Tool Support for Designing, Managing and Optimizing Multi-Device User Interfaces

Proefschrift voorgelegd tot het behalen van de graad van Doctor in de Wetenschappen, Informatica, te verdedigen door:

Jan MESKENS

Promotor: prof. dr. Karin Coninx
Copromotor: prof. dr. Kris Luyten
First of all, I would like to thank Prof. dr. Karin Coninx. As my advisor, she offered me a lot of support and guidance in scientific writing and provided me with many suggestions to improve the quality of our papers and this dissertation. I also would like to thank my co-advisor, Prof. dr. Kris Luyster, for the enthusiasm and passion about my research and for all his contributions to the papers we wrote. He gave me a lot of suggestions and motivated me to submit papers to the right venues. I also would like to thank Kris and Karin for the flexibility and support they offered me when I needed it the most.

Thanks also to all the other members of the jury, Prof. dr. Philippe Bekaert, Prof. dr. Marc Gyssens, dr. Jan Vandenbergh, Prof dr. Jean Vanderdonckt, Prof dr. Gaëlle Calvary, and last, but certainly not least, special thanks go to dr. Jeffrey Nichols. Thanks to Jeffrey’s feedback on early stage ideas, I was able to develop these into actual research contributions.

During my research, I was a member of the research project AMASS++ (IWT SBO 060051) in which I had the opportunity to collaborate with a lot of interesting people, as well inside as outside our university. I have been very lucky to work together with Mieke Haesen. Without her structured working method, I would never have met the deadlines for this project. Moreover, she was also a fantastic colleague who was always willing to provide feedback about my research ideas. Her knowledge about user experiments also helped me a lot when conducting my first user studies.

I would like to thank all room-mates of room 0.02. Geert Vanderhulst learned me how to catch the daily train in time depending on several statistical variables. During our journeys, he provided me with a lot of inspiring tips and tricks that increased my research efficiency. Special thanks go to Jo Vermeulen. His eye for detail and clear vision helped me to improve the quality of almost every paper we wrote. I could discuss research subjects with him at any time and he always pointed me to the right references to improve
my ideas. I also would like to thank Petr Aksenov, our language expert, who gave me a lot of suggestions which improved the writing style of almost all of my papers.

I want to thank the EDM and its management, Prof. dr. Eddy Flerackers and Prof. dr. Frank Van Reeth, together with all the other professors which are involved in making the EDM what it is today. Thanks to all the other researchers at the Expertise center for Digital Media, especially Sofie Notelaers, Steven Maesen, dr. Lode Vanacken, Davy Vanacken, Maarten Cardinaels, Jan Schneider, dr. Chris Raymaekers and Johanna Octavia.

In 2008, I had the opportunity to work together with Alexandre Demeure, back then a senior researcher at our lab. We discussed a lot of research ideas and he helped us to clearly state the contributions of the original Gummy paper. His research also inspired me directly to develop the design toolglass.

Whenever I needed some audiovisual material, Luc Adriaens or Karel Robert helped me out with their professional support. I would like to thank both of them for all their help. I am also grateful to the EDM secretariat, Ingrid Konings and Roger Claes for all the things they arranged for me during the past four years. Thanks also to Peter Quax and Tom Deweyer, who helped me a lot when I was (again) in trouble with some hard- or software.

I also want to thank my parents, Fons and Hélène, who gave me the opportunity to follow my ambitions. They supported me in my studies, work and other choices in my life and were always willing to offer support and guidance.

Last but certainly not least, I want to thank my girlfriend, Kristien De Pauw, for her support at the times that were the most difficult for me. There is no one who has supported me more and I could not have finished this without her.
Abstract

People often access the same information on a variety of devices such as laptops, mobile phones and tablet PCs. On every device, this information is accessed through a user interface that is specifically designed for this device. Currently, designers have to create these multi-device user interfaces manually, which is time consuming and implies switching across a lot of different, often mutually exclusive, design tools. Furthermore, designers need to repeat a lot of design steps in order to keep these designs consistent.

The research presented in this dissertation explores how design tools can be improved in order to better support the design of realistic multi-device high-fidelity prototypes. As a first contribution, we propose GUMMY, a graphical design environment for creating multi-device user interfaces. This design environment allows designers to target multiple devices in one environment without raising the threshold to design user interfaces.

The second contribution concerns techniques for managing multiple device-specific user interfaces. These techniques automate several parts of the design process in order to reduce the amount of repetitive design actions designers have to perform. The proposed techniques allow designers to copy widgets from one device-specific user interface to another and to edit these copies in concert. We also present a design tool macro system, which facilitates designers to record design actions in one design and to reply them afterwards in the same or another design.

When creating multi-device user interfaces, it is important that every device-specific user interface takes into account target device constraints such as screen size, CPU speed or touch screen sensitivity. To ensure this, we propose tools for iteratively optimizing user interfaces until they meet the constraints of the target device. These iterative optimization tools allow designers to switch rapidly between designing and testing in order to verify whether their design decisions work like intended on the target device and to update their
designs if necessary.
# Contents

Acknowledgments i  
Abstract iv  
Contents ix  
List of Figures xiii  
List of Tables xv  

## I Introduction

1 Introduction 3  
1.1 Motivation ........................................ 3  
1.2 Research Questions and Contributions .............. 5  
1.3 Topics Outside the Scope of this Work ............. 6  
1.4 Overview ........................................ 7  

## 2 Related Multi-Device User Interface Design Tools 9  
2.1 Introduction ....................................... 9  
2.2 GUI Toolkits ..................................... 10  
2.3 Model-Based User Interface Design ................. 11  
2.3.1 XML-Based User Interface Description Language .... 13  
2.3.2 Modeling and Automated Design Tools .......... 18  
2.3.3 Model-Based Toolkits .......................... 20  
2.3.4 Discussion ................................... 22  
2.4 HTML 5 ........................................... 23  
2.5 Prototyping Tools .................................. 24
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2 Automatic Transformations</td>
<td>62</td>
</tr>
<tr>
<td>5.2.1 Scenario of Use</td>
<td>63</td>
</tr>
<tr>
<td>5.2.2 Implementation</td>
<td>63</td>
</tr>
<tr>
<td>5.3 Linked Editing</td>
<td>65</td>
</tr>
<tr>
<td>5.3.1 Scenario of Use</td>
<td>66</td>
</tr>
<tr>
<td>5.3.2 Implementation</td>
<td>67</td>
</tr>
<tr>
<td>5.4 Evaluation</td>
<td>67</td>
</tr>
<tr>
<td>5.4.1 Evaluation Set-up</td>
<td>68</td>
</tr>
<tr>
<td>5.4.2 Results and Feedback</td>
<td>69</td>
</tr>
<tr>
<td>5.5 Conclusions</td>
<td>71</td>
</tr>
<tr>
<td>6 Design-Tool Macros (D-Macs)</td>
<td>73</td>
</tr>
<tr>
<td>6.1 Introduction</td>
<td>73</td>
</tr>
<tr>
<td>6.2 Design Tool Requirements</td>
<td>75</td>
</tr>
<tr>
<td>6.2.1 Generalization</td>
<td>75</td>
</tr>
<tr>
<td>6.2.2 Feedback</td>
<td>76</td>
</tr>
<tr>
<td>6.2.3 Contextual Assistance</td>
<td>76</td>
</tr>
<tr>
<td>6.2.4 Design Community</td>
<td>76</td>
</tr>
<tr>
<td>6.3 D-Macs Features</td>
<td>77</td>
</tr>
<tr>
<td>6.3.1 Action Recording</td>
<td>77</td>
</tr>
<tr>
<td>6.3.2 Action Sequence Visualizations</td>
<td>79</td>
</tr>
<tr>
<td>6.3.3 Publishing Design Action Sequences</td>
<td>80</td>
</tr>
<tr>
<td>6.3.4 Reusing Actions in Different Designs</td>
<td>80</td>
</tr>
<tr>
<td>6.3.5 Handling Action Errors</td>
<td>82</td>
</tr>
<tr>
<td>6.4 Architecture and Implementation</td>
<td>83</td>
</tr>
<tr>
<td>6.4.1 Design Action Recording</td>
<td>84</td>
</tr>
<tr>
<td>6.4.2 Replaying Design Actions</td>
<td>85</td>
</tr>
<tr>
<td>6.5 D-Macs Example User Interfaces</td>
<td>86</td>
</tr>
<tr>
<td>6.5.1 Designing User Interfaces Manually</td>
<td>86</td>
</tr>
<tr>
<td>6.5.2 Designing User Interfaces with D-Macs</td>
<td>88</td>
</tr>
<tr>
<td>6.5.3 Reusing Design Action Sequences</td>
<td>90</td>
</tr>
<tr>
<td>6.5.4 Maintaining User Interfaces</td>
<td>91</td>
</tr>
<tr>
<td>6.6 D-Macs Effectiveness Attributes</td>
<td>91</td>
</tr>
<tr>
<td>6.6.1 Reducing Solution Viscosity</td>
<td>92</td>
</tr>
<tr>
<td>6.6.2 Power In Combination</td>
<td>93</td>
</tr>
<tr>
<td>6.7 Conclusion</td>
<td>94</td>
</tr>
</tbody>
</table>
IV Optimizing User Interface Designs

7 Live User Interface Design
  7.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
  7.2 On-device Adaptable Systems . . . . . . . . . . . . . . . . . . 99
    7.2.1 Desktop Adaptable Systems . . . . . . . . . . . . . . . . . 99
    7.2.2 Mobile Adaptable Systems . . . . . . . . . . . . . . . . . 100
  7.3 On-Device UI Design Systems . . . . . . . . . . . . . . . . . . 101
  7.4 Connected-Device UI Design . . . . . . . . . . . . . . . . . . . 101
  7.5 Multi-Device Scalability . . . . . . . . . . . . . . . . . . . . . 102
  7.6 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103

8 Multi-Device Live UI Design
  8.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
  8.2 A Multi-Device Live UI Design Framework . . . . . . . . . . . 106
    8.2.1 Interaction Channel . . . . . . . . . . . . . . . . . . . . . 106
    8.2.2 Design Channel . . . . . . . . . . . . . . . . . . . . . . . 107
  8.3 Design Mirror . . . . . . . . . . . . . . . . . . . . . . . . . . . 108
    8.3.1 Scenario of Use . . . . . . . . . . . . . . . . . . . . . . . 108
    8.3.2 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . 108
    8.3.3 Results and Discussion . . . . . . . . . . . . . . . . . . . 110
  8.4 Design Toolglass . . . . . . . . . . . . . . . . . . . . . . . . . . 111
    8.4.1 Scenario of Use . . . . . . . . . . . . . . . . . . . . . . . 111
    8.4.2 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . 114
    8.4.3 Results and Discussion . . . . . . . . . . . . . . . . . . . 114
  8.5 Multi-Device Continuous Design Pointer . . . . . . . . . . . . . 114
  8.6 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 117

V Conclusions and Future Work

9 Extensions and Future Work
  9.1 Web-Service Integration . . . . . . . . . . . . . . . . . . . . . . 121
  9.2 Automatic Usability Standards Evaluation . . . . . . . . . . . . 123
  9.3 Integration of Contextual Information from Storyboards . . . . 124
  9.4 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . 126
    9.4.1 Tool Support for Specifying GUI Behaviour . . . . . . . . 126
    9.4.2 Better Design Community Support . . . . . . . . . . . . . 127
  9.5 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127
## CONTENTS

10 Conclusions  
10.1 Achievements and Main Contributions . . . . . . . . . . . 129  
10.2 Scientific Contributions and Publications . . . . . . . . . . . 131  

Dutch Summary  

Bibliography
List of Figures

1.1 The techniques and tools that contribute to answer the three proposed research questions. .......................... 8

2.1 User interfaces can be specified at different levels of abstractions and gradually transformed from top level abstract models to a run time user interface. .......................... 12

2.2 The different levels of abstraction in MARIA XML. ............... 18

2.3 The Transformation Environment for inteRactivE Systems representations (TERESA). .......................... 20

2.4 The MARIA tool for transforming the AUI in a desktop CUI. This image is courtesy of Paterno et al [PSS09]. .......................... 21

2.5 The Supple classroom example. This image is courtesy of Gajos et al. [GWW10] ................................................. 22

2.6 In multi-device user interface design, one abstract model can be transformed into multiple platform-specific UIs. This Figure originates from Calvary et al. [CCT+03], redrawn by the author. .......................... 23

3.1 Our initial idea about a multi-device design process. The steps in this process are divided between the designer (left) and design tool (right). ................................................. 34

3.2 The proof-of-concept version of a multi-device design environment, which is used to stimulate discussion about multi-device GUI design tool support. This prototype environment allows designers to create a GUI for one platform (a) and to transform this GUI into a new GUI for another platform (b and c). .......................... 35

4.1 The GUMMY design environment. .......................... 45

4.2 GUMMY’s architecture is inspired by the Model-View-Controller software architecture. .......................... 46
4.3 The structure of GUMMY’s CUI model. It contains the user interface widget tree (a), the layout information (e) and properties of these widgets (b), and the value of every property (c). The model also describes how widgets and properties map on concrete elements (d). .................................................. 49

4.4 The relation between the properties panel and the properties in the CUI model. .................................................. 51

4.5 The layout information of GUMMY’s CUI model is used to place the manipulators at the right place on the canvas. ................. 52

4.6 GUMMY’s MVC architecture is distributed over the PC of the designer and the target device. ................................. 56

5.1 An example scenario showing how cross-device copy-pasting and linked editing can be used in multi-device UI design. ......... 64

5.2 Remapping a combobox on the Windows Mobile platform to a similar widget on the Adobe Flex platform. ..................... 66

5.3 The GUMMY UI model is extended with content properties to support linked editing. .............................................. 68

5.4 Post-experiment questionnaire results. ................................. 69

6.1 Using D-Macs, designers can *demonstrate* an action sequence, *share* these action sequences with other designers and *replay* previously recorded actions. ................................. 74

6.2 D-Macs augments a multi-device GUI builder with a design action recorder. The example user interface design shown here was inspired by the *classroom interface* example of Gajos et al. [GWW10]. ................................. 78

6.3 Action recordings are visualized as icons and human-readable text. .................................................. 79

6.4 Every shared action recording is visualized as a *comic strip* on the central repository’s web interface. ................................. 81

6.5 The state icon of an action indicates if it has to be replayed manually or automatically. ................................. 82

6.6 D-Macs provides suggestions (1) for resolving errors that might occur during action replay. When clicking on a suggestion, all parts of the D-Macs design workspace that contribute to follow this suggestion are highlighted (2). ................................. 83
6.7 Recorded design actions are internally stored as a list of command objects. These command objects can then be published on a central repository. 84
6.8 Replaying design actions from a command object. 85
6.9 Three user interfaces for the news video explorer: for a desktop PC, a mobile device and for web-based access. All interfaces contain an annotated timeline (1), the desktop and mobile device also contain a video player (2). 87
6.10 D-Macs is employed to transform an annotated news timeline from a desktop video explorer into a web-based fish-eye view visualization and a mobile scroll view. 89
6.11 Action sequence A, as depicted in Figure 6.10, can be employed to transform a card deck image gallery into a fish-eye gallery. 91

7.1 An overview of the Live UI Design tools that are discussed in this Chapter. 99

8.1 Architecture for multi-device live UI design. 107
8.2 The design mirror streams every design action to the target device. 109
8.3 Target device and design PC are mostly close to each other in order to benefit maximally from the design mirror approach. 110
8.4 The design toolglass is structured similar to GUMMY. 112
8.5 The design toolglass interaction technique for Android, Windows Mobile and Adobe Flex. 113
8.6 The multi-device design pointer can cross the bezel of the PC screen and end on the target device screen (a). On the target device, this pointer can be used to select, resize and reposition elements (b). 116

9.1 GUMMY’s extension for connecting GUIs with services. 122
9.2 GUMMY’s GUIDE2ux extension for automatically evaluating the usability of a UI design. 123
9.3 A storyboard created in COMuICSer (image courtesy of Haesen et al. [HVdBMP+11]). 125
LIST OF FIGURES
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>The participants for our study, their companies and the platforms they designed for.</td>
<td>37</td>
</tr>
<tr>
<td>6.1</td>
<td>The requirements that D-Macs has to meet.</td>
<td>75</td>
</tr>
</tbody>
</table>
LIST OF TABLES
Part I

Introduction
Chapter 1

Introduction

1.1 Motivation

Typical computer tasks like “browsing the web” or “reading email” do no longer require a personal computer. Today, most of these tasks can be completed using a wide variety of computing devices including tablet PCs, mobile devices, smart phones and interactive TVs. This trend is mainly supported by designers and software developers, who create versions of their applications for every target device. Google and Facebook are examples of companies who realize applications for multiple target devices to reach their services. This way, services like Youtube, Google Maps or Facebook are accessible on different types of mobile phones\(^1\)\(^2\), tablet PCs\(^3\)\(^4\) and of course personal computers.

When targeting multiple computing devices, it is important to take the input and output constraints of every device into account. For example, an application designed for a large touch screen will be hard to use on a mobile device with a small screen and keypad. One way to deal with this situation is to create a suitable User Interface (UI) manually for every targeted device. Uls are created by UI designers, who are skilled at prototyping the look of the application’s User Interface (UI) \[^{MPN+08}\], and software developers, who are responsible for making this prototype interactive. Manually designing multiple device-specific user interfaces has several drawbacks. It is time consuming

\(^1\)http://www.google.com/mobile
\(^2\)http://www.facebook.com/mobile/
\(^3\)http://www.google.com/mobile/ipad/
because a lot of design effort has to be repeated across devices. For example, when adding a widget to one device-specific UI, it is likely that a similar widget has to be added to other device-specific UIs as well. Moreover, it becomes also difficult to manage the design of multiple device-specific UIs because different devices often require different design tools. Switching between these design environments can be disorienting, but also the different styles for designing interfaces and the incompatibilities between design environments cause problems.

Instead of designing multiple UIs manually, Model-Based User Interface Design (MBUID) approaches can be used to generate a UI for every target platform automatically [PSS09, NHA07, GWW10, Sze]. Typically, these approaches require designers to define a “high-level specification” of a user interface which is then used to automatically generate an appropriate user interface for every context of use. Although it becomes easier for designers to make changes that should be propagated to all targeted platforms and to add new target platforms, model-based GUI design has not been widely adopted. One of the major reasons for this is that the resulting user interfaces usually still lack the aesthetic quality and creativity of a manually designed interface. Furthermore, the design process is often not intuitive for designers. They have to master new languages to specify the high-level models and they have to predict what the resulting user interface will look like [MHP00].

The difficulties with the design of multi-device user interfaces discourages designers from prototyping their designs. Prototyping is a crucial step in UI design. Creating and refining prototypes helps designers to gain new insights where they could not have arrived at without producing a concrete manifestation of their ideas [KHT06]. The designs that result from the prototyping stage are handed over to software developers, who turn every design into a device-specific application. It is important that the handed over designs are stable, because design changes become very expensive in the development stage. To realize designs that remain stable during the development stage, we need multi-device prototyping tools that move closer and closer to the real-world code. These multi-device prototyping tools should converge with development tools and techniques used by software engineers who are actually writing the code [Mog07]. This final prototyping stage is also often called high-fidelity prototyping.

In order to make multi-device high-fidelity prototyping easier for designers, this dissertation contributes GUMMY. GUMMY is a flexible user interface prototyping environment, which is able to target a wide range of platforms. Throughout this thesis, GUMMY is used as a basic framework for experiment-
1.2 Research Questions and Contributions

The goal of this research is to create better tools and methods for designing high-fidelity multi-device prototypes. This dissertation contributes to the advancement of multi-device design tools by considering three different research questions. To answer these questions, this dissertation contributes multiple tools and techniques (Figure 1.1). This section briefly discusses the three research questions and the related contributions.

RQ1: How can a design tool enable designers to create realistic high-fidelity prototypes for a broad range of target devices?

To answer this question, we explore how a design tool can target a wide range of devices without raising the expertise threshold to design user interfaces. We propose Gummy (Figure 1.1-1), a design tool that facilitates designers to target a broad range of devices. Gummy’s usage model is inspired by existing design tools, allowing designers to reuse their knowledge from single platform design in a multi-device design approach.

RQ2: How can a design tool help to reduce the complexity when managing multiple device-specific user interfaces?

When the number of target platforms increases, it becomes complex to manage how the different designs should be edited and updated. Designers often have to repeat design actions, which leads to a lot of duplicated design effort. This question is concerned with finding new ways to automate parts of the design process. Gummy is extended in three ways for automating design steps: automatic transformations (Figure 1.1-3) to copy user interface parts from one device to another; linked editing (Figure 1.1-2) to edit user interfaces in concert; and Design tool MACros (Figure 1.1-4) to record design actions and replay them afterwards. These techniques enable designers to manage multiple designs without increasing the necessity to repeat design actions.
RQ3: How can a design tool enable designers to *iteratively optimize* their designs for the targeted devices?

The investigation of the third research question is concerned with finding techniques to iteratively optimize user interfaces so that they meet the input and output constraints of the target device. We support user interface optimization by means of an iterative design-test approach, where designers can test the impact of their design decisions immediately on the target device. We propose three design techniques in order to facilitate an iterative design-test approach: the **design mirror** (Figure 1.1-5), streaming every design update to a running version of the design on the target device; the **design toolglass** (Figure 1.1-6), which is a semi-transparent design tool that can be placed on a running UI to update it; and the **multi-device continuous design pointer** (Figure 1.1-7) to design directly on the target device when it is connected to the design environment.

1.3 Topics Outside the Scope of this Work

My work on multi-device user interface design could have been taken in many directions. This section describes issues related to my research that I do not explore in this dissertation:

- **UI Behaviour prototyping**: besides the look of multi-device UIs, the interactive behaviour of these UIs is also important. This could be prototyped using notations such as state charts [HKB+06], scripting languages [HYA+08] or visual programming systems [YCM09]. A lot of interesting work could be done in this area, but I have chosen to focus on the look of the UI prototypes.

- **Multi-device Application Development**: to create multi-device applications, we need a suitable user interface on every platform and the application logic has to be shared across devices as well. To share the application logic, inspiration could be found in service oriented architectures, where part of the application logic is centralized. The GUMMY architecture is flexible enough to connect user interface designs with web-services, but this is not something that I have looked at as part of my thesis work.

- **Early UI Design Artefacts**: during the early design stages, design ideas are worked out by means of low-fidelity prototypes. These artefacts allow designers to explore several design variants. Although GUMMY
does not support these typical early stage artefacts, designers can use GUMMY to further refine a low-fidelity prototype into concrete high-fidelity prototypes.

1.4 Overview

This dissertation consists of five parts. Below, we give an overview of the items that we are going to describe in each of these parts.

Part I discusses the requirements of multi-device design tools and existing work in this field. In Chapter 2, we survey existing work around multi-device user interface design. The goal of this survey is to identify concepts that we can reuse in our own GUMMY design environment.

In Part II, we introduce tool support for creating user interfaces for a wide range of computing platforms. In this part, we first analyse what kinds of tools designers need to create multi-device user interfaces. Chapter 4 then introduces GUMMY, our multi-device graphical design environment for creating user interfaces.

Part III discusses the techniques that we provide for managing multi-device design projects. Chapter 5 first discusses automatic transformations and linked editing. Next, the Design tool MACro system (D-Macs) is introduced in Chapter 6.

Part IV gives an overview of the developed techniques for iteratively optimizing user interfaces. Before implementing these techniques, we investigate the use of Live User Interface design tools for iteratively optimizing UIs in Chapter 7. Based on this discussion, Chapter 8 introduces several techniques for live multi-device UI design.

Finally, Part V gives an overview of some extensions that were built on top of the GUMMY design tool and discusses future research directions. We then draw the conclusions in Chapter 10.
Figure 1.1: The techniques and tools that contribute to answer the three proposed research questions.
Chapter 2

Related Multi-Device User Interface Design Tools

2.1 Introduction

The work presented in this dissertation builds further on existing multi-device user interface design research. In this Chapter we provide an overview of the important influences and state-of-the-art approaches that are of interest for the work presented in this dissertation. The goal of this literature overview is to identify aspects that can help us to create better multi-device user interface design tools. In Chapter 3, we further investigate how designers would like to see these aspects in design tools.

This Chapter surveys various multi-device design approaches. The next Section discusses how GUI toolkits contribute to multi-device UI design. GUI toolkits are software libraries that contain reusable UI controls for programming user interfaces. In Section 2.3, we discuss Model-Based User Interface Design (MBUID), which has been the subject of research for many years. MBUID allows designers to specify a user interface by means of one or more high level models. From these models, user interfaces will then be generated (semi-)automatically for every target platform. Section 2.4 discusses HTML 5, which is becoming increasingly popular to create web-based solutions for various devices. Finally, prototyping tools for creating multi-device high-, mid- and low-fidelity prototypes are discussed in Section 2.5.
2.2 GUI Toolkits

Toolkits are the primary software libraries for programming GUIs. According to Dix et al. [DFAB03], a toolkit provides a programmer with a set of ready-made interaction objects - alternatively called widgets - which she can use to create her application programs. Examples of these interaction objects are WIMP widgets such as windows, buttons or checkboxes. All widgets can be tailored to the specific situation in which they are invoked by the programmer. For example, the text of a button widget could be a parameter which the programmer can set when creating the button.

The widgets provided by a toolkit abstract the implementation of a user interface from the drawing system of the underlying windowing system. This implies that designers can program user interfaces without having to know the low level details of the windowing system.

The abstraction level provided by a GUI toolkit can also contribute to the portability of a software package across multiple devices. This is the case in virtual toolkits [Mye95], which are toolkits providing virtual widgets that can be mapped to platform-specific widget implementations on each targeted platform. A user interface that is specified using a virtual toolkit will run without change on different platforms and still look like it was designed with that platform's widgets. For example, the virtual toolkit might provide a virtual menu widget, but maps to a Macintosh or Windows menu depending on which machine the application is running.

Some examples of virtual toolkits are the Abstract Window Toolkit (AWT)[awt], Swing [swi], Qt [qt] and Gtk [gtk]. The AWT and Swing toolkits are both typical Java toolkits, running on most Java platforms. Qt is an open source toolkit that can run on the majority of desktop platforms and on various mobile platforms like Windows Mobile, Symbian and MeeGo. The Gtk toolkit can be used for developing desktop systems that run Linux, Windows or Mac OsX.

Virtual toolkits simplify the development of multi-device user interfaces because the same user interface can be deployed to all devices where the toolkit is available. A limitation of this approach is that it is difficult to apply the same UI to target devices with widely varying input and output capabilities using toolkits. This would impose structural GUI changes that go beyond mapping virtual widgets to platform-specific widgets [Nul02]. For example, a GUI dialog designed for a large screen will have to be subdivided in multiple dialogs or tab pages when running this UI on a small screen.
2.3 Model-Based User Interface Design

Model-Based User Interface Design (MBUID) is probably the most explored approach in the context of creating multi-device graphical user interfaces. The goal of this work is generally to allow designers to specify interfaces at a high level of abstraction. This high level specification can then be transformed (semi-)automatically into a running user interface. This approach is called “Model-Based” since these high level specifications are often models such as task, data or domain models. Calvary et al. [CCT+03] describe different levels of abstraction for specifying these models (Figure 2.1):

- **Tasks and Concept (T&C) level**, describing the various interactive tasks to be carried out by the end user and the domain objects that are manipulated by these tasks.

- **Abstract User Interface (AUI) level**, describing user interfaces independent of the interaction modality on the target platform. This description is composed of Abstract Interaction Objects (AIOs) such as “action objects” or “selection objects”.

- **Concrete User Interface (CUI) level**, describing user interfaces independently of any markup or programming language. This description consists of Concrete Interaction Objects (CIOs) such as “buttons” or “combo boxes”. The models at the CUI level are often called presentation models.

- **Final User Interface (FUI) level**, representing the operational UI. A FUI is built out of Final Interaction Objects (FIOs) such as a “java.awt.button” or “<html:input type='select'></html>”.

To produce a user interface from abstract models, transformations have to be employed. These transformations are able to gradually transform the top level tasks and concepts model into a corresponding AUI, followed by a CUI and finally a FUI. For example, the task “select a month” can be transformed into the “selection object” AIO. This AIO can then be further transformed into a “combo box” CUI and finally into an HTML select widget.

Some well-known models that are used in MBUID are domain models, task models, dialog models and presentation models. The domain and task model can be situated at the tasks and concepts level. These models define the type of objects that need to be presented by the interactive system and the tasks the user executes. The presentation model and dialog model are situated at
the CUI level. These models define the look of the user interface dialogs and the logical flow between these dialogs. A popular way to specify user interface models is by means of XML-based **UI Description Languages (UIDLs)**, which we discuss in detail in Section 2.3.1.

Specifying and transforming models are typically supported by a chain of **tools**. According to Szekely [Sze], there are three types of tools that play an important role in MBUID: modeling tools, automated design tools and implementation tools. Designers use **modeling tools** to create the models. The **automated design tools** are used to perform certain design activities that designers either choose or are forced to delegate to the system. For example, automated design tools can automate the transformations between models.

The third type of tools, **implementation tools**, will transform the models into an executable representation that is perceived by the end-users. Implementation tools are often developed by means of Model-Based toolkits.

This section first discusses various XML-based User Interface Description Languages. We then describe three types of tools that support multi-device user interface design: modeling tools, automated design tools and Model-Based toolkits. Afterwards, we discuss the advantages and disadvantages of
2.3 Model-Based User Interface Design

2.3.1 XML-Based User Interface Description Language

A User Interface Description Language (UIDL) [SV03] consists of a high-level computer language for describing the characteristics of a UI. It can be considered as a common way to specify a UI independently of any target language (e.g., programming or markup) that would serve to implement this UI. The most common UIDLs are based on the eXtensible Markup Language (XML), which is a way to encode documents in a machine-readable format. There exist UIDLs for describing UIs at the different levels of abstraction that we introduced by means of Figure 2.1: Tasks and Concepts (T&C), Abstract User Interface (AUI) and Concrete User Interface (CUI).

We believe that the richness of a UI is proportional with the expressive power of the underlying UIDL that is being used to realize this UI. Especially UIDLs that describe the UI presentation model at the CUI level contribute to the expressiveness of a UI. A UI presentation typically describes the widgets that are shown in a UI and their layout. There are two extreme approaches to define a UI presentation model: the common denominator and the meta-widget set approach. While the former identifies a fixed set of widgets that can be used on most devices, the latter allows designers to extend this widget set dynamically. A language that uses a meta-widget approach set can thus support different widgets for different platforms.

In this section, we give an overview of existing XML-based UIDLs. This overview is not exhaustive since there exist many more XML-based UI languages. The goal of this overview is to exemplify the mechanisms that UIDLs use for targeting multiple platforms.

Toolkit-specific XML Languages

Some toolkits allow designers to specify UIs by means of an XML-based UIDL. The tags used in these XML-languages map directly to the widgets that are available in the underlying toolkit. For example, when using a menu tag, this tag will be mapped to a menu widget. Toolkit-specific XML languages provide an additional level of abstraction on top of a GUI toolkit. This makes it possible to reuse a UI specified in an XML language across various implementations of the same toolkit. For example, a GTK UI specified in Glade XML [gla] can be used to access this GTK UI from C/C++, Java, C#, Python and Perl. Also UIs specified in Qt's UI XML [qtx] format can be accessed from Java and C++.
We consider toolkit-specific XML languages as very expressive. Due to the tight coupling with the underlying toolkit, these XML languages allow designers to use the same widgets and manipulate the same properties as the actual toolkit. On the other hand, this tight coupling constrains the diversity of target platforms that can be reached. Only the platforms containing a toolkit implementation can be targeted. The requirement to have the same toolkit on every device also induces some restrictions [Nil02]. When using the same toolkit on every device, it would be difficult to have an optimal speed of execution or user experience on all these devices. Moreover, the resulting user interfaces often have a similar look and feel on every device and are not consistent with the other applications on these devices.

**XML User Interface Language (XUL)**

The XML User Interface Language (XUL) [(MD06] is Mozilla’s XML-based language for describing window layout. It is a language designed specifically for building portable user interfaces. XUL allows designers to implement and modify interfaces quickly and easily. The resulting user interface can be used in four different ways: as a Firefox extension; through a XULRunner application, which is a packaged version of the Mozilla platform for realizing XUL applications; as a XUL package, where the designed UI acts like a separate application in a separate window; and as a Remote XUL application, where XUL code is available on a web server and can be opened in a browser just like you would do with any other web page.

XUL can be combined with other languages. The style of the user interface elements can be specified by means of a style sheet. The style sheet contains information such as the fonts, colors, borders and size of elements. XUL user interfaces can use Javascript to specify interactive behaviour. For declaring the behaviour of XUL widgets, XUL provides XBL (eXtensible Bindings Language). This language makes it possible to create custom components and to change the behaviour of existing elements. For example, XBL could be used to change the way the pieces of a scroll bar function.

We consider XUL also as a very complete and expressive language, which can be extended with custom widgets using XBL. It can even change the behaviour of already existing XUL widgets. However, XUL has the same disadvantage as toolkit-specific XML languages. It will be difficult to guarantee an optimal speed of execution or user experience on every device when using XUL.
X-Forms

XForms [DKMR03] is an XML description format that represents "the next generation of Forms for the Web". The three main components of XForms are:

- the **model**, describing the structure of the data that will be displayed in the form;
- the **instance data**, containing the actual data that is specified conform the model;
- the **abstract user interface**, consisting of the controls that appear in the form, how they are grouped together and what data they display. XForms supports UI controls such as select (choice of one or more items from a list), trigger (activating a process), output (display-only of form data), secret (entry of sensitive information),

Most XForms implementations allow designers to specify how the abstract user interface controls should map on concrete UI elements like radio buttons, checkboxes, ... The fact that the mapping on concrete UI elements can be specified separately makes XForms more suitable for multi-device UI design than toolkit-specific UIDLs or XUL. For every targeted device, a different set of mappings can be used, resulting in a different UI for every targeted device. The major limitation of XForms is that it can only be used to specify form-based UIs. It is also difficult to apply XForms to other UIs than web UIs.

Personal Universal Controller (PUC)

The PUC specification language [NM09] focusses on describing the functional aspects of appliances, which directly influences the design of interfaces for them. This language allows designers to describe the features that an appliance has and how these features relate to each other. It can also be used to describe how contents can be exchanged across devices. This is very useful for creating interfaces that aggregate functions from multiple appliances.

From a PUC specification that describes the functions and features of an appliance, a UI can be generated to control this appliance. These user interfaces are typically generated on mobile devices. The PUC rendering engine [NMH+02] is able to generate user interfaces for various devices like PDAs or Palm PCs. An interesting extension of this rendering engine is UNIFORM [NMR06], which can enforce consistency across the remote control
interfaces of different appliances. This makes it easier for end-users to transfer the knowledge from one remote controller for a certain appliance to another controller for another appliance.

We categorize PUC as a UI specification language at the tasks and concepts level. It allows designers to define the functional concepts of the underlying appliances, which will then be used to generate usable remote controllers for these appliances. Due to its high level of abstraction, PUC has the potential value to reach a broad set of computing platforms.

User Interface Markup Language (UIML)

UIML [HA08] is an XML-based meta-language that allows designers to describe a UI's structure, style, behavior and content. It does not specify any particular widgets or widget sets, but allows to define custom vocabularies with mappings from the desired abstractions and properties onto concrete widgets and widget properties. For example, a UIML UI description could use an abstraction with the name “button”. When rendering this UI on a Java platform, a Java Swing vocabulary will be used. This vocabulary will then map this button to a Java Swing button. On the other hand, when targeting a web-based UI, an HTML vocabulary will be used. This vocabulary will map the button to a HTML button.

This vocabulary approach makes UIML a very expressive language. For every platform, a specialized widget set can be defined to facilitate interface creation on that platform. UIML can use the native toolkit on every targeted device, which contributes to the user experience and speed of execution on these devices. Furthermore, UIML vocabularies also facilitate the integration of custom controls.

Today, there exist several UIML renders for various platforms. Some examples are .NET and compact.NET [LTVC07], iPhone\(^1\), Android\(^2\) and Java [LTVC07].

UsiXML

UsiXML [LV04] is an XML language that facilitates the design of user interfaces for multiple contexts of use. In contrast to most other approaches mentioned in this section, it encapsulates different models and has explicit support for relating different models, transforming models and selecting parts

\(^1\)https://code.launchpad.net/~farinelli-fabrizio/+junk/UIMLiPhone
\(^2\)http://code.google.com/p/android-uiml/
of a model. UsiXML allows designers to describe user interfaces on the four layers of abstraction that are introduced by the Cameleon Reference framework [cal03]: a domain model, a task model, an abstract user interface model and a concrete user interface model. Besides these models, it can also express the context model, mapping model and transformation model.

We consider the UsiXML language as a very complete language that allows designers to combine different user interface models. It is a language that can be used to support a complete Model-Based User Interface Design approach. The weak point of this language is its Concrete User Interface (CUI) model. In contrast with UIML there is no mean to introduce custom abstractions. This induces that the resulting user interfaces always contain the same widget types.

MARIA XML

MARIA XML [PSS09] is a recent model-based language, which allows designers to specify Abstract User Interfaces and Concrete User Interfaces. MARIA’s AUI is independent of the interaction modalities and concrete languages that are provided at the CUI level. The main components of the Abstract User Interface are one or multiple presentations, a data model, and a set of external functions. Each presentation contains a number of user interface elements (interactors) and interactor compositions (indicating how to group or relate a set of interactors), a dialogue model describing the dynamic behaviour of such elements, and connections indicating when a change of presentation should occur. The interactors are classified in abstract terms: edit, selection, only output, control, interactive description, etc.

At the CUI level, MARIA provides several concrete languages that are independent of the implementation language (Figure 2.2). Each of these languages targets a group of platforms that have similar interaction resources. For example, the mobile multitouch platform language targets mobile devices that have similar multitouch interactions than Apple’s iPhone. Maria also provides concrete languages for desktop PCs and voice based systems.

In summary, MARIA is an XML-based UIDL for describing user interfaces at the AUI and CUI level. One AUI description can be refined in one CUI per category of target platforms. For each of these categories, MARIA provides a concrete language for defining these CUIs. These concrete languages contain the widgets that are best suited for a certain category. This approach allows designers to target devices with widely varying interaction capabilities. The limitation of this approach is that it is difficult to add new user interface
components to these concrete languages without having to update the language itself.

2.3.2 Modeling and Automated Design Tools

Modeling tools allow designers to create and relate models. Most of these tools also have an automated component, which designers can use to do transformations across models.

Figure 2.2: The different levels of abstraction in MARIA XML. This image is courtesy of Paterno al. [PSS09].
2.3 Model-Based User Interface Design

One of the first Model-Based User Interface Design systems is Mastermind [SSC+96]. This system allows designers to specify a user interface by means of application, task and presentation models. As the available screen size on the targeted device becomes smaller, these models are adapted by progressively removing less important information. Mastermind can adapt interfaces to fit the screen sizes of workstations, laptops and the first generation handheld devices.

In the ITS system [WBB+90], user interfaces have to be defined at four layers of abstraction: (1) data input and output; (2) the application’s dialog flow; (3) style rules that map the dialogs to concrete widgets, and (4) style programs that compute the layout of these concrete widgets at runtime. This separation of concerns make that ITS can be used for multi-device user interface generation. For every targeted device, one or more layers can be updated without having to touch the other layers. For example, one can use a different set of style rules for a workstation and a handheld device.

TERESA (Transformation Environment for inteRactivE Systems represenTAtions) [Pat00, MPS03] (Figure 2.3) is a tool allowing designers to specify a task model of their interactive system using the ConcurTaskTrees (CTT) modeling language. For every task in the model, it can be specified on which device it should run (Figure 2.3 (a)). From this task model, TERESA will automatically generate an Abstract User Interface (AUI). This AUI is built out of several Abstract Interaction Objects (AIOs) which are composed through a number of different operators. Per target platform, the designer can specify how AIOs and operators map on concrete user interface elements (Figure 2.3 (b)). Using these mappings, TERESA can now generate the final user interfaces for every computing platform.

The MARIA tool [PSS09] is built on top of the MARIA XML language that we described in the previous section. This tool allows designers to define the transformations (Figure 2.4) between the MARIA Abstract User Interfaces and the specific Concrete User Interface. These transformations can be expressed visually by drawing connections from the AIOs to the corresponding CIOs. Designers can specify both one-to-one and one-to-many transformations.

Florins et al.[FV04] present a set of graceful degradation rules that can be used by model-based tools to generate multiple platform-specific user interface from one model. Using these rules, one general model can be transformed in multiple models, i.e one model per target platform. The resulting models are tailored for the targeted platforms and can be easily turned into user interfaces for these platforms. There exist graceful degradation rules for the tasks and
concepts level, the AUI level and the CUI level. Some example rules are:

- at the T&C level, a task will be deleted when this task requires resources that are not available on the target platform. For example, the task “capture a photo” will be removed for devices that have no camera.

- at the AUI level, the tasks that can be achieved at the same time will be reorganized. For example, on a device with a small form factor less tasks can be displayed at the same time. Tasks will then be reorganized in smaller units that are shown sequentially.

- at the CUI level, a widget can be replaced with another widget. For example, this can be the case when a widget takes too much space on the target device.

2.3.3 Model-Based Toolkits

Model-Based toolkits are implementation tools according to the categorisation of Szekely [Sze]. This type of tools is able to turn models into running user interfaces. Compared with traditional GUI toolkits (Section 2.2), designers have to program user interfaces using abstract objects or concepts rather than concrete widgets. For example, Model-Based toolkits could require designers to create a *Choice* AIO instead of a concrete *Combobox* widget. The toolkit

![Figure 2.3: The Transformation Environment for interRactivE Systems representAtions (TERESA). This image is courtesy of Mori et al. [MPS03]](image)
will then compute the graphical representations of these abstractions at runtime, depending on the input and output capabilities of the target platform where the UI is being executed.

Supple [GWW10] is a Java toolkit allowing designers to program a UI at the tasks and concepts level. Designers have to program their own domain objects, which can then be combined into a complete domain model of the user interface. For example, the domain model of a classroom is visually represented in Figure 2.5 (a). All domain objects shown in this figure have to be programmed in Java code. From a domain model, Supple will generate an optimal user interfaces for each targeted platform. Supple’s optimization algorithm can be considered as one of the most powerful UI generation algorithms which can produce aesthetically pleasing UIs. For example, Figure 2.5 (b)

Figure 2.4: The MARIA tool for transforming the AUI in a desktop CUI. This image is courtesy of Paterno et al [PSS09].
shows two GUIs that Supple generated for two different screen sizes.

The COMET (COntext Mouldable widgET) toolkit [DCC08] allows designers to program user interfaces by means of Comets. In short, a Comet groups representations that support a particular user task. For example, a set of radio buttons, a combo-box, a list and a pie menu are grouped in a Comet that supports the user task “select one option from a list of options”. At run-time, the most appropriate representations for every user task are aggregated into a suitable UI for every targeted platform. Designers can select and change the representations for every task while the interface is running.

2.3.4 Discussion

In the context of multi-device user interface design, model-based user interface design has two big advantages. The first advantage is that designers have to specify a user interface only once by means of an abstract specification. From this abstract specification, tools can then automatically generate user interfaces for every desired target platform. This is shown graphically in Figure 2.6, where one AUI specification is transformed into two CUIs and FUIs for two target platforms. The other advantage of abstract user interface specification

![Image](image1.png)

(a) Visual representation of a domain model used in Supple

![Image](image2.png)

(b) User interfaces generated with Supple from the domain model in (a)

Figure 2.5: The Supple classroom example. This image is courtesy of Gajos et al. [GWW10]
approaches is that they are able to target a broad range of platforms. Describing a user interface at the T&C or AUI level results in a description that can be transformed into a running user interface on more varying platforms compared with multi-device toolkits.

The major disadvantage of Model Based User Interface Design is that designers need to use complex modeling languages to specify high level models. When specifying these models, it is very difficult to predict how the resulting UI will look like. The major reason for this is that model transformations use complex algorithms and heuristics which may result in unexpected outcomes [MHP00]. The GrafiXML [MV08] MBUID tool is an exception, allowing designers to design user interfaces visually. For this reason, we consider this tool as a High-Fidelity Prototyping tool and discuss it in detail in Section 2.5.

2.4 HTML 5

With the introduction of HTML 5 [HTM09], the web can be turned into a powerful multi-device design platform. Contrary to its predecessors, HTML 5 has support for new advanced features such as video, geometric shapes, cross-document messaging and drag and drop. Once defining a user interface in

![Diagram](image)

Figure 2.6: In multi-device user interface design, one abstract model can be transformed into multiple platform-specific UIs. This Figure originates from Calvary et al. [CCT+03], redrawn by the author.
24 Related Multi-Device User Interface Design Tools

HTML 5, these interface can be accessed from any device that has access to the internet and a suitable web browser.

Although HTML 5 is able to reach a broad set of computing devices, this approach did not enforce the multi-device design breakthrough [Sie11b, Sie11a]. One reason for this is that it is difficult for HTML 5 UIs to reach the same look, performance and behaviour as native apps. For example, the two most popular types of mobile phones, iPhone and Android, strongly recommend to design user interfaces (for their apps) in the native toolkit rather than using HTML 5. Another reason concerns the slow standardisation of HTML 5. When new technology becomes available, it takes a long time before it is standardized and implemented in web-browsers.

Since HTML 5 is not (yet) the solution for multi-device UI design, we consider it as an type of target platforms. For example, designers should be able to design user interfaces for a native toolkit and a web-based version. This is not yet possible with HTML 5, which can be used to create web-based solutions only.

2.5 Prototyping Tools

Multi-device GUI prototyping tools allow designers to design user interfaces visually. This type of tools exist on multiple levels of fidelity: low-, high- and mid-fidelity. This section introduces these three levels of prototyping tools.

2.5.1 Low-Fidelity Prototyping Tools

Low-fidelity prototyping takes place during the early design stages, where designers need the freedom to sketch rough design ideas quickly and need the ability to test designs by interacting with them [LM01]. In this phase, designers do not focus on issues such as colors, fonts, and alignment, which are more appropriate later in the design.

Although Lin et al. [Lin05] found that there is a need for multi-device low-fidelity prototyping, there is only little tool support available. Most existing low-fidelity tools focus on only one particular device. Some example single-device low-fidelity prototyping tools are Silk [LM01], Denim [NLHL03] and Microsoft SketchFlow [Mic11b], which allow designers to sketch desktop interfaces. Besides sketching, there also exist various wireframing tools like Justinmind [jus], AXURE [axu] and Protoshare [pro].

The most notable multi-device low-fidelity prototyping tool is Damask [LL08], which supports interface sketching for three different platforms: desktop PCs,
mobile devices and speech based interfaces. While sketching these UIs, designers can make use of reusable UI patterns that are presented in a pattern repository. In order to reuse parts of one device’s UI in the UI of another device, Damask employs a fixed set of UI pattern transformations and a widget mapping model. This is comparable to the rule-based mappings that exist in most automatic UI generation systems.

2.5.2 High-Fidelity Prototyping Tools

Compared to low-fidelity prototypes, high-fidelity prototypes offer more realistic interactions and are better at conveying the range of design possibilities [wal02]. This type of prototypes focuses around the concrete look and feel of the intended application rather than broad design explorations. High-fidelity prototypes can be created with almost any commercially available design tool including Microsoft Expression Blend [Mic11a], Qt Designer [qt], Adobe Flash Catalyst [Ado11] and Netbeans Forms Editor [Net11]. However, these tools cannot interoperate with each other and are mostly constrained to one specific GUI toolkit or platform. This makes it difficult to create UIs for a set of mutually exclusive target platforms.

Up till now, only limited research effort has been done to facilitate multi-device high-fidelity prototyping. The most notable multi-device high-fidelity design tool is GrafiXML [MV08]. GrafiXML allows designers to create an interface visually using direct manipulation operations. The designed GUI is created visually and is considered as platform independent. This is because the GUI design is saved as UsiXML (Section 2.3.1), which can be transformed into languages such as Java, XUL or (X)HTML; or it can be rendered using UsiXML interpreters for toolkits such as TCL/Tk and Adobe Flash.

A limitation of GrafiXML is that it does not show the visual layout of the UI design in its design environment. Instead, it shows a Java platform independent layout. For example, updating an Adobe Flash GUI happens by updating the platform independent Java design. This makes it a bit more difficult for designers to map the presentation that is used inside the tool to the final GUI.

2.5.3 Multi-Fidelity Prototyping Tools

Besides low- and high-fidelity prototyping tools, there also exist prototyping tools that combine both levels of fidelity. One approach is to start sketching a low-fidelity prototype and to support the transition towards a high-fidelity prototype. This is the basic idea behind SketchiXML [CKV07]. This tool allows
designers to start sketching a low-fidelity prototype, which can be transformed into a medium-fidelity prototype. A medium-fidelity prototype provides a rough representation of the used UI elements, but still platform independent (i.e. it does not show which window manager, toolkit or environment that will be used in the final design). From this medium-fidelity prototype, a high-fidelity prototype can be obtained for every target platform. All widgets will be visualized as they will appear on each targeted platform. Compared to traditional automatic GUI generation systems, SketchiXML aligns better with traditional design approaches. The major limitation of SketchiXML is that it can only support a limited number of widgets. In addition, it would also be difficult to prototype GUIs for multiple devices that have widely varying interaction capabilities. This would imply the creation of a new low-fidelity prototype for every target device, which is also time consuming.

Canonsketch [CN05] also provides tool support for creating medium-fidelity prototypes. It allows developers to create user interfaces by means of the Canonical Abstract Prototypes (CAP) [Con03] notation. This notation resembles wire frame schematics and contains a standardized set of universal abstract components. Each such Canonical Abstract Component has a specific abstract interactive function, such as toggling a state, creating information, or providing a notification. These interactive functions are then represented by simple symbols. Canonical Abstract Components model not only the interactive functions to be provided by a user interface, but also the position, size, layout, and composition of the user interface features. From the mid-fidelity prototypes created using CAP, a working user interface can be generated for the targeted platforms. Although Canonsketch avoids the construction of complex models, it shares some drawbacks with traditional interface generation systems (Section 2.3): designers have only limited control about the final presentation of the GUI and it is also difficult to generate aesthetically pleasing UIs.

Another type of multi-fidelity prototyping tools are mixed-fidelity tools. In general, these tools produce prototypes that are high fidelity in some respect and low-fidelity in others [MCP+06]. For example, it can be a responsive sketched prototype that allows designers to explore the rich interactivity of the future application. One notable tool in this category is the mixed-fidelity framework proposed by de Sa et al. [dSCDR08]. This framework allows designers to create mixed-fidelity prototypes for mobile- and desktop-systems. The major drawback of this system is that it is constrained to solely one (custom) UI toolkit that only supports a limited set of interactors.
2.5.4 Discussion

Low-fidelity prototyping tools are very useful for exploring ideas during the early design stages. Low-, mid- and mixed-fidelity prototypes help to frame and improve these design ideas [KHT06]. However, low-fidelity prototypes also have some limitations. The mock-ups that are created in this stage differ from the final UI, which may lead to unrealizable design ideas [Hol05]. This is not the case with high-fidelity prototyping, where the resulting prototypes map directly to the final user interface.

2.6 Conclusion

From the techniques discussed throughout this Chapter, Model-Based User Interface Design tools can target the broadest set of computing platforms. These systems allow designers to specify user interfaces at a higher level of abstraction. This abstract specification can then be transformed (semi-)automatically into suitable user interfaces for every targeted platform. Despite the fact that MBUID tools simplify the UI development process, these tools are not oriented towards UI designers. The input models and transformations are mostly too complex for people with a non-technical background.

The High-Fidelity prototyping tools that we discussed are better suited for UI designers. In these tools, designers can work on a concrete representation of the user interface. Working on a concrete UI allows designers to polish UIs until it reaches the desired aesthetic quality. The limitation of most of these High-Fidelity prototyping tools is that designers can only create UIs for one toolkit.

In Chapter 4, we combine the strengths of Model-Based User Interface Design tools and High-Fidelity prototyping tools by means of the GUMMY design tool. GUMMY is our own multi-device design tool that allows designers to work on a concrete UI representation and to target a broad set of computing platforms. Chapter 3 proposes an initial version of our design environment. This design environment will be used to further refine the needs that designers have concerning multi-device user interface design.
Part II

Creating Multi-Device
Graphical User Interfaces
Chapter 3

Multi-Device Design Tool Needs

3.1 Introduction

Chapter 2 showed that there exist many approaches to realize multi-device user interfaces, all having their own way for specifying designs. Some example design methods are GUI programming using toolkits (Section 2.2), abstract models (Section 2.3), and visual prototyping (Section 2.5). Although most of these approaches claim to simplify the creation of multi-device user interfaces, it remains an open question whether these techniques are potentially useful for user interface designers.

In this Chapter, we are going to investigate what kind of tools and techniques designers need for creating multi-device user interfaces. This additional design tool needs analysis is necessary, because UI designers will be the primary users of the tools presented in Chapters 4, 5, 6 and 8. As discussed in Chapter 1, we adopt the definition that UI designers are skilled at prototyping the look of interfaces rather than programming its interactive behavior [MPN+08].

This chapter discusses the details of our analysis. In section 3.2, a set of lessons that we learned from the literature survey in Chapter 2 are presented. Section 3.3 introduces an initial multi-device environment that takes these lessons into account. Next, we used this environment to stimulate discussion with user interface designers about multi-device design tools. From these discussions, we then extracted a set of needs that will be taken into account throughout this dissertation.
3.2 Lessons Learned from Literature

The main objective in this dissertation is to create a multi-device design environment that is suitable for user interface designers. From the literature that we explored in Chapter 2, we deduced four lessons that can help to make multi-device user interface design environments suitable for UI designers:

1. abstract models should be avoided, otherwise designers will be forced to learn new and often complex modeling languages [MHP00]. Abstract models will also raise the expertise threshold that is needed to use a design tool. Furthermore, the time needed to create one high level model usually exceeds the time needed to design each UI by hand.

2. visual UI design on a concrete UI representation is beneficial for UI designers. Working on a concrete high-fidelity prototype reduces the mental burden on the designer [Car88] and helps to polish UIs easily.

3. automation should be comprehensible for UI designers. This is often not the case in MBUID (Section 2.3), where automation is used to transform one model to another and to generate the final user interface. These automation strategies are often based on complex Artificial Intelligence (AI) algorithms and heuristics. The major drawback of this type of automation is that it is difficult to control for UI designers [MHP00].

4. design tool switches should be avoided because different styles for designing interfaces can disorient designers. Since current high-fidelity design tools (Section 2.5) are mostly constrained to design user interfaces for one specific toolkit, designers are forced to switch across multiple mutually exclusive design tools to target a broad set of computing platforms.

These four lessons help to combine the strengths of two different types of multi-device design tools: high-fidelity prototyping tools and MBUID tools. The former allows designers to work on a concrete UI representation but can only target a limited number of target platforms. This means that designers would need to switch between different high-fidelity prototyping tools to target multiple platforms. MBUID tools can typically target a broad range of platforms, but are using models that are not always oriented towards UI designers. Moreover, the transformations from high-level models to concrete UIs are difficult to control and may result in unexpected UIs.

The next section will introduce an initial design environment that takes the lessons that are presented above into account.
3.3 Initial Multi-Device Design Environment

As a first step in the multi-device tool requirements analysis, we created an initial multi-device design environment that takes the lessons that we learned from literature (Section 3.2) into account. This environment embodies our initial ideas about multi-device user interface design and how a design tool could improve the creation of multi-device UIs. The focus of the environment is to stimulate discussion about these ideas, rather than having a complete working design tool. This section presents our envisioned multi-device design approach, followed by a description of a design environment supporting this approach.

3.3.1 Design Approach

To support the design of multi-device user interfaces, we propose a new kind of design process. This process does not use abstract models like in MBUID approaches (Section 2.3). Instead, designers can always work on the concrete representation of their designs. Figure 3.1 depicts the five most important steps in our approach:

1. As a first step, the user interface designer selects the new target platform for which they want to design a user interface. For example, do they want to design a GUI for a mobile phone or a desktop computer?

2. A multi-platform design tool automatically updates a design workspace containing a toolbox with widgets that are suitable for designing interfaces for the selected platform.

3. Based on existing user interfaces, that were previously created for the same application on other devices, the design tool automatically generates an initial design for the selected target platform. When there are no useful existing user interfaces available, the designer creates a user interface from scratch.

4. The designer can refine the initial design until it fits her vision.

5. When reaching the desired quality, the designer saves this design. In a new iteration, which starts after this step, this design can be used to generate an initial design for another device.

In summary, the proposed design process is a combination of manual steps, completed by the designer (Figure 3.1, left), and automatic steps, executed by the design tool (Figure 3.1, right).
3.3.2 Design Environment

To support the design approach described in the previous section, we created an initial design environment. This tool allows designers to create form-based user interfaces for .NET on mobile and desktop platforms. We choose these two platforms because of their different characteristics. The mobile UIs are intended to run on a portable device with a small screen that supports interaction by means of a stylus. Desktop UIs should run on a regular PC, which has a larger screen and traditional input devices like mouse and keyboard.

The workspace of our design tool is structured similar to traditional GUI builders like Visual Studio Forms designer or Glade (Figure 3.2 (a)). Designers can drag components from a toolbox onto the canvas and manipulate the properties of these components using the properties panel. Components can be repositioned and resized using direct manipulation.

A designer first designs a GUI for one of the two supported platforms (i.e. .NET mobile or desktop). When finishing this design, it can be transformed automatically into a design for the other platform. For example, the designer

![Diagram](image)

Figure 3.1: Our initial idea about a multi-device design process. The steps in this process are divided between the designer (left) and design tool (right).
3.3 Initial Multi-Device Design Environment

Figure 3.2: The proof-of-concept version of a multi-device design environment, which is used to stimulate discussion about multi-device GUI design tool support. This prototype environment allows designers to create a GUI for one platform (a) and to transform this GUI into a new GUI for another platform (b and c).
starts with the creation of the mobile user interface (Figure 3.2 (a)). Next, she selects the desktop platform (Figure 3.2 (b)). This action will switch the design tool to show a desktop design that is automatically generated from the mobile design (Figure 3.2 (c)). The designer now has the opportunity to fine tune this design until the desired quality is reached.

To transform a user interface from one platform to another, we use a rule-based transformation algorithm. This algorithm has two steps:

- **mapping** widgets from one platform-specific toolkit to another. Figure 3.2 (c) illustrates these mappings: (1) the button in the mobile design is transformed into a button for the desktop design; (2) the progress bar is also transformed into a desktop version of this widget.

- **scaling** widgets according to the available screen size. By default, the size of the widgets is linearly interpolated with the screen sizes. This is inspired by the *artistic resizing* tool [DCTV05], which allows graphical designers to interpolate multiple vector-based images into a new image with a specific size.

### 3.4 Interviews with Designers

As a next step in our multi-device design tool needs analysis, we conducted formative interviews with five designers across five companies. Four were interviewed in their offices, one during a conference about user interface design. Table 3.1 lists the types of companies for which our participants worked and the devices for which they designed.

As a first goal of these interviews, we wanted to discover the problems designers currently experience while designing user interfaces for multiple computing devices. We asked the interviewed designers to demonstrate their recent projects involving user interface design for multiple devices. The participants did this by introducing the project, showing the design artefacts that were used in this project and the resulting interfaces. During these discussion, we also asked about the problems that showed up while designing the user interfaces. Finally, we asked the participants about how current tools can be improved to overcome these problems.

The second goal of the interviews was to discuss our ideas for a multi-device multi-device UI design tool. This was done by showing our initial multi-device design environment (Section 3.3). We asked them for their reactions and to speculate on how useful such a tool would be. In particular, we wanted to receive feedback about the following two aspects of the design tool:
3.4 Interviews with Designers

Table 3.1: The participants for our study, their companies and the platforms they designed for.

<table>
<thead>
<tr>
<th>Designer</th>
<th>Type of company</th>
<th>Desktop PC</th>
<th>Tablet PC</th>
<th>PDA</th>
<th>Smartphone</th>
<th>Interactive</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>Web Design</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D2</td>
<td>Design Consultancy</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>D3</td>
<td>Design Consultancy</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>D4</td>
<td>Mobile and Desktop Design</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D5</td>
<td>Multimedia Design</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td>√</td>
</tr>
</tbody>
</table>

- the expressive match, do designers like the idea of creating multi-device user interfaces visually?
- can automatic transformations help designers to manage multiple device-specific GUIs?

Following, we discuss the results of the interviews that we conducted. We categorized these results using the three research questions of this dissertation: RQ1 (creating), RQ2 (managing) and RQ3 (optimizing) multi-device user interfaces. While discussing these results, we also refer to existing surveys and research results that support our findings.

3.4.1 Creating

This category describes issues and suggestions designers have about how they would like to create multi-device user interfaces. All interviewed designers were convinced that designing multi-device user interfaces visually in a GUI builder makes sense. This closely resembles to the way they are currently designing interfaces in existing design tools.

One designer (D2) had experience with a high level XML language for specifying multi-device interfaces. This language allowed to specify a UI on a high level of abstraction, and could be used to automatically generate suitable UIs for every targeted platform (mobile and desktop in this case). The designer was not convinced about such approach, because it was very time consuming to
create such an abstract description and the resulting interfaces were not of the desired aesthetic quality. Furthermore, this designer argued that it would be rare that multiple target platforms support exactly the same abstract model. This result is supported by the findings of Myers et al. [MHP00]. They state that there has to be a direct and straightforward mapping between the GUI specification and the final GUI. When there is no such mapping, it would be difficult for designers to predict how a change in the input specification would change the final UI.

Several designers noted the limitation of the presented tool to target only form-based UIs. Web designer D1 would like to work with customized interface controls since this allows them to emphasize certain parts of a website. A common desire of designers D2 and D3 is to reuse components that were created with existing design tools inside a multi-device design environment. They are convinced that the integration of custom interface controls could result in more stylish and aesthetically pleasing interfaces. This result is consistent with the findings of a recent large scale study of the working practices of 259 interface designers [MPN+08]. This study showed that 76% of the studied designers very frequently reuse custom interface components in later projects.

In summary, we can conclude that designers prefer to design UIs graphically. Besides form-based components, they would like to extend this approach with custom GUI controls. This can result in more stylish and aesthetically pleasing interfaces. Part II of this dissertation will take this suggestion into account.

3.4.2 Managing

In multi-device GUI design, several device-specific GUIs are created for the same application or service. For example, a movie rental application can have a specific GUI for an interactive television, a desktop PC and a mobile device. This category discusses issues and suggestions designers have about managing (i.e. updating, extending, maintaining) these device-specific GUIs.

All UI design professionals rejected a tool that automatically transforms a whole UI from one platform to another. They argued that mobile device UIs often support different requirements than desktop UIs. When automatically transforming a complete UI from one device to another, it would be difficult to take these differences into account.

D1 and D5 noted that partial transformations could help to reduce the amount of duplicated design effort across device-specific UIs. For example, when adding a component to a UI for one device, the chance is high that a
similar component has to be added to the other target device’s UIs. For this reason, D1 and D5 argued that transforming parts of UIs across devices could be a better approach. This approach will require a tool that allows designers to edit UI designs for multiple devices in parallel. This is in line with a large scale study with designers [GFIR09]. The study showed that designers need tool support for automating repetitive design actions. It seems to be a major burden of interface designers to repeat the same actions over and over again.

According to D1 and D2, automatic transformations inside a multi-device GUI builder could be very useful for managing the consistency of user interfaces across devices. One of these designers mentioned “it would be great to have a more powerful CSS that can adjust a website from one device to another while preserving its content”. This finding is in line with a study about multi-device user interface design conducted by Lin et al [Lin05]. All the designers that participated to this survey indicated that they experience difficulties to maintain the consistency across device specific interfaces. Of course, interaction techniques can differ across devices, but the designers would like consistency of the interface’s content including terminology, colors and graphics.

In summary, we can conclude that designers do not desire tools that transform a whole UI from one platform to another. Instead, partial transformations of UI components could help to reduce the amount of duplicated design actions across device-specific UIs. Furthermore, this kind of transformations could also support designers to keep the content of UIs consistent across devices. In Part III, we present various automation techniques that help to manage device specific user interfaces.

3.4.3 Optimizing

While discussing existing multi-device projects, several designers mentioned that they experience difficulties when optimizing the look of their designs for multiple devices. For example, when designing user interfaces for touch screens, designers would like to have support to find the optimal size of the objects so that they can be touched easily.

Designers want to have tool support that makes it possible to discover whether the designed UI anticipates on target device constraints such as display size, screen resolution and the supported interaction mechanism. In current tools, this information only becomes visible when deploying the design to the target device or device emulator, which usually takes several minutes.
D1 mentioned: “When building ITV applications, I connect my PC directly to the TV screen. This increases development time because the application can be compiled and tested directly on the PC instead of using the slow deployment process on the ITV emulator”. D2 and D4 mentioned problems with current tools to estimate the size and positions of user interface components due to the different resolutions and DPIs of the target devices. A smarter integration of the design environment and target device could possibly prevent these problems.

This result corresponds to Holmquist et al. [Hol05], stating that the creation of cargo-cult designs should be avoided. Cargo-cult designs are designs that look nice and seem to have a lot of potential, but are very hard to realize. The survey of Grigoreanu et al. [GFIR09] also supports our result about design optimization. This survey found that designers need tools that facilitate testing designs and check whether the design is optimized for the target platform.

In summary, we can conclude that designers need tools that help them finding the optimal look of a UI for a certain target devices. These tools should have a short feedback loop, giving designers the opportunity to perceive their design on the target device while they are designing. This helps them to discover and feel the constraints of the target device. In Part IV of this dissertation, we will discuss various techniques that support the idea of having immediate feedback of design actions on the target device.

3.5 Conclusion

In this chapter, we investigated what kind of tools and techniques designers need for creating multi-device user interfaces. This analysis was done in three steps. First, we deduced lessons from the literature survey in Chapter 2 that helps to make multi-device design tools more ’designer oriented’. Then, we created an initial multi-device design environment that takes these lessons into account. We then used this design environment as a means to stimulate discussion about multi-device design tool needs during interviews with five professional user interface designers.

From these interviews with UI designers, we compiled a set of needs designers have for creating, managing and optimizing their multi-device designs. Designers need a tool that allows them to create multi-device user interfaces visually. This tool should go beyond form-based user interfaces by facilitating designers to integrate custom UI components that were previously created in other design tools. In this dissertation, we address these needs in GUMMY
3.5 Conclusion

(Chapter 4), a multi-device graphical design environment.

For managing multi-device user interfaces, designers need automatic transformations that help them to reduce the amount of repetitive design actions that frequently occur in multi-device UI design. These automatic transformations could also help to keep the content of user interfaces consistent across devices. In Chapter 5, we present linked editing and automatic transformations to address these needs. Chapter 6 also helps to support these needs by means of D-Macs, a tool facilitating designers to create and reuse their own automation strategies.

In order to optimize multi-device UI designs, designers mentioned that they need tools that help them to easily perceive their designs on the actual target devices. This helps them to discover empirically the constraints of the target device and to anticipate on these constraints in their designs. Chapter 8 deals with this need by means of three design techniques that help designers to verify their designs on the actual target devices.
Chapter 4

Gummy: A Graphical Multi-device Design Environment

4.1 Introduction

As introduced in Chapter 1, one of the goals of the research presented in this dissertation is to explore how a design tool can enable designers to create realistic high-fidelity prototypes for a broad range of target devices. To get a better understanding of how designers could benefit from such a tool, Chapter 3 analyzed what kind of multi-device tool support designers need. From this analysis, we concluded that UI designers prefer to create multi-device UI designs visually. Additionally, this visual design approach should go beyond traditional form-based interfaces by reusing custom interface controls.

In this Chapter, we propose Gummy, a graphical multi-device design environment that is able to target a wide range of target platforms. With Gummy, designers prototype the look of multi-platform graphical user interfaces in the same way as they do when dealing with traditional high-fidelity prototyping tools. Unlike these high fidelity prototyping tools (Section 2.5), Gummy is not constrained to a certain toolkit or platform. Toolkit and platform independence is achieved by Gummy’s underlying architecture. This architecture supports the integration of new platforms in Gummy by means of “platform plug-ins”. A platform plug-in can be developed for any target platform because there are almost no technical constraints for creating plug-ins. They can be written in any programming language or run on any operating system.

In this chapter, we first discuss how user interfaces can be created with
Gummy and the structure of this design environment. Next, we explain the most important parts of Gummy’s underlying architecture. Following, we give a detailed description of the presentation model that Gummy uses to describe user interface designs many target platforms. Finally, we discuss how we can integrate new platforms in Gummy by means of plug-ins.

4.2 Designing User Interfaces with Gummy

Gummy is a multi-device design environment that can be used in a similar way to existing high-fidelity GUI design tools. This resemblance to traditional GUI builders reflects one of the main strengths of our approach: designers can use their knowledge of single device design environments in multi-device UI design. Gummy allows designers to alter immediately the visual representation of the UIs instead of having to work with complex artefacts such as abstract models or programming code. This is different from existing multi-platform design tools such as model-based design tools (Section 2.3) or GUI toolkits (Section 2.2).

To support multiple target devices, we subdivided Gummy’s design environment in several individual device-specific design workspaces. When starting Gummy, a set of target platforms can be selected and for each of these platforms a design workspace will be loaded. Workspaces can be accessed through a tabpage at the top of the design environment (Figure 4.1-1). For example, Figure 4.1 shows Gummy for designing Adobe Flex web-based UIs and UIs for Java Swing GUIs. Designers can add additional target platforms at any time.

A design workspace in Gummy is not that different from design workspaces in most traditional GUI builders. It contains a toolbox showing the available user interface elements (Figure 4.1-2), a canvas to build the actual user interface (Figure 4.1-3) and a properties panel to change the style properties of the user interface elements on the canvas (Figure 4.1-4). The selected interface elements on the canvas can be resized or moved through the manipulators that are shown around it (Figure 4.1-5).

The widget toolbox of Gummy’s design workspace (Figure 4.1-2) is flexible in two ways. First, depending on the selected target platform, it will show the widgets that are suitable for this platform. Secondly, the toolbox is not constrained to a fixed set of user interface components. For every platform, the toolbox can be extended by importing custom components which were previously created with other design tools. This can be done using Gummy’s import feature, which allows designers to specify the path to a binary file
4.3 Architecture

One of the strengths of GUMMY is that it is not constrained to design user interfaces for one specific device or GUI toolkit. This is different from most existing high-fidelity prototyping design tools (Section 2.5). In this section, we will give an overview of GUMMY’s underlying architecture to explain how we reach this level of flexibility.

The architecture of GUMMY is inspired by the Model-View-Controller (MVC) software architecture [Bur87]. MVC usually consists of three kinds of objects. The model is the application object, the view its screen presentation, and the controller defines the way the user interface reacts to user interface input. Spreading the software architecture over these three types of objects increases flexibility and reuse.

Figure 4.2 shows the GUMMY MVC architecture. The three MVC components in this architecture are:

- the Gummy User Interface is the view component. As discussed
in Section 4.2, this user interface contains different platform-specific workspaces each having a toolbox with widgets, a canvas to combine these widgets and a properties panel to change the properties of every widget.

- for every platform-specific workspace, there exists a **Gummy model**. This model contains all information that is shown in this workspace by means of two entities: a CUI model and a GUI bitmap. The former describes the structure, style and content of the user interface that is currently being designed. More details about this CUI model can be found in Section 4.4. The GUI bitmap is a screenshot that shows how the designed GUI looks like.

- the **Platform Plug-in** is the controller, which handles the actions that the designer performs in the Gummy user interface and updates the Gummy model accordingly. Per platform-specific workspace, there exists another platform plug-in. The platform plug-ins are described in more detail in Section 4.5.

We will now delve a bit deeper in the communication across these different MVC components. When creating a user interface, a designer performs design actions in **Gummy’s user interface** (Figure 4.2-1). An example action that can be performed in Gummy’s UI is dragging a button from the toolbox to the canvas. This design action is then communicated to the **platform plug-in** (Figure 4.2-2). For example, when adding a button, the platform plug-in
receives an “add button” message containing a description of a button component and the location where it should be added. The platform plug-in then computes how this design action will update the current design (Figure 4.2-3). In our example, the received action will result in a design that contains an additional button.

Once the platform plug-in computed the design changes, it will update the GUMMY model. First, it updates the CUI model (Figure 4.2-4), which describes the user interface design that is currently being created in GUMMY. GUMMY’s user interface queries this model and will be updated accordingly (Figure 4.2-5). In our example, GUMMY’s properties panel (Figure 4.1-4) will contain the properties of the just added button and manipulators (Figure 4.1-5) will become visible around this button. The platform plug-in also updates the GUI bitmap of the GUI model (Figure 4.2-6). In our example, this bitmap will be a screenshot of the user interface containing the new button. Once this GUI bitmap is updated, the updated bitmap will be displayed in GUMMY’s canvas (Figure 4.2-7). This way, designers always have a graphical view of their currently designed user interface.

### 4.4 Gummy CUI Model

As discussed in the previous Section, GUMMY is built using a model-view-controller architecture. When a designer performs an action in GUMMY’s design workspace, this action will be handled by the controller component. In turn, this controller will update GUMMY’s underlying model. This will finally result in an updated design workspace that shows the result of the design action. This approach allows designers to work on a concrete representation of the user interface, a model describing the user interface is only used under the hood.

In GUMMY’s underlying model, user interfaces are being described by means of a Concrete User Interface (CUI) model. According to Calvary et al. [CCT+03], a CUI model describes user interfaces independently of any markup or programming language. This description consists of Concrete Interaction Objects (CIOs) such as “buttons” or “combo boxes”. The models at the CUI level are often also called presentation models.

In this Section, we will describe the basic principles of our CUI model and the structure of the model.
4.4.1 Design Principles

Before and during the design of our CUI model, we developed a set of principles on which to base the design of this model. The four core principles that we kept in mind during model design are:

1. **Gummy uses one CUI model per device-specific UI.** This approach allows designers to create user interfaces with a broad range of look and feel. This is contrary to many MBUID approaches (Section 2.3), which generate all device-specific UIs from one high-level model. These automatic generation approaches are difficult to control and can produce unpredictable outcomes [MHP00]. Further more, the interfaces that are generated from the same abstract model are often called “least common denominator” UIs, because they are composed from a general set of widgets that are available on most platforms or devices [Nil02].

2. the CUI model is oriented towards design tools instead of UI renderers. This makes Gummy’s CUI model different from other presentation models, which usually only describe the concepts that are needed to realize a UI from the model. Besides these concepts, Gummy’s CUI model also contains other information that is needed to build up design tool components like the properties panel or the canvas.

3. Gummy’s CUI model should be dynamically extensible with custom user interface controls. This principle originates from our multi-device design tool needs analysis (Chapter 3), where we found that designers would like to go beyond form-based user interfaces by using custom controls. To support this principle, the CUI model should not be constrained to a predefined set of elements, but it should be able to include new elements at any time.

4. The CUI model is not bound to a certain serialization format. This makes it possible to serialize the model in a format that is most easy and efficiently to parse on a certain platform. For example, when targeting web based platforms, it would be easier to use the Javascript Object Notation (JSON) serialization format rather than XML. In other situations, it might be beneficial to serialize the model in a binary format.

4.4.2 Model Structure

We designed Gummy’s CUI model in such a way that it adheres to the principles that we deduced in Section 4.4.1. In Figure 4.3, the general structure...
4.4 Gummy CUI Model

Figure 4.3: The structure of Gummy’s CUI model. It contains the user interface widget tree (a), the layout information (e) and properties of these widgets (b), and the value of every property (c). The model also describes how widgets and properties map on concrete elements (d).

of the model is visualized. The most important parts of the model are: structure, describing the structure of the widgets that are shown in a GUI, style, describing the style properties of these widgets, and layout, describing the visual layout of every widget. In this section, we will first describe these parts in detail. Afterwards, we will discuss how this model can be mapped to any target platform and how it can be dynamically extended with new components.

Structure

As in many presentation languages and models, the Gummy CUI model describes the structure of a GUI by means of a widget tree (Figure 4.3-(a)). This
tree describes the containment relations between widgets: a child node is contained inside the parent node. Every widget is characterized by two attributes: its type (e.g. Button, ListBox, ...) and name, which is the unique name a designer can associate with a certain widget. It is important to note that the model is not limited to a fixed number of widget types (Section 4.4.3 describes in detail how different types can be handled).

**Style**

Every widget contains a dynamic list of properties, describing the style and content of the widget (Figure 4.3-(b)). This is inspired by the dynamic properties pattern, introduced by Fowler et al. [Fow97]. A dynamic list of properties introduces two levels of flexibility: different widgets can have different properties, and properties can be added at any moment.

Every property has a name and value. Currently, two types of property values are supported (Figure 4.3-(c)):

- **Basic property values** can contain values of the five primitive types of XML Schema \(^1\) (e.g. decimal, string, etc.) and two types that are important for user interface design: image and color. Values can also contain platform-specific types, which map to types of the targeted platform (e.g. java.awt.BorderLayout in the case of a Java AWT interface). **Gummy** will be responsible for decoding values into the desired type.

- **Collection property values** are aggregations of basic property values, all having the same type. An example collection property value is the **items property** of a listbox, which can be defined as an array of strings. In the **Gummy** GUI model this can be represented by a collection property value that aggregates basic string property values.

Figure 4.4 shows how the **Gummy**'s properties panel is related to the underlying CUI model. The **Text** property (Figure 4.4-1) and the **Background** property (Figure 4.4-2) contain both a basic property value, the **Items** property (Figure 4.4-3) contains a collection property value. A designer can specify a collection property value by means of the collection editor, which opens when clicking on the '...' button. For the basic property values, **Gummy** uses their valuertype to decide how the designer can specify their value. In the case of the text property, which has a string valuertype, a text field is shown to type the desired value. When the valuertype is a color, in the case of the background property, a button that pops up a “color picker dialog” is being used.

\(^1\)http://www.w3.org/TR/xmlschema11-2/
Property values also have a range (Figure 4.3-F), which describes in detail all possible values a property can have. This value range can be either a continuous interval or a set of possible options. For example, a property that describes the length of a widget usually will have an integer value that should be between zero and the maximal length of the top window. Another example concerns a property that describes the orientation of a label widget. This orientation property can only have two values: left-to-right and right-to-left. These options are encoded in the model as key value pairs.

**Layout**

The **GUMMY** GUI model also contains layout information about every widget (Figure 4.3-E). This layout contains two types of information: the bounding box of a widget and the layout constraints of an element. **GUMMY** uses this bounding box to place the manipulators of an element correctly on the canvas. Figure 4.5 shows how the bounding box is used to place manipulators around an Adobe Flex label.

The layout information also contains the layout constraints of an element. Example layout constraints are the maximum width of an element, the minimum or maximum number of children an element can contain, etc. These constraints are used by **GUMMY** to enable or disable certain design actions. For example, when an element has reached its maximum size, **GUMMY** will not allow designers to further enlarge it. If an element is added to a layout manager, a designer will not be able to manipulate it directly. Instead, she
Figure 4.5: The layout information of GUMMY’s CUI model is used to place the manipulators at the right place on the canvas.

has to manipulate the properties of the layout manager.

4.4.3 Model Mappings

Two core principles of the GUMMY CUI model (Section 4.4.1) are that it has to be dynamically extensible with new controls and that it has to be able to define a CUI model for any target platform. Few existing presentation models provide support for both principles. Most CUI models that can be used on multiple platforms are constrained to a fixed set of abstractions such as buttons, checkboxes, listboxes, . . . that are available on all targeted platforms. One notable exception is UIML [HA08], which has a separate vocabulary section to define widgets and how these widgets should be rendered on the target platform.

The advantage of the UIML approach is that many platforms can be targeted by creating a custom vocabulary for every target platform and that this vocabulary can be extended with custom components. A limitation of this system is the complexity to add new components and the inconsistencies that can arise between vocabularies. Every new component has to be added to the vocabulary first before it can be used in the UIML UI description. We need a more dynamic system in our platform plug-ins. This is necessary because not all components for designing a user interface are known beforehand. A designer can decide to include a new custom component at any time.

The properties in the GUMMY CUI model that are responsible for mapping the CUI model to the underlying platform and for including new components are known as the mappings (Figure 4.3-D). These mappings describe how widgets and properties should map on concrete user interface elements (Figure 4.3-D). The mapping between widgets in the model and the actual widgets is done using the MapsTo attribute, which contains the class name of the concrete UI element that renders this widget. For example, when a
Properties are linked to widget properties by means of two attributes: *MapsToGetter* and *MapsToSetter*. While the former specifies how a property value can be queried (typically a *getter*) from the concrete UI object, the latter specifies how this property value can be set (typically a *setter*). For example, a text property from a Java AWT Button will contain *getLabel* as *MapsToGetter* and *setLabel* as *MapsToSetter*. In Figure 4.4, it is shown how the *text*, *background* and *items* properties map on the Java toolkit.

In summary, mappings enable the Gummy CUI model to define interfaces for many platforms because it can describe how the model should map on a concrete GUI toolkit. On the other hand, it also allows us to dynamically extend the model with new controls. At any time, new custom widgets can be added since every widget contains its own mapping attributes, describing how it should be rendered. This approach is more flexible than the UIML vocabulary system [HA08] because widgets can be added without having to update a vocabulary section first.

### 4.5 Platform Plug-ins

In Section 4.3, we gave an overview of the Gummy’s model-view-controller architecture. The controllers in this architecture are the platform plug-ins. In every design workspace that is loaded in Gummy, a platform plug-in controls the design actions that are performed by the GUI designers. When receiving a design action, the platform plug-in is responsible for computing how this action should update the GUI design. Then, Gummy’s underlying model is updated in two steps. First, the CUI model describing the design GUI is updated and then this update is reflected in Gummy’s UI. Next, the GUI bitmap that is shown through Gummy’s design canvas is updated in order to visualize the effect of the design action graphically.

In this section, we first describe how the platform plug-in computes the effects of the design actions. This phase is called the *design update* phase. Next, we discuss how the platform plug-ins can communicate with the other components in Gummy. Finally, we demonstrate the flexibility of our plug-in system with some example plug-ins.
4.5.1 Design Update

When receiving a design action, the platform plug-in computes how this action will update the current GUI design. In the current implementation of Gummy, these are the five most important design actions that the platform plug-in receives:

- **adding a new widget** by dragging a widget from the toolbox to the canvas. The corresponding message contains the type of the widget that is added and its location. An example message in Java Swing could contain the `java.awt.Button` type and the coordinates `(2, 2)`.

- **updating a widget’s property** by altering the value of a property in the property panel. This message mentions the name of the widget that is updated and an excerpt of the Gummy CUI model describing the updated property. This CUI model snippet originates from the properties panel, as shown in Figure 4.4.

- **resizing or repositioning a widget** by manipulating the widget’s manipulators on the canvas. In this message, the updated bounding box and the widget name are mentioned.

- **changing the widget’s z-order** by bringing a widget to the front or sending it to the back. This message contains the name of the widget and a field that describes whether the z-order should be increased or decreased.

- **deleting the selected widget** by pressing the delete button. This message contains only the name of the widget that should be deleted.

Based on the information contained in the design action message, the platform plug-in alters a copy of the GUI design’s CUI model. For example, when adding a widget at a certain coordinate, the plug-in searches for the container element and adds the widget to this container. After updating the CUI model, this model is rendered as a running user interface. This **rendering step** is done by traversing the CUI model’s widget tree. Every widget is created by means of reflection using its MapsTo attribute (Section 4.4.3). After creating a widget, its properties are applied. This is also done using reflection ans by means of the SetterMapsTo attribute.

Once the user interface has been rendered, it is shown as a running UI. In this running version, all layout managers have been evaluated and the widgets have the correct positions and sizes. From this running version, we
are now going to extract all necessary information in the reverse rendering step. The widget tree of the running version is traversed and all properties and layout information of these widgets is extracted and stored in the model. Instead of using vocabularies (like in UIML [HA08]), we detect all properties automatically using reflection if possible. For example, in Java, all get and set functions with the same suffix are automatically considered as properties. In .NET, all accessors are automatically converted into properties. Automatic property detection makes it also easier to include custom components: only the MapsTo attribute of a widget has to be provided, all other information is detected automatically.

The reverse rendering step results in a CUI model describing all widgets and their properties. This CUI model is then sent back to GUMMY’s design workspace, which will update its workspace. The platform plug-in also sends a screenshot of the running UI to GUMMY. This screenshot will be used to update the design canvas.

4.5.2 Networked Communication Architecture

In theory, every platform plug-in that respects the communication protocol described in Section 4.3 can be plugged into GUMMY. However, plug-ins that render a GUI for a certain target platform mostly depend on this target-device’s GUI toolkit and on the supported programming languages. These constraints mostly imply that platform plug-ins have to be developed on the target device itself.

In order to make it possible to implement a plug-in on the target device, we chose to employ a networked architecture. This enables plug-ins and design tool to exchange information even when they are not on the same device. To create this networked architecture, we need a communication protocol that is available in multiple programming languages and on multiple platforms. We chose for the IETF standard Extensible Messaging and Presence Protocol (XMPP). Pierce et al. [PN08] have shown that this instant messaging protocol provides the required functionalities and flexibility to build cross-platform distributed applications. Several freely available XMPP libraries exist in different programming languages, which simplifies the development of new rendering engines in almost any programming language on many operating systems. Figure 4.6 depicts what GUMMY’s MVC architecture looks like using the XMPP networking protocol.
4.5.3 Example Platform Plug-ins

To demonstrate the extensibility of our platform plug-in system, this section discusses how we implemented tool support for various computing platforms. As discussed before, Gummy can support a broad range of platforms because the platform plug-ins are loosely coupled to the design tool and thus can be implemented in any programming language. Currently, we have implemented plug-ins in three different programming languages for five different platforms: an Action Scripting (AS3) platform plug-in for producing web-based Adobe Flex UIs, a .NET platform plug-in for targeting Windows desktop and Windows Mobile UIs, and a Java platform plug-ins for designing Java Swing and Android interfaces.

In the AS3 platform plug-in, we use the open source XIFF\(^2\) library for realizing XMPP communication. Dynamic creation of user interface controls is realized using the Haxe reflection libraries for Adobe Flash\(^3\). One limitation that we discovered in the AS3 rendering engine is the lack of multi-threading support in Flash: XMPP traffic has to be handled in the GUI thread as well. Consequently, network interrupts might occur when there are lots of design operations in a small amount of time.

The .NET platform plug-in relies on the open source agsXMPP library \(^4\), which can run on both Windows Desktop and Windows Mobile. There exist three variants of the .NET platform plug-in: two for desktop interfaces (for

---

\(^2\)http://www.igniterealtime.org/projects/xiff/
\(^3\)http://haxe.org/api/reflect
\(^4\)http://www.ag-software.de/agsxmpp-sdk/
4.6 Conclusion

System.Windows.Forms and for WPF) and one for mobile interfaces. The core of these plug-ins is shared in one code base with specific extensions for every target device. This structure is similar to the UIML.net [LTVC07] renderer.

The Java platform plug-in also has two target platforms: desktop (using the Java Swing toolkit) and mobile (Android). Both platform plug-ins also share a core rendering project. For establishing XMPP communication between the Java platform plug-ins and Gummy, we relied on the Smack XMPP library \(^5\).

4.6 Conclusion

This chapter introduced Gummy, a multi-device user interface design tool that facilitates visual GUI design for a broad range of computing devices. Gummy contains multiple design workspaces that are equipped for designing interfaces for the targeted platforms. This allows designers to create multi-device UIs the same way as when designing single-device UIs.

Gummy is built on top of a flexible architecture, which maintains a CUI model of the designed interface and queries the visual representation of the designed interface from a renderer on the target device. We demonstrated the flexibility of Gummy’s underlying architecture by targeting multiple mutually exclusive target platforms like Adobe Flex, Android, Java Swing, Windows Mobile and WPF. Gummy also contributes the integration of custom user interface controls in multi-device GUI design. The integration of custom GUI controls helps to create more aesthetically pleasing user interfaces.

Although Gummy can target various computing platforms, designers still have to create and update every platform specific user interface manually. Gummy does not provide support for managing designs across devices. In Part III, we are going to discuss various techniques that help designers to manage platform-specific GUIs in Gummy.

\(^5\)http://www.igniterealtime.org/projects/smack/
Part III

Managing Device-Specific User Interfaces
Chapter 5

Automatic Transformations and Linked Editing

5.1 Introduction

In the multi-device design tool needs analysis in Chapter 3, we found that designers need tools for creating, managing and optimizing multi-device user interfaces. For creating user interfaces, we proposed the GUMMY design environment in Chapter 4. This tool allows designers to create GUIs for multiple computing devices within the same graphical design environment. When using GUMMY, designers do no longer have to use multiple platform-specific design tools in order to target multiple mutually exclusive target platforms.

Despite the fact that GUMMY lowers the burden for creating multi-device user interfaces, designers still have to create and update every device-specific user interface manually. It does not provide tool support for managing multiple device-specific designs. The lack of managing tool support induces that designers often have to repeat design actions across device-specific GUIs. For example, when adding a button to one device-specific user interface, the chance is high that a similar widget has to be added to the other device-specific UIs as well.

In this Chapter, we explore how we can extend GUMMY with techniques to manage device-specific user interfaces. These techniques automate parts of the design process, so that it becomes easier for designers to create and maintain multiple device-specific user interfaces. The two techniques that will be explained throughout this chapter are automatic transformations and linked editing. The former allows designers to reuse parts of a user interface from one device to another. Linked editing helps to maintain the consistency across user
interfaces by editing widgets in concert. Both automatic transformations and linked editing are supporting graphical user interface design without raising the expertise threshold to design user interfaces.

This Chapter first discusses how linked editing and automatic transformations can be used in GUMMY and how both techniques are implemented. Following, we describe feedback that we gained from HCI professionals during an informal experiment and draw the conclusions.

5.2 Automatic Transformations

Automatic transformations originate from Model Based User Interface Design (MBUID) approaches, which were introduced in Section 2.3. In MBUID, designers create user interfaces by means of abstract models. These models can then be transformed (semi-) automatically into several device-specific user interfaces. A notable example of a MBUID tool that can target multiple devices is TERESA [Pat00, MPS03] (Section 2.3 describes this tool in detail). In short, TERESA allows designers to specify a user interface by means of a task model. This task model is then transformed (semi-) automatically in an Abstract User Interface (AUI). This AUI is composed of Abstract Interaction Objects (AIOs) such as “action” or “input”. These AIOs are then automatically transformed in Concrete Interaction Objects (CIOs) such as “button” or “textfield”. These CIOs can then be used to render the user interface on the target platform.

We extended GUMMY with automatic transformations that are able to transform parts of a graphical user interface from one platform to another. Contrary to MBUID, designers do not have to deal with abstract models to use the automatic transformations. Avoiding abstract models helps to lower the threshold to use this technique while designing user interfaces. The automatic transformations that are integrated in GUMMY are based on the results of the multi-device design tool needs analysis in Chapter 3, where we found that designers could benefit from tools that can transform parts of a user interface from one platform to another.

In the next sections we are going to explain how a designer can use automatic transformations in GUMMY and how these transformations were implemented.
5.2 Automatic Transformations

5.2.1 Scenario of Use

The following scenario exemplifies how automatic transformations can be used in GUMMY. As discussed in Chapter 4, GUMMY contains multiple platform-specific workspaces, one for every targeted platform. In our scenario, a designer starts by selecting one of these workspaces (i.e., the Windows Mobile workspace in Figure 5.1 step 1) and drags a combobox from the toolbox to the canvas. The combobox is now visualized on GUMMY’s canvas and is surrounded by manipulators (Figure 5.1, step 2).

Once a widget is shown on the canvas, a designer can copy this widget by right clicking on it and selecting the copy option in the context menu (Figure 5.1, step 3). Next, she can switch to the Adobe Flex workspace through the corresponding tab (Figure 5.1, step 4). In this workspace, the designer can now right click on the canvas and select the “paste as” option (Figure 5.1, step 5). Here she can select how the element on the clipboard should appear in a different toolkit on a different device. By selecting for example the “fish-eye view” option in this list, the copied combobox will be added as a fish-eye view to the Adobe Flex design. This fish-eye view is a custom horizontal listbox that enlarges the items below the mouse cursor.

5.2.2 Implementation

When copying a UI element from one device to another, GUMMY needs to remap the copied UI element to a suitable element in the other device’s toolkit. First, GUMMY determines all elements in the new device toolkit that are similar to the original UI element and allows the designer to select one of these elements through the “paste as...” menu. GUMMY adds an instance of the selected element to the current design workspace and gives it the same content as the copied element.

In order to compute elements similar to the copied element, GUMMY adopts a similar approach as the user interface semantic networks described by Demeure et al. [DCCV06] and Vermeulen et al. [VVC+07]. These UI semantic networks describe the relationship between concrete UI components and Abstract Interaction Objects (AIOs). AIOs describe UI components independently of any platform and interaction modality (e.g., graphical, vocal, tactile, ...) [Van99]. When copying an element from one device to another, the corresponding elements on the other device can be computed by asking the semantic network for components that are linked to the same AIO as the copied element on the first device.

When a designer starts GUMMY, the UI semantic network is automatically
Figure 5.1: An example scenario showing how cross-device copy-pasting and linked editing can be used in multi-device UI design.
loaded from an XML file. Storing the network in an external XML file allows adding new (custom) controls to the network. Our network employs the same set of high-level AIO types as described by Vermeulen et al. [VVC+07]. These AIOs are differentiated according to the functionality they offer to the user:

1. *input* components allow users to enter or manipulate content;
2. *output* components present content to the user;
3. *action* components allow users to trigger an action;
4. *group* components group other components into a hierarchical structure.

When querying our UI network to find components similar to a given component, we look for all components that have the same AIO type (i.e. input, output, action or group) and have the same *content datatype* as the given component. Currently, we support the five primitive types of XML Schema (e.g. decimal, string, void, etc.), a number of datatypes that are often used in user interfaces (e.g. Image, Color, etc.) and container datatypes that group content items of a certain type together (e.g. a list of strings, a tree of images, etc.).

For example, assume we are querying our network for all Adobe Flex components that can represent a Windows Mobile combobox. Since the combobox is linked to an input AIO (Figure 5.2), the network is first searched for all Adobe Flex components that are linked to an input AIO. This returns a huge list of controls such as checkbox, spinbox, listbox, combobox, fish-eye view, etc. Secondly, this list is searched for all components that support the same datatype as the Windows Mobile combobox (i.e. a list of strings). This finally results in three Adobe Flex components: listbox, combobox and fish-eye view (Figure 5.2). These components are then displayed in GUMMY’s “*paste as...*” menu (see Figure, step 4).

### 5.3 Linked Editing

Linked editing is a technique to keep the content of user interfaces consistent. In Chapter 3, designers suggested that they would like to have tools to keep user interfaces consistent across devices. This finding is supported by a survey about multi-device design tools [Lin05], which showed that designers need tools to keep the content of low fidelity prototypes consistent between device-specific user interfaces.
Linked editing was first introduced by Toomim et al. in the Codelink programming environment to edit duplicated code fragments as one [TBG04]. Hartmann et al. extended this technique to maintain UI design alternatives in the Juxtapose programming environment [HYA+08]. In GUMMY, we share the motivation of the two aforementioned approaches to edit duplicated items in concert. While Codelink and Juxtapose focus on textual programming, we integrate this technique in a graphical design environment to edit the content of duplicated UI elements across different platforms.

In this section we first discuss a typical scenario of use that shows how linked editing can be used in GUMMY. Next, we discuss how we implemented this technique.

### 5.3.1 Scenario of Use

Linked editing is mostly used in combination with automatic transformations (Section 5.2). When an element is copied from one platform to another, GUMMY considers these elements as being linked to each other. GUMMY enables designers to edit the content of these elements in concert using the linked editing mechanism. This way, designers can ensure the consistency of UI content across different devices and toolkits.
5.4 Evaluation

For example, when copying a combobox element from the Windows Mobile design environment as a fish-eye view selection element in Adobe Flex, GUMMY will recognize and remember these are linked. When the fish-eye view is selected, its properties can be modified in the content panel just as would be the case in a regular environment (Figure 5.1, step 6). When the linked editing option is on (presented by a checkbox in the design environment), these properties will be changed for the Adobe Flex fish-eye view as well as for the Windows Mobile combobox. Afterwards, both components will contain the same contents (Figure 5.1, step 7).

5.3.2 Implementation

Linked editing is supported by extending GUMMY’s underlying CUI model (Section 4.4). This model describes the currently designed user interface in GUMMY’s model-view-controller architecture. To facilitate linked editing, we supported this model with content properties. These properties are responsible for the data that is shown in a widget. Some example content properties are properties like the items of a combobox, the text of a button, the image of an icon widget, . . .

In GUMMY’s CUI model, content properties can be represented by means of the Datatype attribute in PropertyValue (see Figure 5.3). If this attribute is set, the property is considered as a content property. The values of the Datatype attribute are similar to those of the content data in the previous section, i.e. the five primitive types of XML Schema (e.g. decimal, string, void, etc.), a number of datatypes that are often used in user interfaces (e.g. Image, Color, etc.) and container datatypes that group content items of a certain type together (e.g. a list of strings, a tree of images, etc.).

When copying a widget across device-specific UIs, these widgets are considered as linked. On the level of the GUMMY CUI model, this means that these widgets have content properties with the same datatype. In the GUMMY design tool, this relation is exploited when updating the property value using linked editing. Updating the value of a content property will update the value of the content properties with the same datatype in the linked widgets as well.

5.4 Evaluation

We conducted an informal use study of GUMMY in order to gain feedback about the usefulness of the two techniques presented in this Chapter: linked editing and automatic transformations. Eight HCI professionals participated
in the study, all having a reasonable level of experience with commercial user interface design tools. Five of them had experience with UI design for other devices than desktop PCs such as multi-touch tables or mobile devices.

5.4.1 Evaluation Set-up

One evaluation session was conducted per participant, and each evaluation session consisted of three parts. First, the participant was asked to read a written tutorial about the Gummy design tool and its basic functionalities. The second part was to get the participant used to interacting with Gummy. We asked the user to add a component to one design workspace, to copy this element to another workspace and to adapt the contents of this element using linked editing.

The final task was a design task, where we asked the participants to create a music player GUI for a Windows Mobile device and an Adobe Flex website. We chose a music-player since this is a well-known application that exists on many devices with a reasonable variety in the different device-specific UIs. The designers were provided with two mockup sketches that gave them an idea of what we expected from the music-player GUIs. The desired music player features were for example selecting a song in a playlist, controlling the volume of the music and navigating through a song.

During the experiment, a think aloud protocol was used to understand the actions and decisions of the participants. Additional questions were asked by the observers if necessary. After the participants finished their task, they filled out a questionnaire asking about Gummy’s core features such as copying UI elements and linked editing in terms of usefulness and ease-of-use. The
5.4 Evaluation

participants rated their usefulness and ease-of-use responses on 5-point Likert scales ranging from “not very useful” to “very useful”, and from “very difficult to use” to “very easy to use”.

5.4.2 Results and Feedback

Automatic Transformations

Figures 5.4 summarizes the results of the post-test questionnaire. From all features we tested in Gummy, the participants ranked copying elements across devices as Gummy’s most useful feature (mean=4.38, median=4.50, $\sigma = 0.74$). On the other hand, the participants rated this feature as the most difficult to use (mean=3.38, median=3.00, $\sigma = 0.92$). This seems to indicate that users felt that this technique’s conceptual idea was on target, but that its current implementation can be improved.

<table>
<thead>
<tr>
<th>How useful was...</th>
<th>mean</th>
<th>median</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked Editing</td>
<td>4.13</td>
<td>4.00</td>
<td>0.83</td>
</tr>
<tr>
<td>Copying Elements</td>
<td>4.38</td>
<td>4.50</td>
<td>0.74</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>How difficult was...</th>
<th>mean</th>
<th>median</th>
<th>$\sigma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked Editing</td>
<td>4.25</td>
<td>4.00</td>
<td>0.46</td>
</tr>
<tr>
<td>Copying Elements</td>
<td>3.38</td>
<td>3.00</td>
<td>0.92</td>
</tr>
</tbody>
</table>

Figure 5.4: Post-experiment questionnaire results.

Two participants suggested to distinguish between “trivial copy operations” that copy a UI element to a very similar element (e.g. copying a Windows Mobile button to an Adobe Flex button) and “complex copy operations” where an element is pasted as a very different widget (e.g. migrating a Combobox as a fish-eye view). They would like to do the straightforward copy operations for multiple items at the same time, while doing complex copy operations one-by-one. Another participant wanted to have more visual feedback about the component that is currently stored on the clipboard: “it would be great to have a ‘paste X as Y or Z’ option, where ‘X’ is the component on
the clipboard and ‘Y’ or ‘Z’ are candidate widgets to represent ‘X’”. It would be fairly easy to support these two suggestions in GUMMY by extending the context menu with the additional options that are requested by the designers.

The current implementation of GUMMY only migrates a component’s content when it is copied from one device to another. Several users wanted to have the option to copy other properties as well. One designer mentioned that “copying would be more useful if the spatial layout would remain more or less intact”. We could include the option to copy multiple properties in GUMMY, but this implies that there should be meta-data available that specifies how the properties of different ‘similar’ widgets are related to each other. For example, this meta-data should indicate for every widget property if it defines the length of a widget, the content, the visibility, etc. This would have a negative impact on GUMMY’s flexibility. When adding a custom widget, all this meta-data should be specified before it can be used.

Concerning cross-device copying, three of the participants also suggested that they would like to have “a mechanism that allows them to define how an element will be copied from one device to another”. Once they defined this, they could reuse this copy strategy at different places in the user interface. This suggestion contrasts with our current approach, where we define copy operations by means of an abstract UI model behind the scenes. We believe that one way to support this option is by means of macros, where designers can record how a transformation should occur and replay this transformation afterwards. Such a macro system is discussed by means of D-Macs in Chapter 6.

Linked Editing

GUMMY’s linked editing was rated as useful (mean=4.13, median=4.00, \( \sigma = 0.83 \)) and easy to use (mean=4.25, median=4.00, \( \sigma = 0.46 \)). During the post-test survey, several participants noticed that this technique would be great during software maintenance or updates of larger multi-device projects. They also mentioned that inconsistency between UIs mostly appears during software maintenance, where updates are done in one version of the UI but not in the other and vice versa. We think that the current version of GUMMY in combination with linked editing and automatic transformations can be used for software maintenance. It can be used here to manage the content of different device-specific user interfaces, which might change over time.
5.5 Conclusions

This Chapter introduced automatic transformations and linked editing, two automation techniques that can help designers to manage the multi-device graphical user interface design process. The goal of these techniques is to reduce the necessity for repeating design steps and to make design parts reusable across devices.

During an informal user study, HCI professionals gave feedback and suggestions about the potential value of these two automation techniques in multi-device user interface design. This study learned that automation support during interface design is useful but that designers want to go further. They would like to define themselves how automation should occur and to reuse these automation strategies later on. In Chapter 6, we present Design Tool Macros as a solution to this problem. This tool enables designers to record how automation should occur and to replay this recording afterwards in the same or in other designs.

Automatic transformations and linked editing also have a negative impact on GUMMY’s flexibility. It becomes more difficult to integrate custom components, because these components now have to be linked to AIOS and to certain data types. The properties of the widgets also have to be augmented with additional information that indicates the properties that contain the widget’s content. These steps are difficult to do for UI designers, since we do not expect that they have knowledge of the underlying models. The macro-system in Chapter 6 could also provide a solution to this problem. Instead of adapting abstract models, designers can record how a custom component has to be transformed from one device to another. This recording can then be replayed afterwards.
Chapter 6

Design-Tool Macros (D-Macs)

6.1 Introduction

One of the goals in this dissertation is to provide tools and techniques for managing multi-device user interfaces. Typically, these tools and techniques automate parts of the design process, which makes it easier for designers to edit multiple device-specific user interfaces. In Chapter 5, we proposed two techniques that help to manage multi-device designs in Gummy: linked editing and automatic transformations. The former helps to edit the content of multiple device-specific user interfaces in concert. Automatic transformations allow designers to reuse parts of a user interface across devices.

From the informal use study that we conducted with linked editing and automatic transformations in Chapter 5, we learned that there is a need for managing techniques where designers can specify themselves how automation should occur and reuse these automation strategies later on. This finding supports the findings of a large scale study on future design tool needs [GFIR09]. This study suggests that designers would benefit from tool support for (1) recording and replaying redundant design actions and (2) sharing a design’s look, feel, code, etc. between designers as sources of inspiration. Fulfilling these needs in one design environment would result in better tool support to realize interactive GUI prototypes on multiple devices. This can help designers to uncover design problems and to generate suggestions for new, and better, designs [KHT06].

The main idea presented throughout this chapter is a new strategy of automating multi-device user interface design by capturing and reusing design
actions across designers and user interfaces. This idea is embodied in Design tool Macros (D-Macs), an extension of GUMMY allowing designers to automate repetitive design actions. The three major steps in the D-Macs approach are outlined in Figure 6.1:

- designers demonstrate the design action sequences that have to be automated;
- designers can share the recorded steps through a central repository with other designers;
- designers can replay and edit their own design actions and those of others.

This chapter first describes the requirements that D-Macs has to meet in order to support the demonstrate, share, and replay functionality. In the next section, we then show all D-Macs features. Following, we give some examples that show the effectiveness of D-Macs’ multi-device design management support. Finally, we discuss this effectiveness also by means of the evaluation framework of Olsen [Ols07a] and we draw the conclusions.
6.2 Design Tool Requirements

Table 6.1: The requirements that D-Macs has to meet.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Demonstrate</th>
<th>Share</th>
<th>Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generalization</td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Feedback</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Design Community</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Contextual Assistance</td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

6.2 Design Tool Requirements

D-Macs tries to find a balance between designer control and automation by means of a three-step approach. Designers demonstrate design actions, share these actions with other designers and replay them in their own design or designs of others. This section describes the requirements that D-Macs has to meet in order to support this functionality. We identified four requirements from previous research and classified them per feature (Table 6.1).

6.2.1 Generalization

Design actions that are recorded in D-Macs can be considered as generalizable if they can be applied to other design contexts than the one in which these actions were captured. This is a challenging goal that originates from Programming By Demonstration systems [Lie01], where users create programming scripts or macros by demonstrating sample input and output values. Notable PBD systems for demonstrating UI behavior are Monet [LL05], DEMO [WF91], Eager [Cyp91], Jade [ZM90] and Gamut [MM99]. The major difficulty in PBD systems is to define how the inferred programming scripts should generalize so that they can work with new input values. This is mostly done by means of machine learning or pattern recognition algorithms, which might not produce the results as intended by the user.

Instead of using complex machine learning or pattern recognition algorithms, we seek for a generalization strategy that can be controlled by UI designers. An interesting approach are mixed-initiative [Hor99] systems, which are systems that divide the work between the human and the computer. This way, designers can be involved in the generalization process without forcing them to learn a complex generalization mechanism. This generalization strategy is similar to the way CoScripter generalizes web interactions [LHML08].
6.2.2 Feedback

In D-Macs, it is important to have a comprehensible notation that helps designers to easily understand their own and each other’s action sequences. This consideration is similar to programming-by-demonstration systems, where providing good feedback about what has been learned from the sample values can facilitate users to understand and eventually correct undesired system behavior [MM01]. Appropriate feedback also lowers the barrier for end-users (i.e., non-programmers) to create scripts by demonstration. Recently, Co-Scripter [LHML08] showed that scripts in human-readable text are easy to use for automating end-user interactions with web applications. Due to their comprehensibility, it is also fairly easy to discuss and share these human-readable scripts across users.

To develop an appropriate feedback mechanism, inspiration can be found in graphic history notations, which are visual depictions of user operation histories. Notable graphic history visualizations are the action history of Adobe Photoshop\(^1\) and comic strip style notations (e.g., [GAL\(^+\)09, MYIM98, MM94]), which use sequences of screenshots together with textual descriptions.

6.2.3 Contextual Assistance

D-Macs should contain a mechanism that provides contextual assistance in order to help designers to overcome errors that might occur during action replay. These errors can occur because of the assumptions that recorded design actions have about the UI they are applied to. For example, an action that updates a button widget can only be applied to a UI that contains a button. In order to guide designers in meeting the requirements of a loaded action sequence, animated visualizations are a good approach to providing contextual assistance. This contextual assistance was preferred over providing written instructions because users rarely pay attention to written instructions [GF10, Har95].

6.2.4 Design Community

In D-Macs, it should be possible to share information with other designers that are member of a design community. Today, it seems to be common practice that designers use other designer’s realizations as sources of inspiration. Reusing and sharing each other’s work is still poorly supported in design tools [GFIR09]. To meet this requirement, D-Macs can find inspiration

\(^{1}\)http://www.adobe.com/products/photoshop/
6.3 D-Macs Features

in various other application domains, such as application development (e.g. HelpMeOut [HMBK10]), web browsers (e.g. CoScripter [LHML08]), information visualization (e.g. ManyEyes [DVWK08]) and recommender systems (e.g. CommunityCommands [MLGF09]). These systems have shown that providing a platform that connects tools with a variety of users empowers these users to discuss and learn from each other’s experiences and expertise.

6.3 D-Macs Features

D-Macs is integrated in GUMMY, allowing designers to prototype multi-device user interfaces visually, using multiple design workspaces. This section discusses the most important interaction techniques and elements that D-Macs offers to augment GUMMY with the desired features. This section is broken down in five parts: action recording, action sequence visualizations, publishing design action sequences, reusing actions in different designs and handling design action errors.

6.3.1 Action Recording

GUI designers can record the actions they perform in the D-Macs design environment by pressing the record button at the bottom of the action recording panel (Figure 6.2-E). In record mode, this panel will visualize the action sequence that is being recorded. An action sequence ends when the action recorder’s stop button is pressed. This way, designers can record multiple distinct action sequences. Figure 6.2 shows three recorded action sequences.

The action recorder allows designers to remove actions from a sequence or add newly recorded actions to this sequence. Actions are deleted by selecting an action and pressing the delete button. In record mode, all actions a designer performs are added to the end of the selected sequence. If a designer selects the first action of this sequence, new actions will be added at the beginning of the sequence. To insert recorded actions between two consecutive actions, the designer first has to select these two actions.

The D-Macs design environment supports a wide variety of design operations that can be recorded. Example operations include direct manipulation actions for resizing, moving or rotating widgets on the design tool’s canvas, actions for adding or removing widgets, property update actions for changing the property values in the properties panel, clipboard actions for copy and paste operations, and tab-switch actions for switching between device-specific designs.
Figure 6.2: D-Macs augments a multi-device GUI builder with a design action recorder. The example user interface design shown here was inspired by the classroom interface example of Gajos et al. [GWW10].
### 6.3 D-Macs Features

#### 6.3.2 Action Sequence Visualizations

D-Macs employs two visualizations to provide rich feedback about how every recorded action affects the GUI design. In one visualization, every action is shown as a combination of a human-readable textual description and a representative icon (Figure 6.3-top). This visualization is similar to how Adobe Photoshop\(^2\) represents its image manipulation action history.

The second visualization concerns a *key frame animation* that appears when pressing the filmstrip icon ![filmstrip icon] on the right of an action. Key frame animations show how a recorded action changed the GUI design. The animation’s first frame shows the GUI design *before* the action was executed, while the last frame depicts the design *after* executing the action. Animations are also produced for complete action sequences and show step-by-step how all individual actions in an action sequence affect the design.

---

\(^2\)http://www.adobe.com/products/photoshop/
6.3.3 Publishing Design Action Sequences

Recorded action sequences can be published to a central repository by pressing the web icon next to every action sequence. When publishing an action sequence, designers can add several annotations such as a title, a textual description and a number of keywords. This metadata might be useful for other designers to understand the rationale behind the sequence.

The central repository can be accessed through a web interface. In the future, this website can become a social platform that gives an overview of all submitted action sequences and stimulates discussion among designers about these recordings. Discussing action recordings can help designers to gain a deeper understanding of each other’s actions, to improve existing action sequences and to learn from more experienced designers. The action sequences listed on this central website can be imported in D-Macs and reused by other designers.

An action sequence that appears on the central website is visualized through a comic strip notation (Figure 6.4). A comic strip presents how a design action sequence affects a GUI design. Every panel in the strip shows a snapshot of the design canvas and a list of the actions that were performed on this snapshot. The comic strip will only start a new panel if there is a visual difference with the canvas snapshot of the previous panel. For example, in Figure 6.4-2 there is no new panel created for the copy action because this does not change the visual layout. Clicking on a strip panel allows designers to add comments about the actions in this panel. An overview of these comments is available through the comment tab.

The comic strip and according discussions can be accessed in D-Macs as well. Clicking on the comments icon next to an action shows the discussion about this action. The comic strip also corresponds to the previously introduced key frame animations that appear after clicking on the filmstrip icon next to an action sequence. Inside D-Macs, key frame animations were favored over comic strips because they are more compact and do not occlude the design workspace.

6.3.4 Reusing Actions in Different Designs

A sequence is replayed in D-Macs by loading the action sequence from the repository and then pressing the play button in the action recorder. During action replay, D-Macs employs a mixed-initiative generalization mechanism that is inspired by CoScripter [LHML08] and allows designers to combine
Figure 6.4: Every shared action recording is visualized as a *comic strip* on the central repository’s web interface.
Figure 6.5: The state icon of an action indicates if it has to be replayed manually or automatically.

...automated actions with manual design actions.

The cornerstone of D-Macs's generalization approach is the state icon (Figure 6.3.4) that indicates for every action if it has to be replayed automatically by the D-Macs system or manually by the UI designer. Designers can easily toggle between the manual and automatic state icon by clicking on it. Manual actions are typically used for the design steps that are difficult to replay in other designs than the one in which they were recorded. When loading an action sequence, D-Macs will automatically put the state of these difficult-to-automate actions to manual.

An action that is typically hard to replay automatically is a selection action. Under the hood, this action selects a component of a certain type at a specific X,Y coordinate. This action will be hard to replay because there is no guarantee about the location of the desired component in other designs. Figure 6.3.4 shows an example of how we employ the D-Macs mixed-initiative approach to solve this automation problem. In this example, we first set the state of the panel selection action to manual. This allows designers to manually select the panel of the current design that has to be affected by the action sequence. After selecting this panel, the remainder of this sequence will take place automatically.

6.3.5 Handling Action Errors

When reusing previously recorded action sequences in other GUI designs, action errors might occur. One example is the selection action that was discussed in the previous section. Another example concerns an action sequence that starts with changing the layout of the selected panel. To execute correctly,
6.4 Architecture and Implementation

This section describes D-Macs’s underlying architecture and discusses the details of its implementation. D-Macs is implemented as an extension of the

---

Figure 6.6: D-Macs provides suggestions (1) for resolving errors that might occur during action replay. When clicking on a suggestion, all parts of the D-Macs design workspace that contribute to follow this suggestion are highlighted (2).

This action requires that there is a panel selected in the current design.

Because we do not want to expose action errors to the designers, D-Macs contains a smooth error assistance mechanism. Assistance is provided by means of two visualizations: suggestions and highlighted regions. Each time an action cannot be executed, suggestions for resolving this error appear below the action. When clicking on a suggestion, all parts of the D-Macs workspace that are needed to follow this suggestion are highlighted using a fade-in/fade-out animation. For example, the action in Figure 6.3.5-1 cannot be executed and the suggestion to “Please select a javax.swing.JPanel” is shown below this action. When clicking on this suggestion, all possible panels in the current GUI design are highlighted in order to emphasize that one of these panels has to be selected (Figure 6.3.5-2).
Gummy design environment. We extended this design tool architecture with the capability to record and replay design actions.

## 6.4.1 Design Action Recording

Action recording is supported through the “Command Object Model” [Ber94]. Every action supported by our GUI builder is implemented as a command object. Each time a designer interacts with the design environment (Figure 6.7-1) to perform a design action, a command object executed (Figure 6.7-2). As commands are executed, they are stored in a command list that serves as a history of all actions that have been taken (Figure 6.7-4). This command list can then be published on the central D-Macs repository (Figure 6.7-5).

Every command object contains a textual and visual description of the interaction that triggered this command. The textual description is supplied as human-readable text during action recording. The visual description, on the other hand, is constructed by a screen grabber (Figure 6.7-3) that takes snapshots of the design tool’s canvas before and after command execution. This combination of visual and textual information is employed by D-Macs to visualize the actions in the action recorder panel (Figure 6.3), to show the key frame animation of every action (Figure 6.3) and to build up the comic strip.

![Figure 6.7: Recorded design actions are internally stored as a list of command objects. These command objects can then be published on a central repository.](image-url)
6.4 Architecture and Implementation

Figure 6.8: Replaying design actions from a command object.

notations on the central repository (see Figure 6.4).

6.4.2 Replaying Design Actions

In order to replay design actions, a previously recorded command list is first loaded into D-Macs. Each command object in this list contains a set of fields (Figure 6.8-1) describing the content that is needed to execute this command. For example, a property update command contains two fields: one for the property name that has to be updated, and one for the value that has to be assigned to this property.

Command objects contain a list of state constraints (Figure 6.8-2). These constraints will check several aspects of the current UI design state in order to determine whether it is possible to execute the command object. For example, a property update action contains a state constraint that checks if the currently selected component has a property that is named like its property name field.

Only if all state constraints are satisfied, a command object can be executed. If one or more state constraints cannot be satisfied, they will produce one or more suggestions (Figure 6.8-6). These suggestions are human readable descriptions that describe how the UI design state should be changed in order to satisfy the state constraint. Every suggestion also involves a number of highlights (Figure 6.8-7), which describe design workspace regions that have to be highlighted in order to draw the designer’s attention. Example regions
that can be highlighted are the design tabs, property fields, UI components on the canvas, UI components in the toolbox, etc.

Once all constraints are satisfied, the actual command object execution can take place. This can be done either automatically or manually. When automatic execution is chosen (Figure 6.8-3), the command object will call its internal `execute` method, which can access the current UI design as well as all command object fields. This execute method can also update the current UI design state. In the manual case (Figure 6.8-4), D-Macs waits for designer input to complete the design action. For this purpose, the command object employs another set of state constraints (Figure 6.8-5). These constraints continuously inspect the UI design state in order to determine if design action execution has been completed. After finishing command object execution, either manually or automatically, the next object in the command list will be notified to start its execution.

### 6.5 D-Macs Example User Interfaces

This section describes how we used D-Macs to create multi-device user interfaces. These examples clarify how D-Macs can be employed in multi-device design projects in order to reduce the amount of repetitive design actions. As a first step in this discussion, we explain how a user interface can be created in GUMMY without automation support. Next, we shows how D-Macs can simplify the creation of this user interface.

#### 6.5.1 Designing User Interfaces Manually

Before we describe how D-Macs can be used to automate multi-device user interface design, we take a closer look at multi-device user interface design in GUMMY without any multi-device design management support. This is explained by means of an example user interface: the *news video explorer* (Figure 6.9). This application allows users to quickly explore large videos in order to find a desired fragment of interest. The news video explorer has a user interface for desktop PCs (Figure 6.9 (a)) and mobile devices (Figure 6.9 (b)), and can also be accessed through a web site (Figure 6.9 (c)). For each target platform, different technologies are being used: the Microsoft .NET framework and the Windows Presentation Foundation (WPF toolkit) for the desktop interface, Android for the mobile application, and Adobe Flex for the web interface.
Figure 6.9: Three user interfaces for the news video explorer: for a desktop PC, a mobile device and for web-based access. All interfaces contain an annotated timeline (1), the desktop and mobile device also contain a video player (2).

The current issues with manual multi-device user interface design are shown by focusing on the design of the news explorer’s central component: the annotated timeline (Figure 6.9-1), which outlines the most important key frames of a video. In the news explorer’s desktop version (Figure 6.9 (a)), this is a custom timeline widget that organizes key frames in chronological order. The mobile design uses a default Android gallery widget (Figure 6.9 (b)),
Design-Tool Macros (D-Macs) allowing users to scroll key frames using touch gestures. In the web-based version, a fish-eye view facilitates users to browse key frames in a 2D grid.

A common problem while prototyping the annotated timeline on the three different devices is the high amount of repetitive design actions. When adding a key frame to the timeline in the desktop version, this key frame also has to be added to the mobile version and the web-based version. This makes it complex for designers to manage and maintain these device-specific GUIs. They have to switch a lot between GUMMY workspaces and repeat similar design actions on each of these workspaces.

The amount of repetitive design actions also raises when keeping the three news explorer interfaces consistent. For example, if a designer wants to update the relevant key frames in the desktop interface, she changes the source property of every key frame accordingly. When she wants to keep the source properties of the key frames in the other designs consistent with the desktop design, she has to adapt these key frames as well.

6.5.2 Designing User Interfaces with D-Macs

We now describe how D-Macs can be used to design the Flex and Android user interfaces of the news video explorer. D-Macs extends GUMMY with multi-device GUI managing support. This way, D-Macs can be used to transform the annotated timeline into the Flex fish-eye view visualization (Figure 6.10-2) and the mobile key frame gallery (Figure 6.10-3). These automatic transformations prevent designers from having to repeat the same design actions over and over again (as described in Section 6.5.1).

Web

For creating the web-based fish-eye view, we first outlined an empty fish-eye view component on the Flex design. This fish-eye view component visualizes an array of images in a grid. We then started to record a design action sequence, which is shown in Figure 6.10-arrow A. This action sequence describes the selection of a key frame image (which is a System.Windows.Controls.Image component) in the desktop .NET design. The value of the source property, which contains the file path of the image, is then copied to the clipboard. Next, this value is pasted in the fish-eye view’s image list on the Adobe Flex design. The image will then appear in the fish-eye view’s grid.

3We used the open source Tile Explorer component, http://demo.quietlyscheming.com/fisheye/TileExplorer.html
Figure 6.10: D-Macs is employed to transform an annotated news timeline from a desktop video explorer into a web-based fish-eye view visualization and a mobile scroll view.

The recorded action sequence now has to be repeated for all images in the timeline. To automate the sequence, the state icon of the key frame selection action is toggled to manual (Figure 6.10-A). The action can now be replayed multiple times. Each time D-Macs reaches the key frame selection action, the designer just has to select the desired key frame that has to be added to the fish-eye view.

Mobile

The mobile version of the news explorer is created by first outlining a scrollable, vertically oriented, custom flow layout panel on the mobile UI. The recorded action sequence for adding the timeline key frames into this list is outlined in Figure 6.10-B. The beginning of this action sequence is similar to the previously discussed action sequence for creating the web application. In the last two actions, a `System.Windows.Forms.PictureBox` widget is added to the flow layout panel and the file path is copied into this widget’s image property.

Action sequence B is automated in a similar way as action sequence A. The
state icon of the second action is toggled to manual. Each time this action sequence is executed, designers have to manually select the key frame that has to appear in the mobile UI.

**Discussion**

When we compare D-Macs with the manual design approach in Gummy (Section 6.5.1), we see that the multi-device UI managing support provided by D-Macs reduces the amount of repetitive design actions. In Gummy, designers had to add a key frame manually to each of the designs. D-Macs follows a different approach. Designers demonstrate once how they copy the source of a key frame from a widget on one platform to a widget on another platform. This recording can then be replayed for all other key frames. While replaying this recording, D-Macs takes a lot of work out of the hands of the designers. The designer only has to point to the frames that she wants to transform from the desktop platform to one of the other platforms.

**6.5.3 Reusing Design Action Sequences**

D-Macs can not only be used to replay design actions in the same design as it was recorded. It can also replay previously recorded design actions in other designs. To show how action sequence A (Figure 6.10-A) can be reused in another UI, we introduce a second example application: the image gallery (Figure 6.11-top). This image gallery is a desktop application created in .NET using the Windows Presentation Foundation (WPF) toolkit. It organizes image thumbnails as a card deck and allows users to view a full-sized image by clicking on its thumbnail.

Using D-Macs, the image gallery UI is transformed into an Adobe Flex fish-eye view gallery (Figure 6.11-bottom). For this transformation, the previously recorded action sequence A (Figure 6.10-arrow A) is reused. On loading this sequence, D-Macs automatically toggles the second step of this action sequence to manual. This gives designers the opportunity to manually select every image they want to use in the Adobe Flex UI. After selecting an image, the remainder of the action sequence is replayed automatically. The result of this transformation is shown at the bottom of Figure 6.11.

Reusing action sequences in other designs than they were recorded is one of the main contributions of D-Macs. This aspect of our tool allows designers to reuse design effort over multiple projects. Reusing design effort helps designers to manage their multi-device user interface design projects more efficiently.
Figure 6.11: Action sequence A, as depicted in Figure 6.10, can be employed to transform a card deck image gallery into a fish-eye gallery.

6.5.4 Maintaining User Interfaces

D-Macs can also be used for maintaining device-specific user interfaces. For example, a D-Macs recording can be used to keep user interfaces consistent. When updating the most important key frames in the Adobe Flex design’s timeline, these key frames also have to be updated in the other two designs. This update actions can be coordinated with sequences similar to the ones shown in Figure 6.11. When updating the source property of one key frame, this value can be copied to the corresponding key frames in the other user interfaces.

6.6 D-Macs Effectiveness Attributes

By means of the example user interfaces in the previous section, we showed that D-Macs is an effective tool for designing multi-device user interfaces. When we compare this tool with our own GUMMY tool without managing support, we see that it reduces the amount of repetitive design actions for
designing and maintaining multi-device user interfaces.

In this section, we are going to further elaborate on the effectiveness of D-Macs. To discuss this effectiveness, we use the evaluation framework of Olsen [Ols07a]. This framework states that the effectiveness of a tool can be discussed by showing that it reduces solution viscosity and supports power in combination. These two concepts will be explained in detail in this section.

6.6.1 Reducing Solution Viscosity

An important characteristic of a good UI design tool is that it reduces solution viscosity, which means that it minimizes the effort required to iterate on many possible design solutions. Three ways in which a tool can reduce solution viscosity are flexibility, expressive leverage and expressive match.

Flexibility

According to Olsen [Ols07a], a UI tool is flexible if it is possible to make rapid design changes that can then be evaluated by users (designers in our case). We believe that D-Macs meets this claim for two reasons. First, it is integrated in GUMMY, an environment to design UIs for many different computing platforms. This allows designers to work in a direct manipulation environment where they can immediately alter the design and directly perceive the result. The second reason why we consider D-Macs as a flexible design tool is because of the recording and replay tool support. A designer can quickly record some actions and replay them at different places in the design or in other design. This allows designers to rapidly make design changes at multiple places and see the result immediately.

Expressive Leverage

Expressive leverage [Ols07b] is when designers can “accomplish more by expressing less”. D-Macs achieves leverage by helping designers to avoid repetitive design steps. Designers can record an action sequence and automatically repeat these sequences multiple times. By sharing and discussing action sequences via a central repository, designers are stimulated to reuse each other’s design action sequences.

A common challenge in achieving leverage is to find a good “generalize and reuse” strategy. D-Macs contains a powerful, yet easy to control, mixed-initiative generalization mechanism. This mechanism allows designers to easily intertwine manual and automatic design steps by toggling the state icon of
each action. When the state icon of an action indicates that it has to be performed manually, designers can complete this action the same way as they usually do with the design environment. They are not required to learn a new, often complicated, user interface in order to express how an action should be generalized. D-Macs assists designers in two ways during the automation process. Firstly, it will mark all actions that are difficult to replay manually when loading an action sequence. Secondly, a smooth error recovering mechanism is applied to help designers in resolving problems with actions that are difficult to replay.

**Expressive Match**

Olson [Ols07a] defines *expressive match* as an estimate of how close the means for expressing design choices are to the problem being solved. D-Macs adheres to this principle by employing a comic strip visualization in the central repository’s web interface. These comic strips tell the story of how a design action sequence can affect a GUI design. Every panel in the strip uses a snapshot of the UI design canvas, which is visually very close to the look and feel of the final user interface. In D-Macs, comic strips can be accessed as key frame animations that provide a compact overview of an action sequence without occluding the design workspace.

### 6.6.2 Power In Combination

*Power in combination* refers to the way UI tools can support new components to create new solutions. This can be supported by minimizing the cost of adding new components and by making these components easy to use and combine.

D-Macs allows designers to add custom widgets in a similar way as in traditional GUI builders. A custom widget can be used in D-Macs just by loading the library that describes this widget. This is certainly a lower cost than existing multi-device UI design tools and UI generation systems that rely on abstract UI models. Adapting these abstract models to support new components is mostly complex and does not suit UI designers, because they often do not master the model formalisms [MHP00]. When the number of supported components grows and the components become more diverse, the effort needed to build a good abstraction model increases as well.

Multi-device design environments usually allow designers to transform UI parts from one target device to another. These transformations are often based on for designers unpredictable artificial intelligence algorithms that may
produce undesired results [MHP00]. D-Macs follows a different approach by describing and visualizing UI element transformations explicitly through understandable and adaptable action sequences. This contributes to a better mechanism for using UI elements in a multi-device design environment.

6.7 Conclusion

This chapter explored a new multi-device user interface design approach through recording and replaying design actions. An important innovation of D-Macs is that it reuses a designer’s expertise instead of complex, often unpredictable, automatic UI transformation algorithms. This is accomplished by a mixed-initiative approach that divides the effort between manual design actions and automatic design decisions. Reuse is stimulated by a central web-based repository that allows designers to share and discuss recorded actions sequences. D-Macs acknowledges the importance of automation feedback and therefore provides several visualizations that help designers to understand how every automated design step works. The combination of all these functionalities makes D-Macs potentially useful in a wide variety of multi-device UI design projects.

D-Macs addresses the issues that we identified with automatic transformations and linked editing in Chapter 5. Instead of using a predefined set of automations, D-Macs empowers designers to create new automations and to control how these should occur. D-Macs also facilitates the integration of custom components and to let them participate in the automation by recording a new set of design actions and replay it afterwards. This does no longer require designers to adapt the underlying models.

One of the limitations of D-Macs is that it does not contain a mechanism to split property values. For example, assume that a widget has a property for its RGB color value, which is specified by means of a comma separated string (e.g. “0,255,34”). If we want to copy this value to a widget on another platform that has one property for red, one for green and one for blue, we need to split the copied value first. This is not yet supported in D-Macs but might be possible using regular expressions. The drawback of regular expressions is that it can be difficult for UI designers to specify such expressions.
Part IV

Optimizing User Interface Designs
Chapter 7

Live User Interface Design

7.1 Introduction

The third research question that we formulated in Chapter 1 concerned how tools can help designers to optimize their designs for the targeted devices. Optimizing UIs means fine-tuning these UIs until they meet the constraints of the target device. These constraints include the available screen size, the input method and the sensitivity of a touch screen. From our multi-device design tool needs analysis (Chapter 3), we learned that designers would like to optimize designs by testing their design decisions immediately on the target device. This is consistent with the idea of creating user interfaces through an iterative design-test process, which originates from the field of user interface prototyping. In this domain, it is common to frame and evaluate a design challenge by working it through, rather than just thinking it through [KHT06, Sch84]. Practically, this means that designers create a part of the user interface and then immediately test this part on the target device. Testing the designs on the target device will help designers to uncover new ideas for optimizing and fine tuning the design.

In this Chapter we discuss Live User Interface Design approaches, which allow designers to design and/or adapt user interfaces on the target device while it is running (i.e. the interface is “live”). Tools supporting such an approach are also known as (user-)adaptable systems. We believe that Live User Interface Design approaches are very suitable to support an iterative design-test process. Most tools that support these approaches allow designers to intertwine design and test actions at any moment without additional
We favored Live User Interface Design tools over adaptive tools, which automatically adapt user interfaces to the context of use. A well known adaptive tool is the Supple UI optimization [GWW10] toolkit (which we also discussed in Section 2.3). Supple first monitors how users can perform tasks like pointing on an object, dragging an object or selecting an item from a list. Based on the monitored properties, Supple will then adapt user interfaces so that they are better suited for the monitored user. For example, a user interface that is optimized for users that can perform rapid but inaccurate mouse movements usually contain relatively large targets. Automatic optimization algorithms have shown to be effective to make user interfaces accessible for motor impaired users [GWW08]. However, despite their advantages, this type of tools is not oriented towards user interface designers. The primary reason for this is the lack of control designers have about the optimization algorithms, which might result in unexpected design adaptations.

This Chapter gives an overview of existing tools and techniques that can support Live User Interface Design. The goal of this literature study is to verify whether existing Live UI Design techniques have the potential to be used in multi-device user interface design. An overview of the Live User Interface Design tools and techniques that are discussed in this Chapter is shown in Figure 7.1. The two major types of Live UI Design tools are:

- **on-device** tools, allowing designers to create UIs on the target device while the UI is running. This type of tools can be further subdivided in on-device adaptable systems, which only facilitate the adaptation of existing UIs, and on-device design systems, which go one step further and allow designers to design UIs from scratch.

- **connected-device UI design**, allowing designers to design a UI in a design environment (on a PC) which is tightly connected with the running UI (on the target device).

In this chapter, we first describe on-device adaptable systems, on-device design systems, and connected-device UI design tools. After introducing these three techniques, we discuss whether these techniques can scale over multiple devices. Based on this discussion, we draw the conclusions about how live UI can be supported in multi-device UI design.
7.2 On-device Adaptable Systems

Live user interface design is mostly supported by means of On-device Adaptable Systems. These systems are mostly oriented towards application end-users and only allow these users to adapt (or fine tune) existing GUIs manually. As shown in Figure 7.1, on-device adaptable systems are typically running on desktop PCs or mobile devices. The following describes systems that are running on one of these devices.

7.2.1 Desktop Adaptable Systems

A notable example for desktop-based user interface adaptation is the User Interface Façades [SCPR06] system. This adaptation system is part of a Linux window manager and facilitates users to copy a part of a user interface from one program to another program. For example, it would be possible to place a part of a calculator program inside a word processor. User Interface Façades
also supports standard rearranging operations like component resizing. Furthermore, components can be replaced with other components. For example, a drop down widget containing country names can be replaced by a map widget where countries can be clicked.

Arnauld [GW05] is a user interface adaptation system that is part of the multi-device Supple toolkit [GWW10]. After generating a user interface, Arnauld can be used to adapt this interface. By right clicking on the rendered interface, designers can choose to remove and replace widgets. Currently, Arnauld can only be used on desktop-based Supple user interfaces and not on their mobile versions.

Demeure et al. [DMLC08] propose a toolglass-based GUI adaptation technique on top of the Comets toolkit [DCCV06]. This approach allows designers to put a toolglass on top of a running system. Through this toolglass, the underlying interface can be adapted by relayouting components and replacing components by other components. The Comets toolglass is only available to adapt desktop-based user interfaces.

The Clip Connect and Clone (C3W) [FLHT04] tool allows users to clip elements from a web page onto a new web page. These elements can then be glued together in order to create new web pages from already existing web pages. C3W supports only the adaptation of web pages on desktop PCs. d.mix [HWCK07] works in a similar way than C3W. It allows designers to reuse elements from different existing websites into new - custom - webpages. This way, web pages can be adapted in order to better fit target device constraints.

7.2.2 Mobile Adaptable Systems

There are only few systems allowing users to adapt a user interface on a mobile device itself. The PageTailor [BRM+07] system allows users to adapt web-pages in order to better fit mobile device screen sizes. PageTailor is a browser plugin that runs on the mobile phone itself. After selecting a webpage element, users can do several manipulations on this element. The selected element can be removed, resized and moved. PageTailor runs best on mobile devices that facilitate stylus based interaction.

As a part of the mixed-fidelity design framework, De Sá et al [dSCDR08] presented a mobile UI adaptable system that is oriented towards designers and end-users. It facilitates user interface fine-tuning on the mobile device once the user interface is running. Some example operations that are supported are moving, deleting or resizing widgets. The major constraint of this mobile UI adaptable system is that it only works on devices that support stylus based
interaction on a screen.

The mobile phones that are currently available on the market also provide mechanisms to adapt web pages. For example, iPhone and most touch-screen Android devices support pinch and zoom gestures while browsing web pages. Using these gestures, users can use zooming and panning to fit a web page within the limited screen space.

7.3 On-Device UI Design Systems

On-device UI Design Systems go further than on-device adaptable systems. Instead of only adapting and fine-tuning user interfaces, these systems allow designers to design GUIs from scratch on the target device. On-device UI Design Systems are often based on dynamic programming languages such as Smalltalk. Morphic [MS95] is a Smalltalk based UI construction environment. It introduces the concepts liveness and directness as a way user interface designers can examine or change the attributes, the structure and the behaviour of user interface components by pointing at their graphical presentation (directness) while the user interface is running (liveness). These properties were also applied during the development of the Parks PDA [OMO03], which showed that doing small amounts of programming directly on the mobile target device saves development effort. Sun’s Lively Kernel [Ing08] is a web-based design environment adopting the liveness and directness concepts. It is written in Javascript, allowing designers to edit those scripts while the web-page is running.

A drawback of the aforementioned approaches is that they are not really oriented towards user interface designers. Morphic’s usage model to update some parts of the interface is comparable with a program debugger instead of a design environment. The parameters of all classes and objects in the interface can be inspected and modified while the interface is running. For adapting scripts in the Lively Kernel interface, designers have to update a script manually, which is comparable to programming.

7.4 Connected-Device UI Design

This section describes existing design environments that embrace the target device as a part of their design workspace. These design tools are tightly connected with the target device, which mostly results in continuous and/or fast interface deployment. This makes it easier for designers to test their designs.
Two connected-device environments are discussed below: d.tools [HKB+06] and Highlight [NL08].

The d.tools [HKB+06] environment enables designers to prototype the physical layout of a device (i.e. sensors, screen, etc.) and an appropriate user interface for this device. Inside the d.tools design environment, which runs on a PC, a designer can create user interfaces by combining predefined widgets. Each time the designer updates this UI design, a running version of this design is automatically updated on the target device. This way, designers can immediately test whether their design works well on this device. An important drawback of d.tools is that the resulting user interfaces are running using a custom toolkit which only contains a limited set of widgets. d.tools is also not able to target multiple device types, they are restricted to devices that are built using their physical toolkit.

Highlight [NL08] is a web-browser extension that allows creating mobile versions of web pages. These mobile website versions are based on the way a user interacts with the desktop version of this website. While interacting with a website, Highlight suggests in real time how this website can be subdivided into several pages (called pagelets). Highlight also provides a mini design environment to adapt the look and feel of the proposed mobile version. Although Highlight did not allow real-time simultaneous viewing on the actual mobile device, it’s underlying architecture can be easily extended to support this feature.

### 7.5 Multi-Device Scalability

In order to find a good multi-device live design strategy, this section discusses how the aforementioned live design environments can be scaled over multiple devices.

All Live User Interface Design tools have two components:

- a **primary user interface**, which is the actual user interface design that is being created;
- a **secondary user interface**, which is the user interface and the interaction techniques that are needed to create the primary user interface.

In the case of on-device adaptable systems, it would be hard to create such a secondary user interface that scales over multiple devices. The differences in input and output mechanisms across devices make it impossible to find this secondary design user interface that can work on all targeted devices.
For example, mobile phones and interactive TVs have different input devices. Mobile phones have touch screens or keypads as input device, while interactive TV has remote controllers. This diversity of input devices makes it impossible to define interactions that allow interface design on all mobile phones and on interactive TVs. Altering user interfaces with a stylus (like in the mixed-fidelity framework of de Sá [dSCDR08]) will only work on mobile phones that support stylus based interactions and not on other mobile phones or digital TVs.

Connected-device design environments always run on a PC and tightly integrate the target device in this design environment. Because they run on a regular PC, there is always a mouse and keyboard available. This availability of mouse and keyboard makes eases defining one design interaction technique that can scale over multiple target devices. This way, designers only have to learn one interaction technique for many devices instead of one per device as in on-device design systems. An example design technique that can scale over many devices is presented in d.tools [HKB+06], which facilitates interface design in its design environment and “streams” every design update immediately to the running design on the target device.

7.6 Conclusions

In this Chapter we gave an overview of Live User Interface Design tools, which allow designers to design and/or adapt user interfaces on the target device while it is running. Live User Interface Design tools are very suitable to support an iterative design-test process because these tools allow designers to intertwine design and test actions without additional deployment effort. An iterative design-test process helps designers to optimize their designs so that these designs are likely to meet the constraints of the target devices.

We discussed three types of Live User Interface Design tools in detail: on-device adaptable systems, on-device UI design systems and connected-device UI design tools. While the first two facilitate interface design directly on the target device, the latter tightly integrates the running UI (on the target device) with the design environment (on the PC). On-device UI Live UI Design tools are difficult to apply in multi-device UI design, because these tools need a different design interaction technique for each device. Connected-device design tools are better suited to scale over multiple devices, because there is always a design tool available on the PC. This way, designers only have to learn one design interaction technique to target many platforms.

In the next Chapter, we will elaborate further on this conclusion and in-
vestigate different techniques that support Live User Interface Design in a multi-device setting.
Chapter 8

Multi-Device Live UI Design

8.1 Introduction

From the exploration of Live User Interface Design solutions in Chapter 7, we concluded that Connected-Device Design tools have the potential to scale over multiple devices. This type of design environment allows designers to create user interfaces in a design tool on a PC and to easily test and observe the user interfaces on the target device. For example, d.tools shows every update that occurs in the design environment in a running version of the design on the target device [HKB+06]. This running version can then be used by the designer to test their designs when the design is not yet finished. These short design-test iterations contribute to the creation of designs that are optimized for the targeted devices.

This Chapter explores how target devices can be closely connected with a multi-device design environment. Several live design techniques that allow designers to intertwine design and test actions are investigated. In order to experiment and develop these design techniques, a multi-device live UI design framework is proposed. This framework builds upon GUMMY’s architecture.

The flexibility of the proposed live UI design framework is demonstrated by implementing three design techniques:

- the **design mirror**, which streams every design action in GUMMY immediately to a running version of the UI design on the target device;

- the **design toolglass**, which is a semi-transparent design canvas that can be placed on top of a running UI to start designing this UI;
the multi-device continuous design pointer, which allows designers to work directly on the target device by moving a mouse pointer between the design environment on a PC and the running UI on the target device.

At certain points during the development of these techniques, informal tests with HCI professionals were conducted to gain feedback about how we could further improve the techniques.

This chapter first discusses the framework for multi-device live UI design tool support. The next section describes the usage model and development of three distributed design interaction techniques by means of this framework. Finally, we draw the conclusions.

8.2 A Multi-Device Live UI Design Framework

In order to experiment with multi-device Live User Interface Design techniques, we extended Gummy’s MVC architecture (as described in Section 4.3). This new architecture has to make it possible to tightly integrate the target devices as a part of the design tool. To reach this goal, the revised architecture (Figure 8.1) contains two communication channels between target device(s) and PC:

- the interaction channel (Figure 8.1-1) for sharing input and output devices and implementing live design techniques;
- the design channel (Figure 8.1-2) for updating the user interface design.

8.2.1 Interaction Channel

The interaction channel is implemented through interaction routers, which are located on the PC (Figure 8.1-3) and target device (Figure 8.1-4). The goal of these interaction routers is to redirect input from the design environment to the target device and vice versa. Interaction routers listen to low-level interaction events and decide where this interaction should take place. For example, when a designer moves the mouse of the PC, the interaction router can be implemented to redirect this mouse movement to the target device. The interaction router on the target device will receive these mouse events and update a mouse pointer on this device. This way, the mouse on the designer’s PC can control a mouse pointer that is located on the target device.
Another example of interaction routing is when a designer touches a widget on the live running UI on the target device. The interaction router on this target device can be implemented to redirect this touch event to the designer’s PC. The part of the design environment that runs on the PC can interpret this event as the selection of a component. This can be visualized by e.g. updating a properties panel that shows all properties of the selected component.

The interaction routers can communicate with each other by means of a TCP and/or UDP connection that exists between target device and PC. We choose to have direct socket communication because of efficiency. For example, when routing mouse coordinates to the target PC there should be immediate feedback from the pointer. This would be difficult to establish with the XMPP protocol, since messages always have to pass an XMPP server before they arrive at the destination.

8.2.2 Design Channel

The design channel (Figure 8.1-2) is responsible for updating the user interface design. This channel uses the same messages as in the original GUMMY MVC architecture (Section 4.3). When a designer performs a design action in the
design environment, this action is sent to the platform plug-in. This platform plug-in then updates a running version of the designed UI so that it shows the effect of the design action. Next, the platform plug-in sends back an updated CUI model and a screenshot of the running UI.

8.3 Design Mirror

8.3.1 Scenario of Use

The design mirror is a live design technique that extends GUMMY by streaming every action that a designer takes in GUMMY directly to a running UI on the target device (see Figure 8.3.1). This technique is inspired by the d.tools design tool [HKB+06], which shows every update that occurs in the d.tools design environment in a running version on the target device. The design mirror extends d.tools by targeting multiple devices and toolkits instead of only one.

The two major components of the design mirror are the GUI builder and the live view (Figure 8.3.1). The live-view is situated on the target device and harvests a running version of the currently designed user interface. When a designer changes the GUI design in the GUMMY GUI builder, each change is immediately visualized in the running GUI of the live-view. For example, if a designer moves or resizes a component in the design workspace, the runtime version of this component is moved or resized in the live view on the target device simultaneously. By visualising every change on the target device immediately, designers do not have to switch between run and edit mode to test the runtime implications of design changes. This way, designers are not forced to invest additional time or effort in acquiring and optimizing the runtime characteristics of their design.

8.3.2 Evaluation

To gain feedback from user interface professionals about the design mirror, we conducted an informal user study. The purpose of this study was to check if the subjects were able to create user interfaces for a Samsung Ultra Mobile PC (UMPC) with D-Macs. More specifically, we wanted to assess whether the participants took the output of the live-view into account and changed their design decisions according to what they perceived in this live-view.

Seven subjects participated in our study, having different backgrounds and experiences. Five of the subjects were experienced programmers, two of the
After reading a written manual, the participants were asked to create a “home automation system” user interface for an UMPC. The UI was described by means of a written list of functionalities the resulting UI should contain: changing the temperature of the heating, controlling the lights in several rooms and activating the alarm. During the experiment a think aloud protocol was used to understand the actions and decisions of the participants. Additional questions were asked by the observers if necessary.
8.3.3 Results and Discussion

We observed that five participants changed their design decisions according to the feedback they got from the live-view. Two subjects first selected a trackbar to change the temperature of the heating. Immediately after they dragged the trackbar to the canvas, they tested its runtime characteristics and discovered that it was difficult to manipulate this widget using the UMPC’s touch screen. This made them decide to use another widget. While one subject replaced the trackbar with two larger buttons to increase or decrease the temperature, the other one opted for a textfield where the temperature can be entered using the UMPC’s keyboard.

Another subject immediately tested the runtime behaviour of the button she dragged to the canvas. This way, she discovered difficulties to hit the
Design Toolglass

8.4 Scenario of Use

As a second live design technique, we extended the design mirror into the design toolglass. This design toolglass (see Figure 8.4) is a see-through interface [BSP+94] that is placed on top of a UI running in a device emulator. If there is also a target device available, the design toolglass will stream the design actions to a running UI on this target device as well (similar to the Design Mirror).

The design toolglass’ workspaces (Figure 8.4-a) are structured in a similar way as GUMMY’s workspaces. The major components are also a toolbox with UI elements that are suited for the selected target platform (Figure 8.4-b), a canvas (Figure 8.4-c), a properties panel (Figure 8.4-e) and a widget tree (Figure 8.4-d). The canvas is a transparent overlay that triggers design actions on top of a running UI design. This UI is can either run in a device emulator (e.g. the Android or Windows Mobile emulator in Figure 8.5) or directly on
the PC of the designer (e.g. Adobe Flex in a webbrowser in Figure 8.5). To select a certain UI element, the designer has to click through the canvas on this element. This results in a bounding rectangle that is drawn on the canvas around the selected element (Figure 8.4-F). The properties of the selected component can be observed and adapted through the properties panel. New components are added by dragging an element from the toolbox to the canvas.

If a target device is available, the design toolglass will also behave as a design mirror and stream design actions immediately to the actual target device. On this target device, design actions can then be tested. However, if the actual device is not (yet) present, designers can minimize the toolglass and test their design on the emulator or directly on their PC. The fact that designers can work with emulators and devices is a big advantage of the toolglass. It allows us to target toolkits and operating systems before devices are present. For example, the toolglass worked on the Android emulator even before a commercial Android device became available.

Figure 8.4: The design toolglass is structured similar to GUMMY.
The design toolglass reduces the UI deployment time dramatically. To switch to testing mode, the design toolglass brings the running GUIs to the front. This reduces deployment time to a few milliseconds (i.e. the time needed to bring a window to the front). The design toolglass is implemented in Windows and can automatically align the running user interface with the see-through canvas (using Windows API calls).
8.4.2 Evaluation

Feedback about the design toolglass was gained by conducting an informal experiment with a group of HCI professionals. This group contained 8 participants, all having a reasonable level of experience with UI design. Seven participants were HCI researchers, one participant was a user interface designer.

After reading a written tutorial, the participants were asked to create a music player GUI for Windows Mobile and Adobe flex. The designers were provided with two mockup sketches that described a music player with features like selecting a song in a playlist, controlling the volume of the music and navigating through a song. During the experiment, a think aloud protocol was used to understand the actions and decisions of the participants. Additional questions were asked if necessary.

8.4.3 Results and Discussion

We observed that the participants mostly switched to testing mode if they were using non-native user interface controls. Especially when designing the flex-based media player interface, we noted that designers frequently switched to the running UI. This can be explained because of the availability of richer custom UI controls like a fish-eye menu and a round slider in this toolkit.

An example of the role of the design toolglass in testing components was the design of the music player’s volume control. In the Adobe Flex design, we noticed that 7 participants used a custom round range slider for this feature. When dragging this component to the canvas, participants were wondering “how this widget should be manipulated in order to raise or lower the volume?” and “do I need to press or turn this rounded button?”. To resolve this problem, six participants switched immediately after adding the widget to the running UI and tested the behaviour of the volume button in the browser window. The participants liked this instant switch, some were surprised about the way the volume button should be manipulated: “Cool, the volume widget can be turned! At first sight, I thought that it was a press button”.

8.5 Multi-Device Continuous Design Pointer

During the evaluation of the design mirror, several designers suggested to add a distributed direct manipulation technique that allows dragging elements from the design tool directly to the target device. The multi-device continuous
**8.5 Multi-Device Continuous Design Pointer**

*design pointer* is inspired by this suggestion. It is a special mouse pointer which can for example start on the desktop PC of the designer, cross the bezel of the screen and end on the screen of the target device (see Figure 8.6(a)).

The multi-device continuous design pointer is inspired by cross-display operations in multi-display environments. Notable examples in this domain are *Stitching* [HRG+04], *Pick-and-Drop* [Rek97] and *HyperDragging* [RS99]. These approaches allow to use a pen or a mouse to e.g. drag elements from a tablet PC to a Personal Digital Assistant (PDA). Besides traditional input devices like a pen or mouse, researchers have proposed other ubiquitous input devices such as the *Smart Phone* [BBRS06] or *Soap* [BSW06]. To manage how different input devices can be used within (large scale) multi-device and multi-display environments, architectures like *iStuff* [BRSB03], *PointRight* [JHWS02] and *Synergy* [syn] have been proposed. Compared with our approach, the aforementioned systems do not focus on the design of user interfaces. We adopt a multi-display mouse pointer as a mean to design user interfaces for different types of mobile devices while they are running.

This mouse pointer allows designers to run the design canvas on the target device by means of the running UI. On the other hand, the design tool’s toolbox and properties panel are hosted by the PC. The special mouse pointer is used for repositioning and resizing UI elements, adding new elements by dragging an element from the toolbox on the designer’s PC to the running UI on the target device and selecting UI elements.

The design pointer is employed to edit a UI design, the normal interaction method (e.g. pressing on a touch screen, the normal mouse cursor, etc.) can still be used to interact with the design. Thus, the actions performed with this design pointer will not affect the internal state of the user interface elements and thus clearly separates design actions from the UI behaviour. For example, *clicking* on a checkbox element with the design pointer will not check or uncheck this element (= design action) whereas *pressing* on the same checkbox through the PDA’s touchscreen will check or uncheck this element (= UI behaviour).

As an example, consider the mouse pointer to add a button by dragging a button element from the toolbox to a windows mobile design. This button can then be tested immediately in order to discover that it might be too small and occluded by the designer’s finger when pressing on it. Using the mouse, and thus the design pointer, the button can be selected, which results in a black modifier around this control (see Figure 8.6(b)). This modifier allows designers to resize and move the control with the design pointer using direct manipulation. The button can now be enlarged until it is large enough.
Figure 8.6: The multi-device design pointer can cross the bezel of the PC screen and end on the target device screen (a). On the target device, this pointer can be used to select, resize and reposition elements (b).
to perceive its feedback.

8.6 Conclusion

In this Chapter, we contributed a framework for implementing Connected-Device Multi-Device UI Design techniques. The proposed framework makes it possible to develop design environments that are closely connected to the target devices. These design environments allow designers to intertwine design and test actions. The flexibility of our architecture was demonstrated by implementing three design techniques: the design mirror, the design toolglass and the multi-device continuous mouse pointer.

During the development of these techniques, we conducted evaluations with HCI professionals. These evaluations helped to gain feedback about the usefulness of these techniques and about how we could further improve our techniques. During these evaluations, we saw that most HCI professionals improved their designs by testing at regular basis on the target device. These observations confirm that connected-device live UI design techniques can help designers to optimize their designs.
Part V

Conclusions and Future Work
Chapter 9

Extensions and Future Work

GUMMY has been freely available for about three years. This gave other researcher the opportunity to further extend this tool. This chapter describes various extensions of GUMMY that have been created in collaboration with other researchers. Next, we discuss some opportunities for further research.

9.1 Web-Service Integration

In the context of the CROSLOCiS project [cro08], GUMMY has been extended with the application glue plug-in. This plug-in allows designers to connect a user interface design with a web-service without having to write programming code. By easily connecting services with UI designs, designers can check how their prototypes will look like when connected to real data.

Figure 9.1 shows GUMMY with the application-glue plug-in. This plug-in can load web-services that are specified by means of the Web Service Description Language (WSDL). The loaded services are shown in the application glue panel (Figure 9.1-1), every service has its own tab-page. The service methods are shown in the table, each having input parameters, a method name and an output value (Figure 9.1-2).

When a service is loaded, a designer can connect the input and outputs of its functions to user interface widgets. This can be done easily by dragging these inputs and outputs from the application glue panel to the widgets on the canvas. When dropping these inputs and/or outputs, the system asks the designer to which property it should be connected. The method name can
Extensions and Future Work

Figure 9.1: GUMMY’s extension for connecting GUIs with services.

also be dragged to a widget. When dropping this name, the system asks which event that should be responsible for executing the service method.

An example of a connected service method is given in Figure 9.1. The GetPictureImage method (Figure 9.1-2) has been connected to three widgets. Its input parameter, the ID of an image, is connected to a textfield’s text property (Figure 9.1-3). The output, a binary representation of an image, is connected to the source property of an image widget (Figure 9.1-4). This GetPictureImage should be triggered when clicking a button, which means that the method name has to be connected to a button’s click event (Figure 9.1-5).
9.2 Automatic Usability Standards Evaluation

In collaboration with the German DFKI institute, we developed GUIDE2ux [MLS+11], an extension of Gummy for automatically evaluating whether a UI design meets specified usability criteria. GUIDE2ux facilitates designers to adjust UI designs if they do not meet certain usability criteria, even before the UI has been completed. These changes are less expensive compared to changes that are made as a result of a traditional evaluation, which usually takes place after the interface or prototype has been built.

While designing a user interface in Gummy, GUIDE2ux continuously verifies if the design is structured conform a predefined set of usability rules. If the design does not follow these rules, usability warnings are shown in a separate tabbed panel (see Figure 9.2-d, first tab page). For example, the label “pasta” of the selected radio button in Figure 9.2 does not start with an upper-case letter. As shown in the warnings panel, this does not correspond to a predefined rule, stating that every label should start with an upper-case letter except for specific cases. This warning will disappear when the designer makes the first letter of this radio button upper case.

Each warning can be further explored by the designer (Figure 9.2-e) by simply right-clicking the warning. In cases when the warning is not relevant, the designer can also choose to hide the warning and avoid cluttering with
irrelevant data. When further exploring a warning, this warning is displayed in a separate window together with a clear example showing when this warning occurs and how it can be resolved (see Figure 9.2-f). If the designer decides to remove the warning, the system will neglect the rule causing this warning in the future.

Besides usability warnings, GUIDE2ux also provides tips about the widgets that are used in the design. Tips are no strict usability rules, they provide advise on how the usage of certain widgets could be improved. For example, in our food menu application, GUIDE2ux advised to place the title ("Select food type") of the radio button list on the left or on top of this list.

GUIDE2ux also gives general tips about the designed interface. For example, while designing the food-menu touch screen application, general tips about touch screen design will be shown. In this example, the system provides general tips about e.g. the fat finger problem: “a finger is larger than a mouse pointer: buttons should be large enough for a finger to touch easily”.

9.3 Integration of Contextual Information from Storyboards

GUMMY is compatible with COllaborative MultIdisciplinary user-Centered Software engineering (COMuICSer) [HVdBM+11], a tool for specifying requirements and contextual information by means of storyboards. In COMuICSer, a storyboard is defined as: a sequence of pictures of real life situations, depicting users carrying out several activities by using devices in a certain context, presented in a narrative format. From this definition, the four primary pieces of information that can be found in a storyboard are clear: users, activities, devices and context.

Figure 9.3 shows a storyboard that has been created with COMuICSer. The storyboard visualizes the future use of a new video searching application in the broadcasting industry. In this storyboard, a professional video searcher queries a video archive for video fragments of a sportsperson. First, the video searcher retrieves interesting videos in his office, behind his own desk (Figure 9.3-1). He then saves the files on his mobile device and hurries to a meeting room where he will discuss the video fragments with some colleagues (Figure 9.3-2). In this meeting, a large display is used to browse through the available video fragments and to talk about the most suitable fragments (Figure 9.3-3). Afterwards, the video searcher goes back to his office and starts assembling the resulting video, which will be displayed on television.
When creating the aforementioned storyboard in COMuICSer, designers can annotate information such as the devices that are being used in the different scenes. These annotations are made in a similar way as the photo tagging features on Facebook or Flickr. Such an annotated storyboard can then be imported in GUMMY. When importing the storyboard, GUMMY automatically adapts its workspaces according to the target platforms specified in the storyboard annotations. For the storyboard in Figure 9.3, GUMMY will load workspaces for a mobile device, desktop device and large screen device. Apart from the workspaces, GUMMY will also visualize the storyboards graphically. Showing storyboards while designing UIs gives designers a clear view on the future context of use of the designed application. This way, storyboards provide guidelines for the designed UI.
9.4 Future Work

In this section, we discuss some possible directions for future research.

9.4.1 Tool Support for Specifying GUI Behaviour

The research presented in this dissertation focuses on prototyping the look of a user interface, we did not consider the implementation of user interface behaviour. In future research, it would be interesting to explore how our tools and techniques can be used for specifying this interface behaviour.

In order to facilitate the specification of UI behaviour, we could extend GUMMY with scripting support. This would allow designers to program UI behaviour by means of a set of scripting methods. However, when scripting UI behaviour, designers first need to learn the scripting language syntax as well as the widget events and the methods that are needed to define the desired behaviour. This results in a high learning curve for designers to start writing scripts.

One way to lower the threshold for creating UI behaviour scripts is by turning the design mirror (Chapter 8) into an interactive event debugger. This interactive event debugger could visualize the events that are triggered when interacting with the running version of the UI design on the target device. For example, when a designer adds a button to her design in GUMMY, this button is also shown in the running UI on the target device. If the designer now presses this button, the “click” event is triggered and visualized in the design tool. This way, the designer knows that the click event has to be used when specifying button behaviour. A good way for visualizing the triggered events could be by means of a time line, which are well known by designers for specifying animations in professional design tools such as Adobe Flash and Expression Blend. Timelines also have shown to be useful for analysing interactions with physical UIs [HKB+06] and web-based UIs [OM09], and for stepping through program behaviour [KM08].

Another area of future research that could lower the threshold for programming UI behaviour is by combining D-Macs with example-centric programming tools like Blueprint [BDWK10]. Blueprint allows designers to easily find example programming code that is available in online forums, code repositories or technical blogs. Once designers find a piece of programming code that expresses the desired behaviour, they can insert this code in their own script. Then they have to adapt it so that it works with their own graphical design. In order to adapt the code, a D-Macs macro could be useful. This macro
9.5 Conclusion

should contain all actions that are necessary to successfully connect the example script with the graphical design. For example, assume a script that animates a button so that it moves around in circles. When a designer wants to use this script in her own design to animate a listbox, she can execute a D-Macs macro that starts with a manual action where she has to select the widget she wants to animate. The next steps in the macro will then automatically adapt the script so that it animates the selected widget. This script will then be added to the current behaviour script.

9.4.2 Better Design Community Support

D-Macs provides a central repository where designers can share their own recordings and search for other designers’ action sequences. We now use our own repository for this, but in the future it would be interesting to integrate this repository in existing social networks like Facebook. This allows designers to form a community that goes beyond sharing design actions and supports more informal contacts as well.

In the current version of D-Macs, designers have to search the central repository and have to decide themselves which sequences are useful for their current design. We believe this effort can be lowered by automatically recommending relevant design sequences in the D-Macs workspace. Automatically recommended actions would further stimulate reuse because designers are continuously informed about the action sequences that are relevant for their current work. An interesting approach that explores this recommendation idea is CommunityCommands [MLGF09].

9.5 Conclusion

In this Chapter, we discussed how GUMMY served as a framework for performing research in the field of Human Computer Interaction (HCI). In collaboration with other researchers, GUMMY extensions were realized to connect GUMMY designs with web services, to evaluate GUMMY designs against usability standards and to use GUMMY in a storyboarding approach.

We also discussed some opportunities for future research. In the future, it would be interesting to investigate how the techniques presented in this dissertation could be extended to specify UI behaviour and to support design communities.
Chapter 10

Conclusions

In this Chapter we discuss how we have answered the research questions that we have postulated in the introduction of this dissertation. Furthermore, we present an overview of the scientific contributions and publications presented in this dissertation.

10.1 Achievements and Main Contributions

In Chapter 1, three research questions were formulated. To answer these questions, this dissertation contributes various tools and techniques for creating, managing and optimizing multi-device user interfaces:

RQ1. Creating multi-device user interfaces

- The creation of multi-device user interfaces is supported by means of a graphical design approach. This approach is supported by means of Gummy (Chapter 4), a design environment that is able to target a broad range of target devices. Compared with existing multi-device design tools, GUMMY has a better expressive match because it facilitates visual UI design. We based GUMMY’s usage model on traditional GUI builders, which allows designers to design multi-device user interfaces without raising the expertise threshold.

- GUMMY served as a framework for performing research in the field of Human Computer Interaction (HCI) (Chapter 9). In collaboration with
other researchers, GUMMY was extended in order to facilitate the integration of web services, to evaluate GUMMY designs against usability standards and to use GUMMY in a storyboarding approach that integrates contextual information.

RQ2. Managing multi-device user interfaces

- to better manage multi-device user interfaces, we extended GUMMY with two new design techniques: automatic transformations and linked editing. The former allows designers to transform a part of a UI from one device to a UI for another device. The latter allows designers to edit these transformed UI components in concert in order to keep their content consistent across devices. Both design techniques are based on an underlying abstract model, which is often difficult to extend for UI designers.

- We augmented GUMMY with Design Tool MACroS (D-MACS), which allows designers to decide by themselves how a part of a UI should be transformed from one device to another. Using D-MACS, designers can record a set of actions and replay these actions afterwards in the same or another design. This approach allows designers to define their own transformations without having to know anything about abstract models.

RQ3. Iteratively optimizing user interfaces for the targeted devices

- To optimize a user interface for its target platform, we propose an iterative design-test process. This process is supported by several live UI design techniques, which facilitate designers to easily switch between designing and testing. The first technique is the design mirror, which visualizes every design update in the design tool immediately to a running version of the design on the target platform.

- A second technique that facilitates short design-test iterations is the design toolglass. This toolglass is a design tool with a semi-transparent canvas. By placing the toolglass on top of a running UI (which can run in e.g. a device emulator), designers can design and update this UI.

- The final design technique is the continuous multi-device design pointer. This is a mouse pointer that can move between the design environment and the running UI on the target device. For example, this
The work presented in this dissertation was performed over three and a half years of research. In this section we present an overview of the publications disseminated from the work discussed in this dissertation.

This paper describes the first version of our multi-device design environment. This environment has been used in Chapter 3 to analyze the requirements for a multi-device design environment.

This paper describes GUMMY’s usage model and architecture. It formed the basis for Chapter 4.

This paper explores the first idea of using a toolglass to design user interfaces. Back then, this toolglass was still closely connected to automatic user interface generation toolkits. This idea has been worked out in more detail in [MLC10b] and is discussed in Chapter 8.

In this paper, we describe the concepts of how a design tool can empower...
designers to optimize user interfaces. This idea forms the basis for the live UI design techniques that we presented in Part IV.


In this paper, the continuous multi-device mouse pointer is discussed. This is a technique for optimizing a user interface for the targeted device, as discussed in Chapter 8.


This paper describes the design mirror, which was called “GUMMY-live” back then. It describes the idea of mirroring design updates to the target device and presents a first evaluation of this technique. This topic has been discussed in Chapter 8.


In this paper, we explain how we integrated cross-device copy-pasting and linked editing in GUMMY. This way, it forms the basis for chapter 5. The paper also gives a description of the GUMMY CUI model, which is discussed in Chapter 4.


D-Macs is the subject of this paper. It describes how we integrated D-Macs in GUMMY and gives details about its implementation. This forms the basis of Chapter 6.
10.2 Scientific Contributions and Publications

Het computer gebruik is de laatste jaren sterk geëvolueerd. De meeste mensen hebben naast een klassieke PC ook een mobiele telefoon, tablet PC en digitale TV. Op al deze toestellen wenst men vaak dezelfde applicaties te gebruiken. Om dit mogelijk te maken, dient er een gepaste User Interface (UI) voorhanden te zijn op elk van deze toestellen. Deze UI’s zijn specifiek ontwikkeld voor elk van deze toestellen en houden rekening met de interactiemogelijkheden dat een toestel aanbiedt.

Het manueel ontwikkelen van UI’s voor elk afzonderlijk toestel is een tijdrovende en complexe procedure. Een UI-designer moet vaak switchen tussen verschillende incompatibele designomgevingen. Tijdens dit proces moeten designstappen ook vaak herhaald worden, wat tot de nodige frustraties kan leiden.

In mijn doctoraat heb ik een nieuwe designomgeving, genaamd GUMMY, gebouwd die het makkelijker maakt voor designers om UI’s voor meerder platformen te maken. GUMMY maakt het makkelijker voor designers om UI’s voor meerdere toestellen te creëren, deze verschillende UI’s te beheren, en te optimaliseren zodat elke UI zo goed mogelijk is afgestemd op de capaciteiten van het onderliggende toestel.

Het creëren van UI’s gebeurt op een grafische manier. Designers kunnen de nodige grafische componenten bij elkaar voegen en aanpassen tot ze de gewenste esthetische kwaliteit bereiken. Dit lijkt sterk op de manier van werken in reeds bestaande design omgevingen. Hierdoor houden we de barrière om met GUMMY te werken laag.

Hoe meer UI’s voor verschillende toestellen men maakt, hoe moeilijker het wordt om het overzicht te bewaren. Om deze situatie het hoofd te bieden bevat GUMMY ondersteuning om de verschillende UI’s te beheren. Het beheren van UI’s houdt in dat GUMMY verscheidene designstappen automatisere. Zo maakt GUMMY het mogelijk om componenten van de ene toestelspecifieke
UI te kopieren naar een UI voor een ander toestel. Deze kopieën kunnen achteraf synchroon gemanipuleerd worden. GUMMY bevat ook een design macrosysteem wat het mogelijk maakt om designstappen op te nemen en deze later ook weer af te spelen in dezelfde of een andere UI. Op deze manier wordt het mogelijk voor designers om zelf te bepalen welke stappen er moeten geautomatiseerd worden en wanneer.

De laatste bijdrage die wordt beschreven in deze thesis is ondersteuning voor het iteratief optimaliseren van UI’s. Optimaliseren betekent het “finetunen” van UI’s zodat deze zo goed mogelijk zijn afgestemd op de mogelijkheden van de onderliggende toestellen. Wij ondersteunen deze optimalisatie door designtechnieken die het mogelijk maken om heel gemakkelijk te wisselen tussen design- en testacties. Op deze manier kan een designer onmiddellijk ingrijpen indien een designactie niet het gewenste effect heeft op het toestel.
Bibliography


[NMR06] Jeffrey Nichols, Brad A. Myers, and Brandon Rothrock. Uniform: automatically generating consistent remote control user


Low- or high-fidelity, paper or computer? Choosing attributes when testing web prototypes, Baltimore, Maryland, 2002. Nokia Research Center.


