extensibility and alteration both on the level of the underlying UI and the target (and handling of the delivery context) is a powerful concept.

Simplicity

The complexity of the user interface has a direct effect on the complexity of providing support for a wide variety of targets and delivery contexts. It is therefore important to address the aforementioned requirements with a design that is optimised towards simplicity. This does not mean that the user interface itself must be designed to cater to the less experienced user, or that product functionality should be kept simple.

Simplicity in the context of Universal Access is relevant at multiple levels:

- Simplicity in the underlying UI design, which positively impacts the development effort on the side of target providers. It also improves the ability to support a wider variety of delivery contexts.
- Simplicity in the concrete user interface implementation, providing support for personalisation and extensibility.
- Simplicity in user interaction, as discussed in section 2.2.3.

2.4.3 GUI design principles vs UA requirements

Providing alternative representations of GUIs is clearly a topic within the realm of Universal Access, and the perspective it puts on HCI. In view of the new definition of GUI (Definition 2.4, page 31), the design principles listed in section 2.2.3 can be revisited within the context of the UA requirements presented in section 2.4.2,

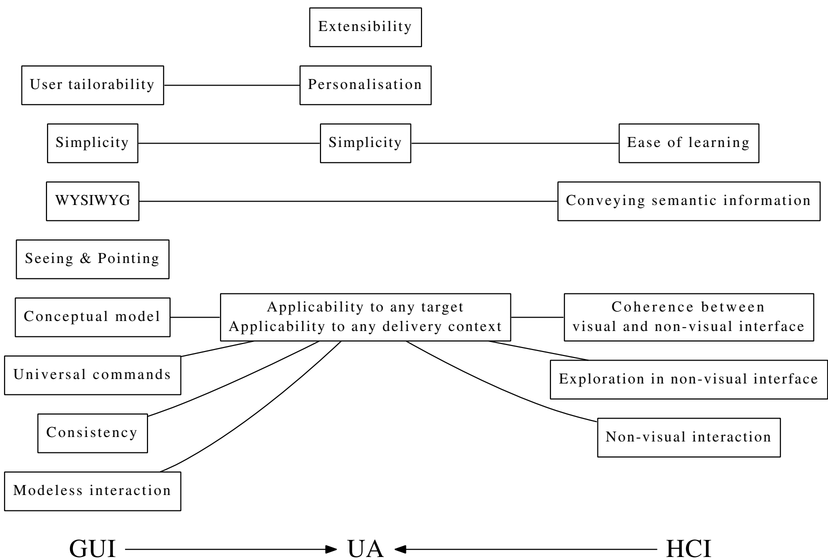


Figure 2.3: Relating GUI to UA and HCI
The figure above shows the correspondence between GUI design principles, UA requirements, and HCI issues related to non-visual access to GUIs

while also considering the HCI concerns related to non-visual access to GUIs (see Figure 2.3):

- *[UA] Applicability to any target and any delivery context*
  This requirement is the heart of the "anyone, anywhere, and at any time" concept. The broad range of contexts that needs to be supported requires product providers to either invest substantial amounts of money in the development of alternative user interfaces, or they must develop user interfaces that can adapt to a variety of contexts [143]. The latter solution has been adopted by industry and research institutions alike.

  The requirement to be able to provide user interface representations for various different contexts is a good fit for abstract user interface descriptions. This technique supports the well established development paradigm to separate presentation from application logic and it has been identified as a necessary concept for providing the flexibility to render the UI for any target [73, 145]. See section 2.5 for a more detailed discussion.

  Gaver described two layers of metaphor for user interfaces (not specifically for GUIs): conceptual and perceptual [50]. The conceptual metaphor relates back to the original GUI design principle of the conceptual model, further augmented and reinforced by the notion of universal commands and

consistency. Both operate at the level of the fundamental user interface, independent from the presentation. The principle of modeless interaction is also relevant here because it makes it much easier to represent the UI for different targets.

From the perspective of HCI issues that relate to non-visual interfaces, coherence between the visual and non-visual interfaces can be achieved based on the underlying abstract user interface. Exploration in the non-visual interface is also supported by this powerful technique, because it involves exploration of the conceptual model rather than trying to make sense of the visual representation. Finally, it also facilitates non-visual user interaction because the user interaction semantics are defined at the abstract level.

It is important to note that while both multiple GUI design principles and some HCI issues related to non-visual interfaces are associated with this UA requirement, they may not be applicable at the same point in time. E.g. modeless interaction (GUI design principle) and exploration (HCI issue) are generally not compatible because non-visual exploration typically requires a specific interaction mode.

- *[UA] Personalisation*
  The ability to modify aspects of the user interface based on user preferences has been a design principle for GUIs from its inception. It has been identified as a requirement for Universal Access as well.

- *[UA] Simplicity*
  As stated earlier, the best designs tend to be more simple in nature. It is therefore no surprise that simplicity is recognised as a GUI design principle, a UA requirement, and a HCI concern. Everyone from user to developer and designer benefit from a focus on simplicity, through consistency, minimising redundancy, and carefully considered design decisions.

- *[HCI] Conveying semantic information in a non-visual interface*
  The need to somehow convey semantic information that is typically presented visually in a non-visual way seems counter-intuitive, and is not really covered by any Universal Access requirement. It is related to the WYSIWYG design principle in Figure 2.3 because it represents the need to convey accurate information about the presentation of information in a specific modality.

  Word processing is a good example. If the end result of the task that the user is performing is a print document that should satisfy specific formatting requirements, a blind user must be able to obtain accurate information about the visual presentation of the document. Otherwise the blind user is unable to perform the task.

The need to be able to exchange modality specific information can be covered under the UA requirement for applicability to any delivery context, as a function of the environment. However, it has significant implications on the way in which UA is implemented and therefore it will be identified as a specific requirement within the context of this work.

- *[GUI] Seeing and pointing*
  The design principle of a pointer device for user interaction in a graphical user interface is an aspect of the presentation of the UI rather than a fundamental component of the conceptual model that lies at the core of the user interface.

  The graphical user interface concept is based on a well-researched conceptual model: the metaphor of a physical office, providing a model with concepts that users are very accustomed to. These familiar concepts are then presented to the user by means of visual metaphors. As discussed in section 2.2.4, this involved two layers of metaphor. The pointer device is a component of the perceptual metaphor, tightly coupled with the visual presentation.

  This specific design principle and its implications for alternative representations of a graphical user interface will be further explored in chapter 6.

- *[UA] Extensibility*
  The Universal Access requirement of extensibility is not present in the original GUI design, nor is it considered as part of HCI issues that relate to non-visual access to GUIs. Traditionally, graphical user interfaces were static entities, defined programmatically during product development. Mynatt, Weber, and Gunzenhäuser [100, 57] formulated the HCI issues based on traditional GUIs, and therefore also did not consider extensibility.

  While this requirement is not explicitly discussed in this work, the use of abstract user interface descriptions as presented in section 5.3.2 provides a good foundation for supporting extensibility (see also section 2.5.1).

2.5 Abstract User Interface descriptions

The UI design and development paradigm to separate presentation from application logic has been well established years ago. The initial benefits were primarily in the area of software development where graphical designers could focus on the "look" of the product, while software developers focused on the actual functionality. Further advances introduced mechanisms to allow the user to control the "Look & Feel"[4] of the application by means of user preferences. Going further still, Java

[4]The term "Look & Feel" became popular as a result of the very controversial and ground breaking court battle between Apple Computer, Inc. and Microsoft Corporation, from 1989 through 1994. The

allows the developer or the user to choose from a (often small) set of different graphical toolkits that are used to visualise the user interface. All this falls within the category of user tailorability rather than usability and/or accessibility.

Abstract user interface descriptions take things further with the decoupling of presentation and semantics. The AUI does not make any assumptions about what modality is used (visual, tactile, auditory, haptic, ...), describing the UI elements and how they relate to the meaning of user interaction scenarios rather than being involved with the mechanics of the user interaction. The AUI ideally describes the user interface in a target and delivery context independent way. As such, the proper use of AUI descriptions supports Universal Access.

2.5.1 Technical requirements for AUI description languages

Trewin, Zimmermann, and Vanderheiden derived high level technical requirements from the requirements listed in section 2.4.2 [143]. They are reformulated here within the context of providing non-visual access to GUIs, and Universal Access to software applications.

Separation of interface elements and their related data from their presentation

The separation between presentation and semantics has already been identified as a crucial component. This requirement however augments the established UI development paradigm with the stipulation that any data that directly relates to the semantics of the UI element is also decoupled from the presentation. Failure to do so might result in details of the presentation inadvertedly influencing the way the data is represented within the application and the AUI. This would likely impact the ability to support other targets and/or delivery contexts. The expanded requirement still follows the general principle of separation of concerns as presented by Parnas [112].

Explicit machine interpretable representation of all UI elements

The AUI description covers information that the user can manipulate, controls to operate, and information to be presented to the user (be it pre-defined or generated). All of these elements must be present at the abstract level, and

“Look” refers to the visual presentation of UI elements, whereas the “Feel” refers to the interaction semantics of the UI elements.

the target must be able to represent them accurately because they are required components in the concrete user interface.

Ideally, no specialised knowledge should be required for the AUI description to be created, because target and delivery context are not expected to be known at development time.

Explicit machine interpretable semantic relations between UI elements

Dependencies between UI elements carry a high degree of semantic meaning that is traditionally presented programmatically, thereby negatively impacting the ability to support a wide range of targets and/or delivery contexts. Often a UI may contain elements that are only meaningful when specific selections are made in the interface. A customer service application might provide an element to mark whether the user would like to leave a contact telephone number. It is common practice to leave the telephone number entry field immutable unless the user has indicated that he or she would like to leave a number. Including this level of dependency at the AUI level yields better UIs.

Often UI elements also relate to one another in a way where such knowledge would be beneficial to the presentation layer. A common example is the use of text elements to provide labels and descriptions for user input elements. By encoding the coupling between labels, descriptions, and input elements, the presentation components are better equipped to provide a meaningful layout of the UI to the user. More so, it enables the presentation to make these decisions rather than either ignoring the relations or encoding layout at the AUI level, breaking the separation between presentation and semantics.

Flexibility in inclusion of presentation information, while maintaining a target and delivery context independent abstraction

This seems to be a contradiction in and of itself, but it is actually a rather powerful requirement that enhances the HCI experience for the user. While the AUI description must capture the essence of the UI from a semantic point of view, effectively modelling the operational characteristics of the application, it is beneficial to augment it with optional presentation-specific information that assists the target in providing an optimal presentation. Note that this information does not carry any semantic meaning. A basic example can be found in alternative texts for labels, where an abbreviated text may be provided for presentation on targets with small screens. Similarly, a button would be defined in the AUI with a text label, whereas additional information may provide an icon image for targets that support graphical information.

Ideally it should be possible to incorporate presentation information from external sources such as third parties for extensibility, both to augment and to possibly replace existing presentation information. Note that it would not be acceptable for such information to affect the actual abstract UI model.

Support for different interaction styles

The AUI defines user interaction semantics at the conceptual level, independent from target or delivery context. This effectively means that basic layout and flow of UI elements is defined as part of the abstraction. Some delivery contexts may have requirements that affect the layout and/or flow of UI elements.

This requirement has not been addressed within the scope of this dissertation, although consideration has been given to how it could be handled. More information can be found in chapter 9.

2-way communication between target and controller, with support for synchronisation

The user interface may at times need to be responsive to changes on the side of the target (e.g. activation of a control or an external event getting triggered), so 2-way communication is required. The controller sends messages to the target concerning UI events, and the target sends messages to the controller concerning external events and target-specific user interaction.

Some targets may also impose delays in the flow of the UI, either due to technical limitations or deliberately as an accommodation in function of the delivery context. Whatever the reason, there needs to be a way to enforce synchronisation between the AUI and the target to ensure consistency.

2.6 Conclusions

When circa 1974 a team at Xerox PARC developed the concept of the Graphical User Interface, they may not have realized that they actually laid the foundation of a much broader (and powerful) concept: the Metaphorical User Interface. This chapter has shown that the extensive research that led to the establishment of design principles for the GUI concept can largely be applied to UIs beyond the context of a specific presentation. As a result, a new definition for GUI can be formulated, conveying the importance of the underlying abstraction that models a conceptual metaphor of the physical world.

When discussing this new definition with David Smith [129], formerly of Xerox PARC, he brought up a very important observation. Technology advances in the area of user interaction modalities have led to the development of visual interfaces that are undeniably graphical in nature, and thus should be considered graphical user interfaces, and yet they are not based on any metaphor of the physical world. The GUI of the Apple iPad is a good representative of this class of user interfaces.

Further analysis of the user interaction semantics at the conceptual level shows however that while there may not be an underlying metaphor of the physical world, nevertheless the user interface is based on familiar concepts. In that sense, it is certainly a conceptual user interface, albeit without the metaphorical connection.

In that regard, it seems prudent to provide a definition for the graphical user interface concept based on the current more generic state of development.

Definition 2.5. *Conceptual User Interface (CUI): An abstract user interface that models the operational characteristics of an application using concepts familiar to the user population.*

Definition 2.6. *Metaphorical User Interface (MUI): An abstract user interface that uses a metaphor of the physical world to augment a conceptual user interface.*

Definition 2.7. *Graphical User Interface: A user interface presentation that utilises visual metaphors to render and interact with the underlying conceptual user interface.*

Definition 2.8. *Non-Visual User Interface: A user interface presentation that utilises non-visual metaphors to render and interact with the underlying conceptual user interface.*

For many years, it was taken on face value that in order to provide access to a GUI environment one should provide access to the GUI rather than to the actual screen image. While this insight by Edwards, Mynatt, and Stockton [45] seems trivial, it was never substantiated based on established research. Section 2.2 provides the analysis to validate their argument based on the original GUI design principles. They also provide clear support for the insight that the screen image is merely a visual representation of a powerful underlying conceptual model, and that accessibility should be aimed at the underlying model rather than on the representation of choice.

Finally, this chapter provides the important analysis that abstract user interface descriptions are not only powerful tools in the design and development of UIs in general, but they also play an instrumental role towards providing Universal Access.

Chapter 3

Target user survey

"The presence of those seeking the truth
is infinitely to be preferred
to the presence of those who think they've found it."
(Terry Pratchett, "Monstrous Regiment", 2003)

In order to gain a better understanding about blind individuals interacting with graphical user interfaces, a survey was conducted. Participation was solicited from totally blind individuals by means of direct email and pass-through email communication[1]. The main questions that this survey intended to address are:

- How do blind users perceive graphical user interfaces?
- Are the users comfortable with the established metaphors?
- What type of mental model(s) do blind users use?
- Is a verbal description of the UI by a sighted peer useful?
- What type of UI concepts are difficult to interact with?

This chapter first presents the participants to the survey in section 3.1, followed by an introduction to GUI concepts that are commonly known to blind individuals presented in section 3.2. Section 3.3 discusses the survey responses concerning UI elements and concepts. Section 3.4 discusses the mental models used by the blind, and section 3.5 expands upon this in view of assistance from sighted peers. Section 3.6 presents common troublesome UI elements within the context of screen readers. The conclusions from the survey are summarised in section 3.7.

[1] All participants were encouraged to share the survey with anyone else they knew to be totally blind.

Respondent	Gender	Age	Exp.	Occupation	Blind since	Braille/ Speech
P1*	M	35	26	AT specialist	Birth	Both
P2	F	54	18	Data entry	Birth	Speech
P3*	M	42	25	Programmer	3	Both
P4*	M	31	17	AT specialist	Birth	Both
P5	F	54	9	Designer		Speech
P6*	F	35	23	Homemaker	Birth	Speech
P7*	F	44	26	Homemaker	Birth	Speech
P8*	F	33	20	Student	23	Speech
P9	M	27	20	Not employed	Birth	Both
P10	M	18	8	Student	4	Speech
P11	M	25	13	Student	Birth	Both
P12	M	59	15	Not employed	Birth	Speech
P13	M	19	11	Student	Birth	Both
P14	M	29	11	Not employed	Birth	Speech
P15	F	36	24	Student	3	Speech
P16	M	29	20	IT repair	Birth	Both
P17	M	57	38	Programmer	Birth	Speech
P18	M	50	22	Admin. Assist.	28	Speech
P19	F	68	25	Retired	64	Both
P20	M	28	18	Teacher	2	Both
P21*	F	41	24	Homemaker	Birth	Speech
P22*	M	35	16	Supervisor	14	Speech
P23	M	51	8	Teacher	Birth	Speech
P24	M	44	24	Self employed	26	Speech
P25	M	58	40	Analyst	Birth	Both
P26	M	47	23	Student	Birth	Both
P27	F	37	20	Homemaker	Birth	Both
P28	F	48	15	Director	Birth	Both

Table 3.1: Demographic information of survey participants
*Respondents marked * responded to a direct invitation to participate.*

3.1 Participation

An initial group of 10 participants was selected from a larger group of acquaintances based on the following criteria:

- Be totally blind.
- Use computers regularly.

- No involvement with in-depth discussions about this doctoral research.

These criteria were chosen in order to ensure that the survey was targeted at the right user population. Individuals with a visual impairment often retain some usable sight, possibly requiring some form of augmentative assistive technology to compensate the limited visual ability. When an individual has no usable sight, he or she is deemed totally blind[2]. The AT needs related to total blindness are quite different from those related to low vision. Within the context of this work, focus lies with the needs related to total blindness only.

The second criterion aims to increase the likelihood that each individual has sufficient experience operating a computer system to be able to articulate a personal perspective on the questions asked. The evaluation of potential participants based on this criterion was very informal, merely taking into consideration whether the individual was known to regularly communicate by email or to use a social networking website.

The final criterion aimed to avoid bias in respondents. Given that individuals who have been indolved in discussions about the subject matter for the survey may retain a preconceived notion about the underlying questions following any such conversations, it was deemed inappropriate to solicit responses from this group of people.

The survey (see section A.1) was distributed to the 10 selected participants by means of direct email. It was accompanied by an introductory note explaining the purpose of the survey, the need for answers based on personal perception and experiences, and inviting each individual to pass the survey along to others. The intent of the request to invite others to respond to the survey was not only an attempt to solicit more responses, but also to limit any potential bias caused by the selection process. The goal was to limit the contribution of directly invited respondents to no more than 50%.

Out of the initial group of 10 selected participants, 8 responses were received (80% response ratio). An additional 20 responses were received from participants who learnt about the survey from one of the participants. As a result, the contribution from directly invited participants to the results of the survey is no more than 29%.

3.1.1 Demographics

The demographic information of all respondents is presented in Table 3.1, identifying the 8 individuals who responded to the survey that was sent to them

[2] Refer to section 1.2.2 for blindness-related definitions.

directly. The purpose of identifying this select group of people is to enable any reader to analyse any potential bias in the data presented in Table A.2 and Table A.3.

In total, 18 men (64%) and 10 women (36%) responded to the survey. The group of respondents covers a rather wide age range ($\overline{age} = 40.5, \sigma_{age} = 12.84$) with the youngest participant 18 years old, and the oldest 68 years old. All respondents indicated to be quite experienced with computer systems, with a minimum of 8 years of overall computer experience ($\overline{exp} = 19.96, \sigma_{exp} = 7.74$). Based on the reported occupation, half of the respondents are gainfully employed[3], covering a wide spectrum of job functions.

Of the 28 respondents, 18 people (64%) reported that they have been blind from birth. An additional 4 people (14%) stated that they became totally blind before age 5[4], and another 4 people (14%) became totally blind between the ages of 10 and 50, leaving one respondent who became blind at the age of 64. One participant did not indicate when they become totally blind.

Three participants (11%) gained experience using computer systems prior to losing their sight completely.

Concerning the use of assistive technologies, 15 respondents (54%) reported that they use synthetic speech as output modality during their interaction with computer systems, whereas 13 respondents (46%) expressed preference for using both tactile (primarily braille) output and synthetic speech. None of the participants indicated sole use of a refreshable braille display during their interaction with computers.

3.2 Concepts of the Graphical User Interface

As discussed in section 2.2.3, the design of the graphical user interface is (in part) based on the fundamental principle that users should be presented with an explicit model of the system that is both familiar and consistent, and it should build upon known objects and activities that users already employ in daily life [8]. After many work-years dedicated to the design of the GUI concept, concensus was reached on what was considered an appropriate model for an office information system: the metaphor of a physical office [130]. Some of its most prominent features are the top surface of an office desk, traditionally the focal point of a person performing his or her job functions in the office, and objects in its immediate

[3] Gainful employment means that a person performs activities intended to provide them with an income.

[4] For the purpose of this survey, individuals who became totally blind before the age of 5 can be considered blind since birth [87].

surroundings, such as filing cabinets, wastebasket, ... All these features are represented in the conceptual model of the GUI.

This metaphor was decided upon in part because the focus of the design was a user interaction system with visual presentation. Non-visual interaction was not taken into consideration. With the introduction of screen readers, one of the concerns has been that blind users may not be able to *use* a system with a graphical user interface, even if the screen content is *accessible*: an issue of whether blind people can grasp the concepts behind the user interface elements they observe through the screen reader.

Now, many years later, the reality is that blind people can access computer systems with GUIs to a variable extent. The survey presented in this chapter is therefore not focused on whether the blind can work with a GUI, but rather on how well the established metaphors and the conceptual model work.

Sarah Morley wrote “Windows Concepts” [94] in an attempt to relieve some of the concern about the accessibility of GUIs by means of screen readers. In this guide, she provides explanations for the various concepts in GUIs, and how to interact with them in a non-visual manner. It is based on the author’s experiences teaching visually impaired users on how to use MS Windows. Her work has undoubtedly influenced the development of various training materials that have since been released. As Edwards wrote concerning Morley’s work [40]:

> “The components of the [graphical user] interface [...] need no description to sighted people, they can simply be shown them. For people who can never have the experience of seeing or using a GUI the concepts are difficult to describe, but anyone who needs such a description might consult Morley [94].”

It should be noted that as a result of graphical user interfaces having been in existence on personal computers for almost 30 years, understanding of the GUI concepts has largely become collective common knowledge, especially for younger generations. The difference can be found in the following two descriptions for the “Desktop” metaphor [94]:

> “[MS] Windows is based on the ’Desktop Metaphor’ to make accessing your computer feel easier. The principle is based on real-life: that you are sitting at your desk (your screen), which has a filing system in drawers (your hard disk). You can arrange both your desktop and your hard disk however you choose. On some areas on your desk you have a variety of things relating to report writing – a typewriter, a dictionary, other documents you are referencing. In another area on your desk there might be things to do with numerical analysis and storage – data sheets, a calculator, financial reports, a statistics manual. In another

> area on your desk, you might have your appointments diary, and an address book. In a single work session, you might want access to all this information, and so you have spread it all out on your desk, some of it is overlapping, and you can see little bits of all of them, and all of some of them. You might just leave some room by your hands, for your notetaker, for example. [MS] Windows allows you to have all of these activities immediately available to you at the same time, without having to put one activity away before starting the next one. ”

versus

> “The desktop is the area on which all windows appear, and if there are no windows open, only the desktop would be visible. The desktop can be full of many open windows, overlapping, of different sizes and shapes, but it could be fairly neat and clear, and does not have to be completely filled by these application windows.”

Not only is the first description much more verbose – it is also perceptual in its description of the metaphor. The latter description is more abstract, but in the present time, it appears to be the more common way to describe the “Desktop” concept in GUIs. In fact, all but three respondents to the survey presented in this chapter described the desktop in a manner that is equivalent to the latter, more abstract description.

It goes to show that the concepts in the graphical user interface are largely learnt concepts that may have lost some of their original metaphorical meaning through frequent use in daily life. While it is not possible to measure the influence that works like “Windows Concepts” have had on the understanding of GUI concepts by blind individuals, the survey presented here certainly seems to indicate that it is far from negligible.

3.3 UI elements: perceptual or conceptual

The survey asked participants to describe three cornerstone GUI elements based on their personal perception and experience: “Window”, “Button”, and “Desktop”. The goal of these questions was to determine to what extent a participant perceives UI elements from a perceptual point of view or from a conceptual point of view. A perceptual view relates to presentation characteristics of the element, such as shape, size, location, layout, ... whereas a conceptual interpretation relates to the functionality of the element.

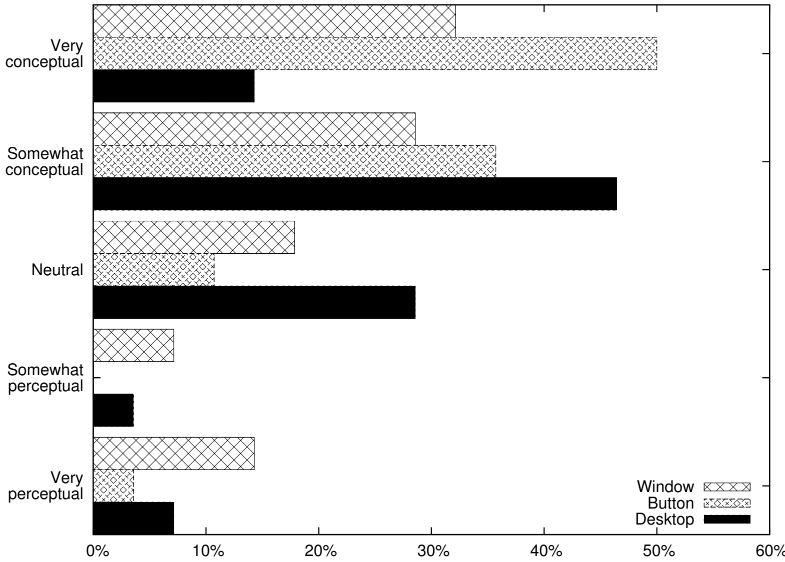


Figure 3.1: UI elements: Perceptual vs conceptual

One respondent (P5) did not seem to understand the questions about UI elements, although answers to other questions indicated a basic understanding of the functionality of the GUI. The overall tone of the submitted survey responses for this person reflected a position of frustration with existing solutions, and a strong desire that “it should just work.” Interestingly, despite the apparent confusion and lack of understanding of GUI concepts, the participant did not indicate in any way that an alternative concept for the UI would be beneficial.

Amongst the other respondents, 79% expressed their interpretation of the “Window” UI element between neutral and very conceptual (Figure 3.1). Their descriptions did refer to a region on the screen in which an application presents all or part of its user interface, but beyond that the primary focus was on the functional aspects of the window and not its appearance (or the physical form that the metaphor relates to).

One participant (P8) expressed a strong perceptual view, providing commentary that explained that she had used computers for 10 years prior to losing her sight, and that she uses a very visual mental image of the user interface when operating a computer. She wrote (translated from Dutch): “I try to imagine [it] based on how it was visualised on the screen when I was still able to see.”

Answers from respondents who indicated expertise in areas like assistive technology and training included perceptual aspects, often in conjunction with

a reference towards what sighted people might perceive.

When asked to describe a “Button”, the answers were more extreme. All but one respondent (96%) expressed a view between neutral and very conceptual (Figure 3.1). Participant P8 again expressed a strong perceptual view, for the reasons stated above. Respondents with a technical background again included some aspects of the presentation as sighted users would view a button, but the interaction was clearly identified by all users as a trigger for an operation, and only two respondents actually referred to activation as “clicking”.

The survey explored the understanding of the “Desktop” metaphor specifically, because of its common use in the underlying conceptual model for GUIs. It is a more abstract concept because one does not typically interact with it, but rather with objects located on it. Similar to the window and button concepts, 89% of the respondents indicated a view between neutral and very conceptual (Figure 3.1). While most descriptions focused on the functionality of the desktop, some did try to describe it as an area that holds other objects. One respondent expressed the notion that it is the main window of the system.

There was clear consensus on it being a container that is central to locating other entities in the system[5].

While only three elements of the GUI metaphor were asked about, the response is overwhelmingly in favour of a conceptual interpretation of the concepts, as evidenced by the chart shown in Figure 3.1. The scoring[6] discussed in section A.2.1 expresses this conclusion more formally (scores are a scale ranging from *very perceptual* (1) to *very conceptual* (5)):

- Window: $\overline{score} = 3.57, \sigma = 1.40$
- Button: $\overline{score} = 4.29, \sigma = 0.94$
- Desktop: $\overline{score} = 3.57, \sigma = 1.03$

3.4 Mental models

Kurniawan, et al. surveyed visually impaired users in view of how they use mental models for interacting in a graphical user environment [80, 81]. Their study determined three categories of mental models:

[5] The term ’locating’ as it is used here does not imply any representation dependent context.

[6] It is important to note that due to practical constraints, the survey answers were scored solely by the author. It is therefore possible that unintentional bias is reflected in the scores. Further analysis with multiple independent scorers is left for future work (see section 9.2).

- *Structural*: Models in this category are characterised by a strong emphasis on how elements in the UI are arranged, i.e. the layout of the UI. In their study, three out of five participants were identified as using a structural mental model.
- *Functional*: These models are based on sequences of operations and commands, independent from the on-screen configuration of the UI. One participant was identified as using a functional mental model.
- *Hybrid*: Models in this category are based on both structural and functional information. One participant was identified as using a hybrid model.

The logical conclusion based on the distribution of participants across the three categories would be that in general, blind users tend to construct a structural mental model for a UI. Kurniawan, et al. report on a followup study (with four of the five original participants) concerning free recall of operations. The results indicated that blind users seem to use a functional mapping for menu items rather than a structural one. It is therefore appropriate to conclude that the users in this study operate using a hybrid mental model, despite the earlier conclusion.

It may be premature to generalise the findings of the study due to the very small number of participants. On the other hand, the three categories of mental models identified by Kurniawan may have been an incorrect starting point altogether. After all, whereas structural models are concerned with elements and how they are laid out, the functional models are concerned with sequences of operations carried out on those elements. It seems unlikely that one could reason in the context of a functional mental model without incorporating the information captured in a structural model. The operations carried out on elements in the mental model are at a minimum in part influenced by the structural composition of the model.

The initial hypothesis of mental modelling was proposed by Craik, stating that people reason by carrying out thought experiments on internal models [34]. Johnson-Laird provided further analysis of this concept, and essentially expressed that a mental model is a structural analog of a real or imagined situation, event, or process, created by the mind in the course of reasoning. The concept of 'structural analog' essentially means that the mental model consists of a representation of the spatial and temporal relations among events and entities, and of the causal structures connecting them [71, 101]. Based on this definition of a mental model the expectation would be that a hybrid of the structural and functional models as studied by Kurniawan is found to be the most accurate categorisation for mental models used by the blind in operating a graphical user interface. Despite the small number of participants, Kurniawan, et al. found evidence to support this hypothesis. The survey reported on in this chapter will also be used to test this hypothesis.

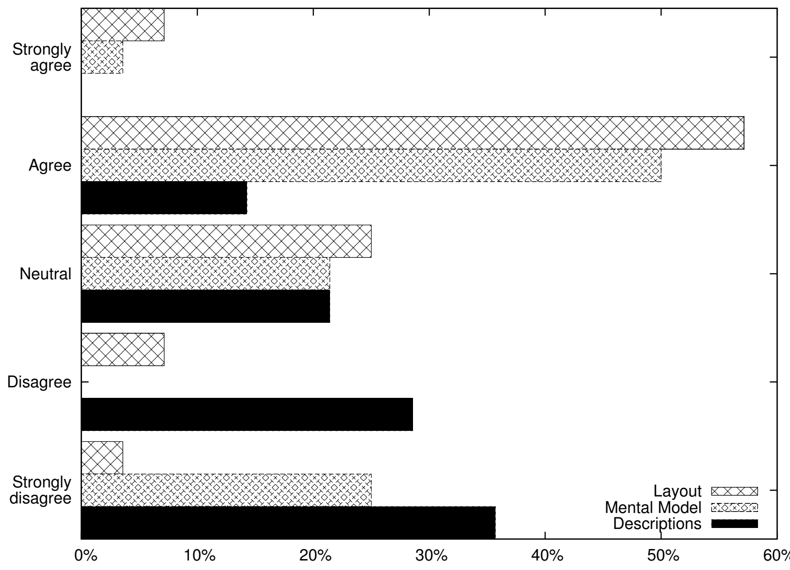


Figure 3.2: Importance of layout, mental model, and verbal descriptions

More importantly however, Kurniawan, et al. reported that the users' mental model is dependent upon the information provided by the screen reader rather than the actual UI, and therefore users may unfairly associate usability problems with the UI rather than recognising the possibility that limitations in assistive technology are the real cause.

This significant problem with establishing an accurate mental model offers support to Landau's position that it is not realistic for an average blind user to create an accurate mental model for a complex layout without some assistance from sighted peers [84].

The survey discussed in this section included questions to study the mental model used by the blind, how it was constructed, and to identify possible additional problems with the creation process. Figure 3.2 provides a chart concerning the scored responses to three questions:

- How important is it to know the layout of elements in the user interface?
- How much does a user rely on a mental model when operating a GUI?
- How useful are verbal descriptions by sighted peers in the construction of a mental model and/or the operation of the UI?
  This question is explored in more detail in section 3.5.

The scoring discussed in sections A.2.2 through A.2.4 provides the following statistical parameters (scores are a scale ranging from *strongly diagree* (1) to *strongly agree* (5)):

- Importance of layout: $\overline{score} = 3.57, \sigma = 0.88$
- Importance of a mental model: $\overline{score} = 3.07, \sigma = 1.30$
- Usefulness of descriptions: $\overline{score} = 2.14, \sigma = 1.08$

Seven respondents to the survey (25%) indicated that they did not in fact create a 'mental image' of a user interface while they explore it. This seemed to be a rather unusual answer and further analysis of answers to other questions confirms that this is likely a misinterpretation of the question because the respondents indicated that they do explore the UI prior to interacting with it, and that the primary information they look for relates to how UI elements are positioned in the UI. It seems appropriate to conclude that these participants do in fact construct a mental model, even if they are not necessarily consciously aware of that fact[7].

All but three respondents (89%) indicated that they explore the UI prior to interacting with it, for the purpose of identifying both layout and functionality. One person specifically identified this as a potential problem because some UI elements are known to trigger functionality when they gain focus rather than requiring an explicit activation.

The exploration of the UI is primarily focused on identifying the elements that comprise the UI, their logical placement[8], and the relationship between elements. This constitutes building a hybrid mental model.

One participant stands out by virtue of indicating that they do not believe that there is any point to exploration when it is known that coherence between the visual presentation of the UI and the OSM is all but present. Answers to other questions indicate that this user has a strong focus on creating a mental model that is an accurate representation of the visualised UI. Based on this analysis, it is fair to conclude that this person strongly favours a structural mental model.

Four participants in the survey noted that their mental models are likely to be inaccurate, and one respondent particularly identified the fact that there is no correlation between what is shown on the screen and what is presented by the screen reader as an important reason.

[7] The survey shows evidence of similar circumstances with most respondents. The term 'mental image' may have been interpreted as implying some aspect of visualisation, and perhaps should have been substituted with 'mental model', although that concept might not have been known to or understood by all participants.

[8] As opposed to physical placement in the visual representation. Logical placement is often referred to as *focus traversal order*.

		Descriptions			
		Agree	Neutral	Disagree	
Layout	Agree	2	6	10	18
	Neutral	2	–	5	7
	Disagree	–	–	3	3
		4	6	18	

Table 3.2: Importance of layout vs usefulness of descriptions

3.5 Assistance from sighted peers

In view of the findings of Kurniawan, et al. [81] and the opinion expressed by Landau [84] (see section 3.4), participants in the survey were asked about assistance from sighted peers in the form of verbal descriptions of the UI, regardless of whether information was provided spontaneously or in response to direct questions. If one accepts the notion that it is not realistic to expect that a blind user can construct an accurate mental model without some assistance from sighted peers, four important questions come to mind:

- How important is it to have knowledge on the layout of the user interface?
- What types of information are requested from sighted peers to improve the accuracy of the blind user's mental model?
- How well do sighted people describe aspects of a user interface?
- Does the information provided by sighted people improve the accuracy of the mental model?

When asked about the importance of knowing the layout of the user interface, 64% of respondents indicated a higher than neutral importance. At the same time, 64% of respondents expressed a less than neutral position on the usefulness of descriptions provided by sighted peers. Table 3.2 provides a breakdown of the responses, relating the importance of layout aginst the usefulness of descriptions by sighted peers. Of the 18 respondents who agree knowing the layout of the UI is important, 10 (55%) do not agree that descriptions by a sighted peer are useful in understanding the layout, while 6 respondents (33%) are neutral on that topic. Respondents who disagree that layout is important wholeheartedly disagree that descriptions are useful either. Those who expressed a neutral opinion concerning the importance of layout indicate a 29% to 71% split on agreement vs disagreement on the usefulness of descriptions.

Perhaps the most significant statistic here is the fact that 89% of the respondents who agree that layout is important still are at best neutral about the usefulness of descriptions by sighted peers.

There is general consensus that descriptions by sighted peers are usually quite vague and (understandably) show bias towards a vision-based interaction model. The most requested information was found to be concerning semantic relations between UI elements, explanations concerning seemingly obscure functionality, and the identification of elements that screen readers fail to recognise correctly (or at all).

One respondent summarised the primary concern with assistance from sighted peers rather eloquently:

> "[. . .] sighted folks are absolutely horrible at putting into words what an interface looks like, especially when asked to focus on such things as location of items, and screen design. They tend to focus on the eye-candy (colours, designs, shading effects, and so on) none of which contribute the least to my understanding of the overall layout of the screen."

Overall, everyone who participated in the survey expressed that in order for assistance by sighted peers in the form of descriptions to be useful, it must involve well balanced descriptions, combining information about logical layout, functionality, and physical configuration. It should contain information that does not only assist in building an accurate mental model, but also information that helps improve communication with sighted users. Such descriptions are rare, and as such there is usually a need to ask more probing questions in an attempt to obtain the necessary information despite the sighted peer's bias. As noted by several respondents, the best assistance is usually obtained from AT professionals, or in general individuals who have a lot of experience with how assistive technology for the blind operates. Ultimately, the source of this complexity stems from the fact that often neither party (blind or sighted) can truly envision how the other party interacts with the UI.

This is further exemplified by participants who used to have full vision, and who experienced sudden and complete loss of vision. The answers provided by those individuals indicate an expectation of being presented with a representation that retains most of the spatial properties of the visual UI. Two of these participants expressed frustration with screen readers that do not present the UI as a reliable replica of the visual form.

Alternatively, participants who have been totally blind since birth expressed an overall lack of interest in the physical properties of UI elements.

Based on this survey, Landau's statement should be restated:

The accuracy of a mental model created by a blind user is dependent upon the ability of the AT solution to accurately reflect the UI, and the expertise of sighted peers that provide assistance.

3.6 Troublesome UI elements

Participants in the survey were asked about specific UI elements or types of elements that were perceived as especially troublesome in terms of user interaction. By their very nature, questions of this kind capture the effects of multiple influences, and it is important to view the answers in that context. Complications in user interaction could stem from limitations in the screen reader support for an UI element, input device limitations, implementation details of the UI element, and/or interaction semantics of the application.

Surprisingly, despite the many variables in the operational context of user interaction, all participants identified essentially the same groups of elements as the source for most of their UI interaction frustrations.

3.6.1 Trigger-happy UI elements

Some UI elements are implemented with “hover” functionality. This means that functionality is triggered by merely moving the cursor to the element, without actively selecting it, or by leaving the cursor on the element for a set amount of time (hence the terminology). The accessibility issue related to this type of elements is that mere exploration of the UI (which has been identified as a common activity for blind users) can cause operations to be performed that the user did not mean to trigger.

Another common example referenced in the survey answers concerns auto-select UI elements. In this case, merely shifting focus to an element causes it to be selected. This is sometimes encountered in groups of radio-buttons[9] where keyboard navigation for cycling through the choices implies selection of the current choice. Again, in this case, exploration of the UI triggers an operation rather than merely providing information to the user.

This problem is a combination of the implementation details of the UI elements and the operation of the screen reader. When the assistive technology is implemented as a derivative of the graphical representation, it is restricted to the user interaction as implemented for the GUI. Moving the cursor or focus to the

[9] A group of radio buttons is a container of buttons where only one button can ever be selected. Selecting one button automatically deselects any previously selection.

element triggers functionality, and the effects of that operation are reported back to the user by the screen reader as it processes notifications from the graphical toolkit (or changes in the UI itself).

Some screen readers implement a special “review mode” to circumvent this problem, where exploration operates entirely on the off-screen model rather than resulting in actually interaction with the UI. This solution requires the user to use this mode for exploration, because it is often not known whether a troublesome element is in use in the UI until the problem presents itself. It is also not always possible to provide full exploration due to limitations in the OSM implementation.

3.6.2 Unidentified UI elements

Many existing assistive technology solutions depend on explicitly defined accessibility information, and when such information is missing or deemed unnecessary by application developers, screen readers may not be able to determine the function of a UI element, or even determine its existence. In other cases, the information is not consistent with the visualisation, causing confusion. This is most often seen in dialogs where by default “OK” and “Cancel” buttons are presented. Many applications have been reported where a button is visually presented with a label other than “OK” yet it is still reported by a screen reader as “OK button”.

More troublesome are UI elements that are truly unidentified. One respondent gave reference to an application where a regular text label turned out to be an element with button functionality. Upon selecting this item, a pop-up menu was presented with multiple items. Some were identified as buttons whereas others were identified as links, despite the fact that visually all items look exactly the same. Users depend on the correct identification to know what form of user interaction is possible on a specific element, and therefore incorrect identification can cause undesirable effects.

Respondents also reported UI elements that the screen reader could not locate, although they were clearly visible on the screen. This type of problems is usually related to limitations in the OSM construction process.

3.6.3 Semantic relations between UI elements

User interfaces often employ labels to commnicate the meaning of entry elements. This association between a label and an entry field is often represented visually. Screen readers may employ heuristics to detect this link, but it is an error prone process and mistakes can render a UI quite difficult to use. Figure 3.3 shows

Figure 3.3: The complexity of entry field label placement

a rather complex example, although visually it is rather obvious. Depending on the actual implementation of heuristics in the screen reader *and* the exact implementation of the UI layout[10], this little form can be presented in many different ways. When testing this form using a commercial screen reader on MS Windows[11], the form was announced as follows:

```
Entry form
First
Middle
Last
Full name Edit field
Edit field
Edit field
No spaces
Occupation Edit field
```

A combination of this type and the previous can be found in entry forms where a single label is used to associate meaning with a group of entry fields, as might be done when entering a postal address. Visually, the size and placement of various entry fields can clearly indicate street name and number, postal code, and city name. However, to the screen reader it might appear that the first entry field has a label “Address”, and the rest of the entry fields do not have a label at all.

3.7 Conclusions

The survey discussed in this chapter provides information about how blind users perceive and interact with graphical user interfaces. Participants were solicited both by direct invitation and indirectly by other participants. Although the number of participants was quite high compared to similar studies, it is still much less than what would be required to achieve statistically significant results. However,

[10] There are usually multiple ways to implement the exact same visual layout – but the screen reader will often read the seemingly identical layouts quite differently.

[11] JAWS for Windows, version 7.0.135.

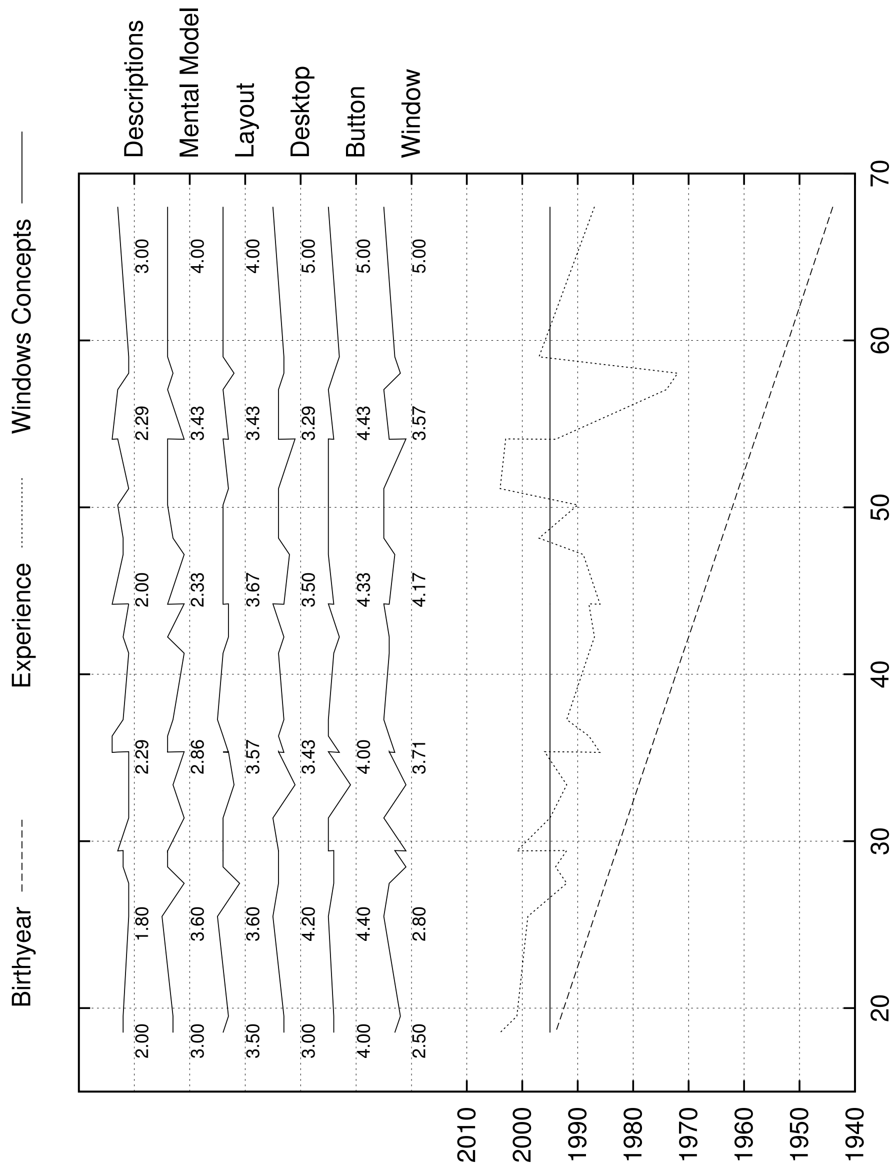


Figure 3.4: Impact of age

the consistency of results in various areas does give confidence that the results are a fair representation.

The demographic data on the participants (Figure 3.1) shows a quite diverse group of individuals who share two main characterstics: they are totally blind, and they have a reasonable level of experience operating computer systems.

During the analysis of the responses to the survey, a question was raised out of sheer academic curiosity: Is there any correlation between the responses and a specific age group. Figure 3.4 charts the scores to all questions discussed in this chapter against the age of the respondents, and a time reference of Sarah Morley's "Windows Concepts" [94]. The chart presents the age on the x-axis, and in the bottom half the year as y-axis, to chart birthyear, and year of first computer use. The top half shows the 6 categories of scored responses, and for each 10-year interval of age, the average score per category. Scoring took place on a 1–5 scale, with 1 and 5 representing opposite choices (3 is the neutral value), so the primary focus is on whether score averages are above or below the neutral point, i.e. in favour of one or the other option.

The score averages per 10-year interval do not show any particular correlation for any of the categories across all age intervals.

Five fundamental questions were being addressed through the survey:

- *How do blind users perceive graphical user interfaces?*
  The analysis of responses concerning UI elements (see section 3.3) indictates that participants think about them mostly in a conceptual way, from a functional point of view. A few people did try to provide a description of the visual appearance of the elements they were questioned about because they believed that this information was requested as well. Almost all respondents were clearly able to determine the semantic meaning of the UI elements.

- *Are the users comfortable with the established metaphors?*
  All respondents demonstrated familiarity with the established metaphors for user interaction. One person indicated not knowing UI elements when considering questions where they were referred to by name, but then demonstrated a working knowledge of using those same UI elements in responses to other questions.

- *What type of mental model(s) do blind users use?*
  Previous research (Kurniawan, et al. [80, 81]) identified three types of mental models are common for blind users: structural, functional, and hybrid (a combination of the previous two types). Under the original hypothesis for mental modelling, mental models are by definition hybrid because they involve entities and the relations between them. This is discussed in

section 3.4. The survey presented in this chapter confirms the hypothesis. Interestingly, multiple people indicated n their survey responses that they do not create mental models for UIs, even though their answers to other questions clearly indicated they do.

Everyone did point out (in various ways) that the accuracy of mental models (or people's understanding of the UI as a whole) is highly dependent upon the information that the model is based on. This information is often obtained from a screen reader, which is unfortunately not always a very reliable source.

- *Is a verbal description of the UI by a sighted peer useful?*
  The usefulness of verbal descriptions by sighted users is debatable. A common concern is that people often do not know what information is most useful to a blind individual, which ultimately relates to the differences in how specific user populations interact with user interfaces. It is quite difficult for a sighted user to truly understand how a blind user operates, and vice versa.

  In addition, multiple participants reported that it is very common for there to be a discrepancy between what is visually shown on the screen and what is presented by the screen reader. This makes it difficult to collaborate at any level.

- *What type of UI concepts are difficult to interact with?*
  Respondents were quite consistent in their identification of UI elements that present significant complications. Their primary concern was found to be with elements that activate functionality while the blind user is exploring the UI. Another frequent complication was found with unidentified UI elements. Such elements are either not reachable[12] by means of operations that the user can use, or they can be reached but they lack nay indication concerning what user interaction can be used to operate them.

The responses to the survey presented in this chapter indicate that blind users are quite familiar with the components of WIMP-based user interfaces. By means of a mental model that incorporates both structural and functional information, users are proficient at interacting with the UI. Discrepancies between visual and non-visual representation are unfortunately common, and they affect the ability to create an accurate mental model. This is further complicated by the fact that sighted users are often not very good at providing useful verbal descriptions. Furthermore, some UI elements present challenges due to their implementation.

It is important to note that regardless of the high level of familiarity with the UI elements that blind users tend to possess, the limitations to being able to create an accurate mental model pose a significant problem. Without an accurate model,

[12] I.e. it is not possible to move focus to them.

user interaction is hindered (and for some elements simply impossible), and collaboration is negatively affected as well.

Chapter 4

State of the art

”The artist is the creator of beautiful things.
To reveal art and conceal the artist is art’s aim.
The critic is he who can translate into another manner
or a new material
his impression of beautiful things.”
(Oscar Wilde, ”The Picture of Dorian Gray”, 1891)

This chapter presents past and current approaches to multimodal user interfaces in the field of Human-Computer Interaction. While not all were designed in function of accessibility, the tools and techniques most certainly can be applied to this problem.

4.1 Introduction

The overview presented in section 1.1 (page 4) illustrates the expansive history of research dedicated to making computer systems accessible. In this chapter, significant past and current contributions will be discussed and compared based on a common set of criteria. This section serves as an introduction to a reference framework that can be applied to the approaches discussed in the remainder of the chapter.

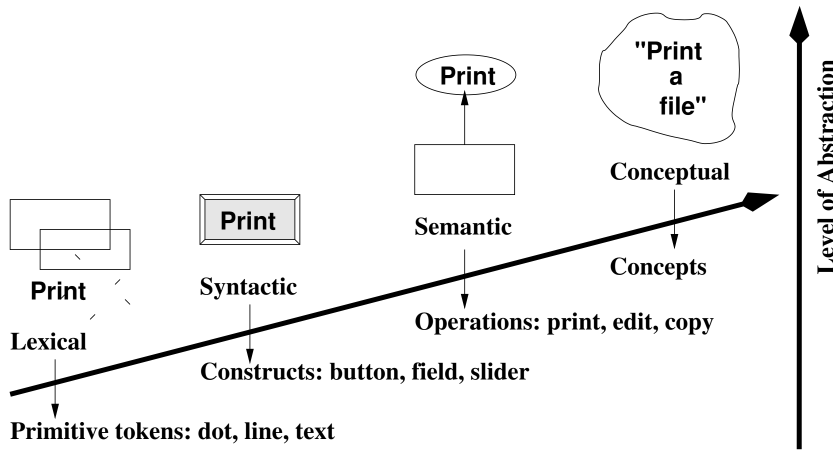


Figure 4.1: Four distinct layers of user interface design

4.1.1 The four layers of user interface design

While UI design is often thought of as a self-contained, straightforward process, four distinct layers of design can be identified (see Figure 4.1) [46, 69, 45][1], and in view of Gaver's work [50] as discussed in section 2.2.4, they can be grouped together in function of the layer of metaphor (or model) they operate on:

- **Conceptual metaphor layer**
  - *Conceptual*
    This layer of design describes the basic elements from the physical world metaphor that are relevant to the UI. It also describes the manipulations that each element supports.
  - *Semantic*
    This layer of design describes the functionality of the system, in an abstract way, independent from any specifics concerning user interaction. It defines the operations that can be performed in the system, and provides meaning to syntactic constructs in a specific UI context.
- **Perceptual metaphor layer**
  - *Syntactic*
    This layer of design describes the operations necessary to perform

[1] Edwards, Mynatt, and Stockton list only three layers in [45], but they limited themselves to a description of the layers of modelling, where the conceptual layer forms the basis for those three layers of modelling.

the functions described in the conceptual layer. This description is modality specific, using the fundamental primitive UI elements to construct a higher order element that either enables some functionality or that encapsulates some information. Examples are: buttons, valuators, text input fields, ... These elements are presented to the user as entry points of interaction to trigger some aspect of the system's functionality as defined at the conceptual layer.

– *Lexical*
  This layer of design maps low-level input modality operations onto higher level fundamental operations of UI elements. At this level, the UI is expressed as a collection of primitive elements, such as dots, lines, shapes, images and text.

The distinction between the different layers and their grouping is important in consideration of accessibility because solutions will generally encompass a specific layer and all those below it (usually in order of the list above). Adaptations at the lexical level may involve the use of haptic input devices or a braille keyboard, whereas assistive technology solutions at the syntactic level are more involved. They affect the perceptual layer as a whole. Common representatives are various screen readers. Approaches to accessibility at the semantic or conceptual level are less common because they usually require an adaptation at the level of application or system functionality.

4.1.2 Unified Reference Framework

Calvary, et al. developed a Unifying Reference Framework (URF)[2] [24, 26, 25] for multi-target user interfaces, specifically intended to support the development of context-aware UIs[3]. The context of use in this framework comprises three components: a target user population, a hardware/software platform, and a physical environment. Each aspect of the context of use may influence the UI development life cycle at any of four distinct levels. The levels of abstraction recognised in the Unifying Reference Framework correspond to the four layers of UI design presented in section 4.1.1 (see Figure 4.2):

- **Tasks & Concepts** (T&C): User interface specification in terms of tasks to be carried out by the user and well-known underlying concepts (objects that

[2] Also known as the CAMELEON Reference Framework.

[3] The Unifying Reference Framework comprises more elements than are presented here. The discussion of the state of the art does not require all elements of the framework, and the scope has therefore been limited to what is sufficient to describe, understand, and compare the various approaches.

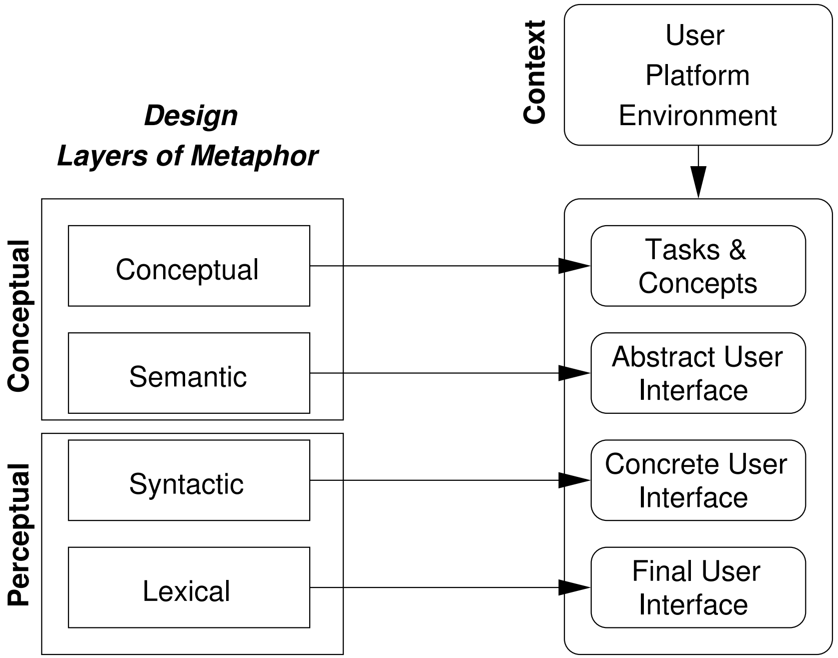


Figure 4.2: Design layers in the Unified Reference Framework

are manipulated during the completion of tasks). This level is computational-independent.

- **Abstract User Interface** (AUI): Canonical expression of the interactions described at the T&C level. The interactions can be grouped to reflect logical relations, commonly seen in multi-step tasks and sequences of tasks. This level is modality-independent.
- **Concrete User Interface** (CUI): Specification of the UI in terms of a specific ”Look & Feel”, but independent from any specific platform. The CUI effectively defines all user interaction elements in the UI, relations between the elements, and layout. This level is toolkit-independent.
- **Final User Interface** (FUI): The final representation of the UI within the context of a specific platform. This is the actual implementation of the UI. It may be specified as source code, compiled into object code, or it may be instantiated at runtime.

The development of a UI (as modelled in the Unifying Reference Framework) can be accomplished by means of transformations between the aforementioned

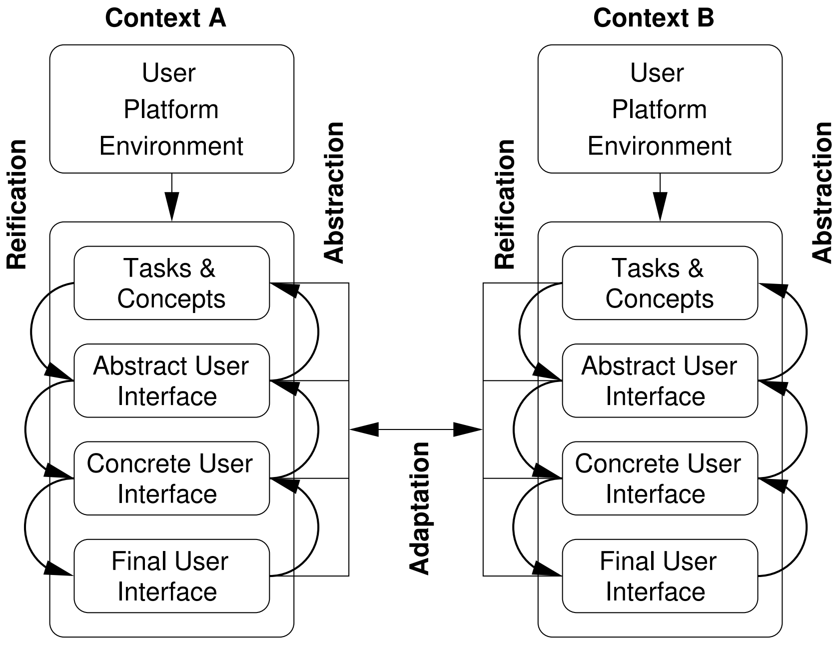


Figure 4.3: Unified Reference Framework

levels. Both top–down and bottom–up transformations are possible, depending on the initial design.

- **Reification** (top–down): A derivation process whereby an abstract specification is made more concrete.
- **Abstraction** (bottom–up): A reverse engineering process whereby an abstract specification is inferred from a more concrete one.

By means of these two operations, the framework is able to model a large variety of UI development processes. E.g. a designer might prototype a UI at the concrete level using a design tool. In this case, reification will yield the final UI, whereas abstraction provides for the specification of the user tasks and underlying concepts.

When multimodal user interfaces are considered, the development of the UI spans multiple contexts of use. The Unifying Reference Framework supports this with the addition of a third operation (see Figure 4.3):

- **Adaptation** (cross context): A transformation process whereby a UI specification at a given level for a specific context of use is translated to

a UI specification (possibly at a different level of abstraction[4]) for a different context.

4.1.3 The CARE properties

In the context of multimodal user interfaces, possible relations between modalities may exist. Coutaz, et al. [33] define a set of four properties to characterise these relations: Complementarity, Assignment, Redundancy, and Equivalence. The formal definition of these properties is based on the following important concepts:

- **State**: A set of measurable properties that characterise a situation.
- **Interaction trajectory**: A sequence of successive states.
- **Agent**: An entity that can initiate the execution of an interaction trajectory.
- **Goal**: A state that an agent intends to reach.
- **Modality**: An interaction method that an agent can use to reach a goal.
- **Temporal relationship**: A characterisation for the use of a set of modalities over time.

In addition, the following predicates are used in the formal definition of the CARE properties:

- $Card(M)$ represents the number of modalities in set M.
- $Duration(tw)$ expresses the duration of time interval tw.
- $Active(m, t)$ indicates that modality m is being used at some point t in time.
- $Pick(s, m, s')$ means that modality m was chosen from a set of modalities to satisfy interaction trajectory $s \rightarrow s'$.
- $Reach(s, m, s')$ means that interaction trajectory $s \rightarrow s'$ cam be satisfied with modality m.
- $SetReach(s, M, s')$ means that interaction trajectory $s \rightarrow s'$ can be satisfied with the modalities in set M.

[4] The initial version of the Unifying Reference Framework [24] defined the adaptation operation as a transformation between representations at the same level of abstraction. Revisions made in support of plasticity of user interfaces (being able to adapt to context changes without affecting usability) introduced adaptation as a cross level operation.

- $Parallel(M, tw) \Leftrightarrow (Card(M) > 1) \wedge (Duration(tw) \neq \infty) \wedge (\exists t \in M \cdot \forall m \in M \cdot Active(m, t))$
  Parallel events (simultaneous use of modalities) over a temporal window are characterised by the fact that there is a point in time where all the modalities in M are active.

- $Sequential(M, tw) \Leftrightarrow (Card(M) > 1) \wedge (Duration(tw) \neq \infty) \wedge (\forall t \in tw \cdot (\forall m, m' \in M \cdot Active(m, t) \Rightarrow \neg Active(m', t)) \wedge (\forall m \in M \cdot \exists t \in tw \cdot Active(m, t))$
  Sequential events are characterised by the fact that there is at most one modality active at any given time in the temporal window, and where all the modalities in set M are used at some point in the temporal window.

The four CARE properties can then be defined as follows:

- **Equivalence**:
  $Equivalence(s, M, s') \Leftrightarrow (Card(M) > 1) \wedge (\forall m \in M \cdot Reach(s, m, s')$
  The equivalence property expresses that interaction trajectory $s \rightarrow s'$ can be accomplished by means of any of the modalities in set M. It therefore characterises choice of modality. It is important to note that no temporal constraint is enforced, i.e. different modalities may have different temporal requirements for completing the interaction trajectory.

- **Assignment**:
  $StrictAssignment(s, m, s') \Leftrightarrow$
  $Reach(s, m, s') \wedge (\forall m' \in M \cdot Reach(s, m', s') \Rightarrow m' = m)$
  $AgentAssignment(s, m, M, s') \Leftrightarrow$
  $(Card(M) > 1) \wedge (\forall m' \in M \cdot (Reach(s, m, s') \wedge (Pick(s, m', s')) \Rightarrow m' = m))$
  Contrary to the equivalence property, assignment characterises the absence of choice. A given modality m is said to be assigned to the interaction trajectory $s \rightarrow s'$ if no other modality is used for that trajectory, either because it is the only possible modality (StrictAssignment), or because the agent will always select the same modality m for the trajectory (AgentAssignment).

- **Redundancy**:
  $Redundancy(s, M, s', tw) \Leftrightarrow$
  $Equivalence(s, M, s') \wedge (Sequential(M, tw) \vee Parallel(M, tw))$
  The redundancy property characterises the ability to satisfy the interaction trajectory $s \rightarrow s'$ with any of the modalities in set M within temporal window tw. Redundancy comprises both sequential and parallel temporal relations.

- **Complementarity**:
  $Complementarity(s, M, s', tw) \Leftrightarrow$
  $(Card(M) > 1) \wedge (Duration(tw) \neq \infty) \wedge$

$(\forall M' \in \mathbf{P}M \cdot (M' \neq M) \Rightarrow \neg Reach(s, M', s')))\wedge$
$SetReach(s, M, s') \wedge (Sequential(M, tw) \vee Parallel(M, tw))$
The complementarity property expresses that the modalities in set M must be used together in order to satisfy the interaction trajectory $s \rightarrow s'$, i.e. none of them can individually reach the goal.

4.2 Evaluation criteria for related work

The remainder of this chapter will present various past and current approaches towards multimodal user interface development. The framework presented in section 4.1.2 and the CARE properties discussed in section 4.1.3 serve as a reference for comparison of the related work. Specifically, the following aspects are considered:

- **URF diagram**: The Unified Reference Framework diagram visualises the transformation process from source specification to the final target representation for the user interface. As such, it identifies:
  - The entry level of abstraction (and context, if applicable) that serves as source for the UI creation,
  - The operations (reification, abstraction, adaptation) that are required to perform the transformation.
  - The exit level of abstraction (and context, if applicable) that corresponds to the outcome of the transformation process.
- **Non-visual access to GUIs**: This represents one of the extreme contexts for multimodal UIs, and it is therefore a good criterion to consider. Based on the work of Mynatt, Weber, and Gunzenhäuser (see section 2.3), the following requirements are reported on:
  - Coherence between visual and non-visual interfaces, both static and dynamic
  - Exploration in a non-visual interface
  - Conveying semantic information in a non-visual interface
  - Interaction in a non-visual interface

  The ”ease of learning” topic is not taken into consideration in this dissertation.
- **CARE properties**: The properties defined by Coutaz (see section 4.1.3) help characterise the influence of multimodality on the user interaction. Input and output modalities are considered independently, to the extent possible.

- **Conceptual model**: The underlying conceptual model for a UI tends to have a significant influence on the overall design, and it is therefore important to consider whether an approach is based on a single (shared) conceptual model, or whether multiple models are used.
- **Concurrency**: Collaboration between users may benefit from both users being able to explore and interact with the system at the same time. The ability to present the UI in different modalities at the same time is therefore an important consideration.
- **Cost factors**: Every approach has a cost associated with it, be it the cost of specialised hardware, cost of the software component, cost of implementing applications based on a specific UI toolkit, ... While it is well beyond the scope of this work to estimate the actual costs[5], it is possible to compare approaches based on the factors that contribute to the cost. E.g. if special hardware is required for a specific approach, the cost barrier tends to be higher[6].

In addition, the advantages and shortcomings of each approach will be discussed within the context of this work. Based on the comparison of related work, and in view of the identified advantages and shortcomings of each approach, requirements for the presented work will be defined.

4.2.1 Classification of related work

The extensive research into the state of the art presented in this chapter covers a quite varied collection of projects and studies. Different methods of classification can be used to group the various works. In view of the criteria presented in this section, classification based on the underlying model in the Unified Reference Framework has been selected for this chapter:

- *Abstraction of a final user interface*
  This classification is still quite broad in view of the work that has been done on multimodal user interfaces and alternative representations of user interfaces. The final UI level in the Unified Reference Framework captures the toolkit dependent part of the UI specification, and therefore approaches that are covered in this class of works may be based on:
  - *Lexical adaptation*: In terms of impact on application development, this technique is by far the least invasive because it primarily relates

[5] Also note that the purchase of assistive technology solutions is often funded (in part or in full) by government programs.

[6] This is even true if the hardware itself is not expensive, because hardware that has a single purpose tends to be less favoured.

to adaptations at the level of the input and output devices. A very representative example with historical significance is the console probe [136].

  - *Graphical screen analysis*: Assistive technology in this category applies some form of image detection processing and/or OCR to the content of the graphical screen. This technique is also less invasive, although it is often combined with a more invasive technique in order to improve its accuracy and/or performance.
  - *Perceptual adaptation*: Approaches in this category typically involve reproducing the lexical and/or syntactic structure of the user interface in an alternative form. Based on a variety of information sources, an off-screen model is composed, and an assistive technology solution uses that model to provide information to the user.

  Works in this classification are discussed in section 4.4.

- *Adaptation of an abstract user interface*
  Approaches to multimodal user interfaces that fall in this class of works share a common path through the Unified reference Framework: an abstract specification of the interface (at the AUI level) is adapted for a new context of use (typically a different modality). Two common forms are observed:

  - *Abstraction from a concrete user interface*: This group captures techniques where the user interface is specified at the final UI or concrete UI level, and an abstract UI is obtained by abstracting the concrete UI (rendering it modality independent). The resulting AUI specification is then adapted for the new context of use.
  - *Reification from an abstract user interface*: In this case the user interface is defined a the abstract UI level, and it is used as-is for adaptation in the next context of use.

  Works in this classification are discussed in section 4.5.

Prior to the discussion of representative contributions in the HCI field of multimodal interfaces for the classifications mentioned above, other important works are presented more briefly in section 4.3. Conclusions drawn from the state of the art, and a statement of requirements for the work presented in this dissertation conclude the chapter in section 4.6.

4.3 Related works

The related works in this section are grouped based on their primary goal. Projects that work towards providing accessibility are discussed in section 4.3.1.

The development of user interfaces based on abstract user interface descriptions, and user interface description languages are presented in section 4.3.2. A discussion of the Web Accessibility Initiative (WAI) Accessible Rich Internet Applications (ARIA[7]) follows in section 4.3.3. While it is specific to web technologies, its concepts are of importance to this dissertation and accessible user interfaces in general.

4.3.1 Accessibility

The Mercator project was a research effort at the Georgia Institute of Technology, replacing the GUI with a hierarchical auditory interface [100]. A speech synthesis system is added to the standard desktop configuration, and both speech and iconic sound cues are used to convey information to the user. An off-screen model is created based on captured X11 protocol information, and by means of toolkit hooks (the X11 Remote Access Protocol), taking in consideration that many features of GUIs are related to limitations of the output modality. Overlapping windows and clipping occur in a GUI environment due to screen size limitations, and can therefore be avoided for non-visual access, albeit at the expense of sacrificing coherence between the visual and non-visual representations. With the emergence of higher-level graphical toolkits, the capturing at the X11 toolkit level is no longer sufficient to determine the semantics of user interaction.

Weber and Mager discussed various existing techniques for providing a non-visual interface for X11 by means of toolkit hooks, queries to the application and desktop objects, and scripting [160]. Blenkhorn and Evans provide similar details for screen readers on MS Windows [11].

Barnicle [6] described obstacles that blind users face when using GUIs by means of assistive technology. Pontelli, et al [114], and Theofanos and Redish [141] discussed obstacles that affect web accessibility. In comparison, the problems are very similar; so much in fact that the successes with World Wide Web (WWW) forms provide strong support for considering abstract descriptions (similar to HTML) as basis for developing UIs in view of accessibility concerns.

4.3.2 Abstract user interface descriptions

Bodart and Vanderdonckt introduced the concept of Abstract Interaction Objects (AIO) [154, 12] as part of a design to automatically generate user interfaces based on application semantics. This can be seen as the early beginning of model based interface development. The AIO work is of significance because it recognised the

[7] The official acronym to refer to this new specification is ”WAI-ARIA”.

importance for presentation independence and separation of concerns. Whereas the AIO defines the behaviour of the interaction object, the presentation (graphical appearance) is provided by a Concrete Interaction Object.

The Views system described by Bishop and Horspool [9] introduced the concept of runtime creation of the user interface representation based on an XML specification of the UI.

Mir Farooq Ali conducted research at Virginia Tech on building multi-platform user interfaces using UIML [1]. By means of a multi-step annotation and transformation process, an abstract UI description is used to generate a platform-specific UI in UIML. This process takes place during the development phase as opposed to the runtime processing proposed in this dissertation. His thesis identifies the applicability of multi-platform user interfaces as a possible solution for providing accessible user interfaces, yet this idea was not explored any further beyond references to related work. The research into the construction of accessible interfaces using his approach is left as future work.

User interface description languages (UIDL) have been researched extensively throughout the past eight to ten years. Souchon and Vanderdonckt [132] reviewed 10 different XML-compliant UIDLs: UIML (User Interface Markup Language) created by Virginia Tech and partners, AUIML (Abstract User Interface Markup Language) by IBM, XIML (Extensible Interface Markup Language) by RedWhale Software Corp., Seescoa XML by a research consortium of four Belgian universities, Teresa XML by the HCI Group of ISTI-C.N.R., WSXML (Web Services Experience Language) by IBM, XUL (Extensible User interface Language) by Mozilla, XISL (Extensible Interaction Sheets Language), AAIML (Alternate Access Interface Markup Language) by the National Committee for Information Technology Standards, and TADEUS-XML. The review focused mostly on ascertaining which UIDLs are more appropriate for developing fully functional UIs. The final conclusion unfortunately indicates that no single UIDL satisfies the requirements, and as the authors note: ”it is meaningless to possess a refined specification of a UI that cannot be rendered or only partially.”

A followup study was conducted by Guerrero-Garcia, et al. [56] based on an updated list of UIDLs. Confirming the conclusion from the previous study by Souchon and Vanderdonckt [132], the authors note that given all the different characteristics that can be identified for the UIDLs it appears difficult to determine which UIDL to use. The authors also express the opinion that the decision process for choosing a UIDL for a project may benefit from matching up project requirements to the characteristics used to compare the UIDLs in their study.

Trewin, Zimmermann, and Vanderheiden [143, 144] present technical requirements for abstract user interface descriptions based on Universal Access and ”Design-for-All” principles, and they evaluated four different UIDLs based on those requirements: UIML, XIML, Xforms, and AIAP. Their findings indicate

that ”existing abstract languages for user interface representations are close to meeting the requirements outlined in this paper.” However, the authors also note that ”[. . .] analysis at a more detailed level and practical experience [is required] in order to provide a realistic assessment of the extent to which they can support the goal of universal usability.”

The Belgian Laboratory of Computer-Human Interaction (BCHI) at the Université Catholique de Louvain developed an UIDL to surpass all others in terms of goals for functionality, ”capturing the essential properties [. . .] that turn out to be vital for specifying, describing, designing, and developing [. . .] UIs”: UsiXML [89]. Of special importance are:

- *The UI design should be independent of any modality of interaction.*
  This goal captures the *applicability to any target* requirement for Universal Access discussed in section 2.4.2.
- *It should support the integration of all models used during UI development (context of use, user, platform, environment, . . .).*
  This goal relates to the *applicability to any delivery context* requirement for Universal Access discussed in section 2.4.2.
- *It should be possible to express explicit mappings between models and elements.*
  This goal reflects the technical requirement discussed in section 2.5.1 concerning *flexibility in inclusion of presentation information*.

UsiXML has also proven to be quite extensible, as exemplified by the research of Kaklanis, et al. on a haptic rendering engine using an extension to the UsiXML CUI model [72]. Their work is aimed at web browsers, where the source HTML document is transformed into an XHTML document that can then be used to generate a UsiXML description for later use. While this technique derives both a haptic rendering and a visual representation from the same HTML source, there is no mechanism in place (yet) to ensure coherence.

Draheim, et al. introduce the concept of ”GUIs as documents” [36]. The authors provide a detailed comparison of four GUI development paradigms, proposing a document-oriented GUI paradigm where editing of the graphical user interface can take place at application runtime. In the discussion of the document-based GUI paradigm, they write about the separation of GUI and program logic: ”This makes it possible to have different GUIs for different kinds of users, e.g. special GUIs for users with disabilities or GUIs in different languages. Consequently, this approach inherently offers solutions for accessibility and internationalisation.” The idea did not get developed further, however.

4.3.3 WAI-ARIA

As World Wide Web (WWW, commonly referred to as ”the Web”) technologies advance, so does the complexity of providing accessibility. With the introduction of Web 2.0[8], the Web has evolved from a repository of data to a world wide interactive platform. As the complexity of Web interactions increased, so did the user interfaces become more sophisticated. These advances have brought forth new accessibility issues that go beyond the recommendations in the Web Content Accessibility Guidelines (WCAG) put forth by the W3C in response to the original Web accessibility concerns [31].

Application user interfaces on computing devices are typically developed based on a specific presentation toolkit that defines a distinct set of widgets (button, label, valuator, . . .). The interaction semantics of these widgets are well defined and it is therefore possible to provide reasonable support for these widgets in assistive technology solutions. Web 2.0 Internet applications often require similar UI elements, yet they are not part of the HTML specification. Developers commonly construct these more advanced elements by means of *span* and *div* elements, combined with images and other elements, and further customised with style sheets and scripting. Developers may even opt to use custom widgets in lieu of standard elements such as buttons in order to have greater control over their appearance and/or behaviour [142].

The Web Accessibility Initiative (WAI) has been in the process of establishing an accessibility specification to address the Web 2.0 advances: WAI-ARIA. It introduces several important concepts to provide information on UI element interaction semantics and to assist assistive technology providers with making Web 2.0 entities accessible [90, 165]:

- Roles: The new *role* attribute specifies the widget or document structure type that is assigned to a specific element, regardless of any semantic inherited from the implementing technology. The available widget roles reflect UI widgets that are commonly found in presentation toolkits and the roles are often already represented in accessibility APIs. Each role has a specific set of states and/or properties associated with it.
- Document landmarks: Landmarks are roles that are used to identify regions in a document as navigational landmarks, e.g. banner, form, navigation, . . . The purpose of landmarks is to make it easier to identify specific sections, and speed up navigation.
- States and properties: Changeable states and properties are attributes of a role that are used to support accessibility APIs. They convey information

[8] A term coined by O’Reilly Media [51].

that is associated with the behaviour of a specific role, e.g. whether a checkable item is checked, or the value of a valuator.

- Live regions: These are perceivable regions in a document that can be updated independently, i.e. a construct to support partial UI updates. Live regions are implemented by means of specific properties on the regions (e.g. priority of updates, atomicity, types of relevant changes, . . .). Regions may be associated with UI elements that control their content updates.
- Tabindex extension: The standard HTML *tabindex* attribute has become available for any visible element in the UI, and can be given the value -1 to indicate that the element can only receive focus by means of a JavaScript operation. This can be used to implement arbitrary keyboard navigation, including arrow-based navigation[9].

Analysis

The Accessible Rich Internet Applications specification fills an important gap in the world of interactive web content. The application of this ontology-based approach is quite diverse because of its nature. Any web content can potentially be made more accessible by ensuring that semantic information is provided for all elements in the UI.

One of the big advantages of this approach is that WAI-ARIA makes it possible to assign semantic information to UI elements based on their contribution to the overall UI rather than based on their presentation. At its core, the design of this technique is rooted in the notion that e.g. a UI element is a button if and only if it acts like a button, regardless of its appearance.

As a result of the fact that WAI-ARIA is based on annotating UI elements with semantic information, one important shortcoming is that the specification of accessibility related information is not an integral part of the UI design process, or rather it is not enforced as a requirement. As a result, it is not uncommon for the WAI-ARIA annotation of the UI to take place well after the UI has been finalised, i.e. when the ”meaning” of all UI elements has been established. This amounts to an adaptation at the concrete UI level, based on an interpretation of the concrete UI.

Because WAI-ARIA annotations are often added to an otherwise finished UI design the quality of the accessibility support depends on the expertise of the people involved in the annotation process. Mikovec, et al. performed a web toolkits accessibility study focused on WAI-ARIA and found significant issues [92].

[9]The standard form of keyboard navigation in HTML documents is by means of the *Tab* key which moves focus to the next focusable element. This is often not adequate.

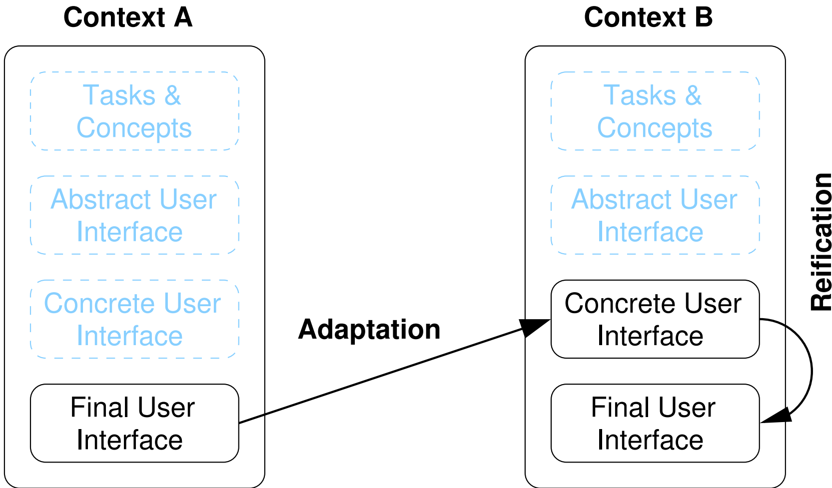


Figure 4.4: Abstraction of a Final User Interface

It is important to note however that WAI-ARIA is still an emerging specification, and as any work in progress it remains to be seen how well adoption will progress and whether the lengthy process towards a firm specification may have resulted in incompatibilities in assistive technology.

When the annotation of the UI elements takes place after the UI is designed, there is also a potential danger that what would amount to design issues at the level of the UI in terms of accessibility gets masked by annotations that essentially resolve the issue. Clearly what should happen is that the design of the UI gets updated to fix the problem.

4.4 Abstraction of a Final User Interface

The least invasive and most desirable method for transforming an existing user interface in a given modality into a representation in a different modality is a transformation of the final UI in the source modality into a final UI in the target modality. In terms of practical considerations, the actual path taken generally involves an abstraction towards an adapted concrete UI in the target modality context, followed by a reification to the final target UI (see Figure 4.4).

It is important to note that a final UI has multiple forms, depending on the implementation of the multimodal UI approach:

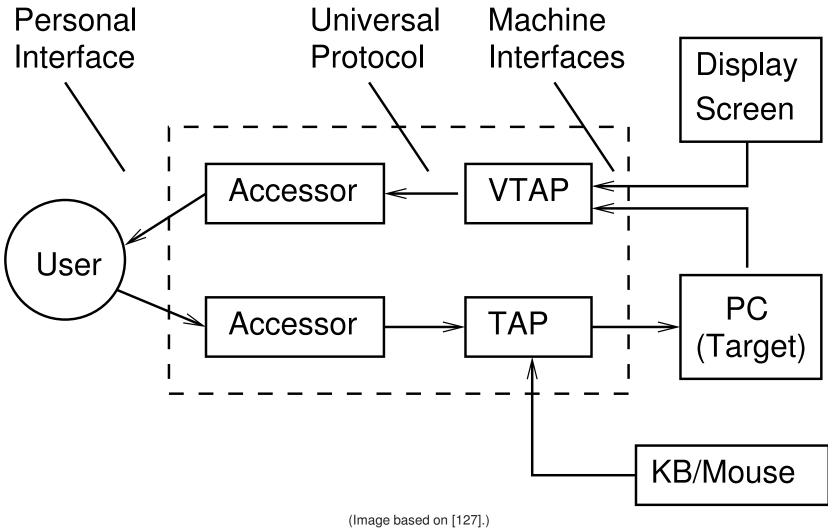


Figure 4.5: Total Access System with VTAP

- Source code: The FUI is either developed as a procedural implementation of a design, or it is generated by a UI design tool.
- Object code: Executable code generated from source code, or an internal representation of the UI created through runtime interpretation.
- Rendering: The final representation of the FUI, rendered in a specific modality and presented to the user.

In this section, approaches are discussed that either capture the rendered FUI (e.g. a screen image), or obtain information about the FUI from specific hooks in the rendering toolkit.

4.4.1 Archimedes a.k.a. Total Access System

The Archimedes project at the University of Hawaii (since September 2003, formerly at the Stanford University Center for the Study of Language and Information) provides non-invasive access to any computer by means of external devices: the Total Access System (TAS) [127]. While this approach primarily focuses on providing support for a wide range of input modalities, the discussion here will consider the Visual Total Access System (Visual TAS). At first glance this may seem to be a narrowing of scope, but the actual design will show that the Visual TAS is actually an extension of the basic system.

Figure 4.5 provides the high level design of the system. Aside from regular keyboard and mouse interaction, users can also interact by means of personalised input devices that present him or her with a preferred input modality. These devices are called ”Accessors”, and they can be tailored to individual needs, be it as a specialised keyboard or pointer device, or in the form of advanced speech recognition, head or eye tracking, or even video-based interpretation of sign language.

The input accessor communicates with the Total Access Port (TAP), providing information about the specifics of the user interaction. The TAP translates the events from the accessor into equivalent simulated keyboard and mouse operations that are then presented to the computer system as if the user did operate a physical keyboard and/or mouse. The work flow from user interaction with the input accessor, through the TAP, into the computer system is what is known as the Total Access System.

Obviously, blind users have the added disadvantage that they cannot observe the results of their interaction on the graphical screen. This is addressed by means of the Visual TAS extension. A Visual TAP (VTAP) captures the video stream from the computer system to the display, and extracts information from it by means of sophisticated image analysis and Optical Character Recognition (OCR) techniques, augmented with advanced pattern matching. The information is used to create an off-screen model that is tightly coupled to the overall Total Access System architecture. An output accessor is used to present the system output in the user's preferred output modality. Supported output options include synthetic speech, and tactile, haptic, and non-speech auditory feedback.

Analysis

The Archimedes project provides for a strict separation between input and output processing. The two components can be used independently, and are therefore treated here as two distinct sub-systems.

Total Access System (input)
The (basic) Total Access System does not address aspects of UI presentation and it is therefore not possible to describe the system in the Unified Reference Framework. What it does provide is support for user interaction in various alternative input modalities, mapping all actions onto equivalent keyboard and mouse operations. This does make the assumption that keyboard and mouse devices are the default input modalities for the system, which is known to be true for the vast majority of computer systems.

As a result of the fact that the TAS addresses input modalities only, an immediate conclusion can be reached that static coherence, non-visual exploration, convey-

ing of semantic information, the support for concurrent representations, and a determination of CARE properties for output are not applicable.

The Total Access System provides users with an interaction mechanism that is tailored to the preferred input modalities of each user by means of accessor. Operations carried out with an accessor generate TAS-specific interaction events that are presented to the TAP, where they are translated into equivalent keyboard and mouse operations. The translated operations are delivered to the computer system as if they were generated directly by the keyboard or mouse. It is therefore possible to **ensure dynamic coherence**. This approach also allows for the accessor to implement custom interactions as needed for the target user population. Therefore, it is possible to **ensure non-visual interaction**.

Furthermore, given that all user interaction must either be delivered to the computer system as simulated keyboard and/or mouse operations, or actual keyboard and/or mouse operations, both **”Equivalence” and ”Redundancy” are provided**.

The Total Access System provides an approach to supporting multimodality of user interaction, i.e. providing support for input by means of a variety of different input devices. This is implemented as an external system that merely delivers (simulated) keyboard and mouse actions to the computer system. As such, the Total Access System does not impose any specific conceptual model of its own design. Instead it exposes the **single conceptual model** that forms the basis for interaction with applications on the computer system.

Visual Total Access System (output)
The Visual Total Access System implements the UI adaptation process described in the Unified Reference Framework diagram shown in Figure 4.4. The final UI representation of the computer system is used as input by capturing its video stream[10]. Analysis of the captured image yields an off-screen model at the concrete UI level, which in turn is reified into a final UI in the context of the end user’s preferred output accessor.

Static coherence is ensured because the Visual TAP creates the off-screen model based on the rendered final UI as presented on the screen of the computer system. Given that the Visual TAP cannot access semantic information about the UI, all elements in the user interface must be presented to the user in the non-visual context in order for the user to be able to make user interaction decisions based on his or her perception of the UI.

Dynamic coherence does not apply for this adaptation approach because the Visual TAS is independent from the user interaction processing. That is handled by the (basic) Total Access System as described earlier in this section. For the

[10]This is about the most extreme form of using a final user interface as source for an adaptation process because it captures the UI post-presentation.

same reason, non-visual interaction and determining CARE properties for input are not applicable here.

Non-visual exploration can be supported in the Visual Total Access System provided that the output accessor includes explicit facilities for user interaction with the final UI rendering in non-visual format, e.g. a tactile pad that provides navigation controls for ”scrolling” so that a user can explore the entire UI regardless of the fact that the tactile pad can only render a small subset of the visual screen at a time. This user interaction is entirely independent from the interaction with the computer system.

The Visual Total Access System relies solely on the final UI presentation in graphical form as its source of information, and therefore may not always be able to determine the semantics behind a visualisation. As a result, its **ability to convey semantic information is partial at best**.

The approach of capturing a screen image and creating an off-screen model by means of analysis can only be successful if it is capable of identifying all user interface elements accurately. By doing so, it ensures that the **”Equivalence” CARE property** applies to the presentation component of the Archimedes project. It also ensures that a **single conceptual model** forms the basis for the multimodal UI presentation, because the non-visual rendering is created by interpreting the visual output.

Since the non-visual representation of the UI by means of an output accessor cannot exist without capturing a video stream from a computer display, the Visual Total Access System provides **concurrent representations**.

Advantages and shortcomings

By far the greatest advantage offered by the Archimedes project’s Total Access System is the fact that it is both modular and external to the computer system it provides access to. The modularity aspect (and its resulting separation of concerns) ensures that accessors remain independent of the target hardware, operating system and applications. The Total Access Port and the Visual TAP have some dependency on hardware and operating systems specifics, but in a very limited way.

Accessors are device independent, i.e. they can operate any device that has a TAP available. This is important because it isolates the accessors (typically the most expensive component of the system) from software upgrades of both the operating system and applications. The communication between an accessor and the TAP uses a universal protocol that is both accessor and operation system independent. As such, modality specific aspects of the user interaction are never

visible to the computer system. Even more so, the computer system will typically not be aware of the fact that an assistive technology solution is in use on the system.

Using the visual rendering of the UI (at the FUI level) as source of information poses significant limitations in being able to interpret the meaning of parts or all of the captured image. While significant advances have been made towards fast and accurate recognition of UI elements, icons and texts, inter-element relations are typically not detectable. The burden of interpreting the UI is placed on the user, often requiring specific knowledge of the underlying graphical system. It must also be noted that despite the use of very efficient image analysis techniques, real time abstraction from FUI to CUI imposes delays in the delivery of a representation of the UI in alternate modalities.

The Visual Total Access System also depends on pre-programmed knowledge (in the VTAP) concerning how to recognise specific parts of a UI. When faced with brand new icons or unusual UI elements, the system cannot make a determination as to the identity or meaning. When this occurs, the interaction dialogue may face significant obstacles and meaningful collaboration will suffer.

The system does not allow exploration of the UI beyond observing what is on the screen, and then only if the output accessor provides modality specific exploration features that are accessed from the output device directly. In some cases this problem can be alleviated by combining the input and output accessors in a single device, e.g. a refreshable braille display with built in keyboard.

The need for specialised hardware is a concern due to the potential high costs involved. The Archimedes project, in support of blind users, requires that an accessor be obtained for UI representation and interaction[11], and both a TAP and VTAP. The (V)TAS design does ensure that an accessor is system independent and therefore can be reused for a longer time and across multiple systems. The cost factor is a consideration however, and the Archimedes Project staff characterises it as follows (in a section named ”Advantages of the VTAS”) [116]:

> ”The [. . .] costs of providing access are shared between the user who provides a personal accessor, and the provider of the IT infrastructure who provides a VTAP [and TAP].”

4.4.2 TIDE/VISA

One of the projects in the European Technology Initiative for the Disabled and Elderly (TIDE) addresses the problems facing visually impaired users due to the

[11] The assumption is made, from a cost factor perspective, that preference is given to an accessor that provides both functions rather than two independent accessors.

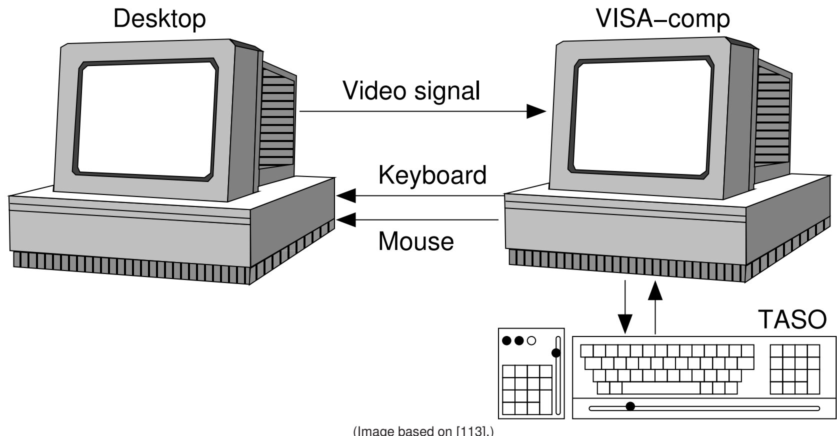


Figure 4.6: The VISA-Comp System

growing use of graphical user interfaces: Video Interface and Signal Analysis (VISA). Much like the later Visual Total Access System, the VISA approach is based on capturing the video signal from the computer system that is to be made accessible, and through analysis and re-interpretation, a presentation based on speech and auditory cues is provided [59, 60, 113].

The VISA-Comp system (see Figure 4.6) is implemented as a desktop computer system where applications are running, a computer system with special hardware for video capturing and for simulating keyboard and mouse events to be sent to the desktop system. Presentation of the UI in non-visual format is done by means of specialised hardware: the Tactile Acoustic Screen Orientation (TASO).

The content of the screen on the desktop system is captured as a bitmap and analysed by means of fast icon and character recognition implemented in a specialised OCR software component. The analysis yields information about visual elements in the UI, which is used to create or update the Current Screen Model (CSM). The TASO uses the CSM as its authoritative source for presenting the UI as text through synthetic speech.

The TASO can present information in two distinct modes:

- Exploration mode: In this mode, all information in and about the UI is represented such that spatial properties are retained. In other words, elements are represented in their original location.
- User mode: In this mode, all information is represented sequentially, per identified category. Spatial information about UI elements is ignored in this mode.

Users can navigate through the information (in either mode) by means of two physical sliders, providing access to a 2-dimensional spatial field, augmented with acoustic feedback. When the desired information is located, the built in speech synthesiser speaks the selection.

Text input can be provided by the user by means of the regular keyboard that is part of the TASO, while UI element based operations can be initiated by means of a specialised numeric keypad.

Because screen image analysis is a rather expensive operation, VISA-Comp implements heuristics to assist with this process. User interaction is used to augment the image analysis process by using operations initiated by the user to predict upcoming changes to the rendering of the UI. E.g. selecting a toggle element will change its boolean state, and therefore the visualisation of that action can be predicted. Of course, the prediction can be wrong, e.g. if the toggle operation triggers a validation, and the change is found to be illegal or possibly requires confirmation by means of a modal dialog. Once the operation has been performed on the desktop system, the effects on the UI will become visible, and this allows the system to recover from a possibly incorrect prediction.

Analysis

The VISA-Comp system relates very well to the Unified Reference Framework shown in Figure 4.4. Video stream intercept hardware captures the screen image of the final UI representation, and OCR-based analysis of the image yields a Current Screen Model at the concrete UI level in the non-visual context. The CSM information is used by the TASO to render the UI in the new final UI.

Static coherence is ensured because the CSM construction process can only be considered successful if all UI elements have been recognised, identified, and are represented in the CSM. In cases where not all UI elements are accurately identified, the conclusion must be reached that VISA-Comp is not capable of providing a non-visual representation.

Dynamic coherence is not upheld because the design of the VISA-Comp system does not automatically provide for direct manipulation operations on UI objects (e.g. dragging an object to the trashcan).

The VISA-Comp system provides an explicit exploration mode in the TASO, which satisfies the requirement for **non-visual exploration**.

Poll and Waterham discuss the disadvantages of deriving and maintaining the CSM based on an analysis of the visual representation of the UI [113]. The visual appearance of two UI elements is sometimes near impossible to distinguish, and semantic relations between elements are typically not visible in the final

UI rendering. Therefore, VISA-Comp **cannot consistently convey semantic information**.

While the VISA-Comp approach provides specific non-visual interaction mechanisms in the TASO, the lack of dynamic coherence clearly indicates that not all interactions are possible, let alone being provided for in a specific non-visual way.

Regular text input and issuing of commands is provided for with the TASO in a manner that is equivalent to the use of a regular keyboard. Mouse interaction is quite different, and this approach provides alternative ways to accomplish many of the typical tasks that would involve mouse actions[12]. The system therefore provides for the **"Equivalence" and "Assignment" CARE properties in terms of input modalities**.

On the UI representation side the **"Equivalence" CARE property applies for output** by virtue of the ability to ensure static coherence.

The conceptual model underlying the visual UI representation is reflected in the structure and content of the final UI in visual form. The VISA-Comp approach is designed to capture this information accurately, and render it in non-visual format. The CSM that is created as intermediary representation at the concrete UI level, and the final UI in non-visual modalities therefore both reflect the same **single conceptual model**.

Since the non-visual representation of the UI through the TASO cannot exist without being able to capture the video signal from the visual representation in real time, the VISA-Comp system provides **concurrent representations**.

Advantages and shortcomings

One of the most important advantages of the VISA-Comp approach is the fact that it is a very non-invasive approach. The desktop system that is made accessible does not require any software or hardware modifications, and the system is not able to distinguish between regular use and use by means of the VISA-Comp. The approach is also independent from the actual GUI system that is used as long as the outward (visual) appearance of its UI elements is known to VISA-Comp.

A major shortcoming in this approach is the fact that all information about the UI is captured from the rendering of the final UI in a visual modality. Quite important semantic information is typically no longer available at this level, and

[12] The actions performed with the TASO are translated into keyboard and/or mouse events, so as far as the desktop system is concerned, it appears as if a regular keyboard and mouse are being used to interact with the system.

it is therefore not possible for the VISA-Comp system to determine the semantic relations between elements, etc...

In order for the system to be able to recognise UI elements it must have some knowledge about the appearance of the elements. When new elements are introduced, or existing elements might be changed, the system must be updated to make it possible to recognise them. This usually involves a quite substantial development effort.

Finally, this approach usually imposes a high cost on the end user because additional hardware is required (including a dedicated device for user interaction).

4.4.3 GUIB

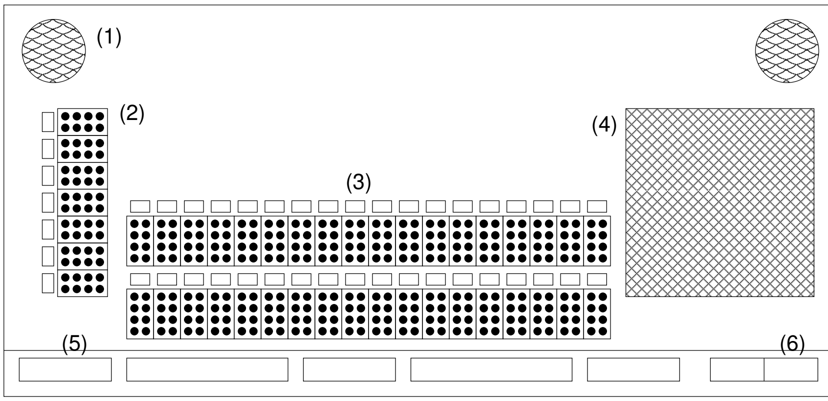
GUIB was a cooperative effort of multiple partners within the European Union, funded in part by the European Commission. It translates the screen contents into a tactile presentation while retaining spatial organisation. A matrix of braille cells with touch-sensors is used as primary input and output modality, augmented with sound [100].

One of the important design decisions at the foundation of GUIB is the desire to continue the use of the spatial metaphor that forms the basis of the graphical user interface. While this is a rather unusual design choice for a non-visual interface, it has proven to be quite powerful. As a result, the spatial configuration of the interface is mirrored onto the tactile pad and braille display used with GUIB.

This approach captures data from a variety of sources in order to be able to provide an accurate non-visual representation of the interface. First of all, keyboard and mouse input is intercepted and analysed. While often the user interaction events are passed on to the system without additional processing, knowledge about the ongoing interaction offers important information that helps augment the off-screen model. Information about the visual presentation is obtained from the GUI directly, both at the lexical level by means of a Virtual Screen Copy (VISC), and at the syntactic level by means of presentation toolkit hooks.

The VISC is a database that contains information that links every display pixel with the character or graphical entity (icon, border, ...) it is part of [75]. The database is constructed by means of two techniques:

- Modified video driver: This component provides information to the VISC during the bitmap construction processing, where characters and graphical elements are visualised by means of specific pixel patterns. This constitutes capturing the results of a forward engineering process.



(Image based on [158].)

Figure 4.7: GUIDE

- OCR: The screen content is captured either directly from video memory or by capturing the video signal. OCR is used to reverse the process performed in the video driver, transforming pixel patterns into their source graphical element or character. This process is less reliable and tends to require significantly more processing resources. This constitutes a reverse engineering process.

The primary source of information is the graphical presentation toolkit because it is able to provide information at a higher abstraction level. An off-screen model is constructed based on this source, creating a hierarchical tree of interaction objects. This tree is augmented with information from the VISC whenever the syntactical level provides insufficient information.

Using the off-screen model, GUIB translates the graphical interface into a textual representation that can be transcribed into braille. The non-visual representation of the UI is provided by the GUIDE input/output device. Figure 4.7 shows a drawing of the device, highlighting the 6 most important components: (1) speakers for auditory output (speech and sound), (2) a vertical refreshable braille display with routing keys, (3) two horizontal refreshable braille displays with routing keys, (4) a touch pad, (5) exploration keys, and (6) mouse button emulation keys.

The screen reader software component of GUIB was developed in an event-response language named GUIB-ERL, based on Hill's Event-Response Language (ERL) [63]. ERL (and GUIB-ERL) is a dialogue specification language that supports the specification of concurrent dialogues. It establishes a rule-based system specifying responses to external events, and actions to be taken when a specific state is entered. In GUIB-ERL, rules are used to transform the

hierarchical off-screen model into braille output, augmented with synthetic speech and non-speech auditory cues where appropriate.

Analysis

The GUIB approach relates to the process described in the Unified Reference Framework diagram shown in Figure 4.4. At first glance, it would seem that adaptation should be modelled at two different levels because GUIB collects information from both the lexical level and the syntactic level. As shown in Figure 4.2, relating the four layers of design to the Unified Reference Framework abstraction levels, the syntactic layer relates to the concrete UI level in the URF diagram. In GUIB however, syntactic information is obtained from hooks in the graphical presentation toolkit which therefore falls at the final UI level.

The lexical and syntactic information is used to construct an off-screen model at the concrete UI level in the non-visual context. Rendering the OSM on the GUIDE device is a CUI-to-FUI reification process within the alternative context.

Static coherence is one of the cornerstones of the GUIB design. The spatial metaphor of the graphical representation is retained, and therefore not only is there a one-to-one correspondence of UI elements between the visual and non-visual interfaces, but spatial location of elements is also consistent.

The two representations are synchronised by translating user interaction events from one context to the other context. Specialised processing is required for pointer device based interactions because braille is a fixed size font whereas most graphical toolkits use proportional fonts.

One of the more complex problems with providing non-visual access to a graphical user interface relates to the observation that the main interaction mechanism in GUIs does not translate well to non-visual interfaces. Alternative forms of interaction are necessary, and GUIB provides two different substitutes for pointer device interaction. First of all, every braille cell can be selected by the user (which can then be translated to its equivalent semantic operation such as selecting a character or entire UI element). Second, a pressure sensitive touch pad can be used to provide direct manipulation interaction. Together with regular keyboard input, GUIB **ensures dynamic coherence**.

GUIB provides **non-visual exploration** based on the spatial metaphor. Two modes are supported: full screen exploration, and application scope exploration. Direct manipulation by means of Braille cell selection and planar touch pad interaction provides for a quite efficient exploration mechanism.

Conveying semantic information is supported by means of two mechanisms: non-speech auditory cues, and rendering information in braille. The primary

approach to rendering semantic or symbolic information is by integrating it into the braille code where possible. The classification of each interaction object is encoded by special characters, e.g. button labels are enclosed in square brackets. Text attributes are also included in the braille notation, providing for a relatively concise form to convey this often important information.

Whenever braille rendering is not feasible or appropriate, synthetic speech and/or non-speech auditory cues are used.

GUIB supports non-visual interaction by means of keyboard input, and two mechanisms to substitute for pointer device interaction (as described above in the context of dynamic coherence). Another important aspect of interaction is notification about changes in the UI. GUIB implements a variety of cursors to mark specific UI changes, such as dialog pop-ups, label changes, new windows, . . . This is also an aspect of conveying semantic information.

User interaction can take place based on both the visual representation by means of the keyboard and the mouse, and the non-visual representation by means of the keyboard, braille cell selection, and the touch pad. Some operations can be accomplished by more than one input modality, while others are to be completed by combining interaction with more than one modality. In addition, there is a level of user interaction that is specific to GUIB and the GUIDE device. In terms of the CARE properties, this provides support for **”Complementarity”, ”Assignment”, ”Redundancy”, and ”Equivalence” for input**.

Aside from the regular visual representation provided by the system, GUIB also provides for output in the tactile modality by means of braille output, and in the auditory modality for synthetic speech and non-speech sounds. While coherence between the visual and non-visual representations is maintained, GUIB does allow for additional information to be presented in the non-visual interface (e.g. as a response to exploration), and some responses require output in multiple modalities. This indicates that GUIB conforms to the **”Complementarity”, ”Assignment”, and ”Equivalence” CARE properties for output**.

The very design of GUIB is centred on the concept of a **single conceptual model** across both representations in support of collaboration between sighted and blind users. In fact, Mynatt and Weber write [100]: ”the visual and nonvisual interfaces [must] support the same mental model of the application interface.”

GUIB supports **concurrent representations** in support of collaboration because it provides a non-visual representation alongside the visual form.

Advantages and shortcomings

As discussed in section 2.3, the HCI concerns related to non-visual access to GUIs were introduced by Mynatt, Weber, and Gunzenhäuser [100, 57], in part in relation to the description of GUIB. It is therefore no surprise that this approach satisfies the derived requirements for non-visual access rather well. The analysis of GUIB in terms of this part of the criteria presented in section 4.2 therefore serves more as a point of reference rather than an evaluation of the approach.

GUIB makes extensive use of multimodality for both input and output, which is a powerful feature because it allows the user interaction to be less cumbersome. It presents a less common approach of retaining the spatial metaphor of the graphical user interface, and also by putting the primary focus on tactile output.

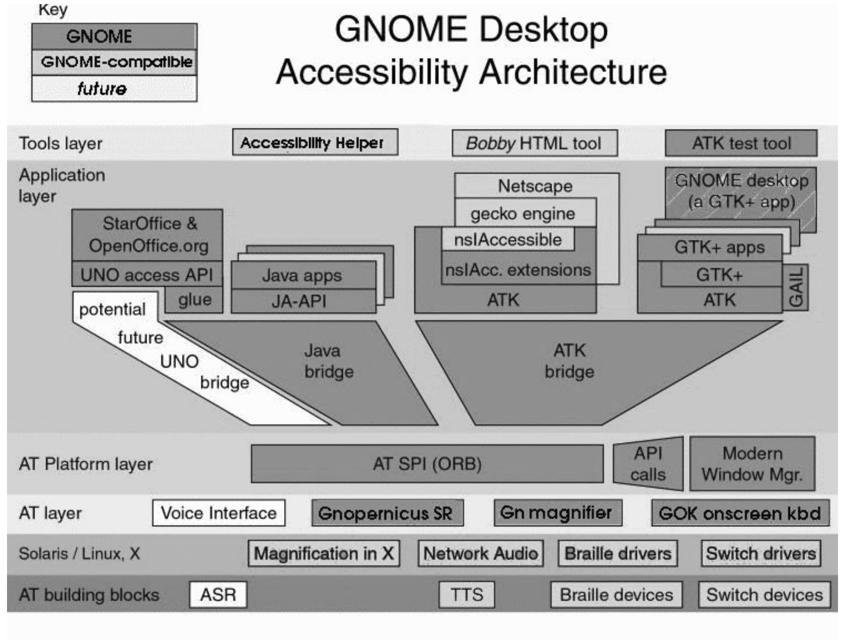
The very strong focus on supporting collaboration between sighted and blind users is a great advantage because it addresses an important need for blind users to be able to be productive in the work environment.

Although GUIB captures information from both the lexical and the syntactic levels, it still depends on the final UI level as source of information because even the syntactic info is toolkit specific. This limits the approach because it is not able to capture semantic properties and relations that are part of the UI design, but whose influence on the UI implementation is the only remaining trace of their existence, and this residual information is often not sufficient to reverse engineering the original semantics.

GUIB combines multiple modalities in a very successful manner, but at some cost in terms of usability. The GUIDE device provides braille output augmented with auditory feedback, but is limited to providing positional input events (braille cell selectors or touch pad). Text input and various aspects of application level navigation are still performed by means of the regular keyboard, which requires the user to switch between the two devices quite frequently.

GUIB was designed to provide a spatial metaphor based representation on a 25 line by 80 character braille cell matrix (effectively 200 by 160 dots), which according to Mynatt and Weber is sufficient to represent a 640 by 480 pixel screen[13]. Current graphical screen technology far surpasses that resolution (1400 by 1200 is not uncommon), yet using larger braille matrix devices is not a practical solution. The specialised hardware required for GUIB is a drawback because it limits its use by a larger population due to the high cost.

[13] The most common graphics standard in use on personal computers in the late 1980s, early 1990s was the Video Graphics Array (VGA) with a standard display resolution of 640 by 480 pixels.



(Reprinted from [58], with permission.)

Figure 4.8: The GNOME Accessibility Architecture

4.4.4 GNOME Accessibility Architecture (Orca)

The GNOME Project was started in 1997 by a team of students, with a very ambitious aim: developing a free desktop environment for UNIX-type systems. As the project matured, and gained both community and corporate support, providing facilities towards accessibility became important. This was in part driven by the realization that acceptance of the GNOME desktop by government and corporate entities would require support for assistive technology. Initial work culminated in the design of the GNOME Accessibility Architecture [137].

Figure 4.8[14] provides a schematic overview of the GNOME Accessibility Architecture [58]. Central to the design is the Assistive Technology-Service Provider Interface (AT-SPI), a system-wide interface that exposes modality-independent accessibility information about applications to AT providers such as screen readers, magnifiers, ... The AT providers are consumers of information posted to the AT-SPI, and they can also explicitly request information from an application through the AT-SPI.

[14] This diagram mentions Gnopernicus as the screen reader for GNOME. It has since been replaced by Orca.

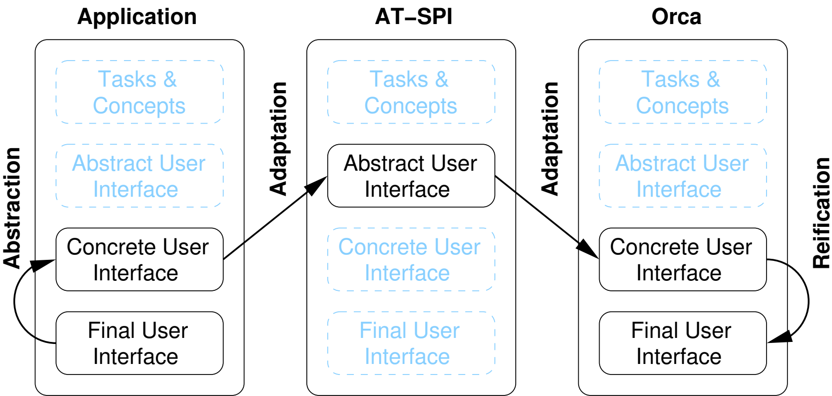


Figure 4.9: URF Diagram for the GNOME Accessibility Architecture

Applications come in many flavours, and the GNOME Accessibility Architecture is designed to cope well with the complexity imposed by the fact that applications may present their UI based on any of several available presentation toolkits. Java applications may use Java AWT or Swing to implement their UI, GNOME desktop applications use GTK+, and a web browser is likely to use the gecko engine for rendering the UI.

On the producer side of the AT-SPI, toolkit specific application UI information is mapped onto a common specification that is toolkit and modality independent. Java applications provide accessibility information through the Java Accessibility API. GNOME applications are typically implemented using GTK+ as the presentation toolkit, augmented with the Accessibility ToolKit (ATK). The Gecko engine uses ATK as well, whereas an application like OpenOffice.org provides its own API. The various distinct toolkits or toolkit extensions provide their information to AT-SPI by means of specific bridge components: the Java bridge, the ATK bridge, ...

A typical work flow therefore commences with user interaction resulting in a state change in a widget that is part of the UI of the application. By means of an accessibility API related to the presentation toolkit, the user interaction and/or its effect is passed through a bridge component to the AT-SPI, where it is made available to assistive technology providers.

From the perspective of the AT providers, applications present their user interface by means of an abstract toolkit, independent of modality or specific presentation.

Analysis

Figure 4.9 provides the Unified Reference Framework diagram for the GNOME Accessibility Architecture. Applications are developed against a specific presentation toolkit, typically implementing the UI programmatically, as a final UI. Accessibility information is provided either programmatically or obtained from an external resource, and it is used to annotate widgets at the concrete UI level (abstracted from the final UI) in order for the widgets to be exposed through an accessibility API.

The API bridge that links the accessibility API to the AT-SPI component essentially performs an adaptation operation that further abstracts the information to the abstract UI level. The AT-SPI exposes a modality and toolkit independent abstract UI to AT providers.

The AT providers use the information obtained through the AT-SPI as source for a reification process to yield the alternative UI representation as a final UI in the context of the AT provider.

The ability for an AT provider to accurately and appropriately present all UI elements from the visual interface is limited by the supplemental information provided through the toolkit specific accessibility API. Unless it is explicitly specified, system defaults are used, and those defaults offer minimal accessibility at best. Therefore, **static and dynamic coherence are partial, at best**.

All AT providers work off information obtained through the AT-SPI. This means that the off-screen model approach that historically has been at the foundation of most AT solutions can often be eliminated within the GNOME Accessibility Architecture. However, the consequence is that exploration without causing side-effects is often no longer possible because the AT provider must consult the application through the AT-SPI to obtain information such as e.g. the content of an as-yet closed menu. In conclusion, only **partial support for exploration** is available.

While the augmentative accessibility information that developers provide is often of reasonable quality, the visualisation of semantic information is not always perceived as requiring additional information. The abstraction process within the toolkit (from FUI to CUI), and again when information is passed to the AT-SPI (abstraction from CUI to AUI) has a reasonable potential for not identifying all semantics because it is often not represented at the concrete and/or final UI levels. There is only **partial support for conveying semantic information**.

The AT-SPI supports interaction events from AT providers to accommodate alternative input modalities. Widgets are exposed through the AT-SPI as abstract entities with defined role and attributes, making it possible to **ensure interaction** by defining modality specific alternatives for the distinct (and limited) set of supported roles.

While an AT provider acts on information from the AT-SPI, it is able to augment the representation at will (and often does). Likewise, interaction by means of alternative input modalities can be filtered to enforce certain modality specific actions. This amounts to **"Assignment" and "Equivalence" support for input and output** in terms of the CARE properties.

The alternative representations created by AT providers are based on a reification of the information provided through the AT-SPI, which is based on abstracting the visual representation at the final UI level. This ensures that there is only a **single conceptual model** involved.

The alternative representations of the UI are rendered by the AT providers as information is obtained from the executing applications, and they therefore are true **concurrent representations**.

Advantages and shortcomings

A significant advantage offered by the GNOME Accessibility Architecture is that it is an open source project. It is supported by an active community of developers and testers, and has received significant contributions from corporate entities as well. Furthermore, it is available for free, which is a very significant advantage over many other accessibility solutions.

As explained in this section, the GNOME Accessibility Architecture provides support for a variety of presentation toolkits. Given the fact that it is quite common to use a wider variety of applications within a single desktop environment on UNIX-type systems, offering cross-toolkit support is important.

In a posting to the GNOME Accessibility mailing list on July 20th, 2004, Peter Korn stated that GNOME was taking the approach that applications must "opt-in" to accessibility. This is typically done by having the application code explicitly call functions in the toolkit's accessibility API to provide information that can be queried later by means of AT-SPI, and by limiting oneself to using widgets and features that are well supported in the accessibility architecture. It also often involves providing additional UI annotations as an external resource that can be loaded by an accessibility component in the toolkit implementation. The downfall of this approach is that the process of keeping the UI description in sync with the application UI code is entirely manual.

The need to explicitly specify accessibility information for widgets opens up various other potential problem areas:

- The quality of the augmentative information provided drives the quality of the accessibility that can be provided for by AT providers. This places a

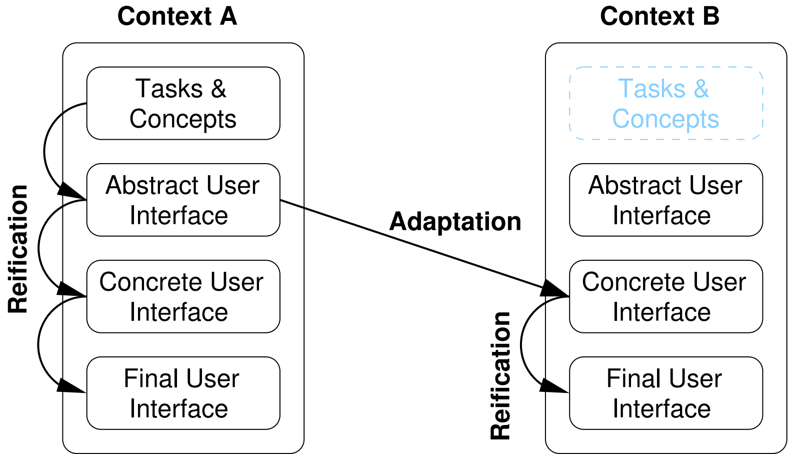


Figure 4.10: Adaptation of a Abstract User Interface

significant burden on application developers, who may not be experienced in this area. Alternatively, the information could be provided by AT experts, but they often do not have the expertise to programmatically specify the additional information.

- This approach makes it possible to add in accessibility support after development has completed. This most commonly implies that accessibility was not an integral aspect of the application design, which is likely to impact the quality of the support that can be offered by AT solutions.

- It may be tempting to developers to provide accessibility annotations for specific needs only rather than in an all-inclusive manner.

It is also important to recognise that there is a significant amount of popular applications that are not developed based on any of the supported toolkits. When these applications are used on a GNOME-based system (which is a quite common scenario), no accessibility information is available and AT providers are unable to assist the user.

4.5 Adaptation of an Abstract User Interface

As discussed in section 2.5, the use of abstract user interface descriptions has become more popular in recent years. The advantages range from supporting the separation of concerns paradigm, allowing expert designers to focus on

the UI design without being burdened with aspects of programming, automated UI implementation code creation, to presenting the UI by means of runtime interpretation.

The related works presented in this section all share a common path through the Unified Reference Framework as shown in Figure 4.10. The diagram shows the theoretical model where in context A the user interface is created as a multi-level reification from the specification of tasks and concepts, through to the final UI representation. Alternative representations are made possible through adaptation at the abstract UI level for a new context of use (B), and then reification in context B towards the alternative final UI. Note that the adaptation at the abstract level can be combined with the reification to the concrete level in context B as a single cross-level operation, and this is reflected in Figure 4.10 because it is commonly done in the works discussed in this section.

The URF model does not place any restrictions on the form in which the UI is specified at any of the abstraction levels, and the related works discussed in this section illustrate some of the variants.

4.5.1 MetaWidgets

Applications that are developed with a graphical user interface are commonly developed using libraries of standard widgets that represent UI elements that users are familiar with. Multimodal applications require a library of widgets defined at a higher level of abstraction, independent of a specific modality, to ensure that they can support interaction with the user in several modalities. Blattner, et al. have proposed widgets that can satisfy the requirements of multimodal application design: metawidgets [10, 163].

A metawidget is an abstract container of presentation widgets for a specific data item, providing methods for the selection of a specific representation. The selection of a representation may have temporal constraints, and even the actual representations may be time dependent. The mechanism for selecting a specific representation for a metawidget takes into account all aspects of the context of use, including user preferences, system resources, and environmental conditions.

One of the more unusual yet revolutionary conditions that were considered in the design of the metawidgets concept is cognitive load. This metric measures the demands on a user's cognitive system due to the multisensory interaction that results from using a multimodal application. As the cognitive load is affected by the current state of the interface, it influences the representation selection mechanism towards using representations that minimise the overall cognitive load. This may cause active representations to undergo metamorphosis in order to reduce the load.

The decision on what representation to use is therefore made at the moment the widget is to be presented to the user, and at any time when the context of use changes.

The fundamental assumptions that drive the metawidget design, and specifically the association with its representations are [52]:

- A metawidget must have one or more representations.
- The representations for a metawidget may be in the same or (preferably) different modalities.
- The representation to be presented by a metawidget is selectable at runtime, based on user preferences, as well as extra- and intra-system information.
- The representation of a metawidget may change (metamorphose) while it is in use, i.e. presented to the user.

Therefore, metawidgets need not be aware what representation has been selected at any given time, and the representation selection is not influenced by the actual information encapsulated by the widget or the modality of the presentation.

The collection of metawidgets comprises an abstract presentation toolkit that is used to develop multimodal applications. The application code interacts with the metawidget in a manner that is equivalent to how it would interact with a modality-specific presentation toolkit widget. The metawidget delegates the functionality to the actual representation.

Analysis

The MetaWidgets approach is rather unique in its support for dynamic selection of representation on a per-widget level. The Unified Reference Framework diagram in Figure 4.11 represents this design. The user interface is defined based on the abstract metawidget toolkit at the abstract UI level. As widgets are selected for presentation, each widget may map to a different context of use as a result of the selection mechanism in function of all aspects of the context of use, including the cognitive load. Furthermore, any widget may be re-presented at any given time. This is equivalent to a redirection of the adaptation operation for that specific widget.

It is obvious from the design discussed in this section that a widget cannot exist in any modality unless it also exists in the abstract UI specification. Furthermore, when a widget is part of the active presentation, it will be represented in one of the

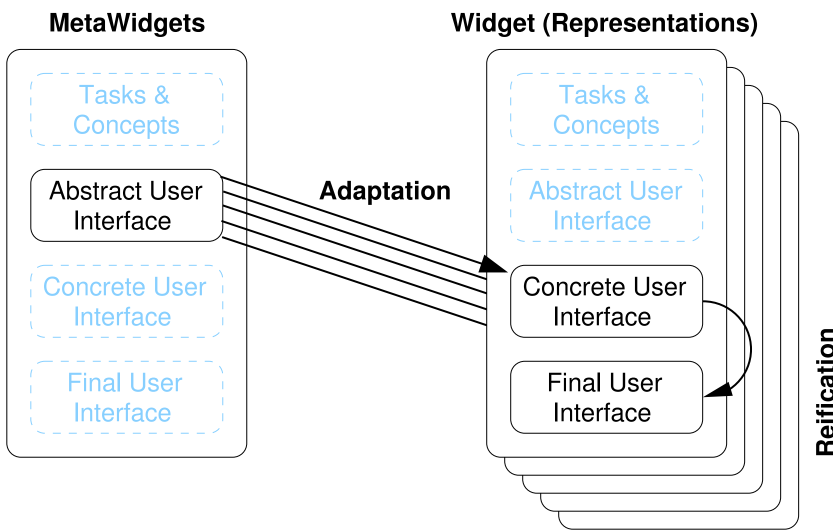


Figure 4.11: URF Diagram for MetaWidgets

active modalities. The interaction with a widget is dependent upon the modality it is presented in, however the overall semantics of the interaction are a reflection of the semantics defined in the abstract UI. In conclusion, **static and dynamic coherence are ensured** regardless of the chosen distribution of widgets across the modalities for any given stable point in time[15]. The fact that all interaction between the application and the user interface is handled by the metawidget component ensures that at all times a **single conceptual model** is represented.

The discussion concerning coherence also provides the basis for being able to conclude that **exploration, interaction, and conveying semantic information is ensured** with the MetaWidgets approach. Again, any failure to satisfy any of thee requirements would be an indication that the implementation of the MetaWidgets concept is flawed.

The very concept that (by design) all presentation widgets for a given metawidget are equivalent for both presentation and interaction specifies that **"Equivalence" for input and output** is provided for.

The design as presented by Blattner, et al. [10], and Glinert and Wise [163, 52] **does not support concurrency**. Some widgets may provide representations that utilise multiple modalities, but that would not constitute concurrency, but rather a multimodal representation for that specific widget.

[15]The condition of stability is introduced here to acknowledge that metamorphosis is not instantaneous and therefore short intervals where coherence is lost will occur when a widget goes through metamorphosis.

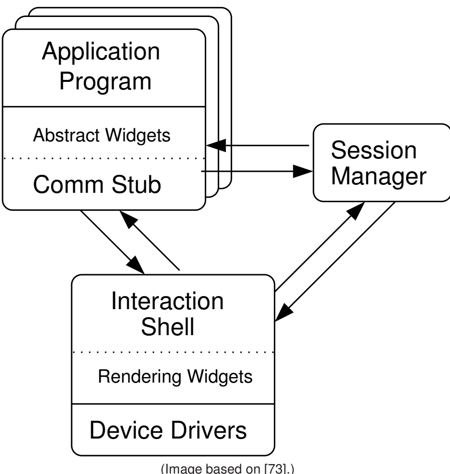


Figure 4.12: The ”Fruit” project

Advantages and shortcomings

MetaWidgets present a very novel approach to multimodal interfaces. With selectable representations at the metawidget level, this approach offers a level of flexibility that is remarkable.

One of the major shortcomings with this approach is the complexity of providing support for a new metawidget. Not only must the semantics be defined at the abstract level, but an implementation is to be provided all desired representations.

In view of collaboration between users, MetaWidgets introduces a complication that may be difficult to overcome. Because of the fact that the user interface changes based on context of use, where some parameters may be less stable, it is more difficult to ensure that a colleague can replicate the configuration of the user interface in order to discuss its operation.

4.5.2 Fruit

Kawai, et al. describe the architecture for a user interface toolkit that supports dynamic selectable modality: ”Fruit” [73]. The system accommodates the needs of users with disabilities and users in special environments by means of a model of semantic abstraction, where a separation of concern is enforced by decoupling the user interface from the application functionality. This allows users to select a UI representation that fits their circumstances as much as possible.

The ”Fruit” project has two immediate goals:

- Allow a user who is operating an application program to suspend his or her interactive session, and to resume it from another computer system.
- Allow for the interaction with an application program to switch from an auditory/tactile interface to a graphical interface without interrupting the execution of the application.

In this system, the application program is developed using a UI toolkit that provides abstract widgets, i.e. widgets that define functionality rather than the representation in a specific output modality. The rendering of the UI is delegated to an interaction shell that uses reified widgets, i.e. widgets that inherit from the corresponding abstract widget, and provide added functionality for the rendering of the widget in a specific modality.

Figure 4.12 shows the architecture of the Fruit project. Essentially, three main components can be identified:

- The communication stub: This is a library of abstract widgets to be linked with the application code. Effectively, the application's UI is designed and developed based on this toolkit library, equivalent to how it would be done based on a graphical toolkit. The abstract widgets implement the semantics of user interaction.
- Interaction shell: All interaction with the user is handled through an interaction shell. It provides input and output facilities through one or more modalities. In general, a user will use a single interaction shell to operate all his or her applications, though it should be possible to use multiple interaction shells simultaneously. The rendered widgets provide the representation of the UI in specific modalities.
- Session manager: This component manages and coordinates the association between applications (through their communication stub) and interaction shells. Applications register themselves with the session manager, and the user is then able to ”connect” to an application by means of their interaction shell of choice. It also supports suspend/resume operations to allow for switching between interaction shells.

Typical operation starts with a session manager running on a specific host[16], and the user starting an interaction shell of their choice on their system[17]. To invoke a new application, the user uses the interaction shell to signal the appropriate

[16] Each host where application code may be executed must have a session manager running.

[17] It is possible for all components to run on a single system.

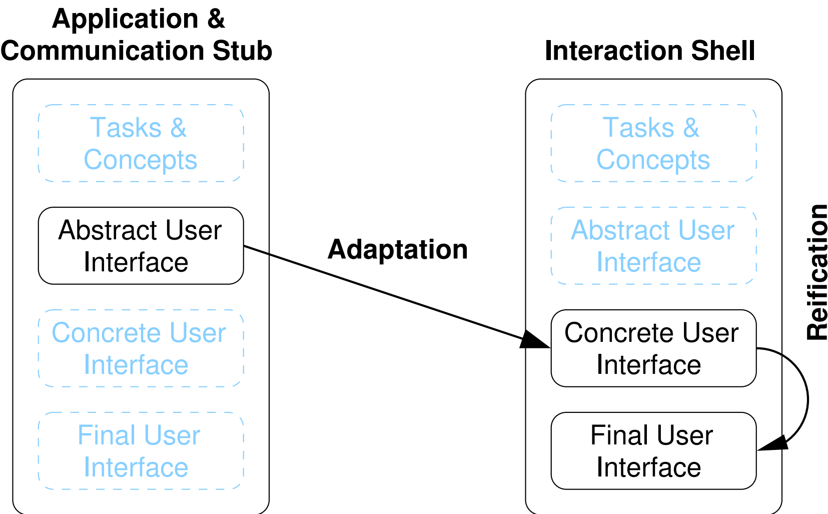


Figure 4.13: URF diagram for Fruit

session manager. The session manager handles the launching of the application, providing it with communication parameters so it can contact its controlling interaction shell.

The communication stub initiates contact with the interaction shell, and receives a reply that indicates the capabilities of the shell, e.g. whether it can process bit-mapped images or whether it support a pointer device. The communication stub uses this information to filter the information that is sent to the interaction shell so that only relevant information is transmitted.

User interaction takes place from this point forward as input is passed from the interaction shell to the application, and output is passed back. At any moment, the user can disconnect from the application, causing it to become suspended. User interaction can be reestablished from the same system or from another system, and with either the same interaction shell or a different one.

Analysis

The Unified Reference Framework diagram for the Fruit system (Figure 4.13) illustrates the overall flow of UI presentation. Note that with this approach, the target context may be one of multiple possible contexts. The application is developed programmatically using an abstract widget toolkit. Given that the toolkit is explicitly developed to be modality independent, this can be considered to be

a development at the abstract UI level. Upon establishing contact between the application's communication stub and the user's interaction shell, the rendering widgets are created to augment the abstract widgets in the application with a concrete reified form. By means of the actual presentation toolkit, the final UI is then generated.

In principle, static coherence should be maintained in this system because the rendering widgets are essentially instantiations of the corresponding abstract widgets. However, the Fruit system provides for filtering at the communication stub level based on capability information provided by the interaction shell, thereby allowing UI elements to be omitted from the UI because they are deemed irrelevant or impossible to render. As a result, Fruit **does not always uphold static coherence**.

The available documentation for this approach does not address input modalities beyond the standard keyboard and mouse. It is therefore (within the scope of the represented UI elements) obvious that **dynamic coherence is ensured**.

The system does not explicitly address the notion of modality-specific exploration of the user interface, yet it is possible to make a determination based on the actual design. The interaction shell is a self-contained entity, regardless of whether it is running on a secondary system or alongside the application, and has both total control over the user interaction and access to the user interface at both the concrete UI level and the final UI level. It is therefore quite realistic to ensure that **exploration is provided for**. Furthermore, this also ensures that **modality-specific interaction is ensured**.

As a result of the filtering of UI information that is applied at the level of the communication stub (effectively the abstract UI level), the **ability to convey semantic information is limited**.

Fruit does not address alternative input modalities, so no determination of CARE properties for input needs to be addressed. As far as output modalities are concerned, it is clear that **"Equivalence" is provided for output modalities**.

A **single conceptual model** drives the entire UI presentation chain as described in the URF diagram in Figure 4.13, because the final UI is created as the multi-level reification from the (adapted) AUI, through the CUI level, to the final UI representation.

Kawai hints at the ability to provide multiple representations by means of multiple interaction shells [73], albeit seemingly in a read-only fashion. It therefore seems prudent to consider this ability **partial concurrency** at best.

Advantages and shortcomings

The Fruit system combines two important goals: remote operation of applications with suspend/resume support, and the ability to switch between multiple user interface representations. This was a quite novel development, recognising the importance of providing multi-device user interface support for applications.

In recognition of the separation of concerns concept, Fruit also provides for a clean API between the communication stub and the interaction shells. This is an advantage in the overall design/development process because application developers need not be concerned with UI presentation details, while UI designers need not be concerned with application logic.

One of the disadvantages in terms of development and change management is that the UI is still constructed in a procedural, programmatic manner, making changes more cumbersome because they require a rebuilding of the application. Any rendering-specific annotations that might need to be specified at the application level need to be predetermined at build time because there is no mechanism to make runtime changes.

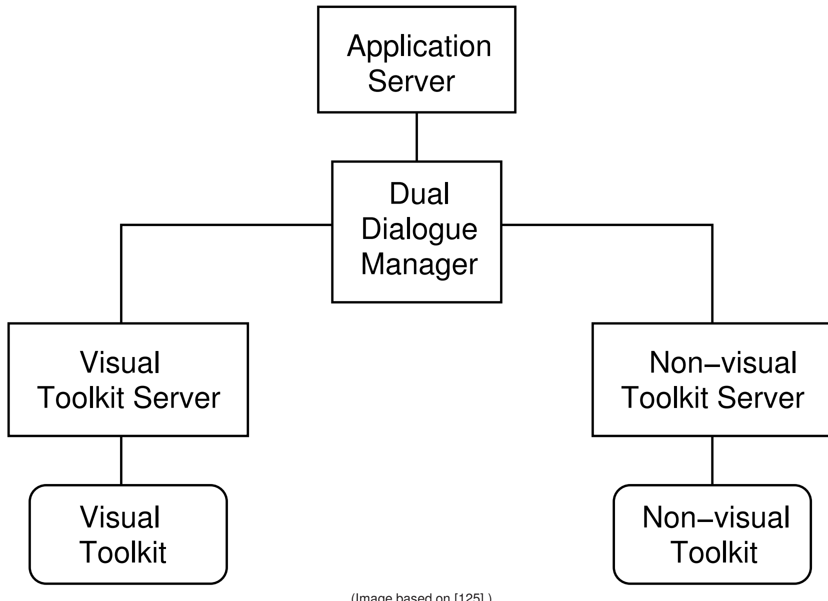
The filtering of information at the communication stub based on capabilities reported by the interaction shell is a concern because it violates the separation of concerns principle. It delegates functionality that is clearly an aspect of the final UI level to the abstract UI level. The filtering is a premature optimisation that perhaps should have been deferred until more data was collected to determine its necessity and to assist in formulating a better design.

4.5.3 HOMER UIMS

The HOMER UIMS [124, 125] is a language-based development framework, aimed at dual interface development: equal user interaction support for blind and sighted people. The dual interface concept is designed to hold the following properties:

1. *Concurrently accessible by blind and sighted people*
   Cooperation between a blind and a sighted user is recognised as quite important to avoid segregation of blind individuals in their work environment. The HOMER system supports cooperation in two modes: both users working side-by-side on the same computer system, or the users (not physically close to one another) working on their own computer systems.

2. *The visual and non-visual metaphors of interaction meet the specific needs of their respective user group, and they need not be the same.*



(Image based on [125].)

Figure 4.14: HOMER UIMS

This property expresses a potential need for interaction metaphors that are designed specifically for the blind. It is important to note that the HOMER UIMS design is in part addressing the perceived notion that the underlying spatial metaphor for the GUI system is by design based on visually oriented concepts, and therefore not appropriate for non-visual interaction. This property is as such not limited to just metaphors that relate to the visualisation of the UI.

3. *The visual and non-visual syntactic and lexical structure meet the specific needs of their respective user group, and they need not be the same.*
   This property (also in view of the exact meaning of the previous property) addresses the possible need for a separate non-visual interface design. The explicit mentioning of the syntactic and lexical structure in this requirement establishes the scope for the non-visual design proposed here as limited to the perceptual layer, although that is not specifically stated.

4. *At all times, the same semantic functionality is to be accessible to each user group through their respective modalities.*
   Essentially, the underlying functionality should be accessible to all users, albeit possibly through alternative modalities. Regardless of the modality, the functionality must be presented to all users in an equivalent manner.

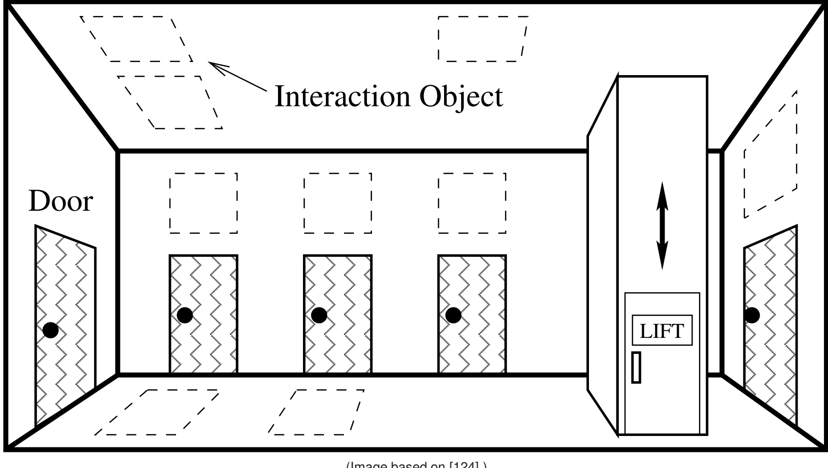
5. *At all times, the same semantic information is to be accessible to each user group through their respective modalities.*
   Similar to the handling of functionality as expressed in the point above, information should be made available to all users, by means of any appropriate modality that can guarantee equivalence.

The second and third properties are largely based on an analysis of sample user interfaces that employ highly visual idioms. Savidis and Stephanidis reach the very valid conclusion that the visual user interface may visualise information in a manner that is not accessible to blind users and likewise interaction techniques may be used that are inaccessible [124, 125]. Under those circumstances, it seems logical that a non-visual user interface (NVUI) would be designed with distinct non-visual features (metaphors and design) to provide an accessible solution. The fourth and fifth properties provide further requirements for the visual and non-visual designs.

Figure 4.14 provides a schematic overview of the ”Dual Run-Time Model” introduced with HOMER UIMS [124, 125]. The application server provides the program functionality for the system, and remains separate from the user interaction handling. All UI processing originates from the dual dialogue manager, where virtual interaction objects provide the underlying semantics of the user interface. An application programming interface channels semantic operational data between the application and the dual dialogue manager in order to maintain a separation of concerns. The actual realisation of the dual UI presentations is handled by means of an instantiation mechanism within the dual dialogue manager, where the visual and non-visual physical interaction objects that are associated with virtual interaction objects are created. These physical objects are rendered according to specific representation toolkits (as appropriate for chosen modalities) by means of the toolkit servers.

It is important to note that the HOMER UIMS does not require that every virtual interaction object is represented by a visual and a non-visual physical object. Likewise, even when dual physical interaction objects do exist, they need only implement those aspects of behaviour of the virtual object that are relevant for the modality in which the physical object is represented. A common use can be found in various visual effects that are associated with user interaction in a graphical user interface.

In view of the need for a non-visual representation of the user interface, Savidis and Stephanidis developed a new metaphor for non-visual interaction [123]: the ”Rooms” metaphor. The rationale for this new development can be found in the observation that existing approaches were ”merely non-visual reproductions of interactive software designed for sighted users” and that these approaches ”explicitly employ the Desktop metaphor for non-visual interaction.” The alternative presented by the authors is shown schematically in figure 4.15.



(Image based on [124].)

Figure 4.15: The ”Rooms” metaphor

The interaction space comprises a collection of virtual rooms, where each room acts as container for interaction objects:

- *Door*: A door is a portal to a neighbouring room at the same (vertical) level.
- *Lift*: A lift provides access to the room directly above or below the current room, allowing for a change in (vertical) level while retaining the same position on the horizontal plane.
- *Switch*: An on/off switch represents a toggle.
- *Book*: A book is a read-only text entity.
- *Button*: A control to activate some functionality.
- . . .

Interaction objects can be placed on any of the six surfaces of the room: the four walls, the floor, and the ceiling.

Analysis

Based on the schematic overview of the HOMER UIMS (Figure 4.14) it would seem that this approach essentially amounts to a dual path reification from a single AUI description to two distinct concrete UI contexts, both further reified into

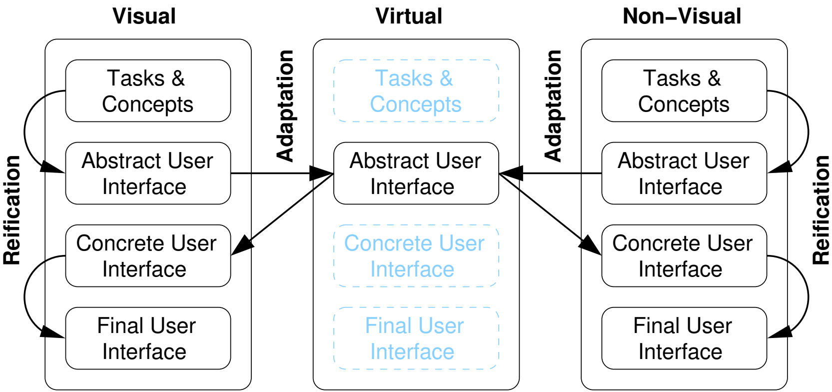


Figure 4.16: URF diagram for HOMER UIMS

thei respective final UIs: one visual, and one non-visual. Closer analysis of the UI development process in the HOMER UIMS however indicates a less trivial model.

The Unified Reference Framework diagram in Figure 4.16 illustrates the flow used for UI development in the HOMER UIMS. The most fundamental concept that HOMER is based on is the notion that the metaphors used in a graphical user interface are not appropriate for blind users, and that there is a need for alternative interaction metaphors that are deemed appropriate for blind users. The development process for the UI begins therefore at the tasks and concepts level, in two distinctly different contexts.

The specification of tasks and concepts in each context of use yields an abstract UI specification. Based on the visual and non-visual abstract UI descriptions, virtual interaction objects are defined that use the visual and non-visual components as alternative representations for a virtual entity. The virtual interaction objects comprise an abstract UI in a neutral context.

The virtual interaction objects (encapsulating both the visual and the non-visual instances) are adapted in both the visual and non-visual contexts, and yield final UIs in their respective modalities by way of reification.

Given that the visual and non-visual representations of the UI in the HOMER UIMS are actually derived from independent tasks and concepts specifications, **static coherence is not ensured**. The very design is based on **two distinct conceptual models**, which makes the **conveying semantic information concept not applicable** because there is no notion of providing a non-visual representation of information from the visual UI because they are for all intents and purposes **concurrent** yet independent.

Dynamic coherence is ensured although primarily on a semantic interaction

level rather than for step-by-step interaction sequences. The HOMER UIMS ensures that both contexts can perform the same tasks, albeit often in quite different ways.

Both contexts (visual and non-visual) provide a complete user interaction experience, including full support for **exploration and interaction**. Both operate in a very unrestricted manner because of the fact that each UI representation is based on what is perceived as the most appropriate modalities with metaphors of interaction that are tailored to the specific needs of the user population.

In terms of CARE properties as they apply to input and output modalities, there is a combination of **”Assignment” and ”Equivalence” for both input and output** because although tasks of semantic interaction are provided for in both environments in an equivalent manner, each environment also provides for some exclusive interaction and representation features.

Advantages and shortcomings

The HOMER UIMS takes a quite revolutionary approach to providing both visual and non-visual access to the same application. The authors made the assumption from the very beginning that the conceptual model (and metaphors) that are used for graphical user interfaces are not appropriate for blind users because they are based on visual concepts. Therefore, the design is strongly rooted in the decision to provide a more appropriate interaction metaphor (and essentially conceptual model) for blind users, and use that to develop a UIMS where a user interface for an application can be modelled and created in two different contexts of use: visual and non-visual. While this is a very interesting and perhaps groundbreaking approach, it also results in a situation where the blind users must be introduced to an entirely new world of interaction.

It also takes away from the opportunities for collaboration between users. Savidis and Stephanidis discuss collaboration as a consideration in the design of the HOMER UIMS [125] and characterise it in terms of proximity: local collaboration or remote collaboration. While this is certainly one of the dimensions of collaboration that can be considered, it is not necessarily relevant when considering blind and sighted people working together.

One of the problems concerning concurrent UI representations is related to the fact that user interaction can take place in multiple UI renderings at the same time. Obviously, some level of synchronisation may be needed. The HOMER UIMS implements a system where user interaction control is passed between users explicitly in what essentially amounts to a manual mutual exclusion protocol. While this system certainly works well, it remains unsure whether there is a true need for such explicit serialisation of user interaction.

4.5.4 Ubiquitous Interactors

Nylander introduced the concept of the Ubiquitous Interactor (UBI) in his doctoral work [105]. It presents itself as a solution to the difficulties of developing cross-device services. In UBI, a service can be represented with device-specific user interfaces, i.e. two devices may present the same service with quite different interfaces. At the same time, the actual interaction semantics remain the same across all devices.

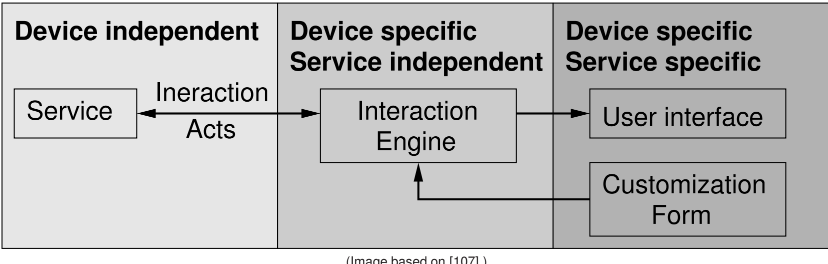
Rather than developing device-specific versions of each service, or using a more generic (but limiting) user interface such as web-based interfaces, UBI provides an approach where the service is developed as a device-independent entity with separable device-specific user interfaces.

The interaction between a service and users is expressed in terms of eight fundamental actions that can be used to describe the interface in a device independent way:

- *input*: Input that the user provides to the service.
- *output*: Output from the service to the user. This is the only interaction act that is sent from the service to the user.
- *select*: Make a selection from a set of alternatives.
- *modify*: Modify some data that is managed by the service.
- *create*: Create a service-specific object.
- *destroy*: Destroy a service-specific object.
- *start*: Start an interaction session with the service.
- *stop*: End an interaction session with the service.

These operations are abstract units of interaction that are independent of device, service, and modality.

The specification of the user-service interaction is written based on these operations, possibly grouping them together as a compound action. Each user interaction carries attributes that enable the service (and user interface) to uniquely identify the action, and to define additional features such as life cycle information, whether an element is modal, ... It is also possible to associate metadata with an interaction act to assist the UI rendering process in laying out the interface. The interaction acts are encoded in an XML-compliant interaction specification language.



(Image based on [107].)

Figure 4.17: The Ubiquitous Interactor

The user interface presentation for a specific device is generated based on the user-service interaction specification, based on device and service specific presentation information. This information is captured in customisation forms, providing directives to link interaction acts to specific UI elements, and resources (e.g. images, sounds, . . .) that are used in the rendering of the UI on the device.

Figure 4.17 provides a schematic overview of the UBI design, illustrating the interdependencies of the various components. A device specific interaction engine provides runtime interpretation of interaction acts and a customisation form to generate a suitable user interface for the service on a specific device. The interaction engine is also responsible for encoding user interaction as an interaction act prior to passing it on to the service. As part of an evaluation of the Ubiquitous Interactors approach, Nylander, et al. implemented interaction engines using Java Swing, HTML, Java AWT, and Tcl/Tk [107].

Analysis

The ubiquitous interactor approach creates user interfaces based on interaction specifications that are modality and toolkit independent, and therefore are at the abstract UI level in the Unified Reference Framework diagram in Figure 4.10. The user interface generation takes place at the final UI level, combining the interaction specification with a customisation form. It could therefore be argued that the adaptation should be modelled as a cross context, cross level operation from AUI level directly to FUI level rather than passing through the CUI level in the target context. However, upon closer investigation it is evident that the interaction engine operates both on the concrete UI level and the final UI level:

- Interpretation of interaction acts from and to the service operates on the concrete UI level because although it may have dependencies on the modality, it does not relate directly to the presentation toolkit.

- Generation of the actual user interface based on the interpretation of the interaction acts and the customisation forms depends on the actual presentation toolkit and therefore operates on the final UI level.

The ubiquitous interactor is designed to present itself with different user interfaces on different devices, providing different presentation information depending on the specific device being used [107]. Therefore, this approach **does not provide static coherence**.

On the other hand, one of the cornerstones of the design is that interaction is kept the same across all devices. This is accomplished with the interaction acts that are defined at the abstract UI level. This mechanism makes it possible to **ensure dynamic coherence**.

The approach presented in this section places its main focus on the generation of different user interfaces for various devices in terms of user interface presentation. User input is translated as *input* interaction acts and sent to the service for processing. The ubiquitous interactor therefore **does not provide non-visual exploration** features.

Based on the underlying principle of implementing the same interaction across all devices, regardless of differences in the presentation of the user interface, it is clear that the UBI supports **conveying semantic information** across all UI representations.

In order for the same interaction to be offered on all devices, context specific forms of interaction must be provided for. While the available sample interaction engines are all providing visual representations, the design behind the UBI approach is generic enough to **ensure non-visual interaction** if an interaction engine for non-visual representation were to be implemented.

As previously discussed, the ubiquitous interactor system does not provide any special processing functionality or support for input modalities. In all, it merely provides for adequate support to ensure that the required interaction acts can be provided on any given device. This does satisfy the requirements for **”Equivalence” for input modalities**.

For output modalities, i.e. representations of the UI, device characteristics and other aspects of the context of use may influence the presentation and can affect the availability of functionality. The TAP Broker service discussed by Nylander, et al. [106, 107] allows viewing of the complete transaction history in the desktop UI, while this option is removed from the UI in the Java AWT based presentation for small screen devices (e.g. cellular phones). The UBI support **”Equivalence” and ”Assignment” properties for output modalities**.

Given that all user interfaces for a service are derived from a single interaction specification, with presentation specific attributes specified by means of device dependent customisation forms, it is obvious that the UBI operates based on a **single conceptual model**.

Concurrency is not supported in the ubiquitous interactor design.

Advantages and shortcomings

The ubiquitous interactor allows service providers to develop services without having to be concerned about the presentation aspects of the user interfaces on various devices. This is a significant advantage in terms of development and maintenance effort because there is only a single service entity. The actual user interface can be adapted based on context of use by means of customisation forms, which need not necessarily be done by service developers. In addition, changes to customisation forms are easy and do not impact the service implementation at all.

This design also provides a very open ended mechanism for supporting a wide variety of devices, and ensures that even future devices can be supported by providing an interaction engine for any such device. Given that there are only eight fundamental operations used in interaction acts, support for a new device is quite a bit more manageable.

While the claim is made that interaction remains the same for all devices, the TAP Broker sample service [106, 107] illustrates that there is no guarantee that the same functionality is present on all devices. It therefore seems prudent to reinterpret the claim to mean that *if* a certain functionality is provided in a user interface, *then* it's interaction semantics will be equivalent to those in alternative interfaces on other devices.

The ubiquitous interactor does not provide device-level input processing that is presentation specific beyond translation from input events to interaction acts. Providing device-dependent exploration support would therefore require an entirely new software component.

The lack of concurrency support combined with the possibility that some interactions are not included in a specific device dependent user interface limits the level of support for collaboration between users in different contexts of use rather significantly.

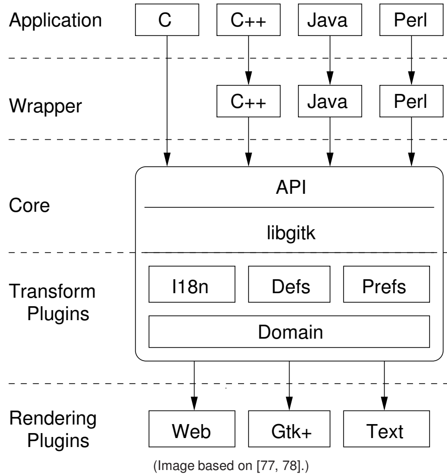


Figure 4.18: GITK

4.5.5 GITK

Stefan Kost developed a system to generate user interface representations dynamically based on an abstract UI description [77, 78]. His thesis centres on the modality-independent aspect of the AUI description, providing a choice of presentation toolkit for a given application by introducing a meta-toolkit: the Generalised Interface ToolKit (GITK).

Rather than providing facilities for programmatic user interface construction, a markup language is used to describe the user interaction dialogues: the Generalised Interface Markup Language (GIML). The GITK transforms the abstract UI description in GIML into a UI representation that is tailored to the context of use.

Figure 4.18 provides an overview of the multi-layered architecture of the GITK. Five distinct layers can be identified:

- **Application**: This layer contains the actual implementation of the application logic. It contains both the data objects and the abstract UI descriptions.
- **Wrapper**: Although GITK is implemented in the C programming language, applications can be written in C, C++, Java, or Perl. Language specific wrappers are provided to allow the applications to interface with the GITK core by means of its C-based API.
- **Core**: The actual transformation from abstract UI description to UI representation takes place in this layer. It encapsulates the API to the

application (possibly by means of a wrapper object) and the transformation logic. It creates a pipeline of transformation plug-ins that take an abstract UI description and yield a transformed UI description that can be used by a rendering plug-in for presentation to the user.

- **Transformation plug-ins**: These components each provide a very specific adaptation to the user interface. Each plug-in performs a GIML-to-GIML transformation, so that it is possible to combine them in any configuration as a processing pipeline.
- **Rendering plug-ins**: The rendering plug-ins use the transformed UI description to compose the final user interface within the context of a specific modality-dependent rendering toolkit.

This architecture allows for a multi-dimensional adaptation. Domain independent transformations are carried out early in the processing pipeline, and the UI description is then handed off to the target domain for further processing (which may involve additional adaptation). These later transformations are domain dependent.

It is theoretically possible to accommodate runtime changes to the adaptation profile, yielding changes in the transformation pipeline. When such a change is necessary, the user interaction processing is temporarily suspended while the change is applied. Once the UI description is re-processed by the transformation pipeline, the user interaction processing is restarted using the new final user interface representation.

Analysis

The GITK approach can be described well using the Unified Reference Framework diagram shown in Figure 4.10. It takes a UI description in GIML at the abstract level, and transforms it into a concrete UI description by a combined process of adaptation and reification. Further reification yields the final UI, rendered by a specific rendering plug-in.

An evaluation of the GITK approach in view of the HCI aspects of non-visual access to UIs can only be done based on theoretical analysis of the design because non-visual representation of the UI is not discussed in [77, 78] except as possible future work.

Static coherence is ensured by the GITK by virtue of its design. The transformation process makes modifications to the various interface elements, adding annotations, default attributes, etc... without changing the logical structure of the interface. Therefore, elements are never filtered out, although

some may be transformed into more simple, more complex, or even compound elements.

Dynamic coherence is ensured as well because of the design contract between the GITK core and the rendering plug-ins. Rendering plug-ins must ensure that the user interaction semantics for any given interface element are supported in the representation it renders.

The GITK approach implements a multi-level reification from abstract UI to final UI, where the handling of both user interaction and UI presentation is delegated to the rendering plug-ins. In the existing design, only a single rendering plug-in can be active at a given time, ensuring that all user input events will be processed by that plug-in. It would therefore be technically possible for the rendering plug-in to implement additional user interaction functionality for the purpose of UI exploration beyond the control of the application, i.e. in such way that the exploration is essentially invisible to the application. However, given that the current design and implementation does not address the possibility for implementing such features, one must conclude that based on the available information, **non-visual exploration is not supported**.

One of the important design choices made in GIML is the need for media-neutral attributes on interface elements. The transformation pipeline will use this information to annotate the UI description with rendering-specific information. E.g. while emphasis in text is often visualised using italics, it would be wrong to conclude that all text in italics is emphasised. The abstract UI description provides semantic information in a media-neutral way, and therefore it is possible to ensure that all rendering plug-ins can **convey semantic information** in a modality appropriate way.

As discussed earlier in this section, the GITK approach does ensure dynamic coherence in its UI representations. Although non-visual representations of the UI is not explicitly discussed in the GITK design, it is possible to make the determination that due to the commitment to dynamic coherence, a non-visual rendering plug-in would **ensure non-visual interaction**.

The very core of the GITK design is to ensure that the user interface of an application can be rendered by means of any of the supported rendering plug-ins, without any loss of functionality. The correctness of a rendering plug-in implementation is essentially based on its ability to render the UI to be equivalent (both in presentation and interaction) with all other representations. Therefore, the GITK approach provides **"Equivalence" for input and output**.

The GITK approach presents a user interface toolkit to the application, with a minimal API. Rather than providing functionality for programmatic construction of the UI, it supports loading abstract UI descriptions in GIML, binding data to (abstract) interface elements, and establishing callbacks. The GIML documents

provide the functional description of the interface, thereby reflecting the underlying conceptual model. The transformation pipeline that applies adaptations to this UI description implements a multi-level reification process, and therefore regardless of the chosen transformations and rendering plug-in, any final UI will reflect the same **single conceptual model**.

Stefan Kost acknowledges the importance of being able to provide multiple interface instances in support of collaboration between users, and more specifically in terms of multimodality, CSCW where some of the users may present with special needs. Some ideas are offered towards a possible design on a theoretical level, but they have been left as potential future work. Therefore, **GITK does not support concurrent representations**.

Advantages and shortcomings

The GITK approach provides multimodal interfaces based on abstract interface descriptions that are truly modality independent. This offers a high degree of flexibility and makes it possible to support a potentially endless number of different representations. The transformation pipeline based adaptation process supports a clear separation of interface aspects throughout the system, and makes it possible to support runtime changes to the context of use. This design also ensures that all representations are based on a single conceptual model, and that both static and dynamic coherence are provided for.

Stefan Kost notes in [78] that interaction serialisation is a problem related to the layout of a representation, and states: "in a usable interface the interaction sequence should match the layout of the interaction objects." However, doing so would render any navigational aspects of the user interaction semantics irrelevant. This approach also implies that there is no consistency between different representations in terms of logical navigation order of interface elements. A better approach would be to have the logical navigational flow through the interface elements influence the modality-specific layout decisions.

The GITK approach provides for a clean separation of interface aspects, and carries all aspects of the interface from abstract description through to the final UI representation provided by the rendering plug-in. This works well for the overall goal of being able to render the interface for an application in one of multiple different modalities, yet it introduces additional complexity when concurrent representations are considered. By delegating both the interaction handling and the presentation to the rendering plug-ins there is no coordination between the representations in terms of e.g. focus management. The solution proposed in [78] of implementing cross-instance communication falls short of being acceptable because it introduces semantics at the presentation level. A more appropriate solution centralises the user interaction processing at a higher level of abstraction.

Separating the user interaction processing from the presentation is also more appropriate in view of upholding the separation of concerns principle.

Another aspect of the lack of separation between interaction and presentation is that input and output modalities are combined as a rendering plug-in context. This coupling of input and output modalities results in duplication of effort in the implementation of rendering plug-ins because many share some (or all) of the input modalities.

4.6 Conclusions

The various projects discussed in this chapter represent the extensive field of research concerning HCI, multimodal interfaces, and accessibility. The results of the analysis of the primary related works presented in sections 4.4 and 4.5 are summarised in Table 4.1 and Table 4.2. For each project, the evaluation criteria introduced in section 4.2 are addressed.

The remainder of this section first presents the primary shortcomings of the state of the art in section 4.6.1, and then concludes with a set of requirements in section 4.6.2 that are derived from the analysis of the state of the art and the identified shortcomings. These requirements will be used as basis for the work presented in this dissertation, and also in chapter 8 to evaluate the presented work.

4.6.1 Shortcomings

The following shortcomings have been identified in the state of the art. They serve as a starting point for formulating requirements for the design of an approach to provide equivalent representations of multimodal user interfaces.

The related works can be separated in two groups based on the primary goal that they are trying to resolve, and the shortcomings will be presented in the same fashion.

Accessibility of user interfaces

- *Lack of semantic information*: When a final UI representation is used as primary source of information for constructing an off-screen model at a higher level of abstraction, most semantic information about the UI

		Archimedes		TIDE/VISA	GUIB	GNOME Accessibility Architecture
		TAS	VisualTAS			
URF Diagram		n/a	T&C T&C AUI AUI CUI CUI FUI FUI	T&C T&C AUI AUI CUI CUI FUI FUI	T&C T&C AUI AUI CUI CUI FUI FUI	T&C T&C T&C AUI AUI AUI CUI CUI CUI FUI FUI FUI
Coherence	Static	n/a	Yes	Yes	Yes	Partial
	Dynamic	Yes	n/a	No	Yes	Partial
Exploration		n/a	Yes	Yes	Yes	Partial
Conveying semantic information		n/a	Partial	Partial	Yes	Partial
Interaction		Yes	n/a	No	Yes	Yes
CARE	Input	RE	n/a	AE	CARE	AE
	Output	n/a	E	E	CAE	AE
Conceptual model		Single	Single	Single	Single	Single
Concurrency		n/a	Yes	Yes	Yes	Yes
Cost factors		Specialised hardware	Specialised hardware	2nd computer, specialised hardware	Specialised hardware	Free

Table 4.1: Comparison of related works – 1 of 2

		Meta-Widgets	Fruit	HOMER UIMS	Ubiquitous Interactor	GITK
URF Diagram		T&C T&C AUI AUI CUI CUI FUI FUI	T&C T&C AUI AUI CUI CUI FUI FUI	T&C T&C T&C AUI AUI AUI CUI CUI CUI FUI FUI FUI	T&C T&C AUI AUI CUI CUI FUI FUI	T&C T&C AUI AUI CUI CUI FUI FUI
Coherence	**Static**	Yes	Partial	No	No	Yes
	Dynamic	Yes	Yes	Yes	Yes	Yes
Exploration		Yes	Yes	Yes	No	No
Conveying semantic information		Yes	Partial	No	Partial	Yes
Interaction		Yes	Yes	Yes	Yes	Yes
CARE	**Input**	E	n/a	AE	E	E
	Output	E	E	AE	AE	E
Conceptual model		Single	Single	Dual	Single	Single
Concurrency		No	Partial	Yes	No	No
Cost factors		Specialised toolkit	Specialised toolkit	Dual design with specialised toolkits, specialised hardware	Specialised design tools	Specialised toolkit supporting dynamic updates

Table 4.2: Comparison of related works – 2 of 2

remains unavailable because it is not explicitly represented in the final UI presentation.

- *Lack of generalisation*: When a final UI representation is used as primary source of information, the approach depends on direct and detailed knowledge about the output modalities used in the primary representation. When changes occur in the rendering of the primary representation, extensive modifications are required to maintain the same level of functionality. It is usually also not possible to use the approach with any other source modality.
- *Lack of separation between contexts*: When a user accesses the presentation of a UI in an alternative modality, while still performing most (if not all) interaction by means of the same input modalities as a user of the primary presentation, user interaction is sub-optimal, and the separation of concerns concept is negated.
- *Lack of extendibility*: Assistive technology solutions tend to target a specific needs category, and fail to generalise their ability to provide alternative representations across other modalities.
- *Poor support for collaboration*: It is not sufficient to provide an alternative representation of the user interface based on the primary representation in order for users with differing abilities and/or needs to be able to collaborate successfully. Both static and dynamic coherence should be maintained to ensure that the representations are equivalent.

Multimodal user interfaces

- *Lack of support for concurrent representations*: While techniques for presenting a UI in a modality of choice offer a high degree of flexibility, multiple concurrent representations are needed in order to support a higher level of collaboration between users with differing abilities and/or needs.
- *Lack of separation between contexts*: When a multimodal UI system provides a choice of output modality independent from the input modalities used for user interaction, the separation of concerns concept is no longer maintained.
- *Lack of extendibility*: Programmatically defined user interfaces based on abstract toolkits must be able to support any available presentation component, and ideally any future presentation components that may be implemented for alternative modalities.

- *Lack of separation between application and presentation*: When the user interface is generated during system development, and compiled into the application as program code based on an abstract presentation toolkit, UI presentation aspects cannot be updated without updating the entire application.

4.6.2 Requirements

An approach that is able to provide the ability expressed in the thesis statement in section 1.3 is essentially one that combines elements from solutions that provide accessibility and elements from solutions that provide multimodal user interfaces.

1. *Multimodal input and output*: The proposed approach shall support the use of multiple distinct input and output interaction modalities. This requirement is implied by the very definition of multimodal user interfaces (Definition 1.4 on page 12), and it is explicitly specified in the thesis statement in section 1.3.
2. *Separation of concerns*: This important software development concept was introduced by Parnas [112]. Multiple shortcomings found in the state of the art relate to a failure to maintain separation of concerns. Specifically, a strict separation between application logic, user interaction semantics, and UI presentation should be maintained.
3. *Equivalent representations*: Analysis of the state of the art (summarised in Table 4.1 and Table 4.2) shows that both static and dynamic coherence are prerequisites for resolving the HCI concerns presented in section 2.3. Ensuring that all representations satisfy static and dynamic coherence amongst one another establishes equivalence between the representations. This requirement is directly related to the concept of equivalent representations specified in the thesis statement.
4. *User interface design that is independent of any modality*: In order to be able to provide UI representations in any modality, the UI design must be truly modality-independent. This means that the user interface design is handled at the abstract UI level. The discussion of the state of the art and the shortcomings derived thereof shows that abstraction from the final UI level does not provide sufficient information to support presenting the UI in other modalities while maintaining dynamic coherence.
5. *Use of abstract user interface descriptions*: The shortcomings derived from the analysis of the state of the art show that programmatically defined user interfaces pose complications in terms of maintainability, and in terms of

support for future presentations. Providing the (abstract) UI specifications in the form of UIDL-compliant descriptions ensures that many updates can be carried out without requiring changes to the application code. The use of abstract user interface descriptions supports modality-independent UI design.

6. *Runtime reification*: When the UI is defined in a UIDL-compliant description, providing multimodal representations amounts to a runtime reification process that yields a final UI in the context of use. Performing the reification process at runtime promotes maintainability and it reinforces the separation of concerns.

7. *Concurrent representations*: Full collaboration between users with differing abilities and/or needs requires that all users can observe and interact with the UI in an equivalent way. When the UI can only be rendered in a single presentation at any given time, users must either collaborate based on recollection (a memorised mental image) or engage in context switching between presentations, which is very counter-productive.

Chapter 5

Parallel User Interface Rendering

"Our grand business undoubtedly is,
not to see what lies dimly at a distance,
but to do what lies clearly at hand."
(Thomas Carlyle, "Signs of the Times", 1829)

The Parallel User Interface Rendering approach to providing alternative representations of graphical user interfaces is based on the significant advantages of abstract UIs, described in a sufficiently expressive UIDL. It also builds on the ability to present a UI by means of runtime interpretation of the AUI description. This chapter introduces the design principles that lay the foundation for this novel technique, and it provides details on the actual design. Based on the discussion of GUIs and accessibility in chapter 2 an analysis is presented of how the PUIR approach satisfies the requirements for an accessibility solution for graphical user interfaces.

5.1 Introduction

Existing accessibility solutions for supporting blind users on graphical user interfaces (on UNIX-type systems and elsewhere) are mostly still "best-effort" solutions, limited by the accuracy of the off-screen model that they derive from the GUI. The OSM is created based on a variety of information sources. Much of the syntactic information can be obtained from hooks in the graphical toolkit,

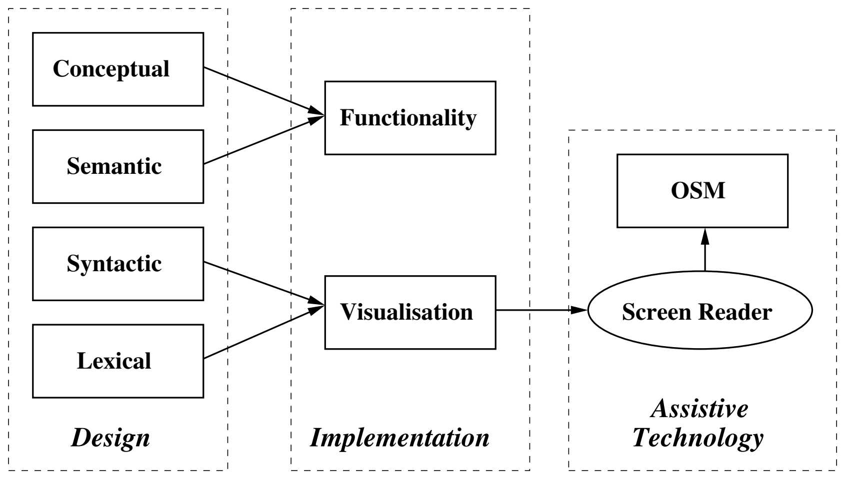


Figure 5.1: Screen reader using the visual UI as information source.

and by means of support functions in the graphical environment. In addition, various toolkits provide an API for AT solutions to query specific information about the GUI. Sometimes, more advanced techniques are necessary, such as OCR and interposing drivers to capture data streams for low-level analysis [59]. Further enhancement of the OSM information often involves complex heuristics and application-specific scripting.

Figure 5.1 provides a schematic overview of the screen reader implementation that is typically used in current AT solutions. As described above, the screen reader derives an off-screen model from the visual representation of the GUI[1]. The visualisation part of the implementation is created based on the syntactic and lexical portions of the UI design, and therefore the screen reader is unable to access important semantic information about the UI [19]. This has proven to be a significant limitation, often requiring application specific scripting on the side of the screen reader to essentially augment the OSM with semantic information. Obviously, this is a less than ideal solution because it requires UI data to be maintained outside of the actual application. Accessibility is also limited in function of the ability of the script writer to capture application semantics accurately.

Clearly, OSM-based screen readers still operate entirely based on information from the perceptual layer, and they are thereby limited to providing a translated

[1] Even though some of the data capturing may take place between the application and the graphical toolkit (e.g. by means of an interposing library), and therefore prior to the graphical rendering, the data can still be considered visual because the application usually either tailors the data in function of the chosen representation, or it passes it to specific functions based on a chosen visualisation.

reproduction of the visual interface. This is an improvement over the traditional approach of interpreting the graphical screen, but it still depends on strictly visual information, or an interpretation thereof. The complications related to this approach are reminiscent of a variation of the “Telephone” game, where a chain of people pass on a message by whispering from one to the next. Only, in this case the first person (Designer) describes (in English) a fairly complex thought to the second person (Implementation), who explains the thought to the third person (Visual Presentation), who writes down the actual information and passes it to the fourth person (Screen Reader, who does not read English fluently), who then verbally provides the last person (You) with a translation of the message. The probability that the message passes through this chain without any loss of content is essentially infinitesimal.

Various research efforts have focused on this problem throughout the past 25–30 years (e.g. [125, 15, 79]), with mixed success and often quite different goals.

Parallel User Interface Rendering is a novel approach based on the following fundamental design principles:

1. A consistent conceptual model with familiar manipulatives as basis for all representations.
   See section 5.2.1.

2. Concurrent use of multiple toolkits at the perceptual level.
   See section 5.2.2.

3. Collaboration between sighted and blind users.
   See section 5.2.3.

4. Multiple coherent concurrently accessible representations.
   See section 5.2.4.

   (a) The same semantic information and functionality is accessible in each representation, at the same time.
   (b) Each representation provides perceptual metaphors that meet the specific needs of its target population.

This chapter commences with a discussion of the design principles for the Parallel User Interface Rendering approach in section 5.2, and they are further refined into the actual design that is presented in section 5.3. A brief comparison with one of the cornerstone paradigms of UI design, the Model-View-Controller, is presented in section 5.4. The chapter closes with the conclusions in section 5.5.

5.2 Design principles

This section provides a detailed discussion of each of the four design principles listed in section 5.1 that form the foundation for the Parallel User Interface Rendering approach. Further refinement towards an actual design is deferred to section 5.3.

5.2.1 A consistent conceptual model with familiar manipulatives as basis for all representations

The conceptual model is by far the most important design principle for any user interface, and therefore lies at the very basis of Parallel User Interface Rendering. Seybold writes (discussing the Star project at Xerox PARC) [128]:

> "Most system design efforts start with hardware specifications, follow this with a set of functional specifications for the software, then try to figure out a logical user interface and command structure. The Star project started the other way around: the paramount concern was to define a conceptual model of how the user would relate to the system. Hardware and software followed from this."

Smith, et al. define a user's conceptual model as [130]:

Definition 5.1. *Conceptual Model: The set of concepts a person gradually acquires to explain the behaviour of a system.*

Based on this very definition one might conclude that a conceptual model is a user specific model, based on personal experiences while operating or interacting with a system. While that is generally true, ample examples can be found that indicate that there are many "systems" for which a collective conceptual model exists, i.e. a model that is shared by most people. Examples include the operation of household appliances, driving an automobile, ... In some cases, the same conceptual model may develop independently for multiple individuals as they interact with a system in the same environment, so that their experiences are sufficiently similar. More often however, a conceptual model becomes a collective conceptual model through teaching, be it direct or through guidance.

Given the need to define a conceptual model for a user interface, designers have essentially two choices: design the user interface based on a existing model employing familiar metaphors, or develop a brand new model[2]. Extensive

[2]This not only involves defining objects and activities (functionality of objects), but also developing strategies to introduce the new model to the anticipated user population.

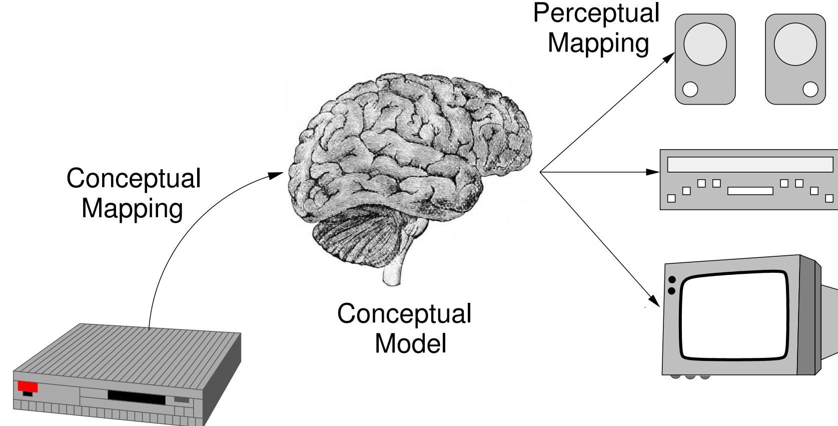


Figure 5.2: The role of the conceptual model

research was done when the original GUI concept was developed, leading to the conclusion that the metaphor of a physical office is an appropriate model [130]. It is however important to note that this conclusion was reached in function of developing a user interface for visual presentation, and non-visual interaction was therefore not taken into consideration.

Is it then possible to have a consistent conceptual model as basis for both visual and non-visual representations, or does the lack of sight necessitate a specialised non-visual conceptual model?

Savidis and Stephanidis suggest that specific interaction metaphors are to be designed for non-visual interaction, because the "Desktop" metaphor is visual by design [123, 125]. This seems to be contradictory to the design principle presented in this section, especially given that Gaver explicitly states that [50]: "The desktop metaphor [...] is the result of a conceptual mapping.", thereby clearly associating the Desktop metaphor with the conceptual model. Savidis and Stephanidis however do not differentiate the multiple layers of metaphors in a user interface the same way as Gaver does, as evidenced by an important statement of scope in their work [123]: "The work [...] concerns the general metaphor for the interaction environment (i.e. how the user realizes the overall interaction space, like the Desktop metaphor) which is different from metaphors which are embedded in the User Interface design." They associate the Desktop metaphor with the perceptual level instead. Their work is therefore not contradicting the design principle presented here. Furthermore, it actually provides support for the design principle of needs-driven perceptual metaphors for representation, as discussed in section 5.2.4.

Kay presents some compelling arguments to avoid the terminology of metaphors in relation to the conceptual model [74], instead referring to “user illusion” as a more appropriate phrase. Indeed, conceptual metaphors used in user interface design are more often than not a weak analogy with their physical counterpart. While the notion of a “paper” metaphor is often used within the context of a word processing application, users are not limited by the typical constraints associated with writing on physical paper. Moving an entire paragraph of text from one location to another on the “paper” is quite an easy task on a computer system whereas it is rather complicated to accomplish on physical paper. Smith, et al. expressed a similar notion when expressing that: “While we want an analogy with the physical world for familiarity, we don’t want to limit ourselves to its capabilities.” [130] It is clear that the underlying conceptual model is therefore a tool to link the internal workings of the computer system to a model that the users can relate to, without requiring the model to be bound by constraints of the physical world, and without any assumptions about representation (see Figure 5.2). The mental model reflects the metaphors that link the computer reality[3] to a task domain.

With the relaxed interpretation of metaphors and under the assumption that the separation between the conceptual and perceptual layers is maintained, the real question becomes whether a blind user can fully comprehend the conceptual model for a user interface. At this level, both sighted and blind users are faced with a mental model[4] that captures manipulatives[5]. The common model represents an office or a desktop, concepts that sighted and blind people are certainly familiar with. Whether a user can conjure up a visual image of the mental model is not necessarily relevant in view of the assumed separation from the perceptual. Still, often people will use visual imagery to represent the model in their mind, regardless of whether that is truly necessary to reason about it, and this applies to both sighted and many blind users. The exact nature of this mental image and whether it is truly visual or perhaps perceptual in an alternative form is a matter of individual preference and/or ability. Edwards states in a very insightful yet unpublished paper (in draft) [39]: “If an existing visual interface is to be adapted, it may be that there are aspects of the interface which are so inherently visual that they will be difficult to render in a non-visual form [. . .] but the suggestion is that this need not be the case, if the designer would not commit to a visual representation at an early stage of the design process.” This early stage would correspond with the development of the conceptual model for the user interface, which by design should be medium-independent in order to support this level of accessibility [41].

[3]The “computer reality” is defined by Gaver as [50]: “the domain in which computer events are described, either by reference to the physical hardware of the system or its operations expressed in some programming language.”

[4]Sighted users may often not even realize that a mental model is involved due to the fact that a GUI is generally presented visually using iconic elements that are closely related to the underlying conceptual (mental) model.

[5]Objects and the manipulations that are possible on and with those objects.

In an inspiring article in The New Yorker, Oliver Sacks wrote about several blind individuals who all experienced the effects of blindness on visual imagery and memory in different ways [120]. One individual effectively lost not only the ability to visualise but in fact the very meaning of visual concepts such as visible characteristics of objects and positional concepts based on visual imagery. Other people reported an enhanced ability to use visual imagery in their daily life. In addition, Sacks also wrote about sighted people who did not possess visual imagery. Noteworthy is that none of the people discussed in Sacks' article seemed to have any difficulties leading a functional and successful life in a predominantly sighted world. Whether one can visualise the physical does not seem to impact one's ability to operate in and interact with the world. Within the context of conceptual mental models, the focus should be on on what objects exist in the model, and what one can do with them, rather than what objects look like or how they function.

The research and analysis presented in this section supports the notion that, when a clear separation between the perceptual and the conceptual is maintained, a single conceptual model for a user interface can be appropriate for both blind and sighted users. There is therefore no need to consider a separate non-visual UI design at the conceptual level.

5.2.2 Concurrent use of multiple toolkits at the perceptual level

Providing access to GUIs for blind users would be relatively easy if one could make the assumption that all applications are developed using a single standard graphical toolkit *and* if that toolkit provides a sufficiently feature-rich API for assistive technology. Unfortunately, this situation is not realistic. While the majority of programs under MS Windows are developed based on a standard toolkit, the provided API still lacks functionality that is necessary to ensure full accessibility of all applications. X Windows does not impose the use of any specific toolkit, nor does it necessarily promote one. It is quite common for users of a UNIX-type system to simultaneously use any number of applications that are each built upon a specific graphical toolkit. Some applications even include support for multiple graphical toolkits, providing the user with a configuration choice to select a specific one.

In order to be able to provide access to application user interfaces regardless of the graphical toolkits they are developed against, the chosen approach must ensure that the provision of non-visual access is not only medium-independent but also toolkit-independent.

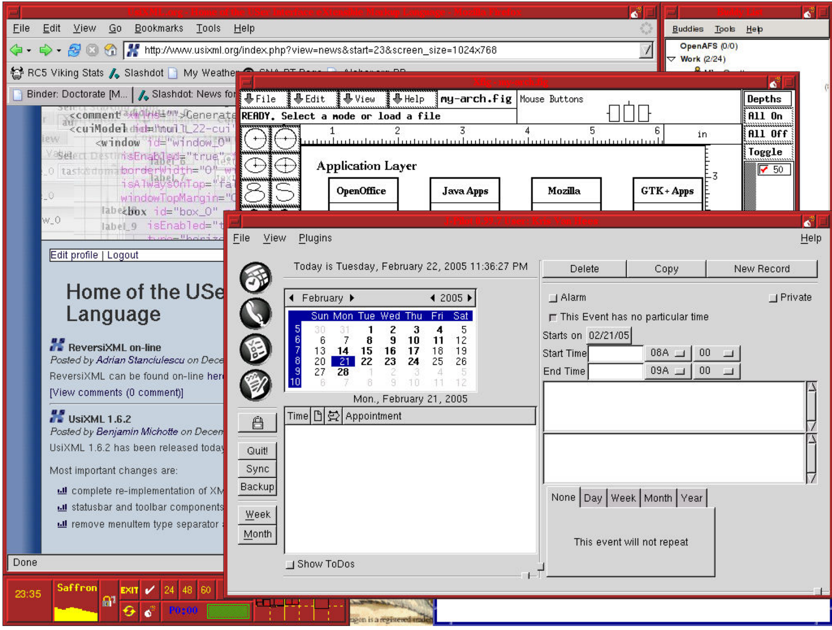


Figure 5.3: X11 session with multiple graphical toolkits

Figure 5.3 shows a fairly typical session on a UNIX-type system, displaying four applications: Firefox, J-Pilot, Xfig, and GAIM. Firefox and J-Pilot are built upon the GTK 1.2 graphical toolkit. Xfig uses the Athena Widget Set, whereas GAIM uses GTK 2.0. The bottom of the figure also shows the FVWM window manager button bar. The "Look & Feel" of the graphical interaction objects for the four applications is quite different, yet sighted users know intuitively how to operate them. All buttons, input fields, or menu bars essentially work the same way, regardless of how they look. To a blind user, the visualisation of the UI element is obviously also irrelevant unless somehow it is used to convey information to the user[6]. The problem therefore lies with the implementation of the accessibility solution and/or the implementation of the toolkits. Toolkits often have very different ways in how they implement the visualisation of a specific UI element which complicates the introduction functionality to support AT needs. Toolkit developers (and vendors) may also be less inclined to keep up with AT changes, etc. . .

Many of the existing approaches based on an accessibility API involve mapping graphical toolkit features (widgets and their interaction semantics) onto a set of accessible features. This effectively amounts to creating an abstraction from a

[6]It is safe to assume for the purpose of this discussion that the element provides user interaction functionality only. Conveying information is an aspect of abstract UI semantics that can be represented in various ways - it is not purely related to visualisation.

realized (concrete) user interface, often with an implicit loss of information[7]. When confronted with multiple graphical toolkits, multiple distinct mappings are required to accommodate abstracting all concrete UIs onto a defined model. This has proven to be quite complex, and typically involves significant limitations as shown above.

Section 5.3 will show that when multiple toolkits operate on the same conceptual model, effectively performing a reification of an abstract user interface within the context of a specific “Look & Feel”, a unified source of information is available for assistive technologies to operate upon. Mapping is no longer required because accessibility information can be derived directly from the abstraction that previously had to be derived from each concrete UI by means of a specialised bridge.

5.2.3 Collaboration between sighted and blind users

The Collins English Dictionary – Complete & Unabridged 10th Edition defines the term “collaboration” as follows:

Definition 5.2. *Collaboration: The act of working with another or others on a joint project.*

In order to ensure that segregation of blind users due to accessibility issues can be avoided, appropriate support for collaboration between the two user groups is important. This collaboration can occur in different ways, each with its own impact on the overall requirements for the accessibility of the environment.

Savidis and Stephanidis [125] consider two types of collaboration:

- Local: Sighted and blind users interact with the application on the same system, and they are therefore physically near one another. This would typically involve one user approaching the other in order to ask a question or share information about interaction with the application. Common occurrences are often characterised by conversations “let me do . . . ”, “let me show you . . . ”, and “how about we do . . . ”.
- Remote: Sighted and blind users access the application from different systems, and they are physically distant from one another. This situation would typically still involve a conversation as illustrated in the previous item, possibly by means of a communication channel outside the scope of the

[7] It is important to note that although information may be lost, accessibility may not be impacted because often only perceptual information is affected,

application (e.g. by phone). It is important to note that both users[8] are using physically distant systems to access the application on a central system.

The distinction of these two types is not sufficient to accurately describe the possible ways in which sighted and blind users may collaborate. Proximity between users[9] does not actually impact collaboration much as long as an adequate communication channel[10] is available. The ability to interact directly with the computer system that the users collaborate about can certainly be a benefit, and this can often be accomplished both locally and by means of remote connectivity. Note that local collaboration between two blind users can be a bit less efficient when a refreshable braille display is used because typically only a single device will be supported for a given computer system, requiring a pass-the-keyboard style interaction. The work presented in this dissertation would allow for multiple refreshable braille displays to be operated simultaneously on a computer system.

Consider a case where a worker walks up to a co-worker and they discuss the user interface of an application. Even without touching the computer system, users can still talk about the operation of the application, e.g. “Select this and that, and then click on the button that reads . . . ”. This requires that the users can understand the user interaction semantics of the UI elements that are part of the discussion, and that the user can either visualise the user interface, or otherwise reason about it based on an alternative perceptual model.

Also consider a situation where a worker needs to call someone for help, and those who can help are at a remote location. In the current age of distributed computing and multi-location businesses this situation is quite common. Again, collaboration may be limited to a communication channel only.

The communication portion of collaboration can occur at two different levels (applying [50] in the broader context of interaction between users as it relates to GUIs):

- Conceptual: Users talk in terms of the semantics of user interaction. Quite typical conversation would resemble: “Select double sided printing, and the collate option, and then print the document.” When direct interaction[11] is possible (be it local or remote), the dialogue is likely to be augmented with

[8] This dissertation may often refer to users in a pure one-to-one sense, even though these principles apply to larger teams of users as well.

[9] Either between blind users, or between a sighted and a blind user. Sighted users often tend to depend on a visual focal point when collaborating about the interaction with a system or an application.

[10] The main requirement for the communication channel is that it provides for sufficient bandwidth to enable efficient and specific exchange of information. A phone connection is often much more constructive to collaboration than e.g. an online chat session.

[11] In the context of conceptual and perceptual collaboration, “direct interaction” means that both participants can operate the computer system during the exchange.

either a demonstration of the interaction, or a verification that the instruction is understood correctly.

- Perceptual: Users talk in terms of how to operate manipulatives that are present in the UI. Conversations would resemble: "Check the double sided check box and the collate check box, and then click on the 'Print' button." Direct interaction would add a demonstration component to the exchange, or alternatively, a mode of verification that the other person is accurately following the directions.

Since proximity between users does not directly impact their collaboration, there is no need to make a distinction between local and remote. On the other hand, the fact that communication between users can take place on two different levels does require us to consider two specific cases[12].

Communication at the perceptual level involves details about the representation of the user interface in a specific modality. It is therefore not an effective way for sighted users and blind users to discuss the operation of an application. It requires that both parties have a good understanding of the actual operational details of the representation of the UI. While many blind users certainly understand the vast majority of visual concepts [94, 120], manipulation of UI elements in the visual representation is often not possible[13] [125]. It is also important to note that an expectation for blind users to be able to communication with others at the perceptual level amounts to reducing the level of accessibility back to the graphical screen rather than the conceptual user interface (see section 2.6).

Definition 5.3. *Perceptual collaboration: The act of working with another or others on a joint project with communication at the perceptual level (i.e. within the context of the representation of the user interface in a specific modality).*

Communication at the conceptual level relates directly to the semantics of user interaction for the application, independent from any representation. In this case, users communicate within the context of the conceptual model that lies at the core of the user interface. Rather than referring to elements of the representation of the UI (visual or non-visual), users refer to elements of the conceptual (often metaphorical) user interface[14].

Definition 5.4. *Conceptual collaboration: The act of working with another or others on a joint project with communication at the conceptual level (i.e. within the context of the modality-independent user interface interaction semantics).*

[12] An almost identical distinction was identified in section 2.2.4 in view of graphical user interfaces, with the conceptual layer of metaphors, and the perceptual layer of metaphors.

[13] Or at least, not possible in an equivalent and/or efficient manner.

[14] See section 2.6 for definitions.

Entry Form
First Middle Last
Full name
No spaces
Occupation

Figure 5.4: Example of a visual layout that can confuse screen readers.

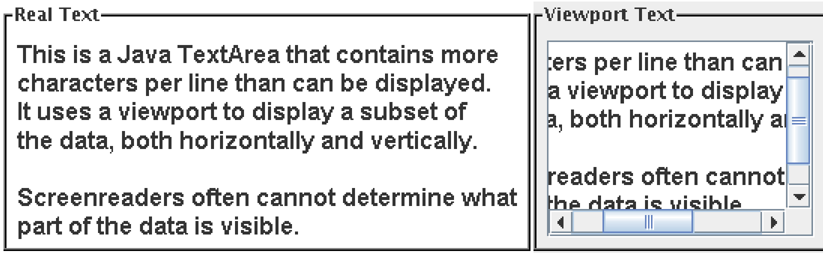


Figure 5.5: Example of the effects of a viewport on text visualisation.

5.2.4 Multiple coherent concurrently accessible representations

A common problem with existing approaches for non-visual access to GUIs is related to the use of an off-screen model: lack of coherence between the visual and the non-visual interfaces. Mynatt and Weber identified this as one of the important HCI issues concerning non-visual access (see section 2.3).

The problem is most often related to the information gathering process that drives the construction of the OSM[15]. Limitations in being able to obtain the following two pieces of information are examples of what can cause this lack of coherence:

- *Semantic relationships between user interface elements.*
  A typical example can be found in the common relation between labels and input fields. If this relation is either not encoded in the GUI implementation, or if it is not available through hooks, the OSM will contain the label and the input field as independent elements. The screen reader will present them as such, which may cause the user to be presented with an input field without any indication what data is expected to be entered there. On screens where multiple input fields occur in close proximity, this can result in significant levels of confusion and potential data entry errors. Figure 5.4 shows an example of how lack of semantic relation information can confuse a screen

[15] Kochanek provides a detailed description of the construction-process for an off-screen model for a GUI [75].

reader. The placement of multiple label widgets in close proximity to a text entry field makes it difficult to determine what label should be spoken when the user accesses the text entry field.

- *Effects of the visualisation process on the actual data contained in user interface elements.*
  This problem is mostly observed with larger text areas, where the text that is actually visible in the GUI may be much less than the actual text that is contained within the UI element. GUI toolkits often use a viewport-technique to render a subset of the information in the screen, in function of the available space on the screen (see Figure 5.5). If the assistive technology solution (e.g. the screen reader) cannot obtain information on what portion of the text is visible to the user, a blind user may be presented with data that a sighted colleague cannot see on the screen. In Figure 5.5, a blind user would typically be presented with an equivalent of the view at the left whereas a sighted user would be presented with the (more limited) view on the right. This leads to significant difficulties if the two users wish to collaborate concerning the operation of the application and the content of the text area.

Both problems can be solved by providing a single user interface representation that is tailored to the specific needs of a user or target group. Coherence is in that case not an issue because only one interface is ever presented at any given time. Unfortunately, this impacts collaboration between users with different needs negatively because not everyone will be able to accurately determine the state of the user interface at any given time. Limiting the user interaction to just one target population at a time is clearly not an acceptable solution.

The discussion on the collaboration design principle (section 5.2.3) shows that direct access to the system is fundamental to working together successfully. The requirements for making that possible can be summarised as:

- *Users can access the system concurrently.*
  This requirement has been discussed in the preceding paragraphs.

- *Users can interact with the system by means of metaphors that meet their specific needs.*
  While the user interface is designed using a single consistent conceptual model with familiar manipulatives (see section 5.2.1), the actual representation that the user interacts with should meet the specific needs of that user. This relates directly to the perceptual layer, covering both the lexical and syntactic aspects of the UI design, and the perceptual metaphors of interaction as appropriate for the needs of a specific user population. Mynatt, Weber, and Gunzenhäuser [100, 57] present HCI concerns related to non-visual representations of GUIs, identifying the

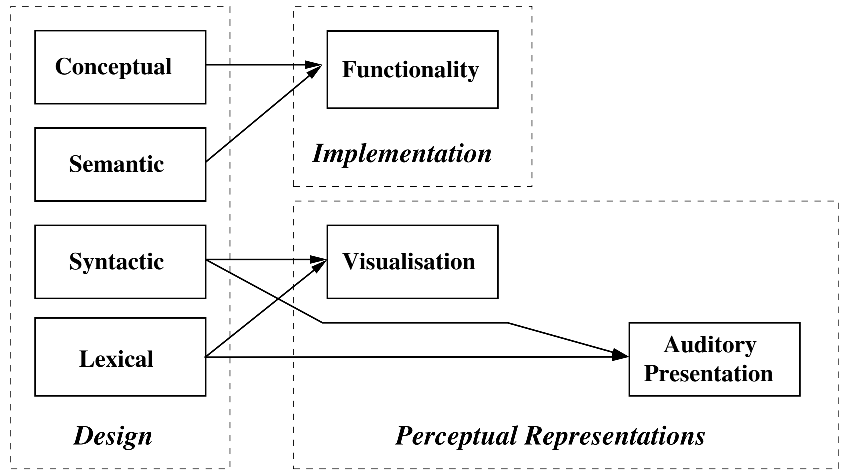


Figure 5.6: Multiple perceptual representations of the same user interface
Concurrent representations of the same user interface, each providing perceptual metaphors that meet the specific needs of a group of users.

need to support exploration and interaction in a non-visual interface. The generalised interpretation of this concern is captured by this requirement, and is discussed further in the remainder of this section.

- *Coherence between UI representations is assured.*
  The preceding two requirements effectively describe a configuration where multiple users can access the system concurrently by means of representations that meet their individual needs. The HCI concerns phrased by Mynatt, Weber, and Gunzenhäuser [100, 57] relate to a more specialised scenario of a dual representation (visual and non-visual). Coherence between the representations was presented as an important concern. In the more general context of multiple concurrent representations, the need for coherence certainly remains. Further discussion on this requirement is presented at the end of this section.

Each representation provides perceptual metaphors that meet the specific needs of its target population

At the conceptual level, all users are presented with the same user interface. No adaptations are necessary because the conceptual model is by design not dependent upon any modality of interaction (see section 5.2.1). Edwards suggested that there is not really a substantial difference between the conceptual models of blind users and sighted users, but rather that the information channels

used are different [39]. Extensive research has been conducted in exploring the information channels that are most conducive to perceptual reasoning in blind individuals [50, 99, 23, 122, 121, 41]. At the perceptual level, tactile and/or auditory presentation of information is by far the most appropriate for non-visual access. It is therefore obvious that blind users would be best served with specific non-visual metaphors of interaction. Likewise, other groups of users with specific needs may benefit from specific metaphors of interaction that are different from those that are provided in e.g. the visual representation of the UI.

Figure 5.6 illustrates the design principle of specialised representations at the perceptual layer. The conceptual layer captures both the conceptual and semantic levels of the design, providing a description of the functionality of the application. The lexical and syntactic levels of the design need to be interpreted within the context of the target population[16], thereby driving the design of the perceptual layer. The presentation of the UI can essentially become a pluggable component[17], interfacing with the conceptual layer of the UI.

The same semantic information and functionality is concurrently accessible in each representation

Edwards, Mynatt, and Stockton define the semantic interpretation of the user interface as [45]: “the operations which the on-screen objects allow us to perform, not the objects themselves.” This refers to the actual functionality that an application provides, but this is only part of the semantic layer in UI design. Trewin, Zimmermann, and Vanderheiden provide a more extensive list of the core elements of a user interface based on what they believe should be represented in an abstract user interface description[18] [143]: variables (modifiable data items) that are manipulated by the user, commands the user may issue, and information elements (output-only data items) that are to be presented to the user.

Therefore, the semantic model of the user interface consists of:

Definition 5.5. *Semantic Information: Data elements in the UI that have meaning within the context of the conceptual model.*

Definition 5.6. *Semantic Functionality: Operations that can be performed in the UI and that have meaning within the context of the conceptual model.*

[16] Specifically, the needs of the target group as those relate to UI interaction.

[17] A pluggable component is one that can easily be replaced by an equivalent component. The term is commonly used in UI contexts to indicate exchangeable presentation components. It is a derivative of the “plug-n-play” hardware concept.

[18] This AUI description effectively defines the user interface at the conceptual and semantic level. It is a formal description of the conceptual model.

The semantic information covers both dynamic data that is encapsulated in UI elements that allow the user to input or manipulate that data, and static data that carries meaning and that is to be presented to the user. It also captures those properties of data items that associate additional meaning to the data.

The semantic functionality captures the manipulations that are possible for each UI element, regardless of representation. The functionality is defined in context of the tasks that can be accomplished with the application.

In order to ensure coherence between concurrent representations of the UI, the same semantic information and functionality must be used to render the UI for each user based on their needs. All users must be able to perform the same user interaction operations at a semantic level, and all users must be able to observe all results of any such operation at the same time[19]. While this seems to be a rather obvious requirement, existing approaches to providing access to GUIs for the blind often provide a sub-optimal implementation where the sighted user is able to interact with a UI element prior to a blind user being able to observe (through synthetic speech or braille output) that the element exists. Similarly, sometimes a blind user can interact with a UI element that isn't actually visible to a sighted user. This is an example of semantic functionality that is accessible in one representation but not in another.

5.3 Design

Using the design principles presented in section 5.2, a framework can be designed for providing access to graphical user interfaces for blind users[20]. This section provides details on the various components while deferring the majority of implementation aspects to chapter 7.

The schematic overview of the Parallel User Interface Rendering approach is shown in Figure 5.7. Rather than constructing the UI programmatically with application code that executes function calls into a specific graphical toolkit, applications provide a UI description in abstract form, expressed in a UIDL. This authoritative AUI description is processed by the AUI engine, and a runtime model of the UI is constructed. The application can interact with the AUI engine to provide data items for UI elements (e.g. text to display in a dialog), to query data items from them (e.g. user input from a text input field), or to

[19]Note that this does not necessarily imply that the result of user interaction is immediate, although it has become common practice to provide near-immediate results in support of the WYSIWYG design principle (see section 2.2.3, page 26).

[20]While the design has a primary focus on providing blind users with user interaction that is equivalent to what their sighted peers use, the techniques presented here are generic enough that they could be applied for providing access in support of a variety of needs.

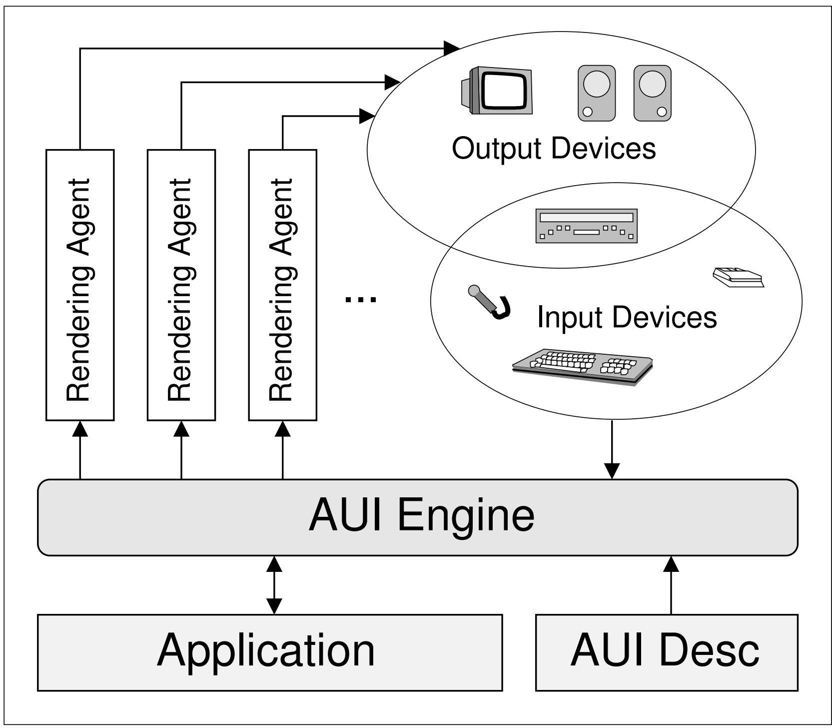


Figure 5.7: Schematic overview of Parallel User Interface Rendering

make runtime changes in the structure of the UI. The AUI engine implements all application semantics, ensuring that the functionality does not depend on any specific modality.

The representation of the UI is delegated to modality specific rendering agents, using the UI model at their source of information. At this level, the AUI is translated into a concrete UI (CUI), and the appropriate widget toolkit (typically provided by the system) is used to present the user with the final UI, by means of specific output devices. Therefore, the UI model that is constructed by the AUI engine serves as information source for all the different rendering agents. All representations of the UI are created equally, rather than one being a derivative of another[21]. The application cannot interact with the rendering agents directly, enforcing a strict separation between application logic and UI representation.

[21] It is important to note that it is not a requirement that all representations are generated at runtime, although development time construction of any representations could imply that dynamic updates to the UI structure are not possible.

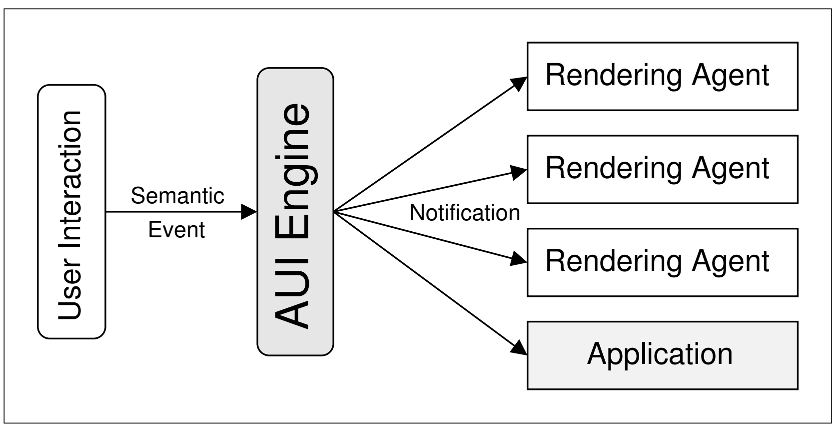


Figure 5.8: Logical flow from interaction to presentation

The handling of user interaction events from input devices[22] occurs at the AUI engine level. The PUIR framework is based on meaningful user interaction, and therefore only semantic user interaction events are given any consideration. Given that events are typically presented to toolkits by means of OS level device drivers, and the fact that these event sources are very generic[23], additional processing is required in order for the PUIR framework to receive the semantic events it depends on.

Figure 5.8 shows the flow of user interaction through the PUIR framework. User interaction is presented to the AUI engine as semantic events (activate element, select element, . . .) Processing of the semantic event results in a notification event being sent to all rendering agents, and the application. This event indicates that the semantic operation completed. Rendering agents will use this event to trigger a presentation change (if applicable) to reflect the fact that a specific semantic operation took place on a specific widget. The application may use the notification to determine whether program logic must be executed as a result of the semantic operation, e.g. activating the submit button for a form might trigger validation and processing of the form content.

A more detailed discussion concerning user interaction event handling can be found in chapter 6.

[22]The physical devices that the user employs to perform operations of user interaction with the application.

[23]Device drivers at the OS level are meant to serve all possible consumers. The events they generate are most commonly very low-level events.

5.3.1 Conceptual model

The most fundamental design principle is the establishment of a conceptual model as the basis for all UI representations. Discussion of this principle in section 5.2.1 not only shows that a single model suffices, but also that the well established metaphors of the physical office and the desktop may be appropriate for both blind and sighted users.

It can be argued that these conceptual models are inherently visual, based on a spatial metaphor from a visual perspective, and that it is therefore advisable to design a specialised non-visual model [123]. This argument, in and of itself, is not sufficient to dismiss the existing models and metaphors as not appropriate for blind users because it makes the assumption that they are not appropriate for the blind just because they have been selected based on visual considerations. The argument also fails to take collaboration between users with different abilities into consideration, and this is another important design principle.

To illustrate the problems with the aforementioned argument, consider the case of print-to-braille transcription. Braille books are known to be quite bulky due to the nature of braille (standard cell size, and limited opportunities to represent a sequence of characters with a shorter sequence of braille cells[24]) and it is therefore more convenient to transcribe books in ways that promote minimising the number of pages needed to transcribe the print text. However, doing so would make it much more difficult to collaborate with sighted peers because correspondence between e.g. page numbers is lost completely. It is for this reason that the Braille Association of North America stipulates that with the exception of some preliminary pages, all pages of text must be numbered as in the print book [17]. Furthermore, it specifies strict rules on how to indicate print page number changes in the middle of a braille page, to facilitate collaboration and to support page-based references to print materials. This is a clear example where the need to support collaboration between sighted and blind users outweighs the advantages of a format that is optimised for blind users, without any regard for being able to consult references.

Blind users live and are taught in a predominantly sighted world, where they learn to interact with many of the same objects and concepts as their sighted peers. While there are often definite differences in user interaction *techniques* between the two groups[25], the semantics of the manipulation are essentially the same. As example, consider the task to lower the volume of a music player. Regardless of whether the control to do so is a slider or a turn knob, once a user knows where

[24] This is known as a contraction in English Braille, American Edition.

[25] It is obvious that even amongst the blind or the sighted, not necessarily everyone will prefer everything the same way. This has been a driving force behind the efforts to provide user customisations for UIs.

Control/Object	Description
Window	Logical container for controls, establishing a context for focusing user interaction.
Button	Control that can be activated, and that can either automatically reset to its default state after activation, or that operates as a switch between two states.
Set of “radio” buttons	Set of buttons where only a single one can ever be selected. It remains selected until another button in the set is selected.
Slide control	A control that allows the user to select an arbitrary value along a predefined range (at the implementation level, this will translate to a value on a discrete range), e.g. a volume setting on a radio.
Input field	A place for the user to enter information.
Label	An object that conveys information to the user, e.g. a description of a control.
Selection list	A control that allows the user to select one or possibly more items in a list., e.g. a checklist for a complex task.

Table 5.1: Examples of controls and objects in the conceptual model

the control is located, he or she generally knows how to operate it[26].

It is also important to observe that although the established terminology seems to refer to visual aspects, the underlying concept is often more abstract. One does not generally consider a GUI “window” in any way equivalent to a physical window, but rather it is seen as a two-dimensional area that usually contains other UI elements. In a sense, it is a top level grouping of all or part of an application UI. Many people are able to identify windows on a screen because they have been taught that a certain entity on the screen is called a “window”, but it could as easily have been named a “tableau”. What really matters is what you can do with it (semantics).

Similarly, a “button” is rarely interpreted as a strict analogy with physical push button controls. Instead, it is intuitively accepted as a UI element that triggers something when it is selected. In general, most users have learnt to think about UI elements in terms of their semantics and less in terms of what they might represent in the physical world.

[26] Generically, this type of control is known as a “valuator”.

The various controls that are represented in the conceptual model for the UI (see Table 5.1 for examples of controls and objects) are somewhat weak analogies for their physical counterparts. As Kay stated, they should perhaps be referred to as “user illusions” [74]. Both sighted and blind users must approach them as concepts that are familiar, yet the mode of interaction is inherently different. Neither user group can truly push a button or move a slider that is merely presented as a visual image or an auditory artefact. Still, users will communication in terms of the metaphor because of the familiarity of the concept.

The PUIR framework is designed around this very notion, allowing for a single conceptual model that is not only familiar to all users, but that has also been a well established model within the context of computer systems for many years. Using the physical office and desktop analogy as underlying model for this work also helps maximise the opportunity for collaboration between users with differing abilities.

5.3.2 AUI descriptions

Edwards, Mynatt, and Stockton suggested that the best approach for creating non-visual user interface representations is a translation of the UI at the semantic level, i.e. render the UI in function of the needs of the user, using perceptual metaphors that meet the needs of the target population (see section 5.2.4, page 140). The perceptual layer is merely a reification of an application’s abstract user interface. Their goal was to provide non-visual access to existing X11-based applications, and their work therefore involved attempting to capture application information at the semantic level by means of toolkit hooks. A hierarchical off-screen model was constructed based on this information: a semantic OSM. While this effectively results in a pseudo-AUI description of the application user interface, it is sub-optimal because it is created as a derivative of the programmatic constructs that realize the visual representation.

The concept of abstract user interface descriptions has been known for a long time already, and it has mainly been intended as a source document for the automated generation of the UI, i.e. generating application source code for the programmatic construction of the UI representation and its interaction with the process logic [86, 89]. Using the AUI as the basis for the UI is a powerful concept because it provides a canonical description of the conceptual model, thereby taking a very prominent place in the design process. Given a sufficiently expressive UIDL, the AUI description can enable application designers and/or developers to implement a true separation of presentation and logic, and it can alleviate part of the burden of implementing a UI by providing for its automation.

Even when the UI implementation is generated automatically based on an AUI

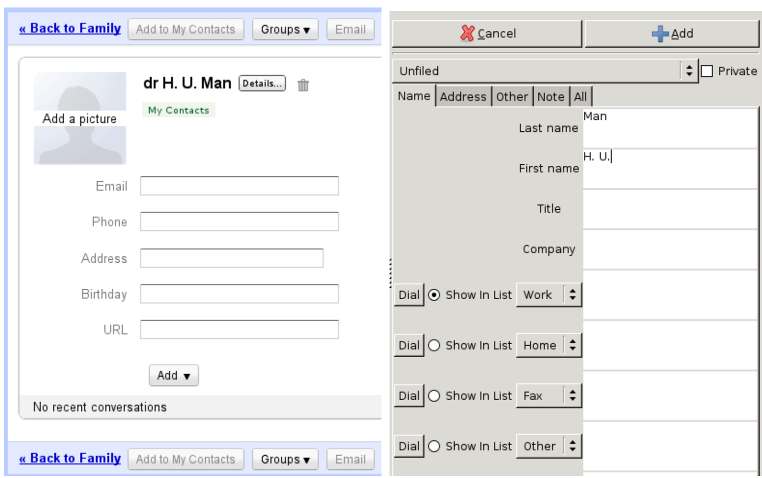


Figure 5.9: Web forms bear a striking resemblance to UI data entry screens. *On the left a web form is shown for a address book contact entry. On the right one can see the UI for an application that implements address book management features.*

description, providing non-visual access to the UI is still sub-optimal because information can still only be obtained from the realized graphical user interface.

A possible solution could be to make the AUI description available alongside the application, to be used as an information source in support of AT solutions (i.e. the Glade project [27]). Because the implementation of the UI is still generally hard coded in the application, this approach does open up the possibility that inconsistencies between the application and the UI description occur[27]. In support of the coherence design principle (see section 5.2.4, page 141), the PUIR framework is designed on the concept that all representations are to be reified from the same AUI. This is a significant paradigm shift from the majority of AT solutions that are still implemented as a derivative of the GUI.

Can the construction of the UI representation(s) be delayed until execution time of the application? In other words, can the UI be rendered by means of AUI runtime interpretation rather than AUI development-time compilation?

When comparing a form on a web page with an application UI where data entry is expected to occur (see Figure 5.9), striking similarities can be observed. Both feature almost identical UI elements: buttons, drop-down lists, text entry fields, and labels. In addition, the obstacles that blind users face when using web forms [114, 141] are known to be very similar to the obstacles they face

[27]This is a common problem in any circumstance where essentially the same information is presented in tow different locations.

when interacting with GUIs [6]. Furthermore, HTML documents are essentially abstract descriptions although specific modality dependent information can be embedded in the document as augmentation to the abstract description. Web browsers handle the rendering of the HTML document, providing the user with a representation that is commonly tailored to the device the user is using, and possibly other user preferences. Based on these observations and research on the use of AUI descriptions (such as the works of Bishop and Horspool [9], and of Stefan Kost [78]), we can therefore conclude that it is possible to describe user interfaces by means of HTML-style documents, to be rendered in function of the output modalities.

This paradigm shift from representing the UI by means of program code in the application to utilising a system that interprets and renders the UI based on an AUI description document has slowly been taking place for the past ten to twelve years. Yet, the shift has not progressed much past the point of using the AUI description as part of the development process. The preceding discussion shows that it is possible (and necessary for this work) to complete the shift to what Draheim, et al. refer to as “the document-based GUI paradigm” [36]. Expanding the notion of the representation of the UI description to the realm of concurrent alternative representations, this can be extended as “the document-based UI paradigm”. The advantages of this approach are significant, although there are also important trade-offs:

- *Separation of concerns between UI and application logic*
  This has been identified as (part of) an important technical requirement for AUI description languages (see section 2.5.1), but the very use of AUI descriptions also enforces this concept through the need for a well-defined mechanism to incorporate linking UI elements to program logic. This also implies a trade-off in flexibility because the application logic is limited in its ability to directly interact with the UI.

- *Maintainability of the application*
  When a UI is described programmatically as part of the application, it typically will have a stronger dependency on system features such as the presentation toolkit that it is developed for. AUI descriptions do not exhibit this complication because because they are toolkit independent. In addition, the document-based nature of AUI makes it much easier to modify the UI for small bug fixes, whereas a code-based UI requires changes in the application program code.

- *Adaptability*
  The adaptability of code-based UIs is generally limited to toolkit-level preference-controlled customisation, whereas the ability to make changes to the AUI description at runtime (e.g. by means of transformation rule sets) provides for a high level of adaptability.

The document-based UI paradigm is powerful, but it does impose some limitations on the designer/developer because some very specialised toolkit features may not be available in all modalities. Toolkits for programmatically defined UI development generally offer a more rich feature set to the developer because they often allow access to the underlying lexical and/or syntactic elements. A rendering agent that creates a UI representation based on an AUI description offers a higher level of consistency and stability through the use of common higher level semantic constructs, at the cost of some flexibility.

While working towards universal access, it is important to be mindful of the creativity and æsthetic insight of designers. It is easy to reduce the user interface to its abstract semantic existence, but ultimately appearance does matter[28]. Empirical observation of both sighted and blind users as they operated computer systems has shown that there is a tendency to favour more æsthetically pleasing representations, and it seems to improve productivity. AUI descriptions in the PUIR framework therefore allow for rendering agent specific annotations to be added to the specification of UI elements[29]. The information is stored by the AUI engine for delivery to a rendering agent that requests it, but beyond that it is ignored by the AUI engine, because it is modality specific.

Despite the very powerful advantages of the document-based UI paradigm, it is important to recognise that the specification of the UI at the abstract semantic level does present a few complications, as illustrated by the following issues.

Dynamic user interfaces

It is common for UIs to contain elements that are not entirely statically defined, i.e. they contain information that is not known at development time. Prime examples are interaction objects that contain user modifiable data and elements that provide for user input. Another common occurrence is a UI element that only allows conditional interaction. GUIs often presents such elements as “grayed out”, and they tend to not respond to user interaction while in that state. All these examples do not alter the composition of the UI presentation, and they therefore do not directly impact non-visual access.

A more disruptive feature involves truly dynamic updates in the user interface. A common example can be found in the almost iconic “File” menu on the application menu bar. It commonly displays (alongside various operations) a list of 5 or 10 most recently used files. The exact content and even the size of this list cannot be

[28] Although “appearance” is commonly interpreted as an aspect of visual perception, it actually carries a much broader meaning, across multiple modalities of perception.

[29] This is fundamentally different from other approaches (e.g. the HOMER UIMS [124, 125]) where UI objects are described multiple times: once in abstract form, and once or more in modality-specific forms.

determined ahead of time. One possible solution may be the implementation of a feature in the AUI that specifies that this specific content must be queried from the application[30]. Alternatively, providing a facility for dynamic updates in the AUI description would provide a more generic solution to this type of problem.

Abstract user interfaces are commonly described in an XML-compliant UIDL, providing a natural hierarchical structure: an object tree. Given that the PUIR framework renders the UI at runtime based on the AUI description, it is possible to support alterations to the user interface by means of adding, removing, or updating parts of the hierarchy (sub-trees). This ensures that dynamic UI changes are possible in a generic way. Components that render the actual representations can then receive notification that an update is in order.

Legacy applications

The adoption of AUI-based application user interface design and development is still ongoing. It is therefore a reality that many legacy applications will not support a UI that is generated at runtime, based on an AUI description. Two possible approaches have been researched in recent years:

- *Reverse engineering the user interface*
  The UsiXML project includes techniques that make it possible to reverse engineer an existing (programmatically defined) UI, and obtain a representative AUI for the legacy application [14, 89, 153].
- *Interposing toolkit library implementations*
  This is essentially a form of reverse engineering by capturing all toolkit function calls using an API-compatible replacement library. This technique provides a non-invasive approach to capturing the programmatic construction of the UI. One such implementation was developed by the Visualisation and Interactive Systems Group at the University of Stuttgart [118].

“Creative programming”

By far the biggest obstacle in providing non-visual access to GUIs is the occasional case of extreme use of features that are provided by user interface toolkits. In its worst form, an enthusiastic developer may implement his or her own toolkit, using a single large image widget from an existing toolkit as canvas for a custom rendering engine. Alternatively, an existing toolkit may be extended with some widgets that are not implemented in a compliant manner. Creative minds

[30] This is commonly known as a “call back” feature.

have even been known to implement buttons in dialog boxes that “run away” from the mouse pointer once it is within a predefined pixel-distance.

The only conclusion that can be reached in view of such creative programming is that it is not likely to be feasible to provide non-visual access to each and every application. It is obvious however that the use of techniques that result in this level of complexity are indicative of a sub-optimal design because separation between application logic/functionality and visual presentation is lost.

5.3.3 AUI engine

As introduced at the beginning of this section (see page 142), the AUI engine is the core of the PUIR framework. It provides the following functionality:

- Translation of the AUI description from its textual UIDL form into the UI object model.
- Focus management (i.e. tracking which widget is to receive context-free input, such as keyboard input).
- Implementation of the user interaction semantics of UI elements.
- Re-routing user interaction events in support of rendering agent specific functionality.

The UI object model used in the AUI engine is a hierarchical model, using a tree structure to represent the UI. The root of the tree, the singleton node that does not have a parent, is the window. Children of the root node are by definition components in the UI. They have exactly one parent: a component that functions as a container, providing a way to group multiple components together in a logical and/or semantic unit. Components that are not containers appear as leaf nodes in the tree structure, whereas containers appear as internal nodes. Figure 5.10 shows a partial UI object tree for a sample UI. In this figure, the parent-child relationship is represented by left-right connections, whereas top-down stacking represent grouping (sibling) relations. Note that this is similar to the commonly used off-screen model [79].

In order to maintain a strict differentiation between grouping and application semantics, a container can only function as a logical grouping of components. As such, it does not have any semantic user interaction associated with it. In fact, the only user interaction that is allowed for containers in the PUIR framework is related to establishing focus.

The objects in the AUI object model are abstract widgets (Table 5.2 lists the supported widgets; see Appendix B for more detailed information), and each

Widget	Description
window	Self-contained portion of a UI
	Window
group	Grouping of related widgets
menu bar	Container for menus
status bar	Notification message
tool bar	Container for easy-access widgets
	Menu bar
menu	Container for menu items
	Menu (and item groups in a menu)
menu	Sub-menu (contains menu items)
menu group	Grouping of menu items
menu item	Menu option that can be activated
mutex	Group of mutually exclusive toggles
toggle	Menu option that can be toggled
	Mutex groups and menu mutex groups
toggle	Mutually exclusive toggle
	Group (non-menu)
button	Button that can be activated
edit select list	Single-select list of items (provides a write-in option)
group	Logical grouping of widgets
multi select list	Multi-select list of items
mutex	Group of mutually exclusive toggles
single select list	Single-select list of items
text	Display text
textField	Text entry field
toggle	Selectable option
valuator	Ranged value entry

Table 5.2: Widgets provided by the AUI layer, by container
Every section is preceded by its container widget, except “window” as the root widget.

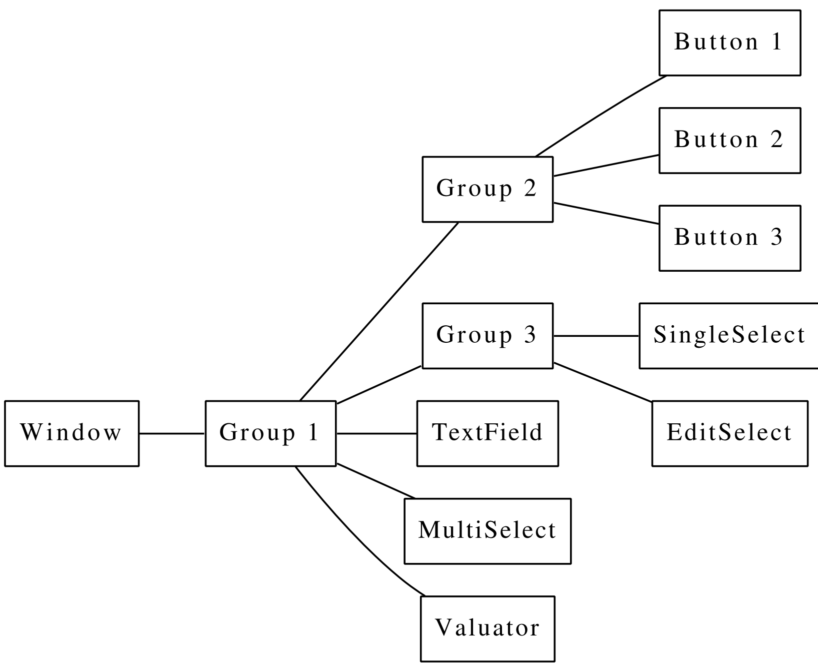


Figure 5.10: Sample hierarchical AUI object model

has been chosen specifically because of the fact that most (if not all) users are familiar with it. Note that the notion of using familiar concepts traces back to the initial design principles for the graphical user interface. Research has shown (e.g. Kurniawan, et al. [80, 81] and the survey discussed in chapter 3) that blind users have a good understanding of most UI elements in terms of their user interaction semantics. Being able to use the same familiar underlying concepts affirms the use of a single conceptual model, and it offers stronger support for collaboration.

It is important to note that the abstract widgets listed in Table 5.2 are only "visible"[31] in terms of their semantics. Some widgets (as mentioned previously) serve as a container for parts of the UI, and they are therefore typically only noticeable by virtue of the effect they may have on focus traversal.

From the context of an application, the AUI engine will handle one or more windows simultaneously. In that sense, the application itself could be considered the root of the overall AUI object model. Yet, it is deliberately omitted in order to maintain separation of concerns.

[31] In this context, being "visible" means that the user can note the existence of the widget. Being part of the AUI, the widget obviously has no perceptual characteristics.

Focus management

While conceptually, the GUI is primarily based on the "seeing and pointing" design principle, empirical evidence shows that automated move-to-next-element functionality and keyboard navigation are essential components of productivity when text entry is required. The mental context switching between coordinating typing and pointer device movements seems to impose a delay[32].

Keyboard input (such as filling in text entry fields) operates without an implicit context as opposed to e.g. pointer device operations. When the user operates a mouse in order to activate a button, the position of the pointer cursor determines what button is being activates. Keyboard input does not explicitly indicate what UI element it belongs to. Instead, an external focus management component takes care of this.

An element is said to be "in focus" or to "have focus" if it has been selected to receive user interaction events that are not explicitly associated with a UI element. The AUI engine manages the process of assigning focus to elements in three different ways:

- Programmatically: the application can request that focus be moved to the next or the previous UI element in the focus traversal order (see below this list for more information). It can also request focus to be given to a specific element. Aside from the application, it may also be beneficial to allow widgets to do the same, e.g. as a default action after an operation is completed.
- User interaction event: the user can navigate the window by moving between UI elements by means of direct user interaction. This is most often used for keyboard navigation (and exploration) based on the focus traversal order. Common navigation operations are: move to next element, move to previous element, move to next group, move to previous group, . . .
- User interaction side-effect: when the user performs an operation on a UI element that is not currently in focus, it is common practice to shift focus to that element. This is essentially a special case of the programmatical assignment of focus.

The "focus traversal order" mentioned in the list above refers to a well defined strict ordering of elements across the hierarchy of objects in the AUI object model. It specifies in what order the various elements receive focus, in a strict linear order. Containers are not included in the list because they have no semantic user interaction associated with them.

[32]More specific research into the impact of mental context switching and related topics is outside the scope for this dissertation.

The AUI engine defines the order in the AUI object model tree as depth-first, left to right. Based on the example in Figure 5.10 (where the order is rightmost-first, top to bottom because the tree is flipped horizontally for display purposes), the focus traversal order will be: *Button 1*, *Button 2*, *Button 3*, *SingleSelect*, *EditSelect*, *TextField*, *MultiSelect*, *Valuator*.

User interaction semantics

As mentioned in section 5.3, the AUI engine is responsible for providing an implementation for the user interaction semantics of all widgets. This is crucial in the design of the PUIR framework because it ensures that the behaviour of UI elements is independent from the representation of the UI.

Various user interaction operations are supported by the PUIR design:

- *Action*: This interaction is used to trigger a specific operation or function. It is one of the most basic forms of the “Seeing and Pointing” design principle for GUIs, because it captures the familiar action of pressing the “On/Off” button on an appliance.
- *Container*: Adding and removing components is the interaction that typically takes place on containers. This is equivalent with the physical world model that containers are based upon.
- *Focus*: This is probably the most obscure of all forms of user interaction. It captures the notion of what the user’s attention is focused on. When a person is filling out a form on paper, it is common to visually locate a specific item, and to then bring one’s pen to the writing space that is associated with that item. The person truly remains focused on the item he or she is filling out.
- *Selection*: When a user is presented with multiple options with the restriction that only one can be chosen, a selection process takes place. Often, a user will consider several options (selecting an option, only to then later dismiss that selection), until a final choice is made, which is then finalised (by means of an *Action* operation).
- *SetSelection*: It is sometimes appropriate to select more than one item from a list of choices (e.g. a buffet). Items that are part of the selection may be consecutive or separate. It is also quite likely that the selection set may change while the user makes up his or her mind. When finally a decision is reached, the appropriate selection is finalised (by means of an *Action* operation).

- *TextCaret*: This operation is (alike the Focus operation) quite obscure because it also relates to the location on which the user is focusing his or her attention. This operation targets text, and is used as the equivalent of placing one's pen tip in a specific position (at a particular character in the text string).
- *TextSelection*: When a user intends to select some text, he or she will commonly utilise a method that requires the least effort. If the user knows an entire line of text is to be selected, locating any point on that line is generally sufficient. Likewise, when trying to select a specific word, any point within the word boundaries is usually acceptable. Only when the text selection is more complex, will a user specifically select starting and end points by character.
- *ValueChange*: Any element that represents a dynamic value, i.e. an element for which the user can select or input a specific value, supports user interaction that modifies the current value. The changing of the value is not a final operation, as it is not uncommon for a user to change their mind (even multiple times) before deciding on the final value (which is then finalised by means of an *Action* operation).
- *Visibility*: For widgets that are not always represented in the UI, a conceptual characteristic of visibility can be assigned. While ordering food in a restaurant, a person typically consults a menu. Once the desired selection is made and communicated to the waiter, the menu is often taken away by the waiter, or it is put aside. No one pays any attention to it unless a followup order (or the intent to) is anticipated.

User interaction is presented to the AUI engine as semantic events, targeted at the widget it operates on. The operations that each widget supports are listed in Table 5.3. The abstract widget implementation handles the event, and (usually) dispatches notification events to rendering agents and the application to indicate that the semantic operation has been processed. More information about each widget can be found in Appendix B.

Re-routing user interaction events

Because the AUI engine is in total control over all user interaction, rendering agents would not be capable of implementing modality-specific operations that assist the user. A very significant example is exploration in a non-visual representation, where the rendering agent must be able to receive interaction events from the user concerning navigation of the user interface.

Containers	**Operation(s)**
group	Container
menu	Container, Focus, Visibility
menu bar	Container
mutex	Container
status bar	Container, ValueChange
tool bar	Container
window	Container, Focus, Visibility

Components	**Operation(s)**
button	Focus, Action
edit select list	Focus, Selection, ValueChange, Action
multi select list	Focus, SetSelection, Action
menu item	Focus, Action
single select list	Focus, Selection, Action
text	ValueChange
text field	Focus, TextCaret, TextSelection, ValueChange, Action
toggle	Focus, Action
valuator	Focus, ValueChange, Action

Table 5.3: Operations supported by abstract widgets

The AUI engine provides functionality that enables rendering agents to register a key stroke combination that uniquely identifies the rendering agent. When a user interaction event is received by the AUI engine for this key stroke, all further user interaction events get re-routed to that rendering agent for processing. This of course means that interaction with the user interface is effectively suspended until the rendering agent reliquishes control. The more immediate result on the rendering agent side is that all user interaction takes place with the rendering agent.

This special functionality exist to make it possible for the rendering agent to provide modality-specific features that the user can control. This can be used to implement configuration controls, or more commonly, it allows the rendering agents to offer a truly safe exploration mode for the application user interface. The AUI engine still responds to query requests concerning the UI, and therefore the rendering agent is capable of providing the user with any and all details about the current state of the user interface.

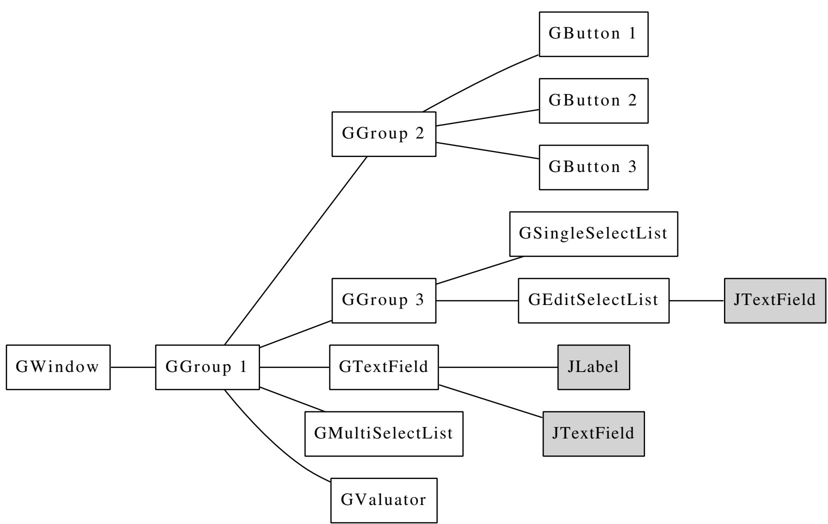


Figure 5.11: Sample hierarchical CUI object model for the AUI in Figure 5.10
The clear boxes indicate components that are created in one-to-one correspondence with the AUI model. The shaded components are necessary additions in order to render the UI correctly in the specific rendering agent.

5.3.4 Rendering agents

The AUI engine described in section 5.3.3 operates entirely within the context of the abstract UI object model, at the conceptual level. In order to be able to present the user with a UI representation within the context of a specific modality, the AUI must go through a reification process. This part of the PUIR framework operates at the perceptual level and is provided by the rendering agents.

Each rendering agent provides AUI reification within the context of one or more modalities. This is generally done as a two-step process:

1. On request of the AUI engine, a concrete UI object model is constructed, incorporating a specific “Look & Feel” based on an established set of interaction metaphors.
2. Based on a modality-specific presentation toolkit, the CUI from the previous step is finalised into the FUI that is presented to the user.

Mapping the AUI model onto a CUI model

Figure 5.11 provides a (partial) example of the CUI object model that a rendering agent might build based on the AUI object model that resides with the AUI engine (Figure 5.10). As illustrated with this example, there is no guarantee for a one-to-one correspondence between the two models, because abstract widgets may very well map onto multiple concrete widgets. This is the case for the abstract TextField widget that e.g. as part of a visual representation agent based on Java Swing is presented as a JLabel object and a JTextField object.

The reverse is certainly possible as well. A rendering agent may have no need for some intermediary container objects, and thereby map multiple abstract widgets onto a single more complex presentation widget. Note that regardless of the mappings between models, the user interaction semantics remain the same.

User interaction

In support of the separation of concerns concept as suggested by Parnas [112], there is no direct communication between the application and the rendering agents whatsoever. Any and all requests (be it from the application or the rendering agent) are to be processed by the AUI engine, which will then provide notification to the rendering agents. Upon receiving a notification event, a determination is made whether the operation requires updating the representation of the UI.

Many commonly available presentation toolkits provide an implementation for both the presentation and interaction components of the UI, rendering the effects of a user interaction immediately, and providing a notification or call back mechanism to the application to allow program logic to react to the user interaction event. Within the requirements of the PUIR design principles, if a presentation toolkit contains a user interaction component, any events from this component must be forwarded to the AUI engine for processing, and the presentation of UI changes as a result of the interaction (e.g. visually showing that a button was pressed, or providing auditory feedback for the same action) must only take place as a response to receiving a notification event from the AUI engine that a specific semantic operation took place. Without this clear separation it would be very difficult to ensure coherence between parallel representations[33].

It is important to note that rendering agents may implement context-specific user interaction. This level of interaction is independent from the actual UI and the

[33]A common problem would be that the modality in which the user interaction was initiated might render the feedback prior to the application logic responding to the operation, whereas all other renderings would render feedback afterwards. This is also commonly observed in assistive technology solutions such as screen readers that are implemented as a derivative to the graphical representation.

application semantics, and therefore not restricted to processing by the AUI engine. Being able to provide this level of interaction is important in order to support exploration of the UI in alternative representations, as a solution to one of the HCI issues related to the usability of alternative UIs (see section 2.3). The user interaction provided by the rendering agent must not result in actual UI interaction, and it therefore operates as a distinct UI mode (see Definition 2.2 on page 28).

Also note that this is a different level of exploration than provided by the user interaction re-routing discussed in section 5.3.3. Re-routing at the level of the AUI engine provides for suspending all user interaction with the user interface, whereas context-specific user interaction will not prevent other input modalities from having their interaction events processed by the AUI engine. As such, exploration by means of context-specific user interaction provides the user with a "view" on the user interface while interaction may be taking place.

Queries to the AUI engine

The rendering agent must also be able to query information from the AUI engine, as needed. Such queries are almost always in response to receiving an event from the AUI engine, but there may be legitimate reasons for the rendering agent to spontaneously request some data from the AUI engine. The most common use is to request additional information about a component in the AUI object model in order to render that component.

Modality-specific limitations

The rendering agent may impose some limitations on the overall UI due to modality-specific limitations. Due to the significant impact imposed on the entire UI, care must be taken to only require this when absolutely necessary. One common relatively low-impact limiting requirement is synchronisation. When only a single representation is used, the UI is primarily self-synchronising because the user cannot interact with components that are not rendered yet. In the presence of parallel representations, it may be necessary to ensure that all agents are presenting the same state of the UI at any given time. Latency with some modalities may therefore require delays to be inserted into the flow of interaction.

Temporal relations within the UI are not addressed in this dissertation, but this is another area where rendering agents may need to provide the AUI agent with limitations in its rendering capabilities. Although this is outside the scope of this work, some insights on this particular aspect are presented for future work in section 9.2.

5.4 Comparison to Model-View-Controller

The Model-View-Controller paradign (MVC) is extremely important in the world of UI design because it provides for an explicit separation of the user input, the modeling of the external world, and the visual feedback to the user [20]. It models a division of labour that is both natural and important to providing flexibility and maintainability.

The so-called MVC triad comprises:

- *View*: This manages the perceptual presentation to the output modality, typically a bitmapped graphical display.
- *Controller*: This component interprets user interaction events from the user, instructing the model and/or the view to change as appropriate.
- *Model*: This manages the behaviour and the data of the application. It provides information about its state and data to the view, and it acts upon requests from the controller to make changes in its state.

The model may be associated with multiple View-Controller pairs, but a view is always associated with a single controller, and a controller has always just a single view. In the MVC model, the link between the view and the controller is considered to be very tight.

The MVC paradigm is similar to the PUIR approach, where the view definitely corresponds to the rendering agent at the final UI level. The model corresponds to an abstract UI widget at the AUI engine level. The controller is more complicated because user interaction in the PUIR framework can be modality-dependent or modality-independent. For modality-dependent interaction such as a pointer device, the controller functionality would definitely be found in the rendering agent. However, for regular keyboard input, an independent entity takes on the role of controller. It is responsible for routing the keyboard based user interaction, yet it is not associated with any output modality.

In conclusion, it is clear that the PUIR approach is quite similar to the MVC paradigm. Model and view correspond well to their respective counterparts. The controller is a bit of an anomaly because in the case of keyboard, or keyboard-alike input, there is no corresponding output modality that provides context.

5.5 Conclusions

In this chapter, the Parallel User Interface Rendering approach has been presented. Fundamental design principles were formulated based on existing research and analysis, forming a solid basis for the design of this novel approach to providing alternative representations of graphical user interfaces in support of the accessibility needs of blind users. A detailed discussion of the design shows how the design principles are incorporated, and how this novel approach satisfies the requirements for providing access to GUIs for blind individuals.

The PUIR approach provides for multiple concurrent representations of the UI model, rendered from a single source as opposed to alternative renderings being created as derivatives of a primary representation. This is a significant contribution to the field of HCI and Universal Access because it elevates accessibility to the level of alternative representation rather than an accommodation. The visualisation of a UI is merely one of many forms of presenting the UI to a specific group of users based on their needs.

It important to recognise that any approach that allows for a high degree of flexibility[34] increases the risk that someone will use that flexibility in a way that interferes with the very design principles that the presented solution is based upon. This work does not intend to guarantee that any UI developed within the PUIR framework will be 100% accessible. However, the presented work promotes sound UI design, and ensures that at a minimum each user is aware of the existence of UI elements, even if it is possible that in rare cases 100% functional user interaction cannot be assured due to potential improper use of features.

[34] Or perhaps better expressed as “creativity”.

Chapter 6

Context-based interaction

"But you can't die. You're a machine."
"No."
"No, you're not a machine?"
"Yes."
"Yes, you are, or yes, you're not?"
"Yes."
"Yes, WHAT?"
"Yes, not."
"Talk about a malfunction."
(Stephanie Speck & Number 5 in "Short Circuit", 1986)

One of the ultimate goals of a graphical user interface, or a UI in general, is to provide a mechanism for the user to interact with an application or system. User interaction requires careful management of input event streams and translation of events within the context where they are generated lest miscommunication run rampant. This is especially true when input mechanisms are associated with multiple simultaneous representations.

6.1 Introduction

The complications addressed in this chapter involve user interaction events that are generated by input devices within the context of a specific application (Figure 6.1). When a user performs a physical action with an input device, an event is generated. This event is typically presented to a UI toolkit for further processing. Given that a computer system is typically providing one single UI

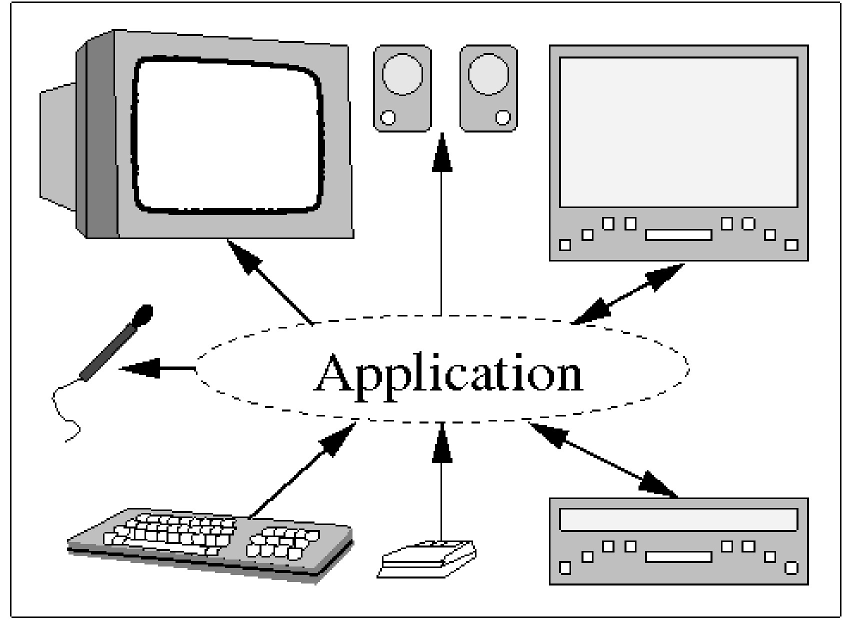


Figure 6.1: Simultaneous device interaction
Clockwise from upper left: display, speakers, planar braille display, linear braille display, mouse, keyboard, microphone.

representation (usually a GUI) to the user, the graphical toolkit can assert total control over the event handling. In Java[1] (similar to most GUI environments) all events are associated with a specific widget, either explicitly (e.g. the position of a mouse pointer) or implicitly (e.g. by the focus manager). Results from the user interaction event are rendered graphically, and notification events are dispatched to any interested parts of the application.

With the introduction of concurrent representations as provided for by the Parallel User Interface Rendering framework presented in this dissertation, handling user input events at the toolkit level is no longer appropriate. When one representation retains its position in being the sole handler of user interaction events, it takes on the role of primary rendering, making all other representations derivatives. This is the situation with the majority of currently available assistive technology solutions for GUI accessibility, where user interaction remains the responsibility of the GUI environment, and the alternative representation is limited to providing a derivative presentation of the output.

[1] The experimental implementation described in chapter 7 is developed in Java. Unless otherwise noted, all references in this chapter to implementation details relate to Java, the Abstract Widget Toolkit, and JFC/Swing.

When instead multiple representations are tasked with handling user interaction events, as a possible alternative to the sole handler scenario, concurrent modifications to a specific abstract UI element could take place, leading to possible synchronisation issues between the representations. Techniques for implementing atomic updates in distributed systems [54] can be used to resolve the concurrent update issue, however the overall system would still render representations where the user interaction did not originate as derivatives in violation of the design principles of the PUIR framework (see section 5.2).

Instead, the design presented in section 5.3 introduces a single central component that is tasked with handling user interaction events: the AUI engine, described in section 5.3.3.

When all user input is routed to the AUI engine for processing, another issue arises: most input events are modality-dependent. This observation is based on event capture data presented in Table 6.2, discussed further on page 175.

Legacy GUI systems operate in a very stable and predictable context of user interaction modalities, providing a single representation that is known to all components in the system. A pointer device event can be passed from the low level device driver component, through various other parts of the operating system, up to the graphical toolkit implementation of the widget it operates on without any modification because at all levels the exact context relative to the representation modality is known.

The introduction of centralised user interaction event handling in PUIR presents a complication here also because the AUI engine is by design independent of any modality. How can an event that depends on a modality-specific context, such as coordinates in a graphical environment, be processed by the AUI engine?

Various devices that relate to specific modalities and the issues addressed in this chapter are presented in section 6.2. The role of the AUI engine as central user interaction event handler is discussed in section 6.3. Section 6.4 addresses the issue of modality-specific events and introduces the PUIR approach to resolving it. The chapter concludes with section 6.5, providing a summary of the contributions related to input processing.

6.2 Devices and events

Devices for user interaction are typically managed by the operating system rather than directly by a toolkit. As the user performs physical actions with the device, information about the interaction is collected from the device by a device driver at the OS level [139]. The device driver (possibly with the assistance of other

system components) presents the user interaction as an event, and it gets added to an application-specific event queue for processing by the toolkit that handles the modality that the device relates to.

Some devices are not supported by the OS directly, and require specialised support in either a toolkit or at the application level. These devices are still handled at the OS level for low-level communication, but the information they provide is interpreted at a higher level. A good example is a braille keyboard that communicates with the computer by means of a USB connection, but that is handled by the OS as a game controller. Specialised software at the application level receives the game controller events from the OS device driver and translates the events into keyboard input events. Those translated events are then presented for placement on the application specific event queue.

The list of user interaction devices is virtually unlimited, and technologies to provide users with novel ways to interact with systems and applications continue to be developed. For the purpose of this work, a limited list of common devices is sufficient to discuss issues and their solutions. In terms of user interaction events, it is important to note that often even non-UI events are posted to the application event queue. This is usually related to a need to perform processing that is somehow synchronised with UI event processing. These non-UI events are not discussed in this chapter.

The following list of devices covers a reasonable subset of device types that are common in computing environments:

Keyboard This is an input modality that is not directly related to any representation. Keyboard input is interpreted in terms of *focus* on a specific UI element, with fall back to its ancestors (container elements). In PUIR, focus tracking is handled at the level of the AUI object model, because it is directly related to the user interaction model. Therefore, keyboard events can be passed on to the AUI engine unchanged.

Mouse A computer mouse is a form of pointer device. Mouse operations are inherently graphical in nature, operating in a two dimensional space with high resolution. This input modality provides an additional dimension of information in terms of mouse button state.

Because mouse operations are explicitly related to the position of the *mouse pointer* in the graphical environment, all user interaction events from this input modality will require additional processing before the AUI engine can handle them.

Haptic pointer device Like a mouse pointer device, haptic devices operate in an inherently graphical environment, in this case based on a three dimensional space with high resolution.

Operations from a haptic device are directly related to the 3D position of the pointer of the device. Additional processing will be required before the events can be handled by the AUI engine.

Braille keyboard While this input modality is quite different from a standard keyboard at the physical level, it is mostly equivalent to a keyboard in terms of user interaction. The most notable difference is the common existence of cell selection keys that present the user with a form of cursor addressable input. This still provides an abstract form of input that can be handled at the AUI object model level.

Microphone Voice input is a modality that largely depends on higher level processing. While it is technically possible for voice input to operate on the graphical level, it is not usually practical. For the purpose of this paper, we'll consider all voice input equivalent to keyboard input, and input events that are generated in response to voice input will be presented as keyboard input.

Braille pad This input modality provides a two dimensional grid with a fairly coarse resolution. The data displayed on the grid often has an immediate relevance to the interpretation of input events [115]. If the grid is used to provide information about the visual placement of UI elements, user interaction must be interpreted in terms of the visual rendering agent. If instead the grid is used to provide a non-visual rendering of data, user interaction is likely to be aimed at the abstract level.

Note that it is perfectly possible for a non-visual rendering agent to use the Braille pad to provide a coarse dot matrix presentation of a graphical element. User interaction may therefore operate on both levels at the same time (e.g. the user may be able to use selection keys at the edges to *scroll* through some graphical data, while the more central selection keys are available for selecting specific UI elements).

Regardless of the mode, additional processing will be required before positional events can be handled by the AUI engine.

Touch screen This device is also meant to cover multi-touch displays. User interactions are inherently graphical in nature, operating in a two dimension space with relatively coarse resolution. Due to the interaction model that is typically used for this device, events are commonly pre-processed by a higher level software component that in turn generates events for further processing [13, 104]. Alternatively, native device events can usually be made available for application level handling as well.

Regardless of whether pre-processing is used or not, both context-free and modality-dependent events are generated by touch screen devices and multi-touch displays.

1	→	MOUSE_MOVED(12, 14) on button
		. . .
2	→	MOUSE_EXITED on . . .
3	→	MOUSE_ENTERED(26, 22) on button
4	→	MOUSE_MOVED(25, 21) on button
		. . .
5	→	MOUSE_PRESSED(22, 13) on button
6	→	FOCUS_LOST on . . .
7	→	FOCUS_GAINED on button
8	→	MOUSE_RELEASED(22, 13) on button
9	→	MOUSE_CLICKED(22, 13) on button
10	→	ACTION_PERFORMED on button

Figure 6.2: Example AWT event sequence

6.2.1 Events

The presentation of various devices in the previous section might give the impression that a physical action with an input device results in a single event that captures the essence of the user interaction. Reality shows a very different picture. Although a user may consider his or her action a single user interaction event, at the device level a large amount of events may be generated throughout the course of completing the action. After all, the device has no knowledge about UI semantics and it can therefore only report on very low level aspects of the interaction. E.g. this means that when a pointer device is used to move the pointer from position (x_0, y_0) to position (x_1, y_1), events will be generated for many positions along that track. As these events are interpreted by higher level components, additional events may be generated to provide information about the action at a specific level of interpretation.

What events are posted to the event queue?

Consider the basic operation of a user using a pointer device to push a button in a GUI screen[2]. First the user will move the on-screen cursor to the button by means of the pointer device. Once the cursor is shown within the boundaries of the graphical representation of the button, the user presses the mouse button. Finally, the user releases the mouse button, and the operation has completed.

[2]The example discussed here uses implementation details of the Java AWT and JFC/Swing toolkits for typical Java GUI applications.

Figure 6.2 shows a sample event sequence[3] posted to the AWT event queue for this basic scenario. The meaning of the various pointer device events that are involved in this user interaction is as follows:

1. *MOUSE_MOVED*
   These events are posted whenever the mouse pointer is moved. Due to implementation details in AWT, multiple events may be combined into a single event. Each event specifies the current position of the mouse pointer. The events have no meaning outside the context of the visual presentation. Note that the movement events shown in Figure 6.2 indicate an association with a specific widget ("button" in this case). This is based on an interpretation of the position of the mouse pointer.
2. *MOUSE_EXITED*
   This event is posted when the mouse pointer leaves the area where the widget is rendered on the screen. While it specifies the location within the widget where the mouse pointer exited, this information is generally not used.
3. *MOUSE_ENTERED*
   This event is posted when the mouse pointer first enters the area of a widget. While it specifies the location within the widget where the mouse pointer entered, this information is generally not used.
4. *MOUSE_MOVED*
   See above.
5. *MOUSE_PRESSED*
   This event is posted when a mouse button has been pressed. It specifies the location of the mouse pointer. The event indicates a mouse device state change that may be relevant in the context of user interaction although this is not common.
6. *FOCUS_LOST*
   This event is posted when a widget has lost focus. In this case, the reason is that the widget where the mouse press took place is gaining focus, and only one widget can have focus at any given time.
7. *FOCUS_GAINED*
   This event is posted when a widget receives focus. The button widget receives focus because a mouse button was pressed while the mouse pointer was located in the window area where the button is located.

[3] The event sequence was captured using a custom event queue. This technique ensured that all events posted to the event queue could be captured, along with higher level event capture techniques that were implemented using the Java Observer pattern [49].

Level	Events
Lexical	Device specific events (pointer device movement, key presses, or pointer device button presses) that are essentially independent of any specific widget.
Syntactic	Widget specific events that indicate some kind of operation on the widget, independent of the underlying functionality.
Semantic	UI specific events that provide notification for functional operations.

Table 6.1: Levels on which events operate

8. *MOUSE_RELEASED*
   This event is posted when a previously pressed mouse button has been released. It specifies the location of the mouse pointer. The event indicates a mouse device state change that may be relevant in the context of user interaction although this is not common.

9. *MOUSE_CLICKED*
   This event is posted to signal the completion of a mouse button "click" operation (sequence of pressed/released pairs). It specifies the location of the mouse pointer. Although some applications may associate meaning to the actual position of the mouse pointer when a mouse click occurs, this meaning is irrelevant outside the context of the visual presentation.

10. *ACTION_PERFORMED*
    This event is posted to inform subscribers that the action associated with the button has been performed.

Based on this sample event sequence, and interpretation of the captured events, it is obvious that not all events relate to the same level of UI implementation detail.

The MOUSE_MOVED events are clearly low level device events, reporting on the location of the mouse pointer. While the reported position is already the result of an interpretation of raw (relative) positional data, it lacks any relation to UI elements. Granted, the event itself indicates an association with a widget but this is purely an artefact of standard higher level interpretation of the pointer location. It carries no meaning.

The MOUSE_EXITED event cannot be issued as a low level device event because it depends on knowledge about the positioning of widgets in the focused window. It is generated as a result of processing a MOUSE_MOVED event with a pointer position outside the area used to render the widget. If the pointer position

falls within the area for another widget in the window, a MOUSE_ENTERED event will be generated immediately after the MOUSE_EXITED event.

The FOCUS_LOST event cannot be generated at the device level nor can it be generated solely based on mouse pointer position vs positioning of widgets. It relates to an even higher level concept: focus tracking. The same applies for the ACTION_PERFORMED event that is generated when the functionality associated with the button has been executed,

Event classification

Similar to the identification of four distinct layers of UI design by Jacob [69], events can be categorised as relating to one of three layers (see Table 6.1).

Events like MOUSE_MOVED, MOUSE_PRESSED, and MOUSE_RELEASED can be classified as *lexical* events. These events relate directly to device specific aspects of user interaction at the lowest level. Often, these events will not be acted upon at the application level. Instead, the graphical toolkit depends on these events in order to generate higher level events that carry meaning for the application.

Events such as MOUSE_EXITED, MOUSE_ENTERED and MOUSE_CLICKED can be classified as *syntactic* events. They either depend on a specific interpretation of modality-dependent information such as pointer position, or they are the synthesis of a sequence of low level events, where a combination of lexical events has meaning at the syntactic level.

The FOCUS_LOST, FOCUS_GAINED, and ACTION_PERFORMED events can be classified as *semantic* events. These high level events relate directly to the semantics of the UI and user interaction as it relates to the conceptual model.

Finally, some events are posted to the AWT event queue for the purpose of providing notification about modality-specific changes such as UI elements being moved (at the perceptual level), or being resized. If the hierarchy of UI elements changes, events are also dispatched to notify interested parties.

Event classification distribution

The classification of events across the lexical, syntactic, and semantics layers is important is terms of making a determination concerning the specific processing that an event requires. Obviously, lexical events are handled quite different from semantic events.

Lexical events	
11760	MOUSE_MOVED
175	MOUSE_RELEASED
175	MOUSE_PRESSED
38	KEY_RELEASED
38	KEY_PRESSED
12186	Total

Syntactic events	
158	MOUSE_CLICKED
104	COMPONENT_RESIZED
79	COMPONENT_MOVED
73	MOUSE_ENTERED
57	MOUSE_EXITED
34	KEY_TYPED
505	Total

Semantic events	
116	FOCUS_LOST
70	FOCUS_GAINED
69	WINDOW_LOST_FOCUS
69	WINDOW_GAINED_FOCUS
69	WINDOW_DEACTIVATED
69	WINDOW_ACTIVATED
34	COMPONENT_SHOWN
34	COMPONENT_HIDDEN
7	WINDOW_OPENED
2	WINDOW_CLOSED
539	Total

13230	TOTAL

Table 6.2: Distribution of AWT events by event level

In order to better understand the importance of event classification, and the demands placed on event processing on all three levels, a more extensive event data collection experiment was conducted. During a five minute user interaction session with the Java implementation of the game "Risk"[4] [91] with the FlashGUI interface[5] all events were captured and written to disk. Only events that were actually posted to the AWT event queue were considered because events that were being delivered by means of the Observer design pattern either originated as events from the event queue anyway, or they were entirely synthetic and therefore not a reflection of direct user interaction. The tabulated results are presented in Table 6.2.

It is clear that lexical events comprise the vast majority of all captured events (12186 out of 13230 events, 92%), with 97% of all lexical events being accounted for as mouse pointer movement events. Interestingly, the application does not require specific mouse pointer movements in any aspect of its UI, i.e any movement events are related to positioning the mouse pointer onto specific UI elements prior to performing some action. These events are important however because they constitute the only events that are generated as a direct response to the physical user interaction with the input device they relate to. This was further confirmed through analysis of the implementation details of Java AWT and Swing, revealing that all captured syntactic and semantic events are in fact synthetic events that are generated either as a derivative of one or more lexical events, or in response to a toolkit-specific action.

The analysis also revealed that the majority of semantic events (462 out of 539 events, 86%) are actually generated as a derivative of a syntactic event, establishing an event derivation chain from lexical, through syntactic, and yielding a semantic event.

In view of the PUIR framework design where all events are to be handled at the AUI engine level, the classification of events is significant because only the final derivatives[6] of an event chain are relevant at the AUI level.

In the introduction, it was stated that "most input events are modality-dependent." The event data captured during the five minute user interaction session discussed in this section provides definite proof for this statement. The input events are classified as lexical events, and of all 12186 input events, only key press and key release events are independent of any modality because all other events are mouse pointer events with a direct dependency on the graphical context they

[4]This application was chosen because it is freely available and it is implemented with a complex visual interface, requiring a combination of pointer device input and keyboard input.

[5]The Java implementation of the game "Risk" allows the user to choose between four different UIs: FlashGUI, Increment1GUI, SimpleGUI, and SwingGUI. Although SwingGUI was the more obvious choice, its implementation made event capturing quite complicated compared to using FlashGUI.

[6]As can be expected, a lexical event may be the origin of multiple event derivation chains, because a seemingly minor event as a result of direct user interaction can have an extensive ripple effect.

relate to due to the importance of the mouse pointer position in the interpretation of the events. Therefore, less than 1% of all input events captured during the experiment are independent of any modality.

While it is potentially dangerous to draw conclusions from a single experiment, less formal data capturing throughout the course of the doctoral work that this dissertation reports on indicates that while the distribution of input events between modality dependence and independence varies slightly, the vast majority of input events consistently falls within the category of depending on a specific modality.

6.3 Synchronising user interaction

In most GUI environments, user input events for mouse and keyboard interaction are supported in full. Events from more specialised devices such as a braille keyboard or a haptic pointer device usually require external support because the default toolkits do not incorporate support for these devices. While it is certainly possible to add device support for additional input modalities directly at the level of the graphical toolkit, this technique is not viable due to the complexity of adding functionality to the low level implementation of various graphical toolkits. In addition, the support for the new devices would have to be re-implemented for every new platform because this kind of code is by its very nature not portable.

Java's AWT is a fair representative for graphical toolkits, and it is also the underlying technology used for the experimental implementation of the PUIR concept described in chapter 7. The remainder of this section will discuss the proposed solution within this context. The technique described here is generic enough that it can certainly be ported to other graphical toolkits.

User interaction events from devices that are natively supported by the toolkit are posted directly to a system event queue, from where they are processed by a dedicated event dispatcher thread. As such, all event processing is strictly single-threaded. This ensures that in the general case of a single (graphical) representation, all event processing takes place in a strict sequential fashion.

When user interaction takes place with devices that are not supported by the toolkit, events are not automatically posted to the system event queue. The obvious solution is of course to enable the external device support components to post device events to the system event queue. Early experiments with Java AWT demonstrated that due to specific design choices reflected in the implementation of the AWT event dispatcher, this approach was not viable. Further research into alternative graphical toolkits has shown that it is not feasible to merely inject user interaction events for specialised devices at a sufficiently low level without the intervention of a device driver.

Deeper analysis into the inner workings of the AWT event system indicated that it is possible to install a custom event queue. Doing so ensures total control over what events can be posted to the queue, and over the functionality of the event dispatcher. With a custom event queue in place, it is possible to enforce a strict sequential event ordering across multiple representations by ensuring that all representations guarantee that user interaction events are posted to the single custom event queue. Doing so does require the custom dispatcher to be aware of the various representations, and it requires events to carry information about the representation they are associated with.

Figure 6.3 shows a schematic overview of the event processing design in the PUIR framework. The processing of an event passes through four distinct stages:

1. An event E is generated by a physical device in response to an action taken by the user, and the event is posted to the central AUI event queue.
2. The dispatcher component of the AUI engine polls the event queue, and processes events in strict sequential order. The dispatcher processing involves a transformation of event E in modality-specific context to an event E' in AUI model context. This is a transformation that results in *abstracting* the concrete event.
3. Once event E' has been handled at the AUI layer, event notifications are dispatched to all rendering agents concerning the event that was handled, and resulting changes in the UI state.
4. Each rendering agent presents the completed event processing, and any associated UI state changes in the modality-appropriate manner. This amounts to a *reification* of the results of the event handling.

The discussion in this section shows that the event synchronisation problem in the presence of multiple parallel UI representations can be resolved with a central multiplexing event queue. In order to make it possible for all event handling to take place at the AUI engine level, an *abstraction transformation* step is added to the event dispatcher component. This ensures that the concrete events related to modality-specific widgets in the final UI are presented to the AUI engine as events that operate on abstract widgets. The next section (6.4) provides further details concerning this important transformation.

6.4 Modality-dependent user interaction events

Having ensured that all user interaction events can and should be handled at the AUI engine level, the remaining problem lies with the transformation from

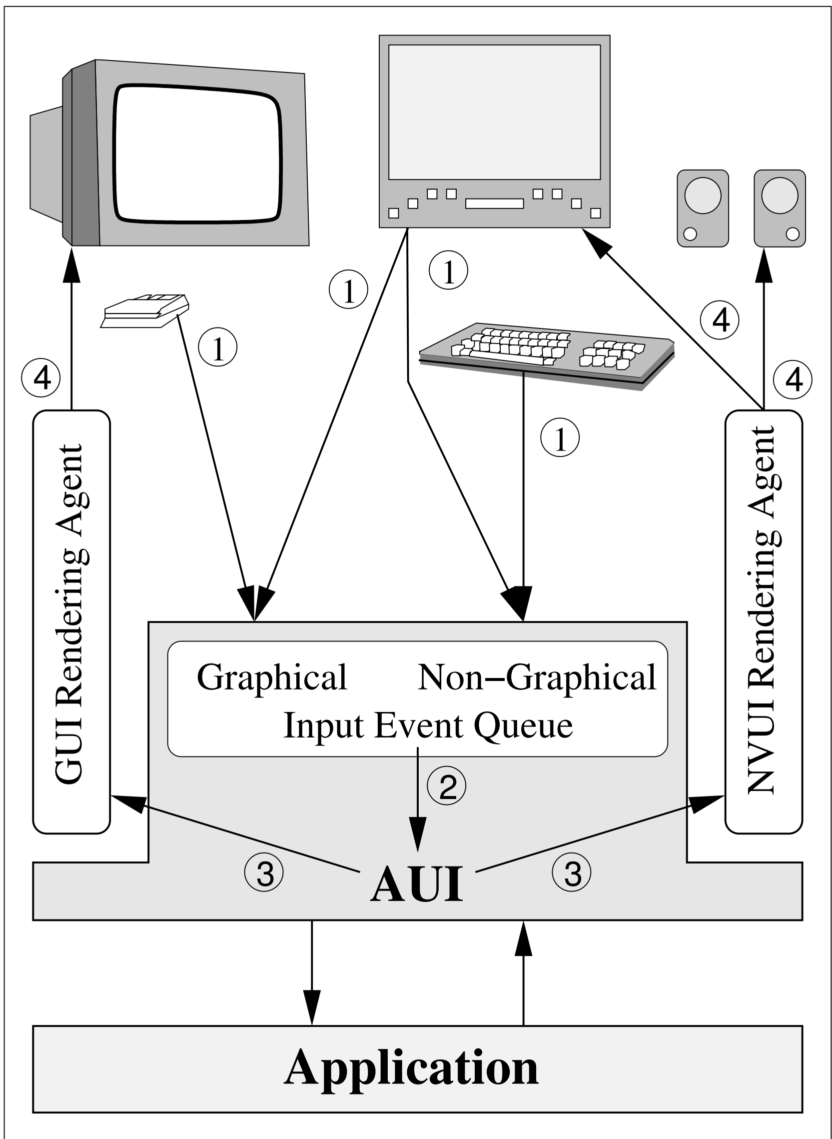


Figure 6.3: Schematic design for synchronised event handling
(1) devices post events to the AUI event queue, (2) AUI engine processes events sequentially, (3) rendering agents receive notification that an event was handled, (4) rendering agents present the effects of event processing.

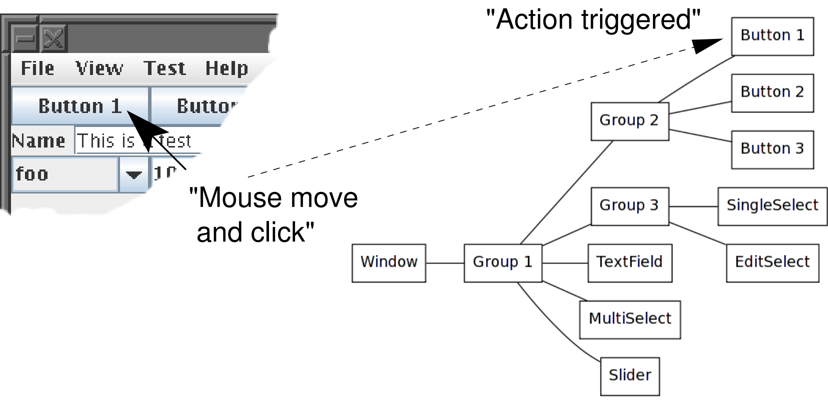


Figure 6.4: Transforming concrete events into abstract events – 1

concrete event to abstract event. Modality-independent input events do not require this transformation because they do not relate to any specific context in and of themselves. Rather, their context is determined by the assignment of focus which is already designed as a function of the AUI engine.

Figure 6.4 provides a schematic illustration of the problem at hand. At the GUI representation level, the user moves the mouse pointer to the button labelled "Button 1", and then presses and releases the mouse button, thereby performing a "mouse click" operation. This multi-step action effectively amounts to a single semantic operation: activating the button labelled "Button 1". At the AUI engine level, this multi-step action is implemented as a single semantic operation: trigger the action associated with button "Button 1". The sequence of multiple lexical events at the GUI level must be transformed into the single semantic event at the AUI level.

Mouse pointer events have been discussed quite a bit in this chapter already, which might give the impression that they are the primary concern for the problem at hand. It is important to note that the solution presented here is in fact aimed at being a generic solution that can be applied to any form of user interaction that generates modality-dependent events. The mouse pointer device is however not only the most commonly used form of user interaction in GUI environments. It is also a good representative for this class of input devices. Haptic devices, touch screens, and multi-touch devices all have in common that they generate events that are associated with a particular position in two- or three-dimensional space, as is the case for mouse pointer devices. Planar braille displays may pose a complication because the limited resolution (in comparison to the graphical screen) necessitates showing only part of the two-dimensional screen space. Sometimes multiple disjunct screen areas are displayed simultaneously, requiring

a mapping between the planar display content and the graphical screen content. As shown in this section, the techniques used for the mouse pointer device (and related devices) is generic, and can therefore easily be implemented for other devices.

At the AUI engine level, all user interaction operates on the semantic level, and therefore only semantic events need to be transformed from their concrete form to an abstract form in the context of the AUI object model. As such, in addition to the event transformation process, an event selection mechanism must be introduced to ensure that only semantic events are processed.

The analysis of events presented in this chapter, specific to the Java AWT and Swing toolkits, strongly suggests that the classification of events and the interpretation of modality specific information is very dependent upon the actual implementation, because the vast majority of all but lexical events are synthetic events, i.e. events generated by the toolkit while processing a lexical event or in response to system state changes. This complexity was also identified (albeit in a rather informal manner) throughout the eight years of research underlying this dissertation. With various Java releases (both minor update releases, and major new version releases) inconsistencies in experiment results were observed. Analysis into the cause for these inconsistencies led to the observation that even between minor releases some implementation details changed sufficiently to render foregone conclusions about event classification and interpretation invalid. It is therefore fair to conclude that establishing a generic mechanism for the classification of events, and with that an event selection mechanism, is an unrealistic goal.

It is clear that intimate knowledge concerning the internal workings of the modality-dependent toolkit is required. Under the assumption that necessary support functionality could be made available by the rendering agent[7], it is important to determine exactly what functionality will be required.

Consider the user interaction shown in Figure 6.4, where the user first moves the mouse pointer to a button, to then perform a “mouse click” operation. This particular interaction by means of a mouse pointer device generates the sequence of lexical events shown at the top of Figure 6.5.

For the purpose of this discussion, the following relations exist between pointer coordinates and toolkit-specific widgets:

- (x_0, y_0) specifies a point within the screen area of the “Name” text field.
- (x_1, y_1) specifies the first point along the path from (x_0, y_0) to (x_2, y_2) that falls within the screen area of the “Button 1” button.

[7] The rendering agent for a specific modality provides controlled access to its underlying toolkit.

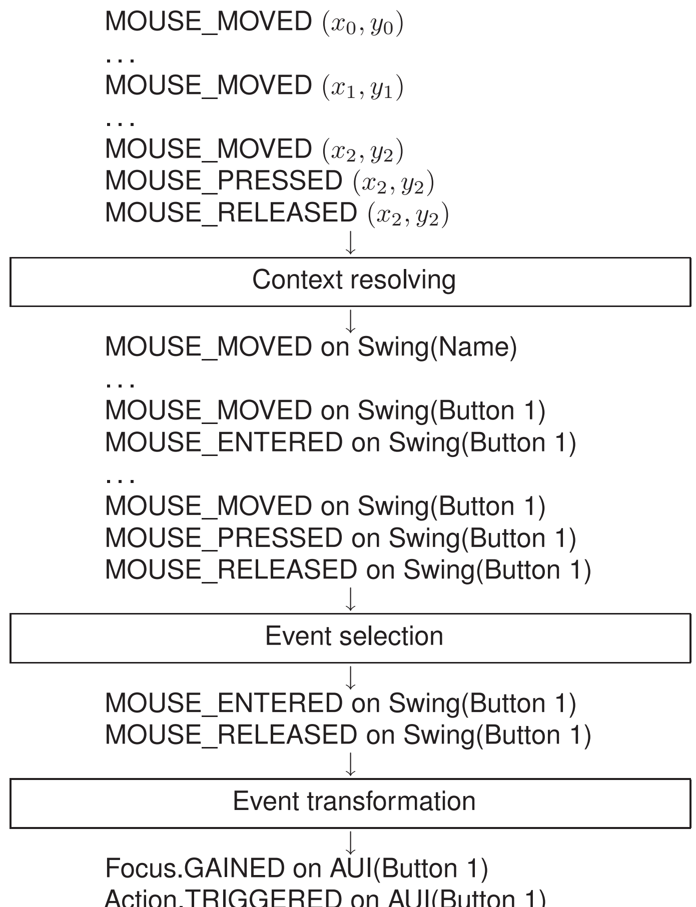


Figure 6.5: Transforming concrete events into abstract events – 2

- (x_2, y_2) specifies a point within the screen area of the “Button 1” button.

Any number of MOUSE_MOVED events will be generated along the path from (x_0, y_0) via (x_1, y_1) to (x_2, y_2), and the “mouse click” operation will be represented by a MOUSE_PRESSED event, followed by a MOUSE_RELEASED event. The pointer positions must first be translated into toolkit-specific widgets in order for further processing to be independent of modality-specific position information. This is accomplished by the context resolving phase, yielding the second sequence of events.

Note the newly inserted MOUSE_ENTERED event. This synthetic event is generated as a result of the mouse pointer entering the screen area for the

"Button 1" button for the first time along its path. It signals the crossing of a widget boundary.

The sequence of context-specific events can then be filtered by the event selection phase. As discussed earlier in this section, the ability to select only specific events is a requirement for being able to ensure that only semantic events are passed to the AUI engine for processing. For the scenario discussed here, only the MOUSE_ENTERED and MOUSE_RELEASED events are retained (third sequence of events in Figure 6.5).

Observe that based on earlier analysis of events in Java (see Table 6.2), these events are respectively syntactic and lexical rather than semantic. This appears to be contradictory to the requirement that only semantic events are to be passed to the AUI engine. Consider however that the information in Table 6.2 is based on capturing events as they occur in a Java AWT and Swing GUI rather than in the context of the PUIR framework. In the current analysis of the user interaction scenario presented in Figure 6.4, the creation of event derivation chains does not take place at the AWT/Swing level. Instead, it occurs in the event transformation process. Rather than *selecting* semantic event, the event selection phase must select events that *will be* transformed into abstract semantic events[8].

Why is MOUSE_RELEASED retained whereas MOUSE_PRESSED is not[9]? Experiments with various different GUI environments led to the observation that the activation of functionality is typically triggered by the completion of the "click" operation, i.e. the moment when the mouse button has been released. This behaviour is applied consistently throughout most GUI environments, possibly to be consistent with the well known "drag-and-drop" operation, where the start of the action is marked by the mouse button being pressed, and the end by the mouse button being released. Also note that many GUI environments invalidate the "click" operation if the mouse pointer was moved into another widget prior to the mouse button being released. Therefore, the MOUSE_RELEASED event is used as trigger for a "click" operation. The event selection process is responsible for ensuring that the MOUSE_PRESSED and MOUSE_RELEASED events are associated with the same widget.

The final phase is the actual transformation of the selected events:

1. Consider concrete event E on widget W (in the context of a toolkit).

[8]The only way to know whether an event would be transformed is to perform a partial transformation during the selection phase. This is quite inefficient. Implementations will combine the selection and transformation phases to work around this dependency

[9]For "drag-and-drop" operations, the MOUSE_PRESSED event is relevant. If a widget can be dragged, the first MOUSE_MOVED event that occurs while the mouse button is pressed will cause a Drag.SELECTED semantic event to be created.

2. Determine abstract widget W' in the AUI object model for which widget W is the representation in the rendering agent.
3. Map concrete event E onto abstract event E' within the context of widget W'.
4. Present abstract event E' on widget W' to the AUI engine for processing..

It is important to determine the AUI widget prior to mapping the event, because any given concrete event may map to different semantic events depending on the widget it relates to. E.g. in the context of a button a mouse "click" triggers activation, whereas in the context of a text field a mouse "click" positions the in-text cursor.

Based on the foregoing analysis, user interaction events that are associated with modality-dependent information for a specific rendering agent require functionality in that rendering agent to:

- Resolve modality-specific information about the target of an event into a specific widget within the context of the rendering agent.
- Identify for each widget in the UI representation what AUI widget it is providing a rendering for.
- For each AUI widget type, provide a mapping of concrete events onto the semantic events it supports.

6.5 Conclusions

This chapter addresses the complications imposed on user interaction handling by the introduction of concurrent representations as provided for by the Parallel User Interface Rendering framework. Two specific problems need to be handled:

- Synchronising user interaction events, generated by devices that each operate within the context of a specific modality.
- Delegating all user interaction event processing to the AUI engine.

Through analysis of graphical toolkit implementations, a determination was made that it is possible to provide a single event queue where all user interaction events shall be posted. This enables the AUI engine to process all events in a strict sequential order. This solution does require the second problem to be resolved in order for the AUI engine to actually handle the events.

Input devices that are associated with a specific rendering agent generate low level lexical events that include modality-dependent information about the target for the event. It is not possible (in a generic manner) to provide for interpretation of this information at the AUI engine level. It is therefore crucial that rendering agents provide functionality to map the modality-dependent information onto a specific widget within their UI representation.

Once it is known what widget an event relates to, a selection and transformation process takes place, making a determination as to what semantic operation is being performed (if any), and what AUI widget the operation is being performed on. For events that indeed indicate a semantic operation, the AUI engine is provided with the transformed event.

The solutions presented in this chapter make it possible for all user interaction to be handled by the AUI engine rather than directly by one or more rendering agents. This ensures consistent semantics for UI elements across all representations, and it also reinforces the principle of providing parallel first-generation representations rather than derivatives.

Chapter 7

Implementation

"Science is knowledge
which we understand so well
that we can teach it to a computer;
and if we don't fully understand it,
it is an art to deal with it."
(Donald Knuth, "Turing Award Lecture", 1974)

The design of the PUIR framework has been substantiated in an experimental implementation to test and demonstrate this novel approach to providing alternative representations of graphical user interfaces. This chapter provides a description of various aspects of the implementation. It is not meant to be exhaustive documentation on its development and operation, but rather a discussion of the methods used to implement Parallel User Interface Rendering. Notable problem areas that were encountered during the implementation are identified and discussed as well.

7.1 Introduction

The experimental implementation for the PUIR framework is written in the Java programming language, using the Abstract Widget Toolkit (AWT) and JFC/Swing[1] as underlying technology for all things graphical. Wherever specific

[1] For the remainder of this work, when *Swing* is mentioned, it is meant to refer to the combination of AWT and JFC/Swing.

implementation details of Java are referenced in this work, they are based on Java SE 6[2] and the Java Platform Standard Edition 6 API Specification [109].

While Java provides a convenient and well-known programming language and runtime environment, it also poses some complications due to some unexpected tight coupling between low-level OS mechanics, AWT, and the Swing toolkit. The reason for this tight coupling is rooted in the fact that although Swing is commonly believed to have been designed based on a Model-View-Controller (MVC) architecture [20], it actually implements a modified MVC architecture where the view and controller parts of each UI element are combined into a single entity [48]: a separable model architecture. In order for user interaction semantics to be implemented in a true modality-independent manner, separation of concerns must be enforced. As such, the experimental implementation required the coupling between the view and controller functions in Swing UI elements to be broken[3]. Section 7.4.3 provides more detailed information on this topic.

A schematic overview of the experimental implementation can be seen in Figure 7.1. This design constitutes a specialisation of the PUIR framework overview shown in Figure 5.7 (page 143). Within the context of providing alternative representations of graphical user interfaces, two specific rendering agents have been developed:

- GUI rendering agent ("GUI Agent"): This component provides the representation of the UI in graphical form. It makes use of the AWT/Swing toolkit as basis for the visualisation. The *look* of the UI in its graphical rendering is therefore virtually identical to a pure Swing-based UI, while the *feel* is defined by the AUI engine design. More details on the implementation of this rendering agent can be found in section 7.4.3.
- Assistive technology rendering agent ("AT Agent"): This component provides the non-visual representation of the UI. Given that it is not within the scope of this work to implement the equivalent of a screen reader, a generic assistive technology solution has been implemented. Observing that most AT solutions operate as stand alone programs on the computer system, the experimental implementation provides support for remote rendering agents, i.e. rendering agents that are not part of the executable process that they represent the UI for. See section 7.4.4 for details on the implementation.

The choice of rendering agents for the experimental implementation also serves as a test bed for the two flavours of rendering agents that are likely to be required

[2] Also known as Java 6, or Java Standard Edition 6.

[3] Modifications to the Swing source code were not deemed acceptable. Instead, rather than decoupling view and controller, the controller functionality was essentially prevented from being activated.

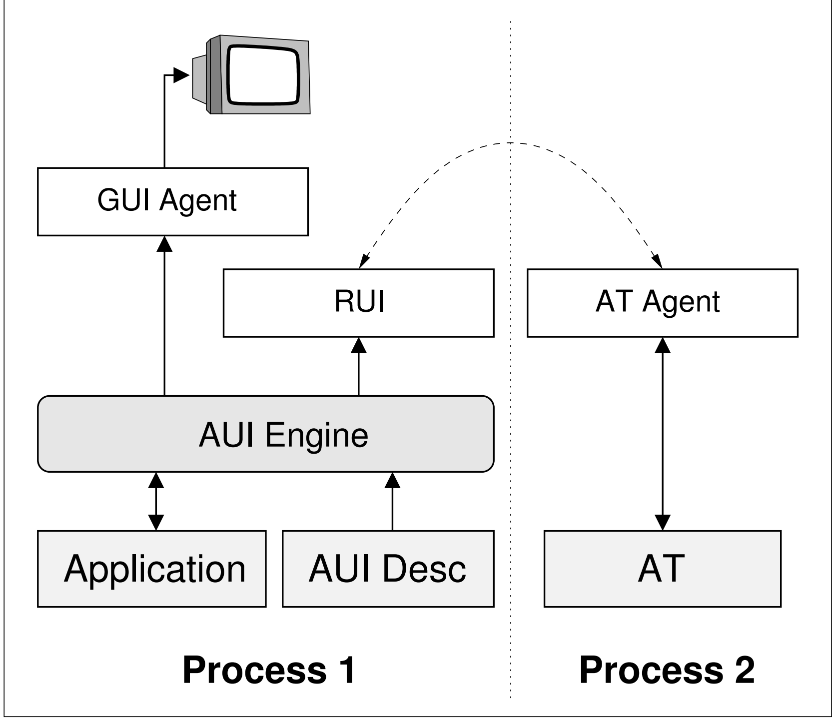


Figure 7.1: Schematic overview of the experimentalconcept implementation

in any general purpose implementation: as part of the application, or as an external process. It is important to note that true to the design of the PUIR framework, the external AT solution (e.g. a screen reader) must interface with the AUI engine rather than hooking into the GUI representation and/or querying the application[4].

Throughout this chapter, the UI visualised in Figure 7.2 will be referenced. It was created as a programmatic implementation of the AUI description listed in Appendix E. The UI is written in Java using the Swing toolkit, and the program code can be found in Appendix D.

The discussion of the concept implementation starts of with one of the core elements of the design: events. Section 7.2 expands upon the analysis presented in chapter 6. The AUI engine implementation is presented in section 7.3, and

[4]An interesting future experiment would be to use an existing screen reader to access the UI through an OSM derived from the GUI representation, and to compare the user experience with operating the UI by means of an AT rendering agent.

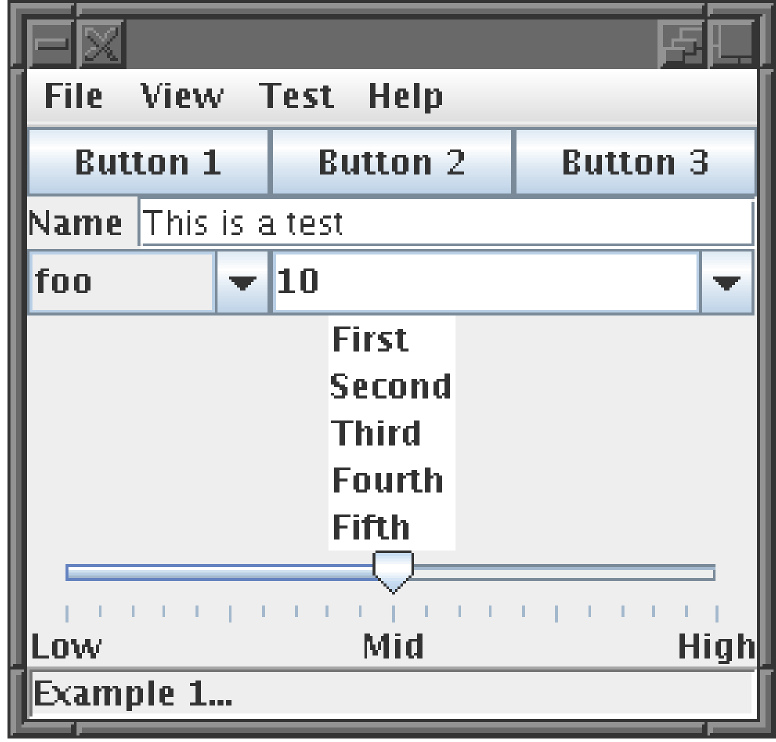


Figure 7.2: GUI representation of the example UI

rendering agents are discussed in section 7.4. In closing, the conclusions are presented in section 7.5.

7.2 Events

By design, AWT/Swing distinguishes between two types of events [157]:

- Low-level events: This is defined as events that are related to the windowing system and user interaction events.
- Semantic events: Everything else.

This somewhat vague definition is reflected in the implementation of AWT and Swing by a distinct lack of actual differentiation between the two types of

events. On the other hand, AWT and Swing provide two different event delivery mechanisms:

- AWT event queue: Events are associated with one specific recipient, thereby establishing a strict one-to-one relationship between the event and the recipient. The events are placed in a FIFO queue, from where they are retrieved by a dispatcher thread. As such, event delivery from the AWT event queue occurs asynchronously. This mechanism is primarily used for events that originate at the OS level, e.g. mouse and keyboard interaction as they are processed by their respective device drivers, but occasionally AWT components post events to the queue as well.
- Observer pattern: With this mechanism, objects can register themselves as listeners ("observers") for specific events, or groups of events [49]. Delivery of events is synchronous, and the observer design pattern establishes a one-to-many relationship between the event source and the listeners. The vast majority of events that are delivered by means of the observer pattern mechanism are notification events, posted at the completion of the operation they relate to.

To complicate matters, most events that are posted to the AWT event queue are also sent to listeners by the event recipient. Capturing events directly from the AWT event queue and by means of listeners will therefore often yield duplicate events in the capture stream. In fact, analysis of captured event logs for both the AWT event queue and all available listener interfaces for components in a UI shows that the vast majority of events on the queue get duplicated, sometimes even more than once because often an event might satisfy the scope of multiple listeners.

In the interest of being able to reuse the user interaction device handling provided by the Java runtime environment, it is important to determine what events are to be captured and transformed for processing by the AUI engine in the PUIR experimental implementation. This analysis is a continuation of the work presented in sections 6.2.1 and 6.4. In order to ensure that all events get captured, custom code has been written to capture events on the AWT event queue and events that are available for delivery to specific listeners. The example UI (see Figure 7.2) was used as test bed for this data collection.

Events were captured from the initial invocation of the program, throughout the creation of the GUI representation, and continuing during some sample user interaction. A total of 12334 events were captured, of which 923 were found to be related to asynchronous processing requests, and they therefore are not at all related to the GUI representation or the semantics of the application. They are not included in the data presented in this section. Another 531 events are object state change notifications that are dispatched according to the Java beans

Queue	Observer	Event
		Semantic Events
0	2	FocusEvent[FOCUS_GAINED]
2	2	WindowEvent[WINDOW_ACTIVATED]
2	2	WindowEvent[WINDOW_GAINED_FOCUS]
1	1	ComponentEvent[COMPONENT_SHOWN]
1	1	FocusEvent[FOCUS_LOST]
1	1	WindowEvent[WINDOW_DEACTIVATED]
1	1	WindowEvent[WINDOW_LOST_FOCUS]
1	1	WindowEvent[WINDOW_OPENED]
		Structural Events
10	10	ComponentEvent[COMPONENT_RESIZED]
6	6	ComponentEvent[COMPONENT_MOVED]
0	1	ContainerEvent[COMPONENT_REMOVED]
0	1	ContainerEvent[COMPONENT_ADDED]
		Notification Events
0	231	HierarchyEvent[ANCESTOR_RESIZED]
0	120	HierarchyEvent[ANCESTOR_MOVED]
0	110	AncestorEvent[ANCESTOR_MOVED]
0	2	AncestorEvent[ANCESTOR_ADDED]
0	2	HierarchyEvent[HIERARCHY_CHANGED]

Table 7.1: AWT/Swing events during UI creation

For each event, a count is given for how many times the event occurred in the captured data from the AWT event queue ("Queue") and how often it was made available to listeners ("Observer").

specification, and they effectively duplicate functionality that is already covered by other events. Therefore, they are also not included in the data presented here. This leaves a total of 10880 events that remain to be considered.

Somewhat surprisingly, 519 events were logged prior to any user interaction taking place. These events are therefore related to the creation of the GUI representation in Swing. Section 7.2.1 presents a further analysis of these events, while section 7.2.2 takes a look at the remainder of the events (10361) that were captured during user interaction.

7.2.1 UI creation events

Given the quite large amount of events that were captured prior to any user interaction, a determination must be made whether any of these events are actually relevant within the context of the PUIR framework.

Table 7.1 shows a breakdown of the UI creation-time events that were captured. The discussion in section 6.2.1 suggests that events can be classified across three levels: lexical, syntactic, and semantic. Yet, this classification does not appear to be appropriate for the events that occur during the creation of the GUI representation of the UI. The reason for this apparent mismatch is that the discussion in section 6.2.1 concerned user interaction events, while the events discussed here are not related to any user activity. Therefore, an alternative classification is used here to guide the discussion.

Notification events account for 90% of all events, with ancestor events being described in documentation as notification-only events that predate the introduction of hierarchy events, and those are also classified as notification events. Structural events account for only about 6%, and these are primarily (32 out of 34) synthetic events that drive internal processing of changes in the overall dimensions of the window.

It is important to note that there are no actual events being posted about the creation of the object hierarchy of the UI (aside from two ancestor events, and they are certainly not representative of actual UI creation because many more components are added to the hierarchy for the example GUI shown in Figure 7.2).

The remaining 20 events (4%) are dispatched by the windowing system and/or the low-level widget implementations. These can be considered semantic events by their very nature, because they indicate important aspects of the UI within the context of the system: determination of focus[5], and determination of visibility[6]. Several of these events seem to occur in predefined sequences, establishing chains of events. E.g. when the windowing system indicates that the application receives focus, the WINDOW_GAINED_FOCUS event is dispatched, which in turn triggers the WINDOW_ACTIVATED event, and also the FOCUS_GAINED event on the component that is to receive in-window focus within the window that gained focus. The reverse is also true (when losing focus), and together they account for 16 out of 20 events. The remaining four events relate to the window becoming visible to the user when it is opened.

In consideration of the observations discussed in this section, it is clear that the

[5] Focus refers to the designated recipient of otherwise context-free user interaction events. Most commonly, it is used to indicate what component should receive keyboard input.

[6] In the context of this work, a more appropriate term would be “noticeability”. Vision-based idioms are commonplace in language, however, and they are generally accepted by the blind.

AWT event queue can be used to determine when a window becomes visible, and when the window gains or loses focus. It does not provide any information about the UI component hierarchy being constructed. Within the PUIR framework that is acceptable because the UI construction is entirely driven by the AUI engine. The event capturing process could be limited to just looking for window focus events at UI creation time while all other events can be discarded. However, the X Window System allows an application to request focus, and therefore the PUIR framework can automatically assign focus to its UI at application startup. This means that there is no need to capture any of the events that occur during the creation of the GUI representation in Swing.

7.2.2 User interaction events

Once the UI has been presented to the user, interaction can take place. During the sample session 10361 events were recorded while the user was using keyboard and pointer devices to interact with the user interface (see Table 7.2). Based on the observations concerning events at UI creation time, several groups of events are known to be irrelevant to the discussion presented here and they are therefore not included in the table:

- Focus events (FocusEvent and most cases of WindowEvent) are semantic events that provide notification of focus changes, and in the PUIR framework those will be generated by the AUI engine. This accounts for 148 events. Note that the WINDOW_CLOSING event on the AWT event queue should not be ignored because it indicates that a request was made from outside the application context to close the window and terminate execution. No alternative way has been identified to determine this state. Therefore, this event must be captured, and after transformation, be presented to the AUI engine for processing.
- Structural events (ComponentEvent and ContainerEvent) are used internally in Swing, and carry no meaning within the context of the PUIR framework. This accounts for another 322 events.
- Some notification events (AncestorEvent and HierarchyEvent) are not at all relevant within the context of the PUIR framework, and can therefore be discarded. This accounts for 2982 events.

This leaves 6909 events to consider. As shown in Table 7.2, these events can be categorised in three groups: semantic events, syntactic events, and lexical events. Each of these groups will be discussed in the remainder of this section.

Queue	Observer	Event
		Semantic Events
8	267	MouseEvent[MOUSE_ENTERED]
8	241	MouseEvent[MOUSE_EXITED]
0	180	ItemEvent[ITEM_STATE_CHANGED]
0	160	MenuEvent[]
0	37	ActionEvent[ACTION_PERFORMED]
0	32	JTextComponent$MutableCaretEvent[dot=]
0	6	ListSelectionEvent[firstIndex=]
0	4	PopupMenuEvent[]
1	1	WindowEvent[WINDOW_CLOSING]
		Syntactic Events
25	50	KeyEvent[KEY_TYPED]
35	33	MouseEvent[MOUSE_DRAGGED]
27	12	MouseEvent[MOUSE_CLICKED]
0	32	MenuDragMouseEvent[MOUSE_RELEASED]
31	0	InputMethodEvent[...]
0	8	MenuDragMouseEvent[MOUSE_DRAGGED]
		Lexical Events
1787	3428	MouseEvent[MOUSE_MOVED]
46	92	KeyEvent[KEY_RELEASED]
46	92	KeyEvent[KEY_PRESSED]
43	93	MouseEvent[MOUSE_PRESSED]
43	41	MouseEvent[MOUSE_RELEASED]

Table 7.2: AWT/Swing events during user interaction

For each event, a count is given for how many times the event occurred in the captured data from the AWT event queue (“Queue”) and how often it was made available to listeners (“Observer”).

Semantic events

The majority of semantic events are not relevant within the context of the PUIR framework because they provide notification of semantic operations after they took place (and usually after they have been rendered visually). The PopupMenuEvent should be pointed out as a very implementation-dependent event, because it reflects the fact that in Swing, pull down menus are represented as pop-up windows with the menu content. The event itself is not relevant in the context of the experimental implementation.

The PUIR approach handles the execution of semantic operations at the AUI engine level, which then dispatches events to the rendering agents to indicate that rendering of the operation should take place.

Some semantic events *are* important within the context of this work. The aforementioned WINDOW_CLOSING event illustrates this quite well. In addition, analysis and experimentation with Swing has shown that only the windowing system provides accurate notification when the pointer device cursor enters or leaves the window representation on the graphical screen. This information can be important for focus tracking, and therefore all instances of these events on the AWT event queue must be retained (17 events).

In conclusion, 928 out of 945 semantic events can be discarded. Note however that in section 6.4 a conclusion was reached that semantic events are what matters, because the AUI engine handles semantic events. Why then discard the vast majority of the captured semantic events? The answers lies in that very section... What matters are events that *will be* transformed into semantic events.

Syntactic events

The syntactic events cover two cases. First, the InputMethodEvent is an internal mechanism in Swing to support alternative text input methods. While this is an interesting feature in Swing, support for the input method framework is beyond the scope of this work. Second, the remaining notification events are essentially synthetic events that are posted as an interpretation of device-specific lexical events. E.g. the KEY_TYPED events are generated whenever a KEY_PRESSED event is posted for a key or key/modifier combination that represents a valid key, and MOUSE_DRAGGED events are generated when a pointer device cursor is being moved while one of its buttons are pressed (and held). None of these events are of importance to the PUIR framework because they are derivative events that reflect user interaction semantics at the GUI representation level. The PUIR framework handles semantics at the AUI engine level.

Therefore, another 253 events should be discarded.

Lexical events

The remaining 5711 events are all lexical events. Pointer device movement events account for 91% and they are mostly irrelevant. Given that the PUIR framework operates on semantic events, and in view of the discussions on user interaction events in chapter 6, it is clear that the position of the pointer device cursor can be relevant when a semantic event is triggered, but that the movement towards that position is meaningless[7]. For the purpose of this discussion, most of the movement events can be ignored.

Keyboard input events for key presses and releases are important because they carry direct user input information. Likewise, events for pointer device button presses and releases are important because they often indicate the activation of a specific function. As mentioned with semantic events, only those that were captured from the AWT event queue are important. This amounts to 178 device events.

7.2.3 Reuse and reduce

In total 195 out of 10361 events (about 2%) are deemed relevant to the PUIR framework, all related either to external circumstances that are not communicated to the system in any other way, or to actual user interaction. All other events are discarded. This has the immediate side effect that the controller part of Swing widgets will not be presented with any events, thereby essentially rendering it null and void. In a roundabout way, this effectively accomplishes the equivalent of the decoupling of view and controller mentioned in section 7.1. All relevant user interaction events must be passed to the AUI engine, where the operation will be processed.

7.3 AUI engine

The AUI engine provides the conceptual layer of the PUIR framework. From an implementation perspective, it is one of the least complex components because it is modality-independent by design and it has no direct dependencies on system level functionality.

For the experimental implementation, the AUI engine was chosen as the first component to be implemented. The rationale behind this decision was simple:

[7] One potential exception could be observed with a drag-and-drop operation, but upon closer examination it is clear that even for that operation only the starting position and the ending position are relevant, and those are provided with the mouse button events.

if it is to be truly independent of any specific rendering, it must be possible to develop the AUI engine prior to any other component, and the implementation of any other component shall not require any changes in the AUI engine.

The design of the AUI engine, presented in section 5.3.3, identifies three areas of functionality:

- *Translation of the AUI description from its textual UIDL form into the UI object model.*
  Although the design of the Parallel User Interface Rendering approach requires the UI to be specified by means of a UIDL-based AUI description, no specific UIDL is identified. Several existing UIDLs were evaluated, yet in the end a custom language was deemed the best option for the experimental implementation. More information about the UIDL, AUI descriptions, and translating the AUI description can be found in section 7.3.1.

- *Focus management*
  While the choice to provide just two rendering agents could avoid the concept of focus management altogether by conveniently excluding any form of user interaction that is not explicitly related to a UI element context, doing so would undermine the validity of the experiment. The implementation of the focus management component of the AUI engine is presented in section 7.3.4.

- *Implementation of the user interaction semantics of UI elements*
  This function of the AUI engine defines the very core of the Parallel User Interface Rendering approach. It ensures consistency in user interaction, and it also makes it possible to provide multiple coherent concurrently accessible representations (section 5.2.4) that are all created as first-generation renderings. The UI elements of the AUI object model are presented in section 7.3.3.

In addition to these three core functions, some additional components of the AUI engine warrant further discussion. As shown in Figure 7.1, all interaction between the application and the PUIR framework takes place via the AUI engine. Therefore, an API is to be defined to allow this interaction to take place in a well-defined manner. The API is presented in section 7.3.2.

The last component to be discussed is crucial as a test for the focus management functionality: keyboard-based user interaction. Section 7.3.5 provides more information on this important topic.

7.3.1 PUIR UI Description Language

The design of the PUIR framework is largely based on the ability to describe a UI in abstract form. While many user interface description languages have been developed in recent years (see section 4.3.2), none were found to be useable for describing user interfaces at an abstract level within the PUIR framework without significant modifications. Rather than spending a significant amount of time trying to make two somewhat mismatched pieces fit together, the decision was made to define a specific UI description language for the PUIR framework: PUDL. The development of PUDL occurred primarily in an on-demand fashion throughout the implementation of the Unified User Interface (U2I) widget toolkit (see section 7.3.3). As such, the specification is quite minimal, albeit sufficient to support the needs of the experimental implementation and to ensure that all aspects of the PUIR design can be exercised based on a PUDL-specified UI. Future development towards production level implementations should definitely include efforts to unify the requirements for UI descriptions for the PUIR framework with a well established UIDL (see section 9.2).

The Document Type Definition (DTD) for PUDL can be found in Appendix C, and an example UI description is listed in Appendix E.

The PUDL description of a user interface contains all information needed to create an AUI object model in the AUI engine, representing all semantic information and functionality. A UI description can (and often will) contain multiple windows. Typically, only one is marked visible at application startup, but any window can be shown or hidden at any time.

As stated in section 5.3.2, the PUIR framework allows for rendering agent specific annotations to be added to the specification of UI elements. Support for this feature is included in PUDL by means of the “agentInfo” element that can be added to any UI element. The annotation must refer to a specific rendering agent, and it can contain an arbitrary number of $(key, value)$ pairs that contain implementation specific information that can be used to enhance the representation of the UI element. Typical uses include adding an image as button label, specifying foreground and background colours, selecting specific visualisation options, . . .

It is important to note that the UI description file is the single source for the creation of an application UI. It is not associated with a source code file, or any other dependency. This makes it possible for non-structural UI changes to be applied without needing to update the application. One potential use for this feature is enhanced accessibility support. Although the PUIR framework is aimed at promoting accessible and useable user interfaces, many developers are likely to still label UI elements with essentially obscure names. It is feasible to include functionality in an assistive technology rendering agent that allows users to assign

labels of their own choice to UI elements. The new assignments can be applied to the UI description file, and from thereon the application will provide the UI with the new labels in place.

7.3.2 AUI engine API

One of the main functions of the AUI engine is to shield the inner workings of the PUIR framework from the application. All interactions must be performed using the API that the AUI engine provides. This section lists the main functions that are provided by this API. Note that path identifiers are used to reference widgets and attributes in widgets. The general structure is:

- `widget`
  This references a specific widget as child of the current root widget. Note that for various API functions, a specific widget can be supplied as root. Supplied `id` arguments are always relative to that root widget.
- `widgetA.widgetB`
  This references a widget as a child of another widget.
- `widget.attr`
  This references an attribute in a widget.

The main API functions are:

- `public static void init(String fileName)`
  This function is called to initialise the user interface based on the UI description file passed as parameter. The UI description will be loaded and the AUI object model will be created. Depending on the implementation of the rendering agents, some representations are likely to also get created at this point.
- `public static void show(String id)`
  This function can be called to request that the window with the given `id` be made visible.
- `public static Component resolve(Component w, String id)`
  This function can be used to gain access to specific widgets within the AUI object model. It resolves the given `id` within the context of the given widget `w`, i.e. it looks for a widget with the given id in the sub-tree rooted at the given widget.
- `public static Object getAttribute(String id)`
  This function can be used to retrieve the value of an attribute in a widget, referenced by the given `id`.

- `public static void setAttribute(String id, Object val)`
  This function is used to set the value of an attribute in a widget.

The provided API is sufficient for the experimental implementation, but it is unlikely to satisfy the requirements for more sophisticated use of the PUIR framework.

7.3.3 Unified UI widgets

Section 7.3 refers to the implementation of user interaction semantics for UI elements as the very core of the PUIR approach. Although this seems to be a rather bold statement, it is quite accurate. The user interface of any application is primarily defined by its functionality and the data it encapsulates. Those two core features are provided by the Unified User Interface (U2I) widget toolkit.

The design of the Parallel User Interface Rendering framework is based on the very notion that the traditional conceptual models that form the basis for UI representations are appropriate for both sighted and blind users as discussed in section 5.3.1. The target user survey presented in chapter 3 indicated that elements of graphical user interfaces, although often considered visual, are easily understood by the blind. After all, blind individuals do operate in a predominantly sighted world, and many aspects of e.g. manipulating controls are not related to vision. Therefore, the design of the U2I widget toolkit incorporates UI elements that all users are known to be familiar with at a conceptual level. Figure 7.3 presents the class inheritance diagram for the widget toolkit provided in the experimental implementation.

It is important to note that the inheritance relations between widgets are generally not captured in the conceptual model. They are merely an artefact of the implementation at the code level. While interviewing some of the participants in the target user survey, it was noted that however that given the explanation that an `editSelectList` widget is essentially a `textField` with a list of default options that one can choose from, the respondents appeared to grasp this concept without much thought. Furthermore, this understanding was observed regardless of the respondent's level of experience with computer systems.

The experimental implementation provides the user interaction semantics presented in section 5.3.3, specifically in function of the operations listed in Table 5.3. Semantic operations are implemented as events (or sub-functions of events), and they are handled according to the work flow presented in Figure 5.8:

1. User interaction causes a semantic event to be dispatched to the U2I widget
2. The semantic event is handled, performing a specific operation on the widget

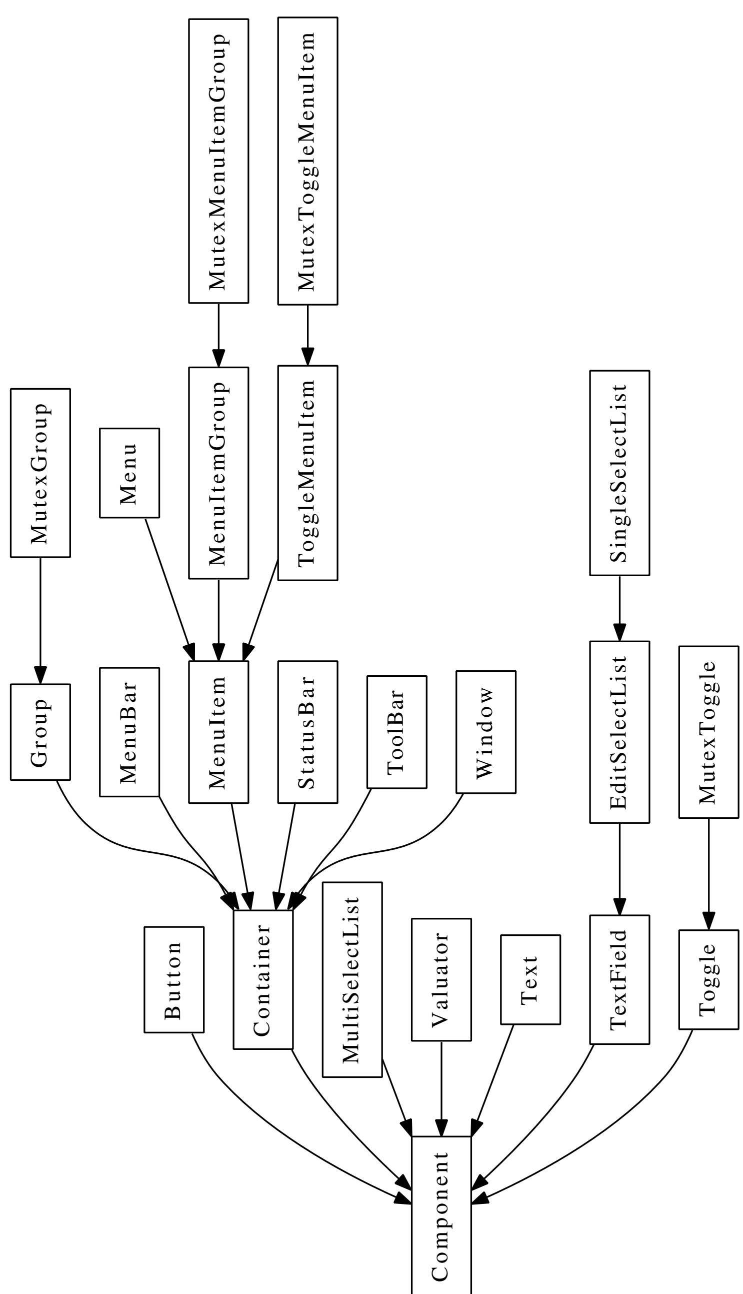


Figure 7.3: Class inheritance tree for U2I widgets

3. A notification event is dispatched to all rendering agents (and the application if it is registered as an observer)

This very simple 3-step algorithm is ultimately one of the most powerful concepts in the PUIR approach because it is applied consistently throughout the entire system. It also ensures that every representation and every modality of user interaction provide the exact same semantics for any given UI element. The perceptual context of interaction may differ greatly, but conceptually all representations operate the same.

7.3.4 Focus management

Focus management is arguably one of the least complicated components of the AUI engine. Focus is maintained at two levels: window and widget.

Window focus is on most computer systems a windowing environment feature, providing a mechanism for application code to request focus to be given to a specific window, and also providing for notification events to be passed to an application when one of its windows receives focus. The AUI engine maintains a reference to the application window that holds focus (if any). If focus lies with another application, this reference will be empty.

Widget focus is maintained by the AUI engine per application window, i.e. in every window there will be a widget that holds focus whenever that window holds focus. When a focus traversal operation takes place (as discussed in section 5.3.3), widget focus moves to the next (or previous) widget according to the focus traversal order. Where many graphical toolkits provide functionality to redefine the focus traversal order, PUIR does not. Moving focus between UI elements is a semantic operation, and allowing application code to change UI semantics violates the design principles that the PUIR framework is based on.

7.3.5 Keyboard-based user interaction

Keyboard-based user interaction covers multiple possible devices. A regular keyboard is certainly the most common device, but e.g. section 6.2 lists additional devices that offer interaction that is equivalent to a regular keyboard. Another commonality between these devices is that the interaction with the device is not explicitly associated with any UI element. From the user's perspective, there is a definite association at a mental level, where the user e.g. types text and he or she "knows" what text field the characters should appear in[8].

[8]An all too common complaint from users related to GUIs has been that the mouse pointer device is often too sensitive, and while operating the keyboard to enter text, focus might suddenly shift to

It is clear that keyboard input is closely coupled with focus management, although it is certainly not a exclusive binding. Focus can change as a result of most semantic operations[9], and it is therefore affected by most forms of user interaction. Keyboard interaction is merely a special case by virtue of being both a producer and a consumer of focus changes.

Keyboard interaction is a very powerful concept, as exemplified by the fact that many experienced computer users tend to favour keyboard navigation over mouse pointer navigation. In part, this may be because switching between user interaction by mouse and by keyboard requires a mental "context switch" which is known to impose a minimal delay. Also, in contrast with other forms of user interaction, it is very specific in how it can interact with a specific widget type. Pointer devices offer movement, and press/release actions on a limited number of buttons, whereas keyboard interaction can provide a large number of key combinations to perform sometimes complex operations on a widget. The navigation, selection, and input operations that can be performed on a text field offer a good example. On the other hand, keyboard interaction is not directly related to any specific representation. Early on in the development of the experimental system, the unfortunate choice was made to implement keyboard actions at the level of the AUI engine. This is obviously a flawed design given that the AUI engine is designed to only handle semantic events. Future development should ensure that the keyboard interaction logic is decoupled from the U2I widget implementation.

7.4 Rendering agents

The perceptual layer of the PUIR framework is implemented by the rendering agents. Theoretically, any number of rendering agents is supported by the system design, although it is not common for more than two or three to be in use at any given time. A rendering agent is responsible for:

- A representation of the UI within the context of a specific modality
  This function is described in section 5.3.4. It is a *reification* process that transforms the conceptual UI at the AUI engine level into a concrete UI, to be presented to the user as the final UI based on a modality-specific toolkit. All interaction from the AUI engine to the rendering agent is event-based, by means of the Observer pattern.

another UI element. This causes frustration because the user maintains a concept of focus in their mental model, and involuntary focus changes render that notion invalid.

[9] In general, when a semantic operation is performed on an unfocused widget, focus will shift to that widget prior to the semantic operation being performed.

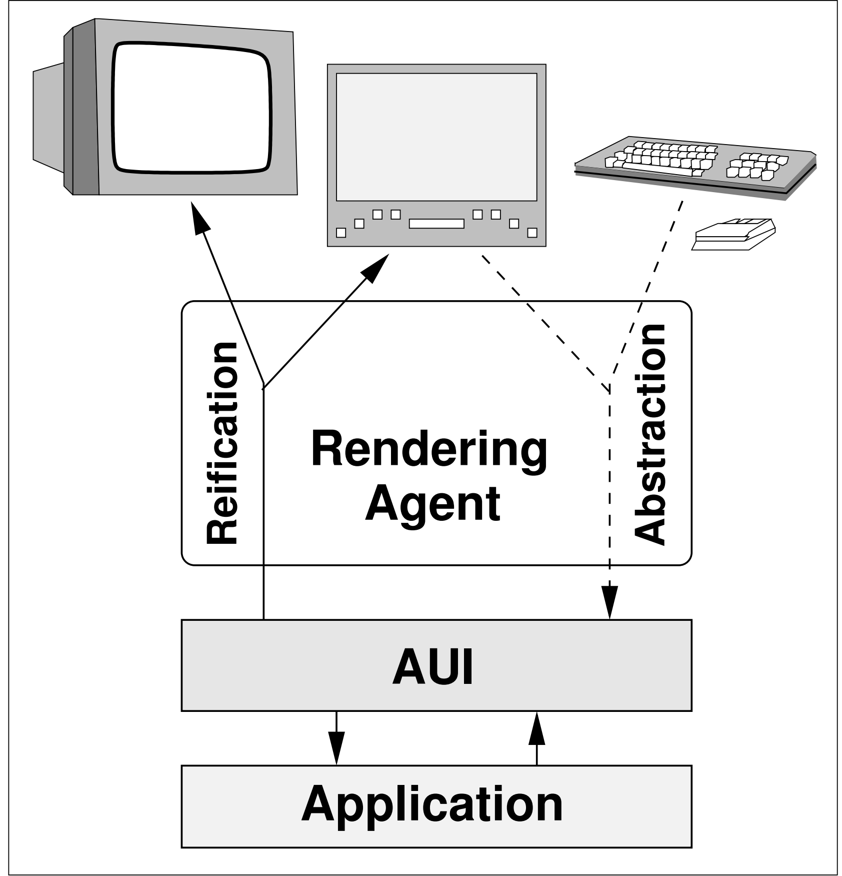


Figure 7.4: Event abstraction vs presentation reification

- A user interaction event selection and transformation process within the context of a specific modality
  This function has been discussed in section 6.4. It handles the fact that some input modalities are tightly coupled with an output modality, and it is therefore not possible to handle user interaction events from those modalities without transforming them into modality-independent semantic events (if appropriate). The process involves making a determination on whether the event is relevant (i.e. whether it carries semantic meaning), and if so, it needs to be transformed into the abstract user interaction it represents.

This section first discusses the reification function provided by a rendering agent in section 7.4.1, followed by a short recap of the abstraction function in section 7.4.2. In-process rendering agents are presented in section 7.4.3, and section 7.4.4 discusses external rendering agents.

7.4.1 Reification

Conceptually, the rendering agent performs a reification of the conceptual UI, yielding a concrete UI that is presented to the user as final UI through a modality-specific toolkit. On the implementation side, two distinct cases are to be considered: structural and perceptual.

Structural reification

The AUI engine provides notification to all rendering agents by means of the `UIEvent` class when the UI is being created, and when it is getting destroyed.

- `CREATED`: This event is dispatched when the UI is first created.
- `ADD_WINDOW`: This event is dispatched when a new window is being added to the user interface of the application. The U2I window object is provided as a parameter to the event. Note that at this point, the window has not yet been populated with widgets.
- `REMOVE_WINDOW`: This event is dispatched when a window has been removed from the user interface. The U2I window object is provided as a parameter to the event. The event indicates that the window is set to be destroyed at the AUI engine level, and no further operations should be attempted in relation to the window once this event has been received.
- `DESTROYED`: This event is dispatched to provide notification that the user interface is about to be destroyed. This is typically used to indicate that the application is terminating.

The `ADD_WINDOW` event is the entry point for the construction of the concrete user interface. Upon receiving the event, an incremental process commences that ultimately results in creating the representation of the UI. The presentation object for a U2I window widget is a root container, i.e. it is a container widget that does not have a parent. Upon receipt of the `ADD_WINDOW` event for a specific U2I window widget, the following algorithm is applied:

1. Determine the appropriate presentation class for the U2I widget.

2. Create the presentation widget.
3. Register the presentation widget as a child of its parent (if any).
4. Associate the U2I widget as peer with the presentation widget.
5. Register the presentation widget as listener for U2I widget events.

If the U2I widget is a container, its presentation widget in the concrete UI will receive `ADD_COMPONENT` events, which will trigger the algorithm above once again. This tree growth process continues until the UI has been populated on the concrete UI side.

Modifications to the user interface can propagate from the conceptual UI to the concrete UI by means of `ADD_COMPONENT` and `REMOVE_COMPONENT` events.

Perceptual reification

The process of creating the concrete UI results in all widgets in the concrete UI having registered themselves with their U2I peer. This enables the widgets at the AUI engine level to dispatch notification events whenever a semantic operation has taken place at the conceptual level. Presentation widgets will provide implementations for all supported notifications. Since events dispatched from the AUI engine are PUIR-specific there is no risk that such event would inadvertedly trigger non-perceptual functionality in the final widget.

7.4.2 Abstraction

The process of user interaction event handling, and the associated abstraction transformation, is discussed in section 6.4.

7.4.3 In-process rendering agents

The experimental implementation provides an in-process rendering agent based on AWT/Swing. Due to the fact that Swing is designed based on a separable model architecture [48], the view and controller parts of the traditional MVC architecture are tightly coupled into a user interface object. In order to ensure that all user interaction semantics are handled by the AUI engine, as is required in the PUIR framework, various levels of functionality had to be bypassed in the Swing-based rendering agent. This was accomplished programmatically by implementing the presentation classes as derivatives of their Swing counterparts.

The event delivery mechanisms used in AWT and Swing allow events to be intercepted, which also happens to be needed in order to provide for the synchronised event handling within the PUIR framework. Unfortunately, it was not possible to accomplish this cleanly in all cases, because of presumably unintentional dependencies between the user interface object and the model component.

An additional complication that was uncovered in the course of implementing the Swing-based rendering agent relates to the implementation of pull-down menus and single select lists (implemented in Swing as the `JomboBox` class). Swing does not provide any functionality to determine the owner of a pop up window (which is used to render the pull down menu content, and the selection list for the combo box). Analysis of the inner workings of these components led to the discovery that both the widget and the associated pop up retained references to a shared model.

Complexities similar to those mentioned in this section are to be expected when implementing a sophisticated novel framework on top of an existing toolkit.

7.4.4 External rendering agents

The framework for assistive technology support has been developed as well as part of the experimental implementation. Given that the majority of AT solutions (screen readers, . . .) are implemented as stand alone programs, the design was extended to include support for remote rendering agents. The overall design of the Parallel User Interface Rendering framework lends itself well to this extension because there is no actual expectation that a rendering agent provides a representation[10].

Figure 7.5 shows a schematic overview of the remote rendering agent support. Rather than implementing a specific rendering agent to provide AT support, a Remote User interface (RUI) stub rendering agent (RUI) was developed. This stub behaves exactly the same as any actual rendering agent, though it's purpose is merely to capture events, and broadcast them to a message bus daemon outside the application process. Assistive technology solutions can connect to this message bus daemon, and register to receive the PUIR events. Furthermore, because the events are passed through the message bus in such way that it is possible to reconstruct them upon receipt, a rendering agent can be implemented as an external process, and act upon the events in a manner that is identical to an in-process rendering agent. From the application side, it is not known that one or more rendering agents are external.

[10] This fact has actually been used quite extensively during the development of the framework, as a mechanism for capturing events in a running system.

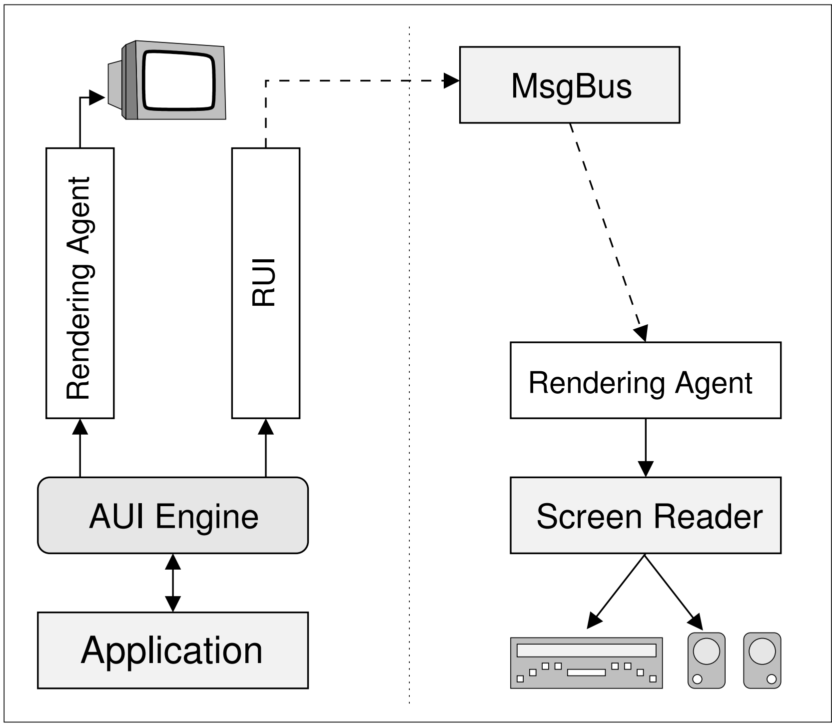


Figure 7.5: Schematic overview of remote rendering agent support

7.5 Conclusions

In this chapter, the experimental implementation was presented. Based on the design of the Parallel User Interface Rendering framework, various aspects of the implementation were given special attention because during the course of the development, several important pitfalls were encountered that are likely to prove common complications when trying to implement this novel approach on existing representation toolkits.

An analysis of the Java AWT/Swing event system was discussed extensively. Based on captured data from various experiments, it was possible to determine that while AWT and Swing generate (and process) a very large amount of events during user interaction, most of these events are actually not relevant to the user interaction semantics. The design of the PUIR framework makes it possible to select only those specific events that can be transformed into semantic events that the AUI engine will handle.

While the description of the experimental implementation is not providing all detail, it is sufficient to show that:

- The separation of perceptual and conceptual functionality in the PUIR framework poses extra complications when implementing a rendering agent because this separation is often not present or incomplete in existing representation toolkits.
- Handling user interaction semantics at the conceptual level ensures that the behaviour of UI elements is consistent across representations.
- Handling user interaction semantics at the conceptual level increases efficiency in the overall system because event handling can be limited to only those events that actually matter in terms of semantics.
- The Parallel User Interface Rendering approach is feasible for providing alternative representations of graphical user interfaces.

Chapter 8

Evaluations

"The truth is rarely pure
and never simple."
(Oscar Wilde, "The Importance of Being Earnest", 1895)

The Parallel User Interface Rendering approach introduced in chapter 5, and further specified in terms of its handling of user interaction in chapter 6 and the implementation in chapter 7 is a novel approach to providing multimodal user interfaces. The framework implements the concepts expressed in the thesis statement introduced in section 1.3, yet the design remains to be validated.

8.1 Introduction

This chapter discusses the evaluation of the framework at two levels: internal and external. The internal validation evaluates the work in terms of the requirements derived from the state of the art as presented in section 4.6.2 and the criteria used to evaluate the related works, introduced in section 4.2. A detailed description of the evaluation and its results are presented in section 8.2.

The external validation involves testing with participants from the target user group; in the case of the experimental implementation this would involve blind users. The goal is to validate the approach in real-life testing scenarios, and to conduct testing aimed at comparing the operation of the PUIR framework against other works. A plan for external testing is presented in section 8.3.

8.2 Internal validation

The internal validation of the Parallel User Interface Rendering consists of two related validation components:

- *An assessment of the design against a well defined set of requirements.* The requirements for the design have been elicited from the analysis of the state of the art in chapter 4. Each of the requirements will be discussed in section 8.2.1.
- *An evaluation of the approach according to a set of criteria.* The state of the art analysis also evaluated each approach based on a set of criteria to allow comparison between the various works. These criteria are presented in section 4.2, and the PUIR framework will be evaluated based on each of these criteria in section 8.2.2.

8.2.1 Assessment of the design against requirements

The requirements discussed in this section have been formulated based on the identified shortcomings in the state of the art (section 4.6.1). The requirements (section 4.6.2) come from two related groups of works: approaches towards GUI accessibility, and multimodal user interface systems. The approach described in the thesis statement (section 1.3) draws from both groups, e.g. concurrency from existing accessibility solutions and specification of the UI at the abstract level from multimodal UI systems.

Multimodal input and output

The Parallel User Interface Rendering approach is based on the notion of concurrent representations of the user interface at the perceptual level. One of the fundamental design principles for PUIR is providing concurrent presentation of the user interface (section 5.2.4). Each presentation is provided by a rendering agent that provides AUI reification within the context of one or more modalities (section 5.3.4).

This requirement has been achieved.

Separation of concerns

The design of the PUIR framework incorporates the separation of concerns requirement in the separation between UI and application logic, which is accomplished by means of abstract UI descriptions because such separation is a technical requirement for abstract UIDLs (section 2.5.1). A second level of separation is attained by implementing the user interaction semantics at the AUI engine level, while delegating the presentation of the UI to rendering agents (section 5.3.3). As such, separation of concerns is provided for with a strict separation between application logic, user interaction semantics, and UI presentation.

This requirement has been achieved.

Equivalent representations

Equivalence between the representations can be broken down as a combination of static and dynamic coherence. The PUIR framework provides presentations of the user interface based on an abstract UI description by means of a runtime reification process (section 5.3.4). Every presentation for a given UI is created based on the same abstract UI description, mapping elements in the AUI model onto a CUI model whereby an abstract widget may be mapped onto one or more concrete widgets, or alternatively multiple abstract widgets may be combined into a single concrete widget. Regardless of the actual mapping, every abstract widget will be represented in the concrete UI, and therefore also in the final UI. This applies to all representations, and therefore static coherence is upheld between them.

Per section 5.3.3, the AUI engine provides the implementation for all user semantics, and therefore dynamic coherence is guaranteed. When user interaction is modality-dependent (e.g. pointer devices in a 2D spatial field), the rendering agent will translate the event into a semantic event that is passed to the AUI engine for processing.

This requirement has been achieved.

User interface design that is independent of any modality

The design of the Parallel User Interface Rendering framework calls for the specification of the user interface in abstract form, expressed in a UIDL (section 5.3). The abstract UI description is by definition modality-independent.

This requirement has been achieved.

Use of abstract user interface descriptions

The design of the Parallel User Interface Rendering framework calls for the specification of the user interface in abstract form, expressed in a UIDL (section 5.3).

This requirement has been achieved.

Runtime reification

In the PUIR framework, a user interface representation is provided by a rendering agent that provides AUI reification within the context of one or more modalities (section 5.3.4). The reification is done at runtime in order to be able to support runtime selection of a rendering agent. The framework provides runtime selection of rendering agents in recognition of the need to support assistive technology solutions that are commonly implemented as standalone programs (section 7.4.4).

This requirement has been achieved.

Concurrent representations

One of the fundamental design principles for the PUIR framework is the provision of concurrent representations in support of closer collaboration between users, especially those from different backgrounds in terms of abilities and/or needs (sections 5.2.4 and 5.2.3).

This requirement has been achieved.

8.2.2 Evaluation of the work against state of the art criteria

The state of the art discussion presented in chapter 4 evaluated major projects in the research field of HCI against a set of criteria presented in section 4.2. It is only fair that the novel Parallel User Interface Rendering framework be evaluated against the same set of criteria, in comparison with the related works. This section provides the analysis of PUIR in function of those criteria.

The comparison of PUIR against four representative related works in terms of the criteria is summarised in Table 8.1.

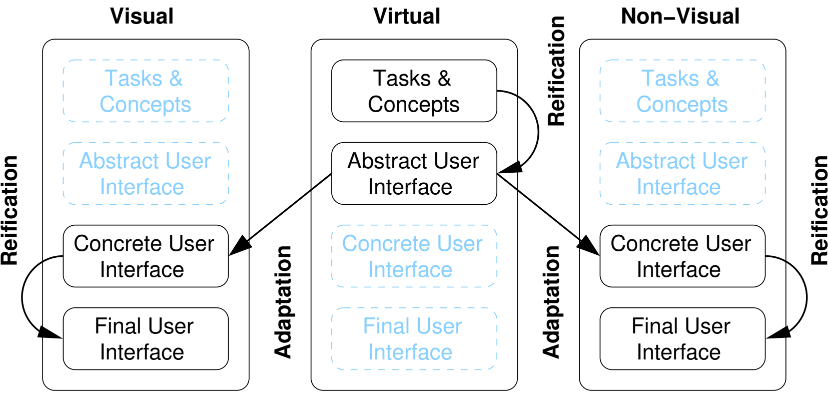


Figure 8.1: URF diagram for PUIR

URF diagram

In the Parallel User Interface Rendering framework, the user interface design yields an abstract UI description that provides the specification of the UI at the abstract UI level. The AUI engine operates at this level, handling the user interaction semantics in a modality-independent manner. The user interface development model used in PUIR follows the Unified Reference Framework diagram in Figure 8.1.

The presentation of the UI is handled by rendering agents that render the UI for one or more specific modalities, using presentation toolkits to present the actual renderings to the user. The process of creating the UI at runtime is an adaptation process from the abstract level to the concrete level in the context of use for the rendering agent. During the adaptation, modality-specific annotation information may be used to assist the rendering agent in presenting the UI. Runtime reification transforms the AUI description into a concrete UI representation, and the actual rendering is presented to the user as a final UI by means of a modality-specific presentation toolkit.

As shown in Figure 8.1, the reification process from AUI to FUI can take place simultaneously in two different contexts of use, or in fact in any arbitrary number of contexts of use. Contrary to other approaches, the PUIR framework keeps operating at multiple levels of abstraction in the URF diagram throughout its life cycle. While the presentation of the UI is handled at the final UI level, along with possible transformation of modality-dependent user interaction events, the processing of user interaction at the semantic level is handled at the abstract UI level.

Static and dynamic coherence

The assessment of the “Equivalent representations” requirement discussed in section 8.2.1 provides the necessary evaluation of the PUIR framework in terms of static and dynamic coherence. PUIR maintains static and dynamic coherence, and in general equivalence between all representations.

Exploration in a non-visual interface

One of the most important requirements for a non-visual interface is the ability to explore the user interface without interference with the application. The PUIR framework provides rendering agents with the ability to request that the AUI engine diverts all user interaction events to the agent (section 5.3.3, *Re-routing user interaction events*). This effectively suspends all user interaction with the application because the AUI engine simply forwards events to the requesting rendering agent. For all intents and purposes, the AUI engine stops processing user interaction until the rendering agent relinquishes control.

This mode is used to implement safe exploration because while the rendering agent can act upon user interaction based on its own interpretation of the events, no interaction with the application is possible. Rendering agents have access to the actual UI representation and can therefore provide accurate information at all times.

A second form of exploration is possible for rendering agents that provide context-specific user interaction, i.e. agents that provide modality-dependent input. As discussed in section 5.3.4 (*User interaction*), rendering agents can implement an exploration mode based on context-specific user interaction, where the user explores the user interface without suspending operations at the AUI engine level. The exploration is still guaranteed to not cause any interaction with the UI.

Conveying semantic information in a non-visual interface

All semantic information and processing takes place at the level of the AUI engine, which is the central source for rendering agents in presenting the user interface. It is therefore guaranteed that rendering agents can convey semantic information.

Interaction in a non-visual interface

User interaction can be modality-independent (e.g. keyboard) or modality-dependent (e.g. pointer device). Modality-independent user interaction events are

transformed into semantic events at the AUI engine level and processed. Modality-dependent events are pre-processed by the rendering agent that controls the corresponding input device, filtering out any events that are not relevant in a semantic context. Those events that are relevant are then transformed into semantic events and passed on to the AUI engine for processing.

Rendering agents can therefore provide modality-specific forms of interaction without the AUI engine needing to be aware. E.g. a rendering agent could implement camera-based hand-gesture detection, translating the gestures into equivalent operations, e.g. arrow key strokes, or for more complex gestures, text input by means of sign language.

The rendering agent can in a similar fashion combine output and input modalities to provide the user with specific modes of interaction. In the context of non-visual representations, an example would be support for haptic devices.

CARE properties for input modalities

All user interaction processing is (eventually) handled at the level of the AUI engine as semantic events, although some filtering and pre-processing may take place at the rendering agent level for modality-dependent input. In this model, “Equivalence” is provided for by virtue of the mapping of user interaction onto a distinct set of semantic events. As a result of the user interaction event processing that takes place for modality-dependent events, and the support for context-specific interaction, the “Assignment” property is also supported.

At the current time, there is no specific support for “Redundancy” and “Complementarity” for input processing.

CARE properties for output modalities

All concurrent representations are equivalent by design, which ensures support for “Equivalence”, and depending on the mode of operation, “Redundancy” is provided for as would be the case when the visual representation is augmented with an auditory representation as an assist.

The very design of the PUIR framework goes against the notion of “Assignment” and “Complementarity” at the output level.

Conceptual model

Because the PUIR framework operates based on a single abstract UI description that is developed as the UI specification, it represents a single conceptual model that is the basis for all representations. The chosen model in the PUIR implementation is the metaphor of the physical office which is also the model for the original GUI design. The target user survey discussed in chapter 3 has shown that blind users tend to understand the conceptual model of the GUI rather well, even if the interaction metaphors are not always appropriate. Since collaboration depends quite heavily on the ability of participants to articulate concepts and interactions on those concepts, it is important that the conceptual models used by participants are reasonable consistent. Recognising the importance of collaboration, the choice was made to implement a conceptual model that everyone was reasonably familiar with.

It is important to note however that the overall design of the Parallel User Interface Rendering framework could be used to implement a system based on other conceptual models. The implementation of the AUI engine defines the actual model that is used, and rendering agents must have matching concrete widgets defined to represent the perceptual level of the UI based on the abstract widgets in the AUI engine.

Concurrency

The assessment of the "Concurrent representations" requirement discussed in section 8.2.1 provides the necessary evaluation of the PUIR framework in terms of whether concurrency is provided for. The PUIR framework is designed to provide concurrent representations.

Cost factors

The Parallel User Interface Rendering approach is based on abstract UI descriptions that are processed at runtime. It provides a user interface management component for applications. As such, in order for an application to be able to benefit from the advantages that PUIR presents, it must be developed specifically with PUIR as its UI component. It is therefore a specialised software component, and carries the implied cost factors associated with specialised toolkits.

The Parallel User Interface Rendering framework will be made available as free software to all interested parties upon completion of the doctoral work presented

in this dissertation, to facilitate contunuing development and in order for the cost factors to be kept at a minimum.

8.3 External validation

Aside from the internal validation of the PUIR framework design, presented in section 8.2, a practical evaluation is important as well. Only through real-world testing can the approach truly be assessed in comparison to other approaches and by itself in terms of functionality and usability.

The external validation comprises three different test scenarios, presented in the following three sections.

8.3.1 Test scenario 1: Basic functionality

The basic functionality test evaluates the functionality of the PUIR framework as an accessibility solution for non-visual operation of a user interface. It is intended to be conducted with test subjects who are totally blind, who have at least 5 years of experience operating a computer system with the help of a screen reader, and who do not have any experience working with the PUIR framework. For the purpose of this test, all test subjects are expected to perform the test on a similar computing environment, be it MS Windows, Linux, MacOS, ... to rule out any influences of the underlying system. The current implementation of PUIR is written in Java, and all systems should have the same version of Java installed.

The test is conducted in two parts:

- Subjects do not have any prior knowledge working with PUIR.
- Subjects have been given 15 minutes to freely explore the test application, during which time they may ask questions to learn more about navigation and operation of UI elements in PUIR. Then repeat the entire test.

Three categories of measures will be collected in this test for both parts separately:

- Quantitative: Time to completion and error count
- Qualitative: Workload, by means of the NASA-Task Load Index (TLX) [62, 61]

		PUIR	Meta-Widgets	HOMER UIMS	Ubiquitous Interactor	GUIB
URF Diagram		T&C T&C T&C AUI AUI AUI CUI CUI CUI FUI FUI FUI	T&C T&C AUI AUI CUI CUI FUI FUI	T&C T&C T&C AUI AUI AUI CUI CUI CUI FUI FUI FUI	T&C T&C AUI AUI CUI CUI FUI FUI	T&C T&C AUI AUI CUI CUI FUI FUI
Coherence	Static	Yes	Partial	No	No	Yes
	Dynamic	Yes	Yes	Yes	Yes	Yes
Exploration		Yes	Yes	Yes	No	Yes
Conveying semantic information		Yes	Yes	No	Partial	Yes
Interaction		Yes	Yes	Yes	Yes	Yes
CARE	Input	AE	E	AE	E	CARE
	Output	RE	E	AE	AE	CAE
Conceptual model		Single	Single	Dual	Single	Single
Concurrency		Yes	No	Yes	No	Yes
Cost factors		Specialised toolkit	Specialised toolkit	Dual design with specialised toolkits, specialised hardware	Specialised design tools	Specialised hardware

Table 8.1: Comparison of PUIR against related works

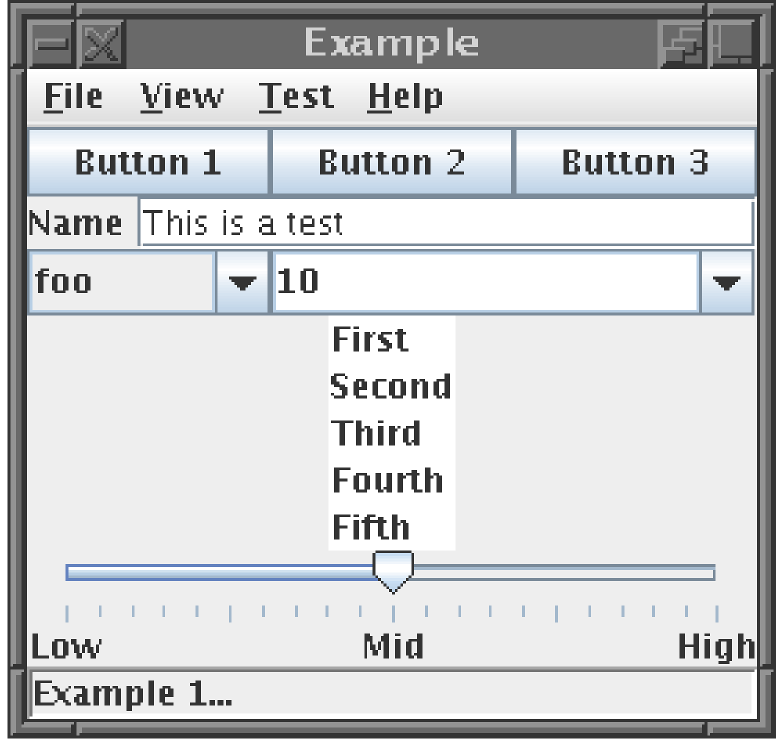


Figure 8.2: User interface of the PUIRDemo application

- Qualitative: Usability, by means of the IBM Post-Study System Usability Questionnaire (PSSUQ) [88]

The test consists of the sample PUIRDemo application included in the PUIR dsitribution. Figure 8.2 shows the user interface presented by this application.

The following tasks are to be performed, in order. The tasks should be given, one at a time, not moving on to the next until the current task has either been completed, or the test subject indicates that he or she is unable to perform the task.

1. Activate “Button 2”
2. Activate “Button 3”
3. Activate “Button 1”
4. Select “20” in edit select list “list2”

5. Write in “45” in edit select list “list2”
6. Type your name in the “name” text field
7. Open the “View” menu from the menu bar
8. Open the “Toolbars” submenu
9. Toggle the “Navigation” menu item
10. Select “Exit” from the “File” menu

8.3.2 Test scenario 2: Basic comparison

The basic comparison test compares the performance of the PUIR framework as an accessibility solution for non-visual operation of a user interface with an established commmercial screen reader. It is intended to be conducted with test subjects who are totally blind, who have at least 5 years of experience operating a computer system with the help of the screen reader used for this comparison. Experience with PUIR is not required, although it is recommended that inexperienced subjects be provided with ample time to learn navigation prior to the testing. For the purpose of this test, all test subjects are expected to perform the test on a similar computing environment, be it MS Windows, Linux, MacOS, ... to rule out any influences of the underlying system. The current implementation of PUIR is written in Java, and all systems should have the same version of Java installed.

The test is conducted with two groups of subjects of equal size:

- Subjects who will perform the test using PUIR
- Subjects who will perform the test using JAWS for Windows

Three categories of measures will be collected in this test:

- Quantitative: Time to completion and error count
- Qualitative: Workload, by means of the NASA-Task Load Index (TLX) [62, 61]
- Qualitative: Usability, by means of the IBM Post-Study System Usability Questionnaire (PSSUQ) [88]

The test consists of the sample PUIRDemo application included in the PUIR dsitribution. Figure 8.2 shows the user interface presented by this application.

The following tasks are to be performed, in order. The tasks should be given, one at a time, not moving on to the next until the current task has either been completed, or the test subject indicates that he or she is unable to perform the task.

1. Open the “Test” menu from the menu bar
2. Select the “6th” menu item
3. Select “apa” from select list “list1”
4. Activate “Button 1”
5. Move “slider” to value 18
6. Activate “Button 3”
7. Open the “View” menu from the menu bar
8. Toggle the “Print view” menu item
9. Open the “File” menu
10. Select “Exit” from the “File” menu

8.3.3 Test scenario 3: Accessibility API

The accessibility API test evaluates the functionality of an application developed using the PUIR framework against an equivalent application developed with a conventional toolkit that provides an accessibility API. It is intended to be conducted with test subjects who are totally blind, who have at least 5 years of experience operating a computer system with the help of a screen reader. Experience with PUIR is not required, although it is recommended that inexperienced subjects be provided with ample time to learn navigation prior to the testing. For the purpose of this test, all test subjects are expected to perform the test on a similar computing environment, be it MS Windows, Linux, MacOS, . . . to rule out any influences of the underlying system. The current implementation of PUIR is written in Java, and all systems should have the same version of Java installed.

The test is conducted with two groups of subjects of equal size:

- Subjects who will perform the test using the PUIRDemo application

- Subjects who will perform the test using the PUIRJavaDemo application and JAWS for Windows as screen reader

Three categories of measures will be collected in this test for both parts separately:

- Quantitative: Time to completion and error count
- Qualitative: Workload, by means of the NASA-Task Load Index (TLX) [62, 61]
- Qualitative: Usability, by means of the IBM Post-Study System Usability Questionnaire (PSSUQ) [88]

The test consists of the sample PUIRDemo and PUIRJavaDemo applications included in the PUIR dsitribution. The PUIRDemo application implements its UI based on the PUIR framework, whereas PUIRJavaDemo implements the equivalent UI using Java Swing[1]. Figure 8.2 shows the user interface presented by this application.

The following tasks are to be performed, in order. The tasks should be given, one at a time, not moving on to the next until the current task has either been completed, or the test subject indicates that he or she is unable to perform the task.

1. Activate “Button 2”
2. Activate “Button 3”
3. Activate “Button 1”
4. Select “20” in edit select list “list2”
5. Write in “45” in edit select list “list2”
6. Type your name in the “name” text field
7. Open the “View” menu from the menu bar
8. Open the “Toolbars” submenu
9. Toggle the “Print view” menu item
10. Select “Exit” from the “File” menu

[1] No special accessibility API calls are made in the PUIRJavaDemo implementation, so any accessibility features observed are provided by the Java classes or the system.

8.4 Conclusions

Validation of the Parallel User Interface Rendering framework is a very important part of this work, and in this chapter both internal and external validation were presented.

The internal validation (section 8.2) comprises two components: assessment against the established requirements, and evaluation according to the state of the art criteria. Analysis of the design presented in this work shows that all requirements were met. The Parallel User Interface Rendering framework provides the functionality proposed in the thesis statement in section 1.3, further motivated in section 1.3.1, specifically:

- Multiple representations
- Parallel presentation
- Functionally equivalent representations

The PUIR framework is able to provide the functionality needed to engage in meaningful collaboration where all participants can observe the user interface with an equivalent representation, and with equivalent user interaction semantics. In comparison with existing approaches to accessibility and multimodal user interfaces, as evaluated based on the criteria used for the state of the art analysis, the PUIR framework is capable of offering a dimension to accessibility that is mostly lacking.

The external validation portion of this chapter provides a plan for conducting practical tests in order to compare the PUIR framework to other approaches, and to assess its overall usability. Given that the external validation could not be performed within the scope of this work, its implementation is left as an important future work.

Chapter 9

Conclusion

"When it comes to the future,
there are three kinds of people:
those who let it happen,
those who make it happen,
and those who wonder what happened."
(John M. Richardson, Jr.)

The main objective of the research presented in this doctoral dissertation was to provide equivalent representations of multimodal user interfaces through parallel rendering. Specifically, the underlying goal was to introduce a novel approach to making GUIs accessible to the blind.

Several contributions were developed throughout the work towards completing the main objective. Section 9.1 evaluates each contribution, and presents the main objective based on the individual contributions. The chapter concludes in section 9.2 with thoughts and insights towards future work.

9.1 Contributions

Underlying all research into GUI accessibility is the very definition of the graphical user interface. The very first objective was to **re-interpret the Graphical User Interface concept** in light of many years of continuing research. Researchers have come to disagree on what constitutes a GUI, and what components should be retained lest one abandon the concept altogether. Especially the "Desktop" metaphor that has long been synonymous with GUI has come under attack

due to accessibility concerns, and some have deemed it inappropriate for blind individuals. The research presented in chapter 2 revives the original design principles behind the GUI concept, and shows that those principles are not only still valid, but there is no basis for deeming them inappropriate for blind individuals. While the original target groups may not have included the blind, the concept has certainly become commonplace. David Smith wrote in personal communication [129]:

> "We realized at the start that it would be impossible (and in any case undesirable) to bring [executives and their staffs] into the computer's world and teach them computer concepts and computer ways of doing things. Instead we would have to bring the computer into their world and teach it office concepts and ways of working. We wanted the users to be able to continue with their familiar methodologies. The computer should augment and facilitate those methodologies, not replace them."

The first objective was achieved through research and analysis, presented in chapter 2.

The second objective was to provide **validation for the "access to GUIs – not graphical screens" argument** phrased by Edwards, Mynatt, and Stockton in their 1994 paper [45]. While the authors did not provide research-based validation for their claim, re-analysis of their position in view of the revived design principles behind the GUI concept and continued research does in fact provide clear support for the insight that the screen image is merely a visual representation of a powerful underlying model, and that accessibility should be focused on developing off screen models based on the UI structure rather than its visualisation.

The second objective was achieved through research and analysis, presented in chapter 2.

The third objective was to provide an updated statement of focus for providing access to GUIs: **provide access to the underlying conceptual user interface, not its visual representation**. By applying established research on the principle of separation of concerns and the various layers of user interface design to the problem of providing access to GUIs for blind individuals, it is possible to further refine the Edwards/Mynatt/Stockton paradigm shift. Research in UI design and development indicates that the accessibility problem should shift focus from the perceptual layer to the conceptual layer.

The third objective was achieved through research and analysis, presented in chapter 2.

The fourth objective was to conduct **a survey to determine familiarity of WIMP-based user interface elements, and the accuracy of mental models for blind**

users. Although an extensive amount of research in past years has been based on the assumption that the underlying metaphor for the original GUI design is not appropriate for blind users, this has mostly been based on opinion rather than analysis of user feedback. The survey presented in chapter 3 augments a study by Kurniawan, et al. concerning the use of mental models [80, 81], and shows that blind users are certainly familiar with commonly used user interface elements, and the metaphors they are based on. The study also shows that the accuracy of the mental model is very dependent upon the information the blind user is able to obtain. Descriptions by sighted peers do not necessarily improve the accuracy of the mental model.

The fourth objective was achieved through analysis of survey results, presented in chapter 3.

The fifth objective was to design a framework for **concurrent representations of the same conceptual model in different modalities**. Based on the earlier objectives and extensive research, a novel approach to provide UI representations was developed. By introducing quite revolutionary approaches to how representations are created, and by shifting the user interaction semantics to the level of the conceptual model rather than a concrete user interface that is dependent upon a specific modality, it is shown that concurrent representations can be provided that are all a reification of the same underlying conceptual model.

The fifth objective was achieved through research and novel design, presented in chapter 5.

The sixth objective aimed to **unify processing of user interaction at a semantic level**. The goal of the objective was to centralise all user interaction processing in order to ensure consistent semantics across all representations. Novel techniques to approach this problem were introduced in chapter 6, and based on important insights concerning separation of concerns, and context-based selection and transformation of user interaction events, unified processing has been shown to be feasible.

The sixth objective was achieved through research and design, presented in chapter 6.

The seventh objective was to develop an **experimental implementation to validate the designs** underlying the PUIR framework. While an implementation of the overall design has been developed, it is possibly too early to conclude that the Parallel User Interface Rendering approach works in general. The experiment has certainly provided invaluable information concerning the complexities surrounding implementing the PUIR framework on top of existing representation toolkits.

The seventh objective was in part achieved through research and development, presented in chapter 7.

The final objective consisted of a **validation of the approach**, by means of both internal and external validation. The evaluation of the Parallel User Interface Rendering framework shows that all requirements are met, and it compares well to the state of the art based on the criteria used in chapter 4. External validation was not performed within the scope of this work, but a test plan comprising three test scenarios has been presented in chapter 8.

The internal validation component of the final objective was achieved through analysis of the work presented in this dissertation as discussed in chapter 8. The external validation plan is presented in chapter 8 as well, although its execution is left as future work.

9.2 Future work

As a first and important future work, the external validation outlined in section 8.3 is to be conducted, and its results analyzed. The real user validation testing is crucial because it will provide important information about the usability and perceived workload impact.

As mentioned in chapter 3, the survey scoring was done solely by the author and while care was taken to avoid introducing bias, there is no objective measurement on the accuracy of the scoring. In order for the survey to be more reliable, and be a more valuable contribution, the scoring should be conducted with multiple scorers (not including the author). The survey results should include the tabulated scores of all scorers, and inter-scorer reliability measurements.

The Parallel User Interface Rendering approach can contribute to the field of accessibility well beyond the immediate goal of providing non-visual access to GUIs [150, 149]. The generic approach behind the design of the PUIR framework lends itself well to developing alternative rendering agents in support of other contexts of use. It is however important to note that the design principle of a consistent conceptual model (see section 5.2.1) implies that all users must be able to understand the concepts that it is based upon.

Especially individuals with cognitive impairments may have difficulties mastering the interaction semantics as they relate to the metaphor of the physical office[1]. Sutcliffe et al. describe the need for quite individual customisation of user interfaces for individuals with cognitive impairments [138], which means that although the underlying conceptual model may remain appropriate, the metaphorical user interface will almost certainly require changes. The PUIR approach can still be used in this case, as a means to provide alternative UI

[1] As discussed in section 2.2.3, the “physical office” metaphor was found to be most appropriate as the underlying concept of the Graphical User Interface.

representations of the customised MUI. Doing so will offer the important benefit of collaboration between groups of users with different abilities (see section 5.2.3).

It should be noted that although the intent of the design implies that a rendering agent provide a truly equivalent representation of the user interface, nothing actually prevents the development of a partial rendering agent. One potential use could be the implementation of a rendering agent that visualises text through computer generated sign language. Alternatively, a rendering agent can be developed to manage the input modality of camera-based runtime translation from sign language to text, which is then posted to the AUI engine as the equivalent of keyboard input.

Another area where the PUIR framework can contribute is automated testing of applications, by providing a way to interact with the application in a programmatical way without any dependency on a specific GUI toolkit [150, 149]. This is a significant advantage because all too often automated testing fails due to cosmetic changes in the GUI or due to graphical toolkit changes[2]. Even small incompatible changes can impact automated GUI testing tools significantly because they tend to depend on details of the actual visual representation. The PUIR framework may alleviate the impact of such changes altogether.

Going forward, the Parallel User Interface Rendering approach is unlikely to progress towards any acceptance from the development community unless its requirements for UI descriptions can be met with a well established UIDL.

An important area of UI design that has not been addressed in this work is the inclusion of temporal relations within the UI. While no effort has been made towards a formal design for this feature, it would likely take the shape of a SMIL-alike specification of temporal relations [164]. Rendering agents could provide the AUI engine with limitations in their rendering capabilities, based on the needs of the users. One might envision a process where limitations from the rendering agents are matched with constraints imposed by the UI.

Parallel user Interface Rendering is a very novel approach to implementing and presenting user interfaces. As it is developed further, new horizons may open.

[2]While it is common practice in software development to perform compatibility testing for components such as graphical toolkits, it is also common for this level of testing to be insufficient.

Appendix A

Target user survey

This appendix provides further details on the target user survey described in Chapter 3. Section A.1 provides the questionnaire for the survey. This is followed by a description of the scoring system in section A.2. The scored results are presented in tables A.2 and A.3.

A.1 Questionnaire

The following list of questions was sent to each prospective respondent, along with a short introductory note explaining that the survey was being conducted in the context of completing a doctorate concerning accessibility of graphical user interfaces. It also explained that the survey is targetted at any totally blind individual. The only instruction provided stated that the objective was to collect brief information on how totally blind users perceive elements of a graphical user interface, and what they mean to them.

Demographics

A. Gender
B. Age
C. Years of experience with computers
D. Job (or equivalent)
E. Blind since (if since birth, write 'birth')
F. Braille, Speech Synthesizer, or Both

Survey

1. Briefly explain what a 'window' in a user interface is, from your perspective?

2. Briefly explain what a ’button’ in a user interface is, from your perspective?
3. Briefly explain the ’desktop’ of a GUI environment, from your perspective?
4. (a) Do you find yourself exploring the overall layout and content of a user interface before you interact with it?
   (b) If so, what kind of things do you look for?
5. (a) Do you find yourself creating a mental image of the user interface as your explore it?
   (b) If so, what does that mental image look like from a high level perspective (not much detail)? What are the main components of the mental image?
6. (a) Do you ever have a sighted person describe the user interface for you?
   (b) If so, what kinds of information do you ask for?
   (c) If so, how well do you feel sighted people describe user interfaces for you?
   (d) What is your most common complaint about people describing user interface to you?
   (e) What stands out with the best descriptions of user interfaces that sighted people have given you?
7. Does it help you to know what user interface elements look like visually, i.e. as a way to make it easier for you to interact with the user interface?
8. (a) Are there any user interface elements in graphical user interfaces that you have found to be difficult to understand in terms of how to interact with them?
   (b) If so, please list the top three difficult elements, and briefly explain why you felt each was difficult?

A.2 Scoring

In order to make analysis of the survey results easier, all survey responses are scored numerically based on the written responses. In some cases, multiple questionnaire items are combined into a single score. The remainder of this section provides a description for each of the items and how they are scored.

A.2.1 User interface elements/concepts: Window, Button, and Desktop

These items are derived directly from questions 1 through 3 in the survey questionnaire. The goal is to determine whether a specific GUI element/concept is considered to be perceptual or conceptual in nature by the respondent. A score is assigned to the response based on its textual content, interpreted in terms of it being predominantly perceptual or conceptual. The following 5-point scoring scale is used:

1. **Very perceptual**: The respondent desccribes the UI element or concept entirely in terms of presentation characteristics such as shape, size, location, layout, . . . The element or concept is described based on its ”Look & Feel”.
2. **Somewhat perceptual**: The respondent describes the UI element or concept mostly in terms of presentation characteristics.
3. **Both perceptual and conceptual**: The description of the UI element or concept is a relatively even mix of presentation characteristics and user interaction semantics.
4. **Somewhat conceptual**: The respondent describes the UI element or concept mostly in terms of the user interaction semantics that are associated with it.
5. **Very conceptual**: The respondent describes the UI element or concept entirely in terms of semantics, i.e. in terms of the functionality associated with the element or concept. The description is not concerned with the outward appearance of the element or concept, but rather with how one interacts with it and/or what it does.

It is important to consider that all participants in this survey are blind, and that blindness in the context of this work implies a total lack of usable vision (see section 1.2.2). It is therefore reasonable to conclude that respondents who describe the UI element or concept with significant perceptual detail are not drawing on information obtained from their own observations of the user interface (except for participants who became blind later in life). Instead, their information typically comes from another source, such as training they may have received, or descriptions provided by sighted peers.

A.2.2 The importance of knowing the UI layout

This item is derived from the responses to questions 4(a), 4(b), 5(b), and 6(b) in the questionnaire. The underlying statement for which agreement or disagreement is scored here is:

> ”The (perceptual) layout of the user interface is important in terms of accessibility and usability.”

The following 5-point scoring scale is used:

1. Strongly disagree
2. Disagree
3. Neither disagree or agree
4. Agree
5. Strongly agree

A.2.3 The significance of a mental model

This item is derived from the responses to questions 5(a), 5(b), 7, 8(a), and 8(b) in the questionnaire. The underlying statement for which agreement or disagreement is scored here is:

> ”My interactions with the user interface are based on a mental model of the UI that I created.”

The following 5-point scoring scale is used:

1. Strongly disagree
2. Disagree
3. Neither disagree or agree
4. Agree
5. Strongly agree

A.2.4 The usefulness of descriptions by a sighted peer

This item is derived from the responses to questions 6(a), 6(b), 6(c), 6(d), and 6(e) in the questionnaire. The underlying statement for which agreement or disagreement is scored is:

> ”A verbal description of the user interface by a sighted person is important to gaining an understanding about how to interact with the UI.”

The following 5-point scoring scale is used:

1. Strongly disagree
2. Disagree
3. Neither disagree or agree
4. Agree
5. Strongly agree

A.2.5 What type of mental model is used?

This item is derived from the responses to all questions in the questionnaire, and is based on the perspective expressed in the responses. Kurniawan, et al. surveyed visually impaired users in view of how they use mental models for interacting in a graphical user environment, and identified three categories of mental models [80, 81]:

- *Structural*: Models in this category are characterised by a strong emphasis on how elements in the UI are arranged, i.e. the layout of the UI. In their study, three out of five participants were identified as using a structural mental model.
- *Functional*: These models are based on sequences of operations and commands, independent from the on-screen configuration of the UI. One participant was identified as using a functional mental model.
- *Hybrid*: Models in this category are based on both structural and functional information. One participant was identified as using a hybrid model.

Respondent	Window	Button	Desktop	Importance of layout	Significance of mental models	Usefulness of description	Mental model type
P1*	3	3	3	3	4	4	S
P2	4	4	4	3	1	4	F
P3*	4	3	3	3	4	2	S
P4*	5	5	5	4	1	1	F
P5	1	5	1	4	4	3	S
P6*	4	4	4	4	4	3	S
P7*	4	4	3	4	4	4	S
P8*	1	1	1	2	3	1	S
P9	4	4	4	1	1	1	F
P10	3	4	3	4	3	2	F
P11	5	5	5	5	5	1	S
P12	3	3	3	4	4	1	H
P13	2	4	3	3	3	2	H
P14	1	5	4	4	4	3	S
P15	4	5	4	4	4	4	S
P16	3	4	4	4	4	2	F
P17	5	5	4	4	4	3	F
P18	5	5	4	4	4	3	S
P19	5	5	5	4	4	3	H
P20	1	4	4	4	4	2	S

Table A.2: Survey results – 1 of 2

*Respondents marked * responded to a direct invitation to participate*

Respondent	Window	Button	Desktop	Importance of layout	Significance of mental models	Usefulness of description	Mental model type
P21*	4	4	4	3	1	1	F
P22*	4	5	4	1	3	1	F
P23	5	5	4	3	4	1	S
P24	5	5	5	3	1	1	F
P25	2	4	3	2	3	1	F
P26	3	5	2	4	1	2	F
P27	5	5	3	5	3	2	S
P28	5	5	4	4	3	2	F

Table A.3: Survey results – 2 of 2

*Respondents marked * responded to a direct invitation to participate*

Appendix B

AUI Widgets

This appendix provides a description of the AUI widgets that are part of the Parallel User Interface Rendering framework as discussed in this dissertation. For production development purposes, additional widgets would likely be needed in support of more complex user interfaces.

The first section describes the container widgets, whereas the second section provides information about regular component widgets.

B.1 Containers

Container widgets are components that contain other components, for the purpose of logical grouping or to augment the functionality of the contained components with some all-encompassing functionality (such as enforcing mutual exclusion on a group of selectable components).

Group

The *Group* widget is the most basic of all containers. It is commonly used as a logical encapsulation of a set of widgets. Grouping of widgets can be used to enforce a specific focus traversal order, because the widgets inside a group are traversed prior to the next sibling of the group, as required by the depth-first traversal algorithm.

User interaction:

- *Container*: Adding or removing components

Menu

The *Menu* widget is a container for menu items and sub-menus. It can contain any combination of regular selectable menu items, toggles, sub-menus, and mutex groups. Also, any part of the content can be encapsulated as a logical group.

A menu belongs either to the menu bar on a window, or to another menu (as a sub-menu). When a menu has not been selected, a labelled placeholder is used to represent the closed (hidden) menu. When a menu is selected, its content is made visible.

User interaction:

- *Container*: Adding or removing menu items or sub-menus.
- *Focus*: Prerequisite for the *Visibility* operations, for user interaction that is not explicitly associated with a specific widget.
- *Visibility*: When the menu is not visible, show a placeholder label; when the menu is visible, display the menu content as well.

Menu bar

The *Menu Bar* widget is an optional component of a window, encapsulating a group of menus. The menu bar and its content are not part of the regular focus traversal chain for the window, and instead the menus form their own chain, allowing for basic first, last, previous and next operations.

The *Menu Bar* widget is essentially a specialised *Mutex* widget, ensuring that only one menu can be selected at any given time.

Only one *Menu Bar* widget can exist per window.

User interaction:

- *Container*: Adding or removing *Menu* widgets.

Mutex

A *Mutex* widget is a special form of the *Group* widget, enforcing mutual exclusion for *Selection* operations on its content widgets, i.e. ensuring that only one component can be selected at any given time.

Only *Toggle* widgets can be added as components in a mutex group.

User interaction:

- *Container*: Adding or removing *Toggle* widgets.

Status bar

The *Status Bar* widget is an optional component of a window, commonly used to communicate status information about the application to the user. In addition, it can also contain non-container widgets. This functionality is sometimes used to present additional information, or to provide convenient access to specific functionality[1] in the application.

Only one *Status Bar* widget can exist per window.

User interaction:

- *Container*: Adding or removing components.
- *ValueChange*: Modifying the status message for the widget. This is not functionality that allows the user to modify the message, but rather for the application to alert the user of the current status.

Tool bar

A *Tool Bar* widget is a logical group of components, that can only be added to a *Window* widget. The content of a *Tool Bar* widget is not included in the regular focus traversal chain for the window. Instead, each *Tool Bar* widget manages its own focus traversal chain.

User interaction:

- *Container*: Adding or removing components.

[1] This is often referred to as ”shortcuts”, ”quicklinks”, or ”quick launch icons”.

Window

A *Window* widget is the root of a self-contained portion of the user interface of an application. It can optionally contain a menu bar, a status bar, and one or more tool bars. More importantly, it must contain a *Group* widget. This group is the container for all regular content in the window.

User interaction:

- *Container*: Adding or removing components. Note that this operation is used for adding all components, including the optional menu bar, optional status bar, optional tool bars, and the mandatory content group.
- *Focus*: Indicate whether the window should be presented with context-free user input. Note that only one window can ever be the recipient for context-free user input at any given time. When a window has focus, there will always be a widget within that window that holds in-window focus. When a window loses focus, the last focused widget is remembered so that when window focus is regained, the same widget will be given focus. This ensures that in-window focus is essentially dormant when the window does not have focus.
- *Visibility*: Creation of the window (becoming visible), and destruction of the window (being removed from view). Regardless of the implementation, the semantics of these operations should be that hiding the window and then showing it again is equivalent to destroying the window, and then recreating it. Any existing state is expected to be lost.

B.2 Components

Components are widgets that allow the user to interact with the application. They provider semantic operations and information. The functionality for the widgets is implemented at the AUI engine level, to ensure coherence across all representations.

Button

The *Button* widget is one of the most basic AUI elements. It provides the user with a control that can be activated as a trigger. Unlike the *Toggle* widget, it does not retain state. Instead, it resets immediately after being activated, ready to be

activated once more. A representative real world example can be found in the switch of a doorbell.

User interaction:

- *Focus*: Indicate whether user input that is not explicitly associated with a specific widget should be directed to this widget.
- *Action*: This operation triggers the functionality that is associated with this button.

Edit select list

An *Edit Select List* widget enables the user to select a single value from a provided list of options, or to write in their own value. As a result of the combined dual function provided by this widget, specific operational semantics apply. If, after a value has been written in, the user selects any of the predefined values in the associated list, the written in value will be replaced with the newly selected value.

It is valid for the default written in value to be different from any of the predefined values in the associated list.

Conceptually, the *Edit Select List* widget is a text entry field that offers the option to select a predefined value from a list. The options list is envisioned to not be presented to the user unless requested (using a *Visibility* operation), because it has no use unless the user is selecting a value. For more information about the operation of the text field portion, refer to the *Text Field* widget below.

The *Edit Select List* widget is a derivative of the *Text Field* widget.

User interaction:

- *Focus*: Indicate whether user input that is not explicitly associated with a specific widget should be directed to this widget.
- *Selection*: Select a specific item from a predefined list of items.
- *TextCaret*: Indicate the position in the text that the user input is focusing on, i.e. set the cursor position. This is the position where further user interaction will be based on.
- *TextSelection*: Select part of the text for further operation(s), and use it as a form of restricted focus. A text selection can be an entire line of text, a specific word, or text between two specific character locations.

- *ValueChange*: Any changes to the text constitute a value change operation.
- *Action*: This operation finalises the value of the widget, either as an item selected from the list of options, or as a written in value, and triggers the functionality that is associated with this widget.
- *Visibility*: The options list is hidden until it is needed. The visibility of the options list is handled by this operation.

Menu item

The *Menu Item* widget is equivalent to the *Button* widget, aside from the fact that the former widget can only occur in menus, whereas the latter is a content widget.

Menu items in a menu can be envisioned as a doorbell panel at an apartment complex, where one selects a specific doorbell.

User interaction:

- *Focus*: Indicate whether user input that is not explicitly associated with a specific widget should be directed to this widget.
- *Action*: This operation triggers the functionality that is associated with this menu item.

Multi select list

A *Multi Select List* widget enables the user to select any number of values from a predefined list of options. Through a series of manipulations, the user defines a (possible non-contiguous) subset of the available values, and when completed, an *Action* operation finalises the selection.

User interaction:

- *Focus*: Indicate whether user input that is not explicitly associated with a specific widget should be directed to this widget.
- *SetSelection*: Define a possibly non-contiguous subset of values across the predefined list of options.
- *Action*: This operation finalises the value set of the widget, and triggers the functionality that is associated with this widget.

Single select list

An *Single Select List* widget enables the user to select a single value from a provided list of options.

Conceptually, the *Single Select List* widget is a button with a specific default value as label text, chosen from the predefined list of values. The options list is envisioned not to be presented to the user unless requested (using a *Visibility* operation), because it has no use unless the user is selecting a value.

The *Single Select List* widget is a derivative of the *Edit Select List* widget.

User interaction:

- *Focus*: Indicate whether user input that is not explicitly associated with a specific widget should be directed to this widget.
- *Selection*: Select a specific item from a predefined list of items.
- *Action*: This operation finalises the item selection from the list of options, and triggers the functionality that is associated with this widget.
- *Visibility*: The options list is hidden until it is needed. The visibility of the options list is handled by this operation.

Text

The *Text* widget is arguably not strictly a user interaction element because it's purpose is to display a predefined text without any user interaction taking place. However, the content of the text can be updated by the application or authorised entities. It is therefore prudent to provide support for alerting the user that the value of the widget has changed.

User interaction:

- *ValueChange*: Indicate that the value of the widget has changed.

Text field

The *Text Field* widget provides the user with a text entry field that provides text editing features for the user's convenience. Text input operates with a more localised notion of focus, known as the cursor position. This is the focal point

of the user's attention while interacting with the text in this widget. It indicates where modifications and additions to the text will take place.

Aside from adding characters to and removing characters from the text entry field, the widget also provides the ability to manipulate arbitrary contiguous subsection of the text. This is done by first selecting the character range that will be involved in the text editing operation, and then performing the text editing action on that character range.

User interaction:

- *Focus*: Indicate whether user input that is not explicitly associated with a specific widget should be directed to this widget.
- *TextCaret*: Indicate the position in the text that the user input is focusing on, i.e. set the cursor position. This is the position where further user interaction will be based on.
- *TextSelection*: Select part of the text for further operation(s), and use it as a form of restricted focus. A text selection can be an entire line of text, a specific word, or text between two specific character locations.
- *ValueChange*: Any changes to the text constitute a value change operation.
- *Action*: This operation finalises the value of the text field, and triggers the functionality that is associated with this widget.

Toggle

The *Toggle* widget is one of the most basic AUI elements. It provides the user with a control that alternates between two states with every activation. It is equivalent to on/off controls on appliances, e.g. a speaker on/off selector on an amplifier.

User interaction:

- *Focus*: Indicate whether user input that is not explicitly associated with a specific widget should be directed to this widget.
- *Action*: This operation triggers toggling the state of the widget between enabled and disabled.

Valuator

The *Valuator* widget enables the user to select any integer numeric value within a predefined (min, max) interval. The value range may be in either ascending or

descending order, and support is provided for stepping through the value range either using single steps, or by means of 10% steps.

User interaction:

- *Focus*: Indicate whether user input that is not explicitly associated with a specific widget should be directed to this widget.
- *ValueChange*: Indicate that the value of the widget has changed.
- *Action*: This operation finalises the value of the valuator, and triggers the functionality that is associated with this widget.

Appendix C

PUIR UI Description Language DTD

This appendix contains the Document Type Definition (DTD) for the PUIR UI Description Language (PUDL). It defines the structure and syntax of the UIDL that is used to provide UI descriptions in the Parallel User Interface Rendering framework.

```
<!--
   | File:    pudl.dtd
   | Author:  Kris Van Hees <kris@alchar.org>
   |
   | $Id: pudl.dtd,v 1.2 2012/01/16 05:18:21 aedil Exp $
   |
   | This document provides the Document Type Definition
   | (DTD) for UI descriptions in the PUIR framework.
   |
   | A PUIR UI description is an XML document, and it must
   | therefore begin with a XML declaration:
   |      <?xml version="1.0" encoding="UTF=8" ?>
   | This is to be followed by a document type declaration:
   |      <!DOCTYPE PUIR SYSTEM
   |           "http://www.alchar.org/puir/dl/pudl.dtd">
  -->
<!--
   | The 'named' entity defines attributes for elements
   | that are addressable by a specific 'id' and that have
   | a display 'name' associated with them.
```

```
  -->
<!ENTITY % named
     "id CDATA #REQUIRED
     name CDATA #IMPLIED"
>

<!--
   | The 'PUIR' element encapsulates the entire UI for the
   | application.  It must contain one or more window
   | widgets.
  -->
<!ELEMENT PUIR (window+) >

<!--
   | The 'window' widget encapsulates a specific self-
   | contained portion of the application UI.  It is the
   | root of a UI object hierarchy.
   |
   | A window can contain an optional menu bar, zero or
   | more toolbars, a group of content widgets, and an
   | optional status bar.
   |
   | The 'window' widget is a named widget.  It also has
   | an optional attribute 'visible' to indicate whether
   | the window should be presented to the user (default:
   | true).
  -->
<!ELEMENT window (menuBar?, toolBar*, group, statusBar?,
                  agentInfo*) >
<!ATTLIST window
     %named;
     visible (true|false) "true"
>

<!--
   | The 'menuBar' widget encapsulates a collection of one
   | or more menus.
  -->
<!ELEMENT menuBar (menu+, agentInfo*) >

<!--
   | The 'menu' widget groups functionality together under
   | a single selectable item.  Its content is made visible
   | when the menu is selected.
```

```
    |
    | The widget contains one or more of:
    |      menu       - sub-menu
    |      menuGroup  - logical grouping of menu items
    |      item       - regular selectable menu item
    |      toggle     - switch-style control (on/off)
    |      mutex      - mutual exclusive grouping of toggles
    |
    | The 'window' widget is a named widget.  It also has
    | an optional attribute 'key' to declare an optional
    | key-stroke that can be used to activate the menu
    | (for keyboard navigation).
  -->
<!ELEMENT menu ((menu | menuGroup | item | toggle |
                 mutex)+, agentInfo*) >
<!ATTLIST menu
    %named;
    key CDATA #IMPLIED
>

<!--
    | The 'menuGroup' widget groups functionality together
    | within a menu or an enclosing group.  The content is
    | typically visible if the enclosing menu is visible.
    |
    | The widget contains one or more of:
    |      menu       - sub-menu
    |      menuGroup  - logical grouping of menu items
    |      item       - regular selectable menu item
    |      toggle     - switch-style control (on/off)
    |      mutex      - mutual exclusive grouping of toggles
  -->
<!ELEMENT menuGroup ((menu | menuGroup | item | toggle |
                      mutex)+, agentInfo*) >

<!--
    | The 'item' widget defines a menu item that can be
    | selected to activate the associated functionality.
    |
    | The 'item' widget is a named widget.  It also has an
    | optional attribute 'key' to declare an optional
    | key-stroke that can be used to activate the menu
    | item (for keyboard navigation).
  -->
```

```
<!ELEMENT item (agentInfo*) >
<!ATTLIST item
    %named;
    key CDATA #IMPLIED
>

<!--
   | The 'toggle' widget defines a UI element that holds
   | a boolean state that can be toggled by selecting
   | the widget.
   |
   | The 'toggle' widget is a named widget.  It also has
   | an optional attribute 'key' to declare an optional
   | key-stroke that can be used to toggle the state of
   | the UI element (for keyboard navigation).
  -->
<!ELEMENT toggle (agentInfo*) >
<!ATTLIST toggle
    %named;
    key CDATA #IMPLIED
>

<!--
   | The 'mutex' widget encapsulates toggles together
   | within a grouping that enforces mutual exclusion.
   | Only one of the enclosed one or more toggles can be
   | set at any given time.
   |
   | The widget has an optional attribute 'orientation'
   | that defines whether the grouping is considered to
   | have a horizontal ordering or a vertical ordering.
   | A horizontal grouping is considered to be more tight
   | than a vertical grouping.
  -->
<!ELEMENT mutex (toggle+, agentInfo*) >
<!ATTLIST mutex
    orientation (horizontal|vertical) #IMPLIED
>

<!--
   | The 'toolbar' widget is yet to be implemented in
   | full.
  -->
<!ELEMENT toolBar (agentInfo*) >
```

```
<!--
   | The 'group' widget provides a container for widgets
   | that can be grouped together.  Logical groups can be
   | used to enforce a specific focus traversal order in a
   | UI.
   |
   | The widget contains one or more of:
   |     button       - element that can be activated
   |     singleSelectList
   |                  - single-select list of values
   |     editSelectList
   |                  - single-select list of value, with
   |                    option to write in a value
   |     group        - logical grouping of widgets
   |     maskedTextField
   |                  - text entry field (hiding input)
   |     multiSelectList
   |                  - multi-select list of values
   |     mutex        - mutual exclusive grouping of toggles
   |     slider       - ranged selector
   |     text         - display text
   |     textField    - text entry field
   |     toggle       - switch-style control (on/off)
  -->
<!ELEMENT group ((button | singleSelectList |
                  editSelectList | group | list |
                  maskedTextField | mutex | slider |
                  text | textField | toggle)+,
                  agentInfo*) >

<!--
   | The 'button' widget represents a UI element that can
   | be activated.
   |
   | The widget is a named widget.
  -->
<!ELEMENT button (agentInfo*) >
<!ATTLIST button
    %named;
>

<!--
   | The 'singleSelectList' widget encapsulates a list of
```

```
    | values from which exactly one can be chosen.
    |
    | The widget is a named widget.
  -->
<!ELEMENT singleSelectList (value+, agentInfo*) >
<!ATTLIST singleSelectList
     %named;
>

<!--
    | The 'value' element provides a data item for widgets
    | that provide a list of items.  The content is a string
    | of characters.
    |
    | The widget allows for an optional 'selected' boolean
    | attribute (default: false).
  -->
<!ELEMENT value (#PCDATA) >
<!ATTLIST value
     selected (true|false) "false"
>

<!--
    | The 'editSelectList' widget encapsulates a list of
    | values from which one can be chosen, or alternatively
    | a value can be entered as if the widget were a text
    | entry field.
    |
    | The widget is a named widget.
  -->
<!ELEMENT editSelectList (value+, agentInfo*) >
<!ATTLIST editSelectList
     %named;
>

<!--
    | The 'multiSelectList' widget encapsulates a list of
    | values from which one or more items can be chosen.
    |
    | The widget is a named widget.
  -->
<!ELEMENT multiSelectList (value+, agentInfo*) >
<!ATTLIST multiSelectList
     %named;
```

```
>

<!--
    | The 'maskedTextField' widget provides a text entry
    | field of a given length in which entered characters
    | are hidden (each character appears as '*').
    |
    | The widget is a named widget.  It also provides for an
    | optional 'text' attribute that holds the initial text
    | for the entry field, and a required 'size' attribute
    | that indicates the amount of characters the field can
    | hold.
  -->
<!ELEMENT maskedTextField (agentInfo*) >
<!ATTLIST maskedTextField
      %named;
      text CDATA ""
      size CDATA #REQUIRED
>

<!--
    | The 'slider' widget provides a control to select a
    | discrete numeric value from a specific [min, max]
    | range.  Optional labels can be specified for this
    | widget.
    |
    | This widget is a named widget.  It also requires 'min'
    | and 'max' attributes to define the range of legal
    | values.  Optional attributes 'majorInterval' and
    | 'minorInterval' can be specified to indicate at which
    | intervals major or minor ticks are to be defined.  The
    | 'stdLabels' boolean can be specified to indicate that
    | in the absence of explicitly defined labels, standard
    | labels should be generated for this widget.
  -->
<!ELEMENT slider (label*, agentInfo*) >
<!ATTLIST slider
      %named;
      min CDATA #REQUIRED
      max CDATA #REQUIRED
      majorInterval CDATA #IMPLIED
      minorInterval CDATA #IMPLIED
      stdLabels (true|false) "true"
>
```

```
<!--
   | The 'label' element is used to associate a textual
   | label with a specific value, for use in a slider
   | widget.
   |
   | The widget requires a 'value' attribute and a 'name'
   | attribute.
  -->
<!ELEMENT label (agentInfo*) >
<!ATTLIST label
    value CDATA #REQUIRED
    name CDATA #REQUIRED
>

<!--
   | The 'text' widget is used to display text in a window.
   |
   | The widget requires a 'text' attribute to specify the
   | text to be displayed. An optional 'id' attribute can
   | be specified to allow the widget to be addressable
   | (e.g. to change the text). An optional 'size'
   | attribute can be specified to reserve space for later
   | updates to the text.
  -->
<!ELEMENT text (agentInfo*) >
<!ATTLIST text
    id CDATA #IMPLIED
    text CDATA #REQUIRED
    size CDATA #IMPLIED
>

<!--
   | The 'textField' widget provides a text entry field of
   | a given length.
   |
   | The widget is a named widget. It also provides for an
   | optional 'text' attribute that holds the initial text
   | for the entry field, and a required 'size' attribute
   | that indicates the amount of characters the field can
   | hold.
  -->
<!ELEMENT textField (agentInfo*) >
<!ATTLIST textField
```

```
    %named;
    text CDATA ""
    size CDATA #REQUIRED
>

<!--
   | The 'statusBar' widget is used to communicate a status
   | to the user.  It is a specialised display text widget,
   | and (if used) it is the last focusable widget in a
   | window.
   |
   | The widget requires a 'message' attribute that holds
   | the text to be displayed.
  -->
<!ELEMENT statusBar (agentInfo*) >
<!ATTLIST statusBar
    message CDATA #REQUIRED
>

<!--
   | The 'agentInfo' element encapsulates render agent
   | specific information for the enclosing widget.
   |
   | The element has a required 'agent' attribute to state
   | what rendering agent the contained information relates
   | to.
  -->
<!ELEMENT agentInfo (agentInfoPair+) >
<!ATTLIST agentInfo
    agent CDATA #REQUIRED
>

<!--
   | The 'agentInfoPair' element provides a key/value pair
   | that encodes some arbitrary piece of information for
   | a rendering agent.
   |
   | The element has a required 'key' attribute that can be
   | used to identify the data item.  An optional 'val'
   | attribute can be provided to state a value for the
   | data item.  In the absence of a value, the element can
   | be considered a boolean data item (present means
   | true).
  -->
```

```
<!ELEMENT agentInfoPair EMPTY >
<!ATTLIST agentInfoPair
    key CDATA #REQUIRED
    val CDATA #IMPLIED
>
```

Appendix D

Example: Programmatic UI creation

This appendix provides the source code for a programmatic implementation of the sample UI discussed in chapter 7. The implementation provided here uses Java AWT and Swing. The visual representation is shown in Figure D.1.

```
import java.awt.*;
import javax.swing.*;
import java.util.Hashtable;

/**
 * Example of a programmatically defined user interface
 * using AWT and Swing.  This class creates a UI that is
 * equivalent to the UI created using the PUIR framework
 * based on the AUI description in Appendix E.
 */
class Example
  extends JFrame {
    /**
     * Create the "File" menu.  This menu contains items
     * for "New", "Open", and "Close" operations, then a
     * sub-menu with the three more recently used files,
     * and then finally an item for the "Exit" operation.
     */
    private JMenu mkFileMenu() {
        JMenu                         menu;
        JMenu                         sub;
```

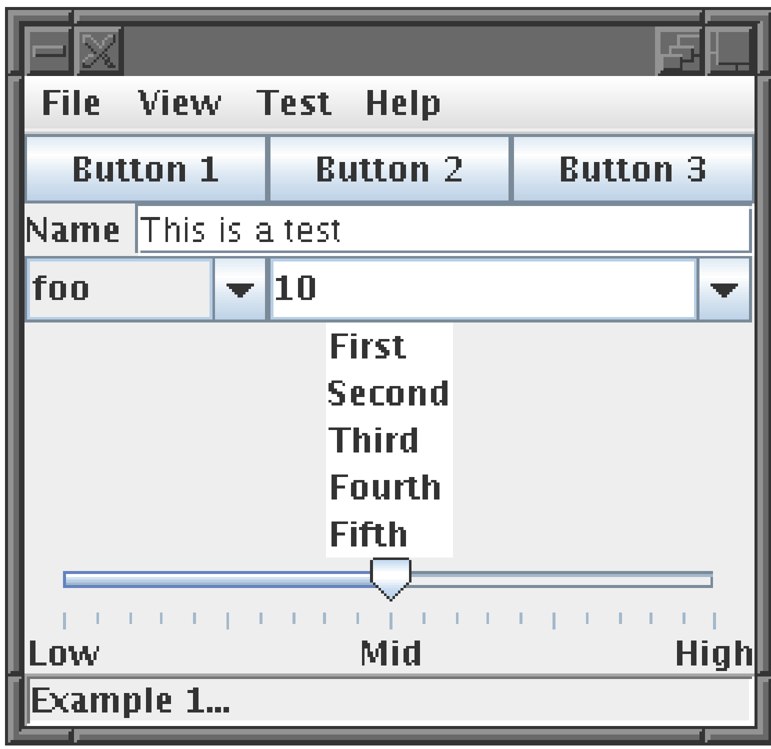


Figure D.1: GUI representation of the example UI using Java/Swing

```
        sub = new JMenu("Recently used file");
        sub.add(new JMenuItem("File1"));
        sub.add(new JMenuItem("File2"));
        sub.add(new JMenuItem("File3"));

        menu = new JMenu("File");
        menu.add(new JMenuItem("New"));
        menu.add(new JMenuItem("Open"));
        menu.add(new JMenuItem("Close"));
        menu.add(sub);
        menu.add(new JMenuItem("Exit"));

        return menu;
    }

    /**
```

```
 * Create the "View" menu.  This menu provides a demo
 * of checkbox menu items, radio button menu items,
 * and a few regular menu items.
 */
private JMenu mkViewMenu() {
    JMenu                   menu;
    JMenu                   sub;
    JRadioButtonMenuItem    rb;
    ButtonGroup             bg;

    sub = new JMenu("Toolbars");
    sub.add(
        new JCheckBoxMenuItem("Navigation Toolbar"));
    sub.add(
        new JCheckBoxMenuItem("Bookmarks Toolbar"));
    sub.add(
        new JMenuItem("Customise..."));

    menu = new JMenu("View");
    menu.add(sub);

    bg = new ButtonGroup();
    bg.add(menu.add(
            new JRadioButtonMenuItem("Full page")));
    bg.add(menu.add(
            new JRadioButtonMenuItem("Print view")));
    bg.add(menu.add(
            new JRadioButtonMenuItem("Fit window")));

    menu.add(new JMenuItem("Zoom"));

    return menu;
}

/**
 * Create the "Test" menu.  This menu is merely an
 * equivalent of the menu with the same name in the
 * PUIR version of the UI, which serves as a test for
 * groupings in menus.
 */
private JMenu mkTestMenu() {
    JMenu                   menu;

    menu = new JMenu("Test");
```

```
        menu.add(new JMenuItem("1st"));
        menu.add(new JMenuItem("2nd"));
        menu.addSeparator();
        menu.add(new JMenuItem("3rd"));
        menu.add(new JMenuItem("4th"));
        menu.addSeparator();
        menu.add(new JMenuItem("5th"));
        menu.addSeparator();
        menu.add(new JMenuItem("6th"));
        menu.addSeparator();
        menu.add(new JMenuItem("7th"));
        menu.add(new JMenuItem("8th"));
        menu.addSeparator();
        menu.add(new JMenuItem("9th"));

        return menu;
    }

    /**
     * Create the "Help" menu.  This is a basic menu
     * with three regular menu items.
     */
    private JMenu mkHelpMenu() {
        JMenu                        menu;

        menu = new JMenu("Help");
        menu.add(new JMenuItem("Help Contents"));
        menu.add(new JMenuItem("Release Notes"));
        menu.add(new JMenuItem("About PUIR"));

        return menu;
    }

    /**
     * Create the menu bar for the single-window UI.  It
     * contains four menus.
     */
    private JMenuBar mkMenuBar() {
        JMenuBar                     menuBar;

        menuBar = new JMenuBar();
        menuBar.add(mkFileMenu());
        menuBar.add(mkViewMenu());
        menuBar.add(mkTestMenu());
```

```
        menuBar.add(mkHelpMenu());

        return menuBar;
    }

    public Example() {
        super();

        setJMenuBar(mkMenuBar());

        // Overall vertical group
        Box                        vBox

        vBox = new Box(BoxLayout.Y_AXIS);

        // Top horizontal group (3 buttons)
        Box                        hBox;
        JButton                    b;

        hBox = new Box(BoxLayout.X_AXIS);
        b = new JButton("Button 1");
        b.setAlignmentX(Component.CENTER_ALIGNMENT);
        hBox.add(b);
        b = new JButton("Button 2");
        b.setAlignmentX(Component.CENTER_ALIGNMENT);
        hBox.add(b);
        b = new JButton("Button 3");
        b.setAlignmentX(Component.CENTER_ALIGNMENT);
        hBox.add(b);

        vBox.add(hBox);

        // Text field (20 characters):
        // horizontal group (label + field)
        JTextField                 tf;
        JLabel                     lbl;

        tf = new JTextField("This is a test", 20);
        lbl = new JLabel("Name");

        hBox = new Box(BoxLayout.X_AXIS);
        lbl.setLabelFor(tf);
        hBox.add(lbl);
        hBox.add(hBox.createHorizontalStrut(5));
```

```
hBox.add(tf);

vBox.add(hBox);

// Horizontal group (single select list and edit
// select list)
JComboBox                cb;

hBox = new Box(BoxLayout.X_AXIS);
cb = new JComboBox(
        new String[] { "foo", "bar",
                       "baz", "apa" });
cb.setEditable(false);
hBox.add(cb);
cb = new JComboBox(
        new Integer[] { 10, 20, 30, 40, 50 });
cb.setEditable(true);
hBox.add(cb);

vBox.add(hBox);

// Multi select list
JList                    lst;

lst = new JList(new String[]
                { "First", "Second", "Third",
                  "Fourth", "Fifth" });

vBox.add(lst);

// Valuator
JSlider                  sld;
Hashtable                tbl;

tbl = new Hashtable();

sld = new JSlider(0, 20);
sld.setMajorTickSpacing(5);
sld.setMinorTickSpacing(1);
sld.setPaintTicks(true);
tbl.put(0, new JLabel("Low"));
tbl.put(10, new JLabel("Mid"));
tbl.put(20, new JLabel("High"));
sld.setLabelTable(tbl);
```

```
        sld.setPaintLabels(true);

        vBox.add(sld);

        add(vBox);

        // Status bar
        JPanel                    pnl;

        pnl = new JPanel(new BorderLayout());
        lbl = new JLabel("Example 1...");
        lbl.setAlignmentX(JLabel.LEFT_ALIGNMENT);
        lbl.setBorder(
                BorderFactory.createLoweredBevelBorder());
        pnl.add(lbl, "Center");

        add(pnl, "South");

        pack();
        setVisible(true);

        CaptureSwingEvents.addWindow(this);
    }

    public static void main(String[] argv) {
        Toolkit.getDefaultToolkit().getSystemEventQueue()
                .push(new CaptureEventQueue());

        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new Example();
            }
        });
    }
};
```

Appendix E

Example: AUI description for the PUIR framework

This appendix provides the PUDL description for the sample UI discussed in chapter 7. The visual representation of this UI description, rendered using the ”Swing” rendering agent is shown in Figure E.1.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE PUIR SYSTEM
          "http://www.alchar.org/puir/dl/pudl.dtd">
<PUIR>
  <window id="example" visible="true">
    <menuBar>
      <menu id="File" key="f">
        <item id="New" key="n" />
        <item id="Open" key="o" />
        <item id="Close" key="c" />
        <menu id="Recent" name="Recently used files"
              key="r">
          <item id="File1" />
          <item id="File2" />
          <item id="File3" />
        </menu>
        <item id="Exit" key="x" />
      </menu>
      <menu id="View" key="v">
        <menu id="Toolbars" key="t">
          <toggle id="Navigation" name="Navigation Toolbar"
```

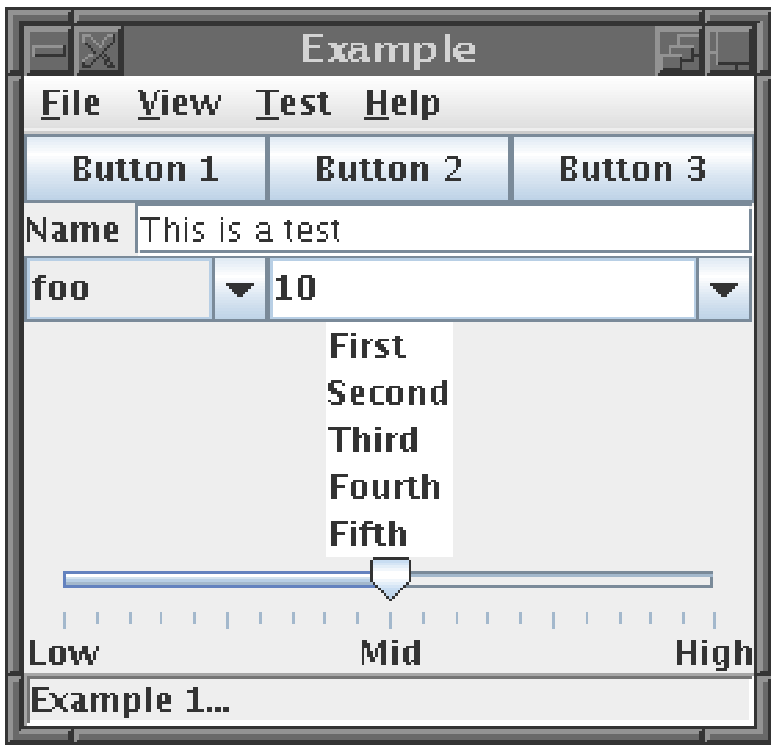


Figure E.1: Visual representation of the example UI using PUIR

```
            key="n" />
        <toggle id="Bookmarks" name="Bookmarks Toolbar"
            key="b" />
        <item id="Customise" name="Customise..."
            key="c" />
    </menu>
    <mutex>
        <toggle id="Full" name="Full page"
            key="f" />
        <toggle id="Print" name="Print view"
            key="p" />
        <toggle id="Window" name="Fit window"
            key = "w" />
    </mutex>
    <item id="Zoom" name="Zoom..." key="z" />
</menu>
<menu id="Test" key="t">
```

```
    <item id="1st" />
    <item id="2nd" />
    <group>
      <group>
        <group>
          <item id="3rd" />
          <item id="4th" />
        </group>
        <item id="5th" />
      </group>
      <group>
        <item id="6th" />
        <group>
          <item id="7th" />
          <item id="8th" />
        </group>
      </group>
    </group>
    <item id="9th" />
  </menu>
  <menu id="Help" key="h">
    <item id="Contents" name="Help Contents"
          key="c" />
    <item id="Notes" name="Release Notes"
          key="n" />
    <item id="About" name="About PUIR"
          key="a" />
  </menu>
</menuBar>
<group orientation="vertical">
  <group orientation="horizontal">
    <button id="button1" name="Button 1" />
    <button id="button2" name="Button 2" />
    <button id="button3" name="Button 3" />
  </group>
  <textField id="field1" name="Name"
             text="This is a test" size="20" />
  <group orientation="horizontal">
    <singleSelectList id="list1">
      <value>foo</value>
      <value>bar</value>
      <value>baz</value>
      <value>apa</value>
    </singleSelectList>
```

```
        <editSelectList id="list2">
          <value>10</value>
          <value>20</value>
          <value>30</value>
          <value>40</value>
          <value>50</value>
        </editSelectList>
      </group>
      <multiSelectList id="list3">
        <value>First</value>
        <value>Second</value>
        <value>Third</value>
        <value>Fourth</value>
        <value>Fifth</value>
      </multiSelectList>
      <valuator id="slider" min="0" max="20"
              majorInterval="5" minorInterval="1"
              standardLabels="true">
        <label value="0" name="Low" />
        <label value="10" name="Mid" />
        <label value="20" name="High" />
      </valuator>
    </group>
    <statusBar message="Example 1..." attr="value" />
  </window>
</PUIR>
```

Bibliography

[1] Mir Farooq Ali. *A transformation-based approach to building multi-platform user interfaces using a task model and the user interface markup language*. PhD thesis, Virginia Polytechnic Institute and State University, 2004.

[2] Gary B. Anderson and David W. Rogers. An inexpensive braille terminal device. *Commun. ACM*, 11(6):417–418, June 1968.

[3] John Robert Anderson. *Cognitive psychology and its implications*. W. H. Freeman and Company, 1980.

[4] Rudolf Arnheim. *Visual Thinking*. University of California Press, 1971.

[5] A. Awde, M.D. Hina, C. Tadj, A. Ramdane-Cherif, and Y. Bellik. Information access in a multimodal multimedia computing system for mobile visually-impaired users. In *Industrial Electronics, 2006 IEEE International Symposium on*, volume 4, pages 2834–2839, July 2006.

[6] Kitch Barnicle. Usability testing with screen reading technology in a Windows environment. In *Proceedings of the 2000 Conference on Universal Usability*, CUU '00, pages 102–109. ACM, 2000.

[7] Eric Bergman and Earl Johnson. Toward accessible human-computer interaction. In Jakob Nielsen, editor, *Advances in human-computer interaction (vol. 5)*, pages 87–113. Ablex Publishing Corp., 1995.

[8] William L. Bewley, Teresa L. Roberts, David Schroit, and William L. Verplank. Human factors testing in the design of Xerox's 8010 "Star" office workstation. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, CHI '83, pages 72–77. ACM, 1983.

[9] Judith Bishop and Nigel Horspool. Developing principles of GUI programming using views. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, SIGCSE '04, pages 373–377. ACM, 2004.

[10] M.M. Blattner, E.P. Glinert, J.A. Jorge, and G.R. Ormsby. Metawidgets: towards a theory of multimodal interface design. In *Computer Software and Applications Conference, 1992. COMPSAC '92. Proceedings., Sixteenth Annual International*, pages 115–120. IEEE Computer Society Press, September 1992.

[11] Paul Blenkhorn and Gareth Evans. Architecture and requirements for a Windows screen reader. In *IEE Seminar on Speech and Language Processing for Disabled and Elderly People (Ref. No. 2000/025)*, pages 1/1–1/4, 2000.

[12] François Bodart and Jean M. Vanderdonckt. Widget standardisation through abstract interaction objects. In *Advances in Applied Ergonomics*, pages 300–305. USA Publishing, 1996.

[13] Matthew N. Bonner, Jeremy T. Brudvik, Gregory D. Abowd, and W. Keith Edwards. No-look notes: Accessible eyes-free multi-touch text entry. In Patrik Floréen, Antonio Krüger, and Mirjana Spasojevic, editors, *Pervasive Computing*, volume 6030 of *Lecture Notes in Computer Science*, pages 409–426. Springer Berlin / Heidelberg, 2010.

[14] Laurent Bouillon, Jean Vanderdonckt, and Kwok Chieu Chow. Flexible re-engineering of web sites. In *Proceedings of the 9th international conference on Intelligent user interfaces*, IUI '04, pages 132–139. ACM, 2004.

[15] Amina Bouraoui and Mejdi Soufi. Improving computer access for blind users. In Khaled Elleithy, editor, *Advances and Innovations in Systems, Computing Sciences and Software Engineering*, pages 29–34. Springer Netherlands, 2007.

[16] Lawrence H. Boyd, Wesley L. Boyd, and Gregg C. Vanderheiden. The graphical user interface crisis: danger and opportunity. *Journal of Visual Impairment and Blindness*, pages 496–502, 1990.

[17] Braille Authority of North America. *Braille Formats: Principles of Print to Braille Transcription 1997*. American Printing House for the Blind, 1998.

[18] Herb Brody. The great equalizer: Pcs empower the disabled. *PC/Comput.*, 2(7):82–93, July 1989.

[19] P. Brunet, B. A. Feigenbaum, K. Harris, C. Laws, R. Schwerdtfeger, and L. Weiss. Accessibility requirements for systems design to accommodate users with vision impairments. *IBM Syst. J.*, 44(3):445–466, August 2005.

[20] Steve Burbeck. Applications programming in Smalltalk-80: How to use model-view-controller (MVC). Available online at: http://archive.sunet.se/pub/lang/smalltalk/st-docs/mvc.rtf, 1992.

[21] Richard Burridge. My first blind email. Available online at: http://blogs.sun.com/richb/entry/my_first_blind_email, November 2005.

[22] W. Buxton, R. Foulds, M. Rosen, L. Scadden, and F. Shein. Human interface design and the handicapped user. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '86, pages 291–297. ACM, 1986.

[23] William Buxton, William Gaver, and Sara Bly. Auditory interfaces: The use of non-speech audio at the interface. Draft manuscript, 1994.

[24] Gaëlle Calvary, Joëlle Coutaz, and David Thevenin. A unifying reference framework for the development of plastic user interfaces. In *Proceedings of the 8th IFIP International Conference on Engineering for Human-Computer Interaction*, EHCI '01, pages 173–192. Springer-Verlag, 2001.

[25] Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289–308, 2003.

[26] Gaelle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Nathalie Souchon, Laurent Bouillon, Murielle Florins, and Jean Vanderdonckt. Plasticity of user interfaces: A revised reference framework. In *Proceedings of the First International Workshop on Task Models and Diagrams for User Interface Design*, pages 127–134. INFOREC Publishing House Bucharest, 2002.

[27] Mitch Chapman. Create user interfaces with glade. *Linux J.*, 2001(87):90–92,94, July 2001.

[28] Mark H. Chignell and John A. Waterworth. WIMPs and NERDs: an extended view of the user interface. *SIGCHI Bull.*, 23(2):15–21, March 1991.

[29] Kevin Christian. Design of haptic and tactile interfaces for blind users. Available online at: http://otal.umd.edu/UUGuide/kevin/, 2000.

[30] Congress of the United States of America. *42 U.S.C. – The Public Health and Welfare, Section 1382(a)2)*. GPO, 1997.

[31] Michael Cooper. Accessibility of emerging rich web technologies: web 2.0 and the semantic web. In *Proceedings of the 2007 international cross-disciplinary conference on Web accessibility (W4A)*, W4A '07, pages 93–98. ACM, 2007.

[32] Joëlle Coutaz. Multimedia and multimodal interfaces: A software engineering perspective. In *Proceedings of the East-West International Conference on Human-Computer Interaction (EWHCI'92)*, 1992.

[33] Joëlle Coutaz, Laurence Nigay, and Daniel Salber. Multimodality from the user and system perspectives. In *Proceedings of the ERCIM'95 workshop on Multimedia Multimodal User Interfaces*, 1995.

[34] Kenneth Craik. *The Nature of Explanation*. Cambridge University Press, 1943.

[35] Guido de Melo, Frank Honold, Michael Weber, Mark Poguntke, and André Berton. Towards a flexible UI model for automotive human-machine interaction. In *Proceedings of the 1st International Conference on Automotive User Interfaces and Interactive Vehicular Applications*, AutomotiveUI '09, pages 47–50. ACM, 2009.

[36] Dirk Draheim, Christof Lutteroth, and Gerald Weber. Graphical user interfaces as documents. In *Proceedings of the 7th ACM SIGCHI New Zealand chapter's international conference on Computer-human interaction: design centered HCI*, CHINZ '06, pages 67–74. ACM, 2006.

[37] Alistair D. N. Edwards. Modelling blind users' interactions with an auditory computer interface. *International Journal of Man-Machine Studies*, 30(5):575–589, 1989.

[38] Alistair D. N. Edwards. Evaluation of outSpoken software for blind users. Technical Report YCS150, University of York, Department of Computer Science, 1991.

[39] Alistair D. N. Edwards. The difference between a blind computer user and a sighted one is that the blind one cannot see. Interactionally Rich Systems Network, Working Paper No. ISS/WP2, 1994.

[40] Alistair D. N. Edwards. The rise of the graphical user interface. *Library Hi Tech*, 14(1):46–50, 1996.

[41] Alistair D. N. Edwards and Evangelos Mitsopoulos. A principled methodology for the specification and design of nonvisual widgets. *ACM Trans. Appl. Percept.*, 2(4):442–449, October 2005.

[42] Allan Edwards, Alistair D. N. Edwards, and Elizabeth D. Mynatt. Enabling technology for users with special needs. In *Conference companion on Human factors in computing systems*, CHI '94, pages 405–406. ACM, 1994.

[43] W. Keith Edwards, S. H. Liebeskind, Elizabeth D. Mynatt, and William D. Walker. A remote access protocol for the X Window System. In *Proceedings of the 9th Annual X Technical Conference*, pages 245–256. O'Reilly & Associates, Inc., 1995.

[44] W. Keith Edwards and Elizabeth D. Mynatt. An architecture for transforming graphical interfaces. In *Proceedings of the 7th annual ACM symposium on User interface software and technology*, UIST '94, pages 39–47. ACM, 1994.

[45] W. Keith Edwards, Elizabeth D. Mynatt, and Kathryn Stockton. Providing access to graphical user interfaces – not graphical screens. In *Proceedings of the first annual ACM conference on Assistive technologies*, Assets '94, pages 47–54. ACM, 1994.

[46] James D. Foley and Andries van Dam. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley Publishing Co., 1982.

[47] American Foundation for the Blind. Assistive technology timeline. Available online at: http://www.afb.org/Section.asp?SectionID=4\&DocumentID=4368.

[48] Amy Fowler. A swing architecture overview. Available online at: http://java.sun.com/products/jfc/tsc/articles/architecture/, 1998.

[49] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Co., 1994.

[50] William W. Gaver. The SonicFinder: an interface that uses auditory icons. *Hum.-Comput. Interact.*, 4(1):67–94, March 1989.

[51] Becky Gibson. Enabling an accessible web 2.0. In *Proceedings of the 2007 international cross-disciplinary conference on Web accessibility (W4A)*, W4A '07, pages 1–6. ACM, 2007.

[52] Ephraim P. Glinert and G. Bowden Wise. Adaptive multimedia interfaces in polymestra. In *Proc. of the First European Conference on Disability, Virtual Reality, and Associated Technologies (ECDVRAT 96)*, pages 141–150, 1996.

[53] John Goldthwaite. Accessibility standards for operating systems. *SIGCAPH Comput. Phys. Handicap.*, 75:2–3, January 2003.

[54] Jim Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481. Springer-Verlag, 1978.

[55] The Open Group. History and timeline. Available online at: http://www.unix.org/what_is_unix/history_timeline.html.

[56] Josefina Guerrero-Garcia, Juan Manuel González-Calleros, Jean Vanderdonckt, and Jaime Muñoz-Arteaga. A theoretical survey of user interface description languages: Preliminary results. In *Latin American Web Congress, 2009. LA-WEB '09*, pages 36–43, November 2009.

[57] Rul Gunzenhäuser and Gerhard Weber. Graphical user interfaces for blind people. In K. Brunnstein and E. Raubold, editors, *13th World Computer Congress 94, Volume 2*, pages 450–457. Elsevier Science B.V., 1994.

[58] Bill Haneman and Marc Mulcahy. The GNOME accessibility architecture in detail. Presented at the CSUN Conference on Technology and Disabilities, 2002.

[59] S. Harness, K. Pugh, N. Sherkat, and R. Whitrow. Fast icon and character recognition for universal access to WIMP interfaces for the blind and partially sighted. In E. Ballabio, I. Placencia-Porrero, and R. P. d. l. Bellcasa, editors, *Rehabilitation Technology: Strategies for the European Union (Proceedings of the First Tide Congress)*, pages 19–23. IOS Press, Brussels, 1993.

[60] S.J. Harness, K. Pugh, N. Sherkat, and R.J. Whitrow. Enabling the use of windows environment by the blind and partially sighted. In *IEE Colloquium on Information Access for People with Disability*, pages 10/1–10/3, May 1993.

[61] Sandra G. Hart. NASA-Task Load Index (NASA-TLX); 20 years later. In *Proceedings of the Human Factors and Ergonomics Society 50th Annual Meeting*, pages 904–908. Human Factors & Ergonomics Society, 2006.

[62] Sandra G. Hart and Lowell E. Staveland. Development of NASA-TLX (Task Load Index). In P. A. Hancock and N. Meshkati, editors, *Human Mental Workload*, pages 239–250. North Holland Press, 1988.

[63] Ralph D. Hill. Supporting concurrency, communication, and synchronization in human-computer interaction – the sassafras uims. *ACM Trans. Graph.*, 5(3):179–210, July 1986.

[64] Mark Hollins. *Understanding Blindness: An Integrative Approach*. Lawrence Erlbaum Associates, 1989.

[65] Institute of Electrical and Electronics Engineers. *610.12-1990, IEEE Standard Glossary of Software Engineering Terminology*. IEEE, Los Alamos, CA, 1990.

[66] International Organization for Standardization. *ISO/IEC 9126, Information Technology, Software Product Evaluation, Quality Characteristics and Guidelines for their Use*. ISO, Geneva, 1991.

[67] International Organization for Standardization. *ISO/IEC 9241-11, Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs), Part 11: Guidance on Usability*. ISO, Geneva, 1998.

[68] International Organization for Standardization. *ISO/IEC 13066-1:2011 Information technology – Interoperability with assistive technology (AT) – Part 1: Requirements and recommendations for interoperability*. ISO, Geneva, 2011.

[69] R. J. K. Jacob. User interfaces. In A. Ralston, E. D. Reilly, and D. Hemmendinger, editors, *Encyclopedia of Computer Science, Fourth Edition*. Grove Dictionaries, Inc., 2000.

[70] Robert J.K. Jacob, Audrey Girouard, Leanne M. Hirshfield, Michael S. Horn, Orit Shaer, Erin Treacy Solovey, and Jamie Zigelbaum. Reality-based interaction: a framework for post-WIMP interfaces. In *Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, CHI '08, pages 201–210. ACM, 2008.

[71] Philip Johnson-Laird. *Mental Models*. Harvard University Press, 1983.

[72] Nikolaos Kaklanis, Juan González Calleros, Jean Vanderdonckt, and Dimitrios Tzovaras. A haptic rendering engine of web pages for blind users. In *Proceedings of the working conference on Advanced visual interfaces*, AVI '08, pages 437–440. ACM, 2008.

[73] Shiro Kawai, Hitoshi Aida, and Tadao Saito. Designing interface toolkit with dynamic selectable modality. In *Proceedings of the second annual ACM conference on Assistive technologies*, Assets '96, pages 72–79. ACM, 1996.

[74] Alan C. Kay. User interface: A personal view. In Brenda Laurel, editor, *The Art of Human-Computer Interface Design*, pages 191–207. Addison-Wesley Publishing Co., 1990.

[75] Dirk Kochanek. Designing an offscreen model for a gui. In Wolfgang Zagler, Geoffrey Busby, and Roland Wagner, editors, *Computers for Handicapped Persons*, volume 860 of *Lecture Notes in Computer Science*, pages 89–95. Springer Berlin / Heidelberg, 1994.

[76] Peter Korn. Solaris 10 – another first, for accessibility. http://blogs.sun.com/korn/entry/solaris_10_another_first_for, 2005.

[77] Stefan Kost. Dynamically generated multi-modal application interfaces. In Marc Luyten, Kris Abrams, Jean Vanderdonckt, and Quentin Limbourg, editors, *Developing User Interfaces with XML: Advances on User Interface Description Languages*, pages 25–30, 2004.

[78] Stefan Kost. *Dynamically generated multi-modal application interfaces*. PhD thesis, Technische Universität Dresden, Dresden, Germany, 2006.

[79] Michael Kraus, Thorsten Völkel, and Gerhard Weber. An off-screen model for tactile graphical user interfaces. In Klaus Miesenberger, Joachim Klaus, Wolfgang Zagler, and Arthur Karshmer, editors, *Computers Helping People with Special Needs*, volume 5105 of *Lecture Notes in Computer Science*, pages 865–872. Springer Berlin / Heidelberg, 2008.

[80] Sri Hastuti Kurniawan and Alistair G. Sutcliffe. Mental models of blind users in the Windows environment. In Klaus Miesenberger, Joachim Klaus, and Wolfgang Zagler, editors, *Computers Helping People with Special Needs*, volume 2398 of *Lecture Notes in Computer Science*, pages 373–386. Springer Berlin / Heidelberg, 2002.

[81] Sri Hastuti Kurniawan, Alistair G. Sutcliffe, and Paul L. Blenkhorn. How blind users' mental models affect their perceived usability of an unfamiliar screen reader. In Matthias Rauterberg, Marino Menozzi, and Janet Wesson, editors, *Human-Computer Interaction INTERACT '03*, pages 631–638. IOS Press, 2003.

[82] James A. Kutsch, Jr. A talking computer terminal. In *Proceedings of the June 13-16, 1977, national computer conference*, AFIPS '77, pages 357–362. ACM, 1977.

[83] Claire Laberge-Nadeau. Wireless telephones and the risk of road crashes. *Accident Analysis & Prevention*, 35(5):649–660, September 2003.

[84] Steven Landau. Tactile graphics and strategies for non-visual seeing. *Tresholds*, 19:78–82, 1999.

[85] Lisa Larges. Personal communication (used with permission).

[86] Ole Lauridsen. Abstract specification of user interfaces. In *Conference companion on Human factors in computing systems*, CHI '95, pages 147–148. ACM, 1995.

[87] Vincent Lévesque. Blindness, technology and haptics. Technical Report TR-CIM-05.08, McGill University, Centre for Intelligent Machines, Haptics Laboratory, 2008.

[88] James R. Lewis. IBM computer usability satisfaction questionnaires: psychometric evaluation and instructions for use. *Int. J. Hum.-Comput. Interact.*, 7(1):57–78, January 1995.

[89] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, Murielle Florins, and Daniela Trevisan. UsiXML: A user interface description language for context-sensitive user interfaces. In Kris Luyten, Marc Abrams, Jean Vanderdonckt, and Quentin Limbourg, editors, *Proceedings of the ACM AVI'2004 Workshop "Developing User Interfaces*

with XML: Advances on User Interface Description Languages", pages 55–62, May 2004.

[90] Marino Linaje, Adolfo Lozano-Tello, Miguel A. Perez-Toledano, Juan Carlos Preciado, Roberto Rodriguez-Echeverria, and Fernando Sanchez-Figueroa. Providing RIA user interfaces with accessibility properties. *Journal of Symbolic Computation*, 46(2):207–217, 2011.

[91] Yura Mamyrin. Java implementation of the game "Risk". Available online at http://jrisk.sourceforge.net/, 2007.

[92] Zdenek Mikovec, Jan Vystrcil, and Pavel Slavik. Web toolkits accessibility study. *SIGACCESS Access. Comput.*, pages 3–8, June 2009.

[93] George Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychology Review*, 63(2):81–97, 1956.

[94] Sarah Morley. *Window Concepts: An Introductory Guide for Visually Disabled Users*. Royal National Institute for the Blind, 1995.

[95] Computer History Museum. Timeline of computer history: 1971. Available online at: http://www.computerhistory.org/timeline/?year=1971.

[96] Computer History Museum. Timeline of computer history: 1977. Available online at: http://www.computerhistory.org/timeline/?year=1977.

[97] Computer History Museum. Timeline of computer history: 1981. Available online at: http://www.computerhistory.org/timeline/?year=1981.

[98] Elizabeth D. Mynatt. Transforming graphical interfaces into auditory interfaces for blind users. *Hum.-Comput. Interact.*, 12(1):7–45, March 1997.

[99] Elizabeth D. Mynatt and W. Keith Edwards. Mapping guis to auditory interfaces. In *Proceedings of the 5th annual ACM symposium on User interface software and technology*, UIST '92, pages 61–70. ACM, 1992.

[100] Elizabeth D. Mynatt and Gerhard Weber. Nonvisual presentation of graphical user interfaces: contrasting two approaches. In *Proceedings of the SIGCHI conference on Human factors in computing systems: celebrating interdependence*, CHI '94, pages 166–172. ACM, 1994.

[101] Nancy J. Nersessian. Model-based reasoning in conceptual change. In L. Magnani, N. J. Nersessian, and P. Thagard, editors, *Model-Based Reasoning in Scientific Discovery*, pages 5–22. Kluwer Academic Publishers, 1999.

[102] Alan F. Newell. CHI for everyone. *Interfaces*, 35:4–5, 1997.

[103] Laurence Nigay and Joëlle Coutaz. A design space for multimodal systems: concurrent processing and data fusion. In *Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, CHI '93, pages 172–178. ACM, 1993.

[104] Tatuso Nishizawa, Daisuke Nagasaka, Hiroshi Kodama, Masami Hashimoto, Kazunori Itoh, and Seiji Miyaoka. Realization of direct manipulation for visually impaired on touch panel interface. In Klaus Miesenberger, Joachim Klaus, Wolfgang Zagler, and Arthur Karshmer, editors, *Computers Helping People with Special Needs*, volume 6180 of *Lecture Notes in Computer Science*, pages 377–384. Springer Berlin / Heidelberg, 2010.

[105] Stina Nylander. The ubiquitous interactor – mobile services with multiple user interfaces. Licentiate thesis, Department of Information Technology, Uppsala University, 2003.

[106] Stina Nylander, Markus Bylund, and Magnus Boman. Mobile access to real-time information – the case of autonomous stock brokering. *Personal Ubiquitous Comput.*, 8:42–46, February 2004.

[107] Stina Nylander, Markus Bylund, and Annika Waern. Ubiquitous service access through adapted user interfaces on multipl e devices. *Personal and Ubiquitous Computing*, 9(3):123–133, 2005.

[108] Željko Obrenović, Julio Abascal, and Dušan Starčević. Universal accessibility as a multimodal design issue. *Commun. ACM*, 50(5):83–88, May 2007.

[109] Oracle Corporation. GNOME 2.0 desktop: Developing with the accessibility framework. Available online at http://download.oracle.com/javase/6/docs/api/, 2011.

[110] Sharon Oviatt. Ten myths of multimodal interaction. *Commun. ACM*, 42(11):74–81, November 1999.

[111] Peter Parente and Brett Clippingdale. Linux screen reader: extensible assistive technology. In *Proceedings of the 8th international ACM SIGACCESS conference on Computers and accessibility*, Assets '06, pages 261–262. ACM, 2006.

[112] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[113] Leonard H. D. Poll and Ronald P. Waterham. Graphical user interfaces and visually disabled users. *Rehabilitation Engineering, IEEE Transactions on*, 3(1):65–69, Mar 1995.

[114] E. Pontelli, D. Gillan, W. Xiong, E. Saad, G. Gupta, and A. I. Karshmer. Navigation of HTML tables, frames, and XML fragments. In *Proceedings of the fifth international ACM conference on Assistive technologies*, Assets '02, pages 25–32. ACM, 2002.

[115] Denise Prescher, Gerhard Weber, and Martin Spindler. A tactile windowing system for blind users. In *Proceedings of the 12th international ACM SIGACCESS conference on Computers and accessibility*, ASSETS '10, pages 91–98. ACM, 2010.

[116] The Archimedes Project. Visual tas (VTAS). Available online at: http://archimedes.hawaii.edu/Visual_TAS.htm.

[117] Morteza Amir Rahimi and John B. Eulenberg. A computing environment for the blind. In *Proceedings of the May 6-10, 1974, national computer conference and exposition*, AFIPS '74, pages 121–124. ACM, 1974.

[118] D. Rose, S. Stegmaier, G. Reina, D. Weiskopf, and T. Ertl. Non-invasive adaptation of black-box user interfaces. In *Proceedings of the Fourth Australasian user interface conference on User interfaces 2003 - Volume 18*, AUIC '03, pages 19–24. Australian Computer Society, Inc., 2003.

[119] Richard Rubinstein and Julian Feldman. A controller for a braille terminal. *Commun. ACM*, 15(9):841–842, September 1972.

[120] Oliver Sacks. The mind's eye: What the blind see. *The New Yorker*, pages 48–59, 28 July 2003.

[121] Norihiro Sadato, Alvaro Pascual-Leone, Jordan Grafman, Marie-Pierre Deiber, Vicente Ibañez, and Mark Hallett. Neural networks for braille reading by the blind. *Brain*, 121:1213–1229, July 1998.

[122] Norihiro Sadato, Alvaro Pascual-Leone, Jordan Grafman, Vicente Ibañez, Marie-Pierre Deiber, George Dold, and Mark Hallett. Activation of the primary visual cortex by braille reading in blind subjects. *Nature*, 380(6574):526–528, 11 April 1996.

[123] Anthony Savidis and Constantine Stephanidis. Building non-visual interaction through the development of the rooms metaphor. In *Conference companion on Human factors in computing systems*, CHI '95, pages 244–245. ACM, 1995.

[124] Anthony Savidis and Constantine Stephanidis. Developing dual user interfaces for integrating blind and sighted users: the HOMER UIMS. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '95, pages 106–113. ACM Press/Addison-Wesley Publishing Co., 1995.

[125] Anthony Savidis and Constantine Stephanidis. The HOMER UIMS for dual user interface development: Fusing visual and non-visual interactions. *Interacting with Computers*, 11(2):173–209, 1998.

[126] Richard S. Schwerdtfeger. Making the GUI talk. *BYTE*, pages 118–128, December 1991. Available online at: ftp://service.boulder.ibm.com/sns/sr-os2/sr2doc/guitalk.txt.

[127] Neil G. Scott and Isabelle Gingras. The total access system. In *CHI '01 extended abstracts on Human factors in computing systems*, CHI EA '01, pages 13–14. ACM, 2001.

[128] Jonathan Seybold. Xerox's "star". *The Seybold Report*, 10(16), 1981.

[129] David Canfield Smith. Personal communication (used with permission).

[130] David Canfield Smith, Eric F. Harslem, Charles H. Irby, Eric B. Kimball, and William L. Verplank. Designing the Star User Interface. *BYTE*, pages 242–282, April 1982.

[131] David Canfield Smith, Charles Irby, Ralph Kimball, and Eric Harslem. The star user interface: an overview. In *Proceedings of the June 7-10, 1982, national computer conference*, AFIPS '82, pages 515–528. ACM, 1982.

[132] Nathalie Souchon and Jean Venderdonckt. A review of XML-compliant user interface description languages. In Joaquim A. Jorge, Nuno Jardim Nunes, and João Falcão e Cunha, editors, *Interactive Systems. Design, Specification, and Verification*, volume 2844 of *Lecture Notes in Computer Science*, pages 391–401. Springer Berlin / Heidelberg, 2003.

[133] Adrian Stanciulescu. *A Methodology for Developing Multimodal User Interfaces of Information Systems*. PhD thesis, Université Catholique de Louvain, Louvain-la-Neuve, Belgium, 2008.

[134] Constantine Stephanidis. Adaptive techniques for universal access. *User Modeling and User-Adapted Interaction*, 11(1-2):159–179, March 2001.

[135] Constantine Stephanidis. User interfaces for all: New perspectives into human computer interaction. In Constantine Staphanidis, editor, *User Interfaces for All - Concepts, Methods, and Tools*, pages 3–17. Lawrence Erlbaum Associates, 2001.

[136] Theodor D. Sterling, M. Lichstein, F. Scarpino, and D. Stuebing. Professional computer work for the blind. *Commun. ACM*, 7(4):228–230, April 1964.

[137] Sun Microsystems. GNOME 2.0 desktop: Developing with the accessibility framework. Technical report, Sun Microsystems, 2003.

[138] Alistair Sutcliffe, Steve Fickas, McKay Moore Sohlberg, and Laurie A. Ehlhardt. Investigating the usability of assistive user interfaces. *Interacting with Computers*, 15(4):577–602, 2003.

[139] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation (3rd Edition)*. Prentice Hall, 2006.

[140] Jim Thatcher. Screen reader/2programmed access to the gui. In *Proceedings of the 4th international conference on Computers for handicapped persons*, pages 76–88. Springer-Verlag New York, Inc., 1994.

[141] Mary Frances Theofanos and Janice (Ginny) Redish. Bridging the gap: between accessibility and usability. *interactions*, 10(6):36–51, November 2003.

[142] Peter Thiessen and Charles Chen. Ajax live regions: chat as a case example. In *Proceedings of the 2007 international cross-disciplinary conference on Web accessibility (W4A)*, W4A '07, pages 7–14. ACM, 2007.

[143] Shari Trewin, Gottfried Zimmermann, and Gregg Vanderheiden. Abstract user interface representations: how well do they support universal access? In *Proceedings of the 2003 conference on Universal usability*, CUU '03, pages 77–84. ACM, 2003.

[144] Shari Trewin, Gottfried Zimmermann, and Gregg Vanderheiden. Abstract representations as a basis for usable user interfaces. *Interacting with Computers*, 16(3):477–506, 2004.

[145] Kris Van Hees and Jan Engelen. Abstract UIs as a long-term solution for non-visual access to GUIs. In *Proceedings of the 3rd International Conference on Universal Access in Human-Computer Interaction*, 2005.

[146] Kris Van Hees and Jan Engelen. Abstracting the graphical user interface for non-visual access. In Alain Pruski and Harry Knops, editors, *Assistive technology from virtually to reality. 8th European conference for the Advancement of Assistive Technology in Europe*, pages 239–245. IOS Press, 2005.

[147] Kris Van Hees and Jan Engelen. Non-visual access to GUIs: Leveraging abstract user interfaces. In Klaus Miesenberger, Joachim Klaus, Wolfgang Zagler, and Arthur Karshmer, editors, *Computers Helping People with Special Needs*, volume 4061 of *Lecture Notes in Computer Science*, pages 1063–1070. Springer Berlin / Heidelberg, 2006.

[148] Kris Van Hees and Jan Engelen. Non-visual access to guis: Coordinating user interaction. (Unpublished paper), 2007.

[149] Kris Van Hees and Jan Engelen. Parallel User Interface Rendering: Accessibility for custom widgets. In *Proceedings of the first International ÆGIS Conference*, pages 17–24, 2010.

[150] Kris Van Hees and Jan Engelen. PUIR: Parallel User Interface Rendering. In Klaus Miesenberger, Joachim Klaus, Wolfgang Zagler, and Arthur Karshmer, editors, *Computers Helping People with Special Needs*, volume 6179 of *Lecture Notes in Computer Science*, pages 200–207. Springer Berlin / Heidelberg, 2010.

[151] Kris Van Hees and Jan Engelen. Concurrent multi-target runtime reification of user interface descriptions. In Adrien Coyette, David Faure, Juan Gonzales, and Jean Vanderdonckt, editors, *Proc. of 2nd Int. Workshop on User Interface Extensible Markup Language UsiXML'2011 (Lisbon, 6 September 2011)*, pages 214–221. Thales Research and Technology, Paris, France, 2011.

[152] Kris Van Hees and Jan Engelen. Equivalent representations of multi-modal user interfaces. *Universal Access in the Information Society*, 2012. Under review.

[153] Jean Vanderdonckt, Quentin Limbourg, Benjamin Michotte, Laurent Bouillon, Daniela Trevisan, and Murielle Florins. UsiXML: a user interface description language for specifying multimodal user interfaces. In *WMI '04: Proceedings of the W3C Workshop on Multimodal Interaction*, 2004.

[154] Jean M. Vanderdonckt and François Bodart. Encapsulating knowledge for intelligent automatic interaction objects selection. In *Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, CHI '93, pages 424–429. ACM, 1993.

[155] Gregg C. Vanderheiden. Accessible design of consumer products: Working draft 1.7. Technical report, Trace Research and Development Center, 1992.

[156] Gregg C. Vanderheiden, Wesley Boyd, John H. Mendenhall, Jr., and Kelly Ford. Development of a multisensory nonvisual interface to computers for blind users. *Proceedings of the Human Factors Society 35th Annual Meeting*, pages 315–318, 1991.

[157] Kathy Walrath, Mary Campione, Alison Huml, and Sharon Zakhour. *The JFC Swing Tutorial: A Guide to Constructing GUIs (2nd Edition)*. Prentice Hall, 2004.

[158] Gerhard Weber. Adapting direct manipulation for blind users. In *INTERACT '93 and CHI '93 conference companion on Human factors in computing systems*, CHI '93, pages 21–22. ACM, 1993.

[159] Gerhard Weber. Programming for usability in nonvisual user interfaces. In *Proceedings of the third international ACM conference on Assistive technologies*, Assets '98, pages 46–48. ACM, 1998.

[160] Gerhard Weber and Rolf Mager. Non-visual user interfaces for X Windows. In *Proceedings of the 5th international conference on Computers helping people with special needs. Part II*, pages 459–468. R. Oldenbourg Verlag GmbH, 1996.

[161] Gerhard Weber, Helen Petrie, Dirk Kochanek, and Sarah Morley. Training blind people in the use of graphical user interfaces. In Wolfgang Zagler, Geoffrey Busby, and Roland Wagner, editors, *Computers for Handicapped Persons*, volume 860 of *Lecture Notes in Computer Science*, pages 25–31. Springer Berlin / Heidelberg, 1994.

[162] Mark Weiser. The computer of the 21st century. *American Scientific*, 265(3):66–75, September 1991.

[163] G. Bowden Wise and Ephraim P. Glinert. Metawidgets for multimodal applications. In *Proceedings of the RESNA'95 conference*, pages 9–14, 1995.

[164] World Wide Web Consortium. Synchronized multimedia integration language (SMIL 3.0). Technical Report REC-SMIL3-20081201, W3C, 2008.

[165] World Wide Web Consortium. Accessible rich internet applications (WAI-ARIA) 1.0. Technical Report CR-wai-aria-20110118, W3C, 2011.

[166] Judy York and Parag C. Pendharkar. Human-computer interaction issues for mobile computing in a variable work context. *International Journal of Human-Computer Studies*, 60(5-6):771–797, 2004.

Curriculum vitae

Kris Van Hees was born June 12th, 1970 in Schoten, Belgium. He received a Bachelor's degree in Applied Sciences from Katholieke Universiteit Leuven (KU Leuven), Belgium, in 1988, a Bachelor's degree in Computer Science from KU Leuven in 1993, and a Master's degree in Computer Science from KU Leuven in 1995.

Kris moved to the United States of America in 1995, where he pursued a professional career in computer science. He is currently working for Oracle Corporation. Aside from his professional career, as of December 2003, he has been pursuing a doctor of engineering degree at the Department of Electrical Engineering, SCD/DocArch, KU Leuven, Belgium, under the supervision of prof. em. dr. ir. Jan Engelen (KU Leuven) and prof. dr. Gerhard Weber (Technische Universität Dresden, Germany).

Throughout his career, Kris has been involved in the development of world-wide distributed financial trading systems, fault-tolerant distributed filesystems, Linux kernel development, accessibility consulting, and various programming tools such as compilers, debuggers, ... In his spare time, he has worked on object oriented programming language design, speech synthesis, and computer system accessibility.

Kris is actively involved in advocacy efforts for students with special needs in the USA, primarily concerning assistive technology and inclusive education. He serves on the board of the Pennsylvania Education-for-All Coalition. Kris is also a member of the ACM SIGACCESS and SIGCHI special interest groups.

List of Publications

[1] Kris Van Hees and Jan Engelen. Abstract UIs as a long-term solution for non-visual access to GUIs. In *Proceedings of the 3rd International Conference on Universal Access in Human-Computer Interaction*, 2005.

[2] Kris Van Hees and Jan Engelen. Abstracting the graphical user interface for non-visual access. In Alain Pruski and Harry Knops, editors, *Assistive technology from virtually to reality. 8th European conference for the Advancement of Assistive Technology in Europe*, pages 239–245. IOS Press, 2005.

[3] Kris Van Hees. Unix screen-readers. *Infovisie Magazine*, 19(4):17–19, December 2005.

[4] Kris Van Hees and Jan Engelen. Non-visual access to GUIs: Leveraging abstract user interfaces. In Klaus Miesenberger, Joachim Klaus, Wolfgang Zagler, and Arthur Karshmer, editors, *Computers Helping People with Special Needs*, volume 4061 of *Lecture Notes in Computer Science*, pages 1063–1070. Springer Berlin / Heidelberg, 2006.

[5] Kris Van Hees and Jan Engelen. Non-visual access to guis: Coordinating user interaction. (Unpublished paper), 2007.

[6] Kris Van Hees and Jan Engelen. PUIR: Parallel User Interface Rendering. In Klaus Miesenberger, Joachim Klaus, Wolfgang Zagler, and Arthur Karshmer, editors, *Computers Helping People with Special Needs*, volume 6179 of *Lecture Notes in Computer Science*, pages 200–207. Springer Berlin / Heidelberg, 2010.

[7] Kris Van Hees and Jan Engelen. Parallel User Interface Rendering: Accessibility for custom widgets. In *Proceedings of the first International ÆGIS Conference*, pages 17–24, 2010.

[8] Kris Van Hees and Jan Engelen. Concurrent multi-target runtime reification of user interface descriptions. In Adrien Coyette, David Faure, Juan

Gonzales, and Jean Vanderdonckt, editors, *Proc. of 2nd Int. Workshop on User Interface Extensible Markup Language UsiXML'2011 (Lisbon, 6 September 2011)*, pages 214–221. Thales Research and Technology, Paris, France, 2011.

[9] Kris Van Hees and Jan Engelen. Equivalent representations of multi-modal user interfaces. *Universal Access in the Information Society*, 2012. Under review.

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Electrical Engineering (ESAT)
SCD / DocArch
Kasteelpark Arenberg 10 Box 2442, 3001 Heverlee, Belgium

KATHOLIEKE UNIVERSITEIT LEUVEN