Diss. ETH No. 24648

Investigating Tool Support for Cross-Device Development

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH

(Dr. sc. ETH Zurich)

presented by

Maria Husmann

Master of Science ETH in Computer Science

born January 16, 1984

citizen of Hochdorf LU and Wolhusen LU, Switzerland

accepted on the recommendation of

Prof. Dr. Moira C. Norrie, examiner
Prof. Dr. Harald Reiterer, co-examiner
Prof. Dr. Michael Nebeling, co-examiner

2017

Copyright © 2017 by Maria Husmann

Abstract

As the number of consumer computing devices at our disposal has grown, we find ourselves in situations where we have access to more than a single device. Even though people have started using multiple devices both sequentially and in parallel, applications and operating systems offer limited support for these scenarios. Support is often limited to data synchronisation via the cloud or restricted to devices from the same manufacturer. The research community has been exploring how applications could adapt to the set of devices at hand and has devised new cross-device interaction techniques. Frameworks and tools have been developed for designing and prototyping such cross-device applications. However, less attention has been paid to what is needed to go beyond research prototypes and to engineer products.

In this thesis, we investigate how we can support the development of cross-device applications across the whole engineering lifecycle from design, prototyping, and implementation to testing, debugging, and usage analysis. We start with an analysis of existing cross-device research as well as general purpose developer tools for the whole engineering lifecycle. We focus on tools for working with web technologies as we have chosen to use the web as a foundation for this thesis due to its cross-platform nature. We then introduce a set of novel tools for cross-device engineering that implement the requirements we identified in the analysis.

For the design and prototyping phase, we contribute MultiMasher, a visual tool for creating functional cross-device mashups. The tool allows developers and designers to experiment with cross-device ideas based on existing applications without the need to write code.

The implementation phase is addressed with XD-MVC, a cross-device library. XD-MVC consists of a layered architecture to provide various levels of support. At the lower end, a JavaScript API offers basic functionality such as connection management, state synchronisation, and device roles. At the other end of the spectrum, we offer high level components for device pairing and UI distribution based on patterns.

XD-Tools offers support for informal testing and debugging. It caters to the need of being able to test and debug many different device combinations and easily switch between different ones. This is achieved both with the integration of real devices as well as the emulation of devices on the developer machine. At the same time, the tool alleviates the issues of fragmentation of the code across devices by aggregating information for existing debugging tools such as the console.

Formal, automated testing is addressed with XD-Testing. It allows developers to write parametrised UI tests in a new domain specific language. These tests can be repeated across multiple device combinations to test functionality independent of the set of devices used. At the same time, a developer can use explicit device selectors to test the distribution of the UI for a given device combination.

Finally, we contribute XD-Analytics for usage analysis across multiple devices. We introduce a set of metrics that are of particular interest in cross-device scenarios. We have implemented these metrics in a prototype and discuss how we used the tool to track the introduction of a cross-device feature in an existing application.

To evaluate our tools, we have implemented a number of sample applications and conducted case studies as well as user studies.

Zusammenfassung

Da die Anzahl der Geräte mit Rechenleistung wie Mobiltelefone und Tablets gewachsen ist, befinden wir uns immer häufiger in Situationen, in denen wir Zugang zu mehr als einem einzigen Gerät haben. Auch wenn einige Nutzer bereits angefangen haben, mehrere Geräten sowohl nacheinander als auch parallel zu benutzen, bieten Anwendungen und Betriebssysteme eine begrenzte Unterstützung für diese Szenarien. Die Unterstützung ist oft auf die Datensynchronisation über die Cloud oder auf Geräte des gleichen Herstellers beschränkt. Die Forschungsgemeinschaft hat untersucht, wie sich Anwendungen an die eingesetzten Geräte anpassen können und hat neue geräteübergreifende Interaktionstechniken entwickelt. Für das Entwerfen und Prototyping solcher geräteübergreifenden Anwendungen wurden bereits Frameworks und Tools entwickelt. Was benötigt wird, um über Forschungsprototypen hinauszugehen und Produkte zu entwickeln, hat hingegen weniger Aufmerksamkeit erfahren.

In dieser Arbeit untersuchen wir, wie wir die Entwicklung von geräteübergreifenden Anwendungen über den gesamten Entwicklungszyklus von Design, Prototyping und Implementierung bis hin zu Testen, Debugging und Nutzungsanalyse unterstützen können. Wir beginnen mit einer Analyse der vorhandenen geräteübergreifenden Forschung sowie generellen Entwicklertools für den gesamten Entwicklungszyklus. Wir konzentrieren uns auf Tools für die Arbeit mit Web-Technologien, da wir das Web wegen seiner plattformübergreifenden Natur als Grundlage für diese Arbeit gewählt haben. Wir stellen anschliessend eine Reihe von neuartigen Tools für die geräteübergreifende Entwicklung vor, welche die in der Analyse identifizierten Anforderungen implementieren.

Für die Design- und Prototyping-Phase tragen wir MultiMasher bei, ein visuelles Tool zur Erstellung von funktionalen geräteübergreifenden Mashups. Das Tool ermöglicht Entwicklern und Designern, mit geräteübergreifenden Ideen zu experimentieren unter der Verwendung bestehender Anwendungen und ohne Code zu schreiben.

Die Implementierungsphase wird mit XD-MVC, einer geräteübergreifenden Softwarebibliothek, angesprochen. XD-MVC besteht aus einer geschichteten Architektur, um dem Entwickler verschiedene Ebenen der Unterstützung zu bieten. Am unteren Ende bietet eine JavaScript-API grundlegende Funktionalität wie Verbindungsmanagement, Zustands-Synchronisation und Geräterollen. Am anderen Ende des Spektrums bieten wir Komponenten für Gerätepaarung und Benutzeroberflächen-Verteilung auf Basis von Mustern an.

XD-Tools unterstützt informelles Testen und Debugging. Es ist darauf ausgerichtet, viele verschiedene Geräte-Kombinationen testen zu können und leicht zwischen diesen zu wechseln. Dies geschieht sowohl mit der Integration von echten Geräten als auch mit der Emulation von Geräten auf der Entwicklermaschine. Gleichzeitig verringert das Tool die Probleme der Fragmentierung des Codes über mehrere Geräte, indem es Informationen für vorhandene Debugging-Tools wie die Konsole zusammenfasst.

Formales, automatisiertes Testen wird mit XD-Testing angesprochen. Es ermöglicht Entwicklern, parametrisierte Benutzeroberflächen-Tests in einer neuen domänenspezifischen Sprache zu schreiben. Diese Tests können über mehrere Gerätekombinationen wiederholt werden, um die Funktionalität unabhängig vom eingesetzten Set von Geräten zu testen. Gleichzeitig kann ein Entwickler explizite Geräte-Selektoren verwenden, um die Verteilung der Benutzeroberfläche für eine gegebene Gerätekombination zu testen.

Schliesslich steuern wir XD-Analytics für die Anwendungsanalyse über mehrere Geräte hinweg bei. Wir stellen eine Reihe von Metriken vor, die für geräteübergreifende Szenarien besonders interessant sind. Wir haben diese Metriken in einem Prototypen implementiert und diskutieren, wie wir das Tool verwendet haben, um die Einführung einer geräteübergreifenden Funktion in einer bestehenden Anwendung zu verfolgen.

Um unsere Werkzeuge zu evaluieren, haben wir eine Reihe von Beispielanwendungen implementiert und Fall- sowie Anwenderstudien durchgeführt.

Acknowledgements

I would like to start by thanking Prof. Moira Norrie for giving me the opportunity to pursue my PhD in her GlobIS group. When I was a Bachelor student, Moira's positive feedback encouraged me to focus my studies on information systems. During my Master thesis and as a student research assistant, I had a first impression of the culture and research of the GlobIS group. These experiences had a significant weight when I was considering whether I should pursue doctoral studies or not. During these studies Moira has been a great mentor. While she has given me a lot of freedom to explore my own ideas, she was always ready to give advice when I needed it.

I am thankful to Prof. Harald Reiterer and Prof. Michael Nebeling for agreeing to be co-examiners of my thesis. Michael was a postdoc with the GlobIS group when I started as new doctoral student and he took me under his wing. He helped me find my thesis topic and taught me how to navigate academia. This thesis owes a lot to his guidance and support and I am very thankful for that.

My colleagues in the GlobIS group have contributed to the enjoyable time that my PhD studies have been, whether it was with chats in the coffee area with David Weber and Karl Presser or philosophical discussions over lunch with Tilmann Zaeschke and Matthias Geel. Alfonso Murolo and Christoph Zimmerli have both been amazing at helping with anything technical or otherwise. Stefania Leone, my first office mate, helped me to get settled into PhD life. Her successor, Linda Di Geronimo was also a fantastic office mate who was always ready to help in any way she could, for example by bringing treats from Italy to the office. I learned a lot from Fabrice Matulic about designing and conducting user studies. I also value the conversations about academic careers I had with former members of the group, including Beat Signer, Alexandre de Spindler, and Michael Grossniklaus. Alex, in particular, has been very helpful in planning the next steps after my PhD.

Many thanks go to all the students who have worked with me on some aspect of this thesis. They took over some of the burden (and fun) of refining and implementing my ideas in prototypes. These students were great partners in discussions and I value their contributions. In particular, I would like to mention Nina Heyder, Silvan Egli, Madelin Schumacher, Stefano Pongelli, Fabian Stutz, Dhivyabharathi Ramasamy, Alexander Richter, Marko Zivkovic, Sivaranjini Chithambaram, Aryaman Fasciati, Michael Spiegel, Jil Weber, and Nicola Marcacci Rossi.

Beate Bernhard and Claudia Günthart have made my life at ETH easier in many ways in their roles as administrative assistants to the GlobIS group and I thank them for that.

I am grateful for all the support that I received from the Moelbert family from Hochdorf over the years. They hired me and taught me programming when I was fresh out of high school and had no formal training in computer science. Their trust in my

abilities was a big confidence boost and having worked on real world problems was helpful for my later studies at ETH. Susanne and I had many discussions about her experience of doing doctoral studies and she was an excellent mentor.

Finally, I thank my family and my partner Reto for all their love and support.

Table of Contents

1 Introduction **1**

1.1 Motivation . . . 3
1.2 Challenges . . . 5
1.3 Contribution . . . 6
1.4 Thesis Overview . . . 8

2 Background **11**

2.1 Devices People Own and Use . . . 11
2.1.1 Quantity and Types of Devices . . . 12
2.1.2 Usage Patterns . . . 13
2.1.3 Issues in Multi-Device Use . . . 15
2.2 Cross-device Systems . . . 16
2.2.1 Interaction Techniques . . . 17
2.2.2 Single-User Scenarios . . . 19
2.2.3 Multi-User Scenarios . . . 21
2.3 The Development Process . . . 23
2.3.1 Design and Prototyping . . . 24
2.3.2 Architectures and Implementation . . . 27
2.3.3 Debugging . . . 32
2.3.4 Testing . . . 36
2.3.5 Usage Analysis . . . 37
2.4 Conclusion . . . 39

3 MultiMasher: Design and Prototyping **41**

3.1 Concepts and Requirements . . . 41
3.2 MultiMasher . . . 44
3.2.1 Global View . . . 45
3.2.2 Mashup View . . . 45
3.2.3 Co-browsing Feedback . . . 48
3.3 Architecture and Implementation . . . 49

3.3.1 Alternative Architectures . . . 49
3.3.2 Implementation . . . 50
3.4 Evaluation . . . 51
3.4.1 Conceptual Evaluation . . . 51
3.4.2 Technical Evaluation . . . 53
3.5 Discussion . . . 57

4 XD-MVC: Implementation 59
4.1 Concepts . . . 59
4.1.1 Layered Architecture . . . 60
4.1.2 JavaScript API . . . 60
4.1.3 Polymer Components . . . 63
4.2 Architecture and Implementation . . . 69
4.2.1 Device-to-Device Communication . . . 69
4.2.2 System Architecture . . . 71
4.2.3 Implementation . . . 73
4.3 Evaluation . . . 73
4.3.1 XD-Bike . . . 74
4.3.2 Hotel Booking . . . 76
4.3.3 Webcam Viewer . . . 77
4.3.4 Maps . . . 78
4.3.5 Voting . . . 79
4.4 Discussion . . . 79

5 XD-Tools: Debugging 81
5.1 Developer Tasks . . . 81
5.1.1 Composition . . . 82
5.1.2 Comprehension . . . 83
5.1.3 Debugging . . . 83
5.2 Requirements . . . 84
5.2.1 Emulation of Multiple Devices . . . 84
5.2.2 Integration of Real Devices . . . 84
5.2.3 Switching of Device Configurations . . . 85
5.2.4 Integration of Debugging Tools . . . 85
5.2.5 Automatic Connection Management . . . 85
5.2.6 Coordinated Record and Replay . . . 86
5.3 XD-Tools . . . 87
5.3.1 Emulation of Multiple Devices . . . 89
5.3.2 Integration of Real Devices . . . 89
5.3.3 Switching of Device Configurations . . . 91

5.3.4 Integration of Debugging Tools 91
5.3.5 Automatic Connection Management 94
5.3.6 Coordinated Record and Replay 94
5.4 Architecture and Implementation 96
5.5 Evaluation 98
5.5.1 Participants and Setup 98
5.5.2 Tasks and Procedure 98
5.5.3 Results 100
5.6 Discussion 102

6 XD-Testing: Automated Testing 105
6.1 Motivating Example 106
6.2 Parametrised Tests with Device Templates and Scenarios 106
6.3 A Domain Specific Language for Cross-Device Tests 107
6.3.1 Explicit Device Selection 108
6.3.2 Implicit Device Selection 110
6.3.3 Device Set Commands 111
6.4 Flow Recording and Visualisation 112
6.4.1 Recording 112
6.4.2 Visualisation 113
6.5 Architecture and Implementation 114
6.6 Case Study 115
6.6.1 Gallery Application and Sample Test Cases 115
6.6.2 Evaluation of Web Engineering Exercise 117
6.7 Discussion 119

7 XD-Analytics: Usage Analysis 121
7.1 Cross-Device Analytics Use Cases 121
7.2 Metrics 122
7.3 XD-Analytics 123
7.3.1 Overview 123
7.3.2 Architecture and Implementation 126
7.4 Case Study 127
7.4.1 Cross-Device Extension 128
7.4.2 Results 128
7.5 Discussion 132

8 Conclusion & Outlook 135
8.1 Analysis 137
8.2 Outlook 138

A Student Contributions **141**

1 Introduction

Over 25 years ago Mark Weiser expressed his seminal vision of the computer of the 21st century [192]. Now, more than a decade into that century, let us examine what parts of the vision have been achieved, what is still missing, and where we have diverged. Weiser imagined that computers would become ubiquitous commodity devices that disappear into the background, rather than being treasured personal devices that command a user's attention. Rooms would be filled with a large number (Weiser mentions over 100) of computing devices ready to be used at anyone's disposal.

These devices would come in different sizes: from inch-scale tabs to foot-scale pads to yard-scale boards. The devices would need to be aware of their own and the users' locations, communicate with each other and work together seamlessly. At the same time, user interaction would be so effortless that users would not consciously interact with a computer but rather focus on the task at hand which the computer would serve in the background. Weiser compared this process to writing which is constantly present in our lives and which transmits information without us having to spend much active attention on the process of reading.

Weiser identified the need for devices (hardware), software, and network technologies to fulfil his vision and singled out software and networks as the bigger challenges. Examining the current state, we can confirm this observation to a large extent. Since the release of the first iPhone in 2007, smartphones have become mainstream. In terms of size, these correspond to Weiser's smallest category, tabs. Smartwatches, which are increasingly gaining in popularity, would also fall into this category. Tablets on the other hand could be classified as pads. Smart TVs, interactive whiteboards, (semi-) public displays are all yard-scale devices. We can summarise that, in terms of hardware, we have come close to the future that Weiser imagined. However, looking at the how we use these devices we are still pretty far away. Phones and watches are still very much personal devices. While tablets and TVs are less personal and often shared within a household, they are certainly not in the background yet but rather still command our attention. Furthermore, we are still far away from the number of devices per room that Weiser envisioned.

Looking at devices that we do not immediately perceive as computers but that essentially include a computer, we can also examine smart or connected homes. While there have been efforts to create smart home technologies, these are still not found in every household and are mainly heard of when they fail. A recent example for a failure is an internet-connected pet feeder that stopped dispensing food when a server failed[1]. Nest, one of the bigger internet of things companies, has also produced headlines with failing thermostats[2] and shut down services[3]. On other hand, robot vacuum cleaners and smart lights have seen more success. These devices are connected to the internet and can be controlled by the owner through their phone (even remotely) or, in the case of the lights, detect the presence of the owner based the geolocation of their phone and switch on or off automatically. However, typically each device needs the owner to install a new application for controlling it and communication among devices is usually limited to the phone, which is far from a seamless interaction in the background.

In summary, with the proliferation of smartphones and tablets and the increasing popularity of smartwatches and smart TVs we are now often surrounded by multiple computing devices. Nevertheless, in terms of interaction we are not yet where Mark Weiser imagined we would be as there is still limited communication between devices. Even though we do have network technology for devices to communicate (for example Wifi or Bluetooth), they are still predominantly used in isolation or require the user to manually coordinate interactions. Synchronisation between devices is typically limited to shared accounts, such as email, or cloud-based services such as Dropbox for files. Sharing information from one device to another still requires relatively much effort and people employ workarounds such as emailing themselves information such as links to web articles. These workarounds are unsatisfactory and studies have revealed a need for more synchronisation between devices [87, 161, 37]. Despite the limited support, these studies have detected patterns in how people already use or attempt to use multiple devices in combination, either sequentially or in parallel.

Facilitating the use of multiple devices in combination can produce benefits to the user if done well. Different devices have different characteristics. Smaller devices such as phones are ready to hand, equipped with sensors and touch screens, but their screens are relatively small. In contrast, devices with larger screens are not mobile and typically do not have the same range of sensors. In particular, smart TVs are more cumbersome to interact with via remote controls and interaction with public screens may be even more limited. Combining these devices, for example a phone with a TV, would lead to richer input and output capabilities. When no large screens are at hand, for example when on the go, combining multiple smaller devices such as phones and tablets would produce increased screen real estate.

Research on multi-device or distributed user interface (DUI) systems has a long tradition. Smart rooms where display walls, tabletop computers and personal computers are combined have been proposed as early as the 90s [180, 181, 184]. These systems were

[1] http://www.dailymail.co.uk/news/article-3714186/Dogs-cats-left-without-food-remote-smart-feeder-PetNet-fails-dispense-meals-suffering-server-issues.html Accessed on 11. 08. 2016

[2] http://www.nytimes.com/2016/01/14/fashion/nest-thermostat-glitch-battery-dies-software-freeze.html Accessed on 11. 08. 2016

[3] http://www.wired.com/2016/04/nests-hub-shutdown-proves-youre-crazy-buy-internet-things/ Accessed on 11. 08. 2016

typically at a larger scale with a dozen or more devices in a single, specially equipped room. The set of devices used was relatively static and tailored to the intended use case.

The advent of mobile phones and tablets has sparked interest in more dynamic multi-device systems. Proxemic interaction [3, 117] has focused on the interaction between individual devices and users based on their relative positions. Walking into a room would activate the system running on the TV. Data could be shared from one device to another by tilting the first device towards the second one. However, obtaining the required fine-grained position information requires expensive tracking systems (Vicon) or at least a specially equipped room (Kinect), making it unsuitable for a wide range of scenarios where such technology would not be available.

More recently, flexible systems have been explored that, in contrast to those built for a specific set of devices, adapt to the set of devices at hand. For these systems the terms cross-device [197] and liquid [129] applications have been used.

Despite over 20 years of research in the area, we have seen only a few cross-device applications outside academia. When it comes to engineering a product, not a research prototype that never leaves the lab, there is still limited support in terms of tools for the usual stages in software development. Most research has focused on the design and prototyping stage and there are a number of cross-device frameworks to help with implementation. However, debugging, testing, and analysing the usage of an application have not received much attention, even though there are challenges specific to cross-device applications that are not met adequately with existing tools for single device development.

1.1 Motivation

The goal of this thesis is to investigate how better support could be provided for the whole development workflow of cross-device applications, starting from design and prototyping to implementation, debugging, testing and analytics.

Rather than trying to impose a new way of working, our aim was to look at how we could build on existing methods and tools and extend them for cross-device scenarios. We chose web technologies as a basis for our work since web applications are supported on all major mobile and desktop platforms, albeit with minor differences. This interoperability is a crucial property for viable cross-device applications. Limiting an application to a subset of platforms would violate the vision of applications that adapt to the set of devices at hand. Furthermore, web applications require no installation and are easy to share via URLs, which can also be represented as QR codes or transmitted easily using Bluetooth or NFC. This simple way of sharing and accessing an application is an important step towards the effortless computing outlined by Weiser. The alternative, native applications, first need to be installed, for example through an app store, and thus come with a significant overhead for first time use.

Building fully fledged applications with web technologies is a viable option nowadays. Writing code that runs on all platforms is more cost-effective than re-implementing the same applications for different platforms such as Android, iOs, and possibly also providing a desktop version. Web technologies have come a long way from simple hypertext documents. Recent and emerging standards make web technologies a strong

competitor compared to native technologies. Device sensors such as accelerometers and gyroscopes can now be accessed via JavaScript. All major browsers except Safari allow the browser to access the camera and Chrome has started to implement an API for Bluetooth[4]. These sensors and technologies provide a basis for implementing new interaction techniques such as mid-air gestures. A whole set of specifications have been proposed for progressive web apps[5]. This new class of applications are written with web technologies but increasingly behave like native applications, if supported by the browser, delivering push notifications and working even when offline. They can be installed to the home screen from where they can be launched full screen and without browser controls so that the application becomes indistinguishable from a native application to the user.

Crucial for cross-device applications, new communication protocols have been introduced. WebSockets are full-duplex communication between clients and server. The technology was first introduced in 2010 and is implemented by all major browsers today. WebRTC is a newer protocol for direct (peer-to-peer) communication between clients. As of now, it is supported or under development by all major browsers. These technologies enable the synchronisation between multiple devices that is needed for cross-device applications.

Since the advent of new interaction modalities (such as touch) and smaller screen sizes with smartphones and tablets, web developers and designers have searched for ways to adapt and produce optimal user experiences. Initially, a prevalent technique was to detect the type of device and have the server deliver a tailored HTML document to mobile devices. This technique is now increasingly being replaced with responsive web design. Websites or applications following this approach deliver the same document to all devices, however, it is written in such a way that it adapts to the device. For example, flexible grids are used rather than pixel-based layouts to adapt to the screen size. Furthermore, with media queries, different CSS rules can be applied to different devices. The HTML5 picture tag allows a developer to provide different versions of the same image and media queries that specify which version should be used on which device. For smaller devices such as phones that may also suffer from limited connectivity or run on expensive data plans, smaller versions of the image can be downloaded, saving the user time and money. However, these technologies allow a website to adapt to the device at hand but not to different sets of devices.

In addition to new technologies, a range of tools to support the developers have been introduced, some directly integrated into the browser. For example, browsers can emulate the viewport size and pixel density of mobile devices and thus allow the developer to quickly switch between different models to check the layout. Furthermore, browsers can also emulate touch input and network conditions, so that a developer can verify how a fast an application loads on a slow network even when working in perfectly fast conditions. However, only a single device can be emulated at a time. In order to debug mobile devices, these can be connected via USB to a developer's machine. Using the tools of their desktop browser, the developer can step through the code that is executed on the mobile device and inspect the run-time state. In addition to tools

[4]`https://developers.google.com/web/updates/2015/07/interact-with-ble-devices-on-the-web` Accessed on 12. 08. 2016

[5]`https://developers.google.com/web/progressive-web-apps/` Accessed on 12. 08. 2016

built into the browser, there are services that run an application on a diverse set of real or emulated devices and produce screenshots or execute test suites. For testing cross-device applications, these services are unsuitable in their current state as they treat each device in isolation and offer no communication among the devices.

After an application has been deployed, analytics tools such as Google Analytics[6] provide insights into the types of devices that are used to access a website. Analytics can be used to measure business goals and to further improve a website. For example, an online shop business could detect that more shopping carts are abandoned on mobile than on desktop devices and start further investigations as to why that is happening. In 2012, Google introduced Universal Analytics with Cross Device reports[7] allowing a user to be tracked across different devices. While these reports can show sequential device paths, for example that a user started shopping on a mobile device and then checked out on the desktop, they provide no information about parallel usage of multiple devices. However, this information is crucial to identify whether a cross-device application is used as intended. Users might not be aware that an application can be used with multiple devices or, if the user experience is deficient, stick to a single device. Another possibility would be that an application has been optimised for a specific device combination, say TV and phone, but it turns out that it is also used in significant proportions with phone and tablet. This could prompt an update to the design.

We hope that, with the right tools, developers will be able to build cross-device applications more easily and of better quality. In the long run, we would like to see better user experiences that span multiple devices and come a step closer to Mark Weiser's vision of seamless interaction with multiple devices.

Our main research questions can be summarised as follows.

- **RQ1** How well do existing tools support cross-device development across the whole development lifecycle?
- **RQ2** What are the requirements of tools specific for cross-device development?
- **RQ3** Can we provide better tools for cross-device development, in particular for testing, debugging, and usage analysis?

1.2 Challenges

Two characteristics of cross-device applications cause some of the main challenges in application development. First, having to address many different possible device combinations influences all stages in the workflow from design to analytics. When designing, an application could be built for some specific device combination(s) or else it could be designed to be more dynamic and adapt to any given set of devices. Designing an application that looks good on any given set of devices is even more challenging than responsive design, which just needs to adapt to a single device. It is not feasible to specify a concrete design for every possible combination of devices, thus the design stage needs some way of specifying more general designs that can be applied to a concrete set of devices. Similarly, it would not be feasible to have different implementations for any

[6] `https://analytics.google.com/` Accessed on 16. 08. 2016
[7] `https://support.google.com/analytics/answer/3234673` Accessed on 16. 08. 2016

possible device combination. Thus the implementation also needs to be flexible. Bugs could occur in some device combinations but not others and manually testing all possible combinations is not a real option. Even testing only a small number of expected device combinations manually becomes tedious when tests need to be repeated. Finding out what device combinations are actually used, would provide interesting information and could feed back into the design process, however, current usage analysis tools do not report on parallel usage. Finally, these device combinations are not necessarily static. Devices may join or leave at run time, requiring the application to adapt on the fly.

Second, the fragmentation of the application across multiple devices also impacts all stages of the development. During the design stage, the developer has to decide what parts of the user interface should be displayed on which device. During the implementation, the fragmentation requires some communication between the devices for synchronisation. Pressing a button on one device may cause an update on another device. A developer needs to take this distribution into account. The fragmentation of the logic across multiple devices also impacts heavily on debugging. It prevents the use of a single debugger on a single device to sequentially step through the code. Instead, multiple devices need to be coordinated and the network adds another possible point of failure. Similarly, for testing, multiple devices need to be coordinated and test cases must be written so that time for synchronisation between the devices is taken into account. Finally, usage analysis is also affected by the fragmentation in that multiple devices must be associated somehow and the visualisation of the tracked data should reflect the fact that multiple devices were used simultaneously.

1.3 Contribution

To address the challenges outlined in the previous section, we investigated tool support for all stages of the development process with particular attention given to the later stages of testing, debugging, and analytics of cross-device applications. While the tools that we provide are not tightly coupled, they are intended to be used in combination across the development cycle of an application. This thesis makes five contributions from prototyping to analytics.

For the design and prototyping stage, we contribute MultiMasher, a visual tool for creating cross-device mashups. MultiMasher allows existing websites to be mashed up and re-distributed across multiple devices, so that designers can experiment and explore new ideas with little effort. MultiMasher produces functional prototypes that could, for example, be tested with users. The tool is visual and does not require any programming skills beyond a basic understanding of website structure and events. We experimented with different architectures for MultiMasher and settled on a centralised server implementing a remote control metaphor.

For the next phase, the implementation, we contribute XD-MVC, a cross-device programming framework. With the new web standards outlined above, many building blocks for web-based cross-device applications are there, however putting them together can be challenging. Furthermore, developers risk re-inventing the wheel every time they build a cross-device application without a specialised library or framework, focusing on the low-level details of juggling the devices rather than application specific behaviour.

XD-MVC takes over these responsibilities and takes care of the connection management, data synchronisation, and devices and their roles in the system. XD-MVC consists of multiple layers and components, the basic layer being JavaScript. This architecture allows developers to choose the level of support that they need in their implementation and to work with their preferred MVC framework such as React[8] or Angular[9]. While the JavaScript layer provides basic support, even more help is available to developers when they use the provided Polymer[10] integration that consists of both visual and non-visual web components. These components handle common tasks in cross-device applications such as pairing devices via QR codes or URLs, and give access to the JavaScript functionality in a declarative way. The Polymer integration includes pre-defined layouts that implement cross-device design patterns such as pagination or the remote control pattern [138], requiring a developer to write only a few lines of code to distribute their application across devices. The architecture of XD-MVC is designed to reduce latency as much as possible while supporting a wide range of devices with its hybrid architecture. Low latency is important for good user experience across multiple devices and identified by previous work [197] as a remaining challenge. To improve latency, XD-MVC builds on direct peer-to-peer connections among devices when supported by the browser and falls back to communication via a central server when that is not the case. Our performance tests showed that the first strategy drastically lowers latency compared to the second approach when the devices are co-located and the server is remote. This is a constellation that we expect to be common, as servers could be hosted anywhere, while the co-operating devices are expected to be mostly close to the user. If the communication is going via the server, latency suffers when the server is remote, however, applications are still usable though not quite as responsive.

The next contribution of this thesis is XD-Tools, a set of tools for debugging and informally testing cross-device applications. Inspired by existing tools for responsive design, we created XD-Tools to support the typical programming tasks of composition, comprehension, and debugging [173]. We give examples of challenges specific to cross-device development and identify requirements for tools to address these challenges. As with tools for responsive design, both emulated devices and real devices should be integrated. However, while responsive development tools treat devices as isolated, cross-device tools should be aware of the coordination among devices and support it. To address the challenge of testing and debugging an application that should adapt to many different device combinations, it should be easy to switch between these combinations. At the same time, handling all of these devices manually would require a developer to repeatedly execute connection management tasks, such as pairing devices every time the application is reloaded. This is a burden that could be taken from the developer by automating the connection management. Established and much used debugging tools such as the JavaScript console should be integrated and tailored to the cross-device use case. Finally, there are a number of record and replay tools (for example Selenium[11]) that allow a developer to record user interaction on a device and replay it on the same or another device. However, they also assume a single device while, in a cross-device application, interaction may occur on multiple devices. With XD-Tools we created a

[8] `https://facebook.github.io/react/` Accessed on 31.03.2017

[9] `https://angular.io/` Accessed on 31.03.2017

[10] `https://www.polymer-project.org/` Accessed on 31.03.2017

[11] `http://www.seleniumhq.org/` Accessed on 31.03.2017

prototype implementation to meet these requirements. A preliminary evaluation with 12 developers received enthusiastic feedback.

As a fourth contribution, we address the testing phase with our XD-Testing library. Automated testing is considered good practice in software engineering and a number of testing frameworks, tools, and libraries exist. However, these do not handle the fragmentation of cross-device applications or take into account that an application should run on different device combinations. Hence a developer has to manually handle these challenges when they script tests for cross-device applications. With XD-Testing, we provide a library that can be used in combination with existing test runners such as Mocha[12]. Testing can occur at different levels of granularity. With XD-Testing, we address end-to-end testing based on the user interface. The system imitates user interactions, such as clicks, and verifies that the system reacts according to the specifications, for example that a label subsequently shows the correct text. XD-Testing allows a developer to address devices explicitly based on properties, such as the screen size, or let the library choose the appropriate device implicitly based on the command to be executed. Tests can be parametrised with different device combinations and executed repeatedly. Device combinations can be either hand crafted by the developer to test a specific set of devices or randomly generated. XD-Testing comes with a visual component that displays screenshots recorded during the test execution. A tester can mark important steps in the execution with checkpoints and a screenshot will be generated for each. Additionally, failures in a test can be captured and are marked as such. The visual tool allows a tester to examine visually what happened during a certain test. In addition, the tool allows multiple recordings to be displayed side by side so that a tester can compare and contrast the same test when it is executed on different device combinations, for example a TV and phone compared against a phone and tablet.

Finally, as a fifth contribution, we present XD-Analytics, a system for usage analysis of cross-device web applications. We introduce a set of metrics of interest in cross-device scenarios that are not tracked in traditional analytics systems and show how they can be used to answer interesting questions about cross-device usage. For example, we can detect if a system is at all used by multiple devices simultaneously and, if so, what types of devices are used together. We introduce our reference implementation XD-Analytics that tracks these metrics and report on the results of an in-the-wild study where we observed the introduction of a cross-device feature in an existing educational application.

1.4 Thesis Overview

This thesis is structured as follows: Chapter 2 analyses related scientific work and existing developer tools outside academia. We identify gaps in the existing work that have not yet been addressed and extract requirements for cross-device tools.

Chapter 3 introduces our approach for prototyping cross-device application based on mashing up existing single device applications or prototypes. We discuss our visual tool, MultiMasher, which is based on direct manipulation of UI elements. We discuss two architectures that we experimented with and reason why the remote control architecture

[12] `http://mochajs.org/` Accessed on 31.03.2017

was chosen in the end. We report on a technical evaluation based on 50 popular websites and discuss the system along the dimensions of a conceptual framework [150].

Chapter 4 is dedicated to the implementation stage and introduces our cross-device framework XD-MVC. We outline the main concepts and its layered modules as well as the hybrid communication architecture. We report on performance tests for peer-to-peer and client-server communication and present showcase applications that were implemented with the framework.

We then go on to the stage of informal testing and debugging in Chapter 5 where we present XD-Tools. We discuss common developer tasks and the challenges introduced in a cross-device environment. Based on these tasks and challenges, as well as our own experiences in developing cross-device applications, we introduce a set of requirements for testing and debugging cross-device applications. We introduce XD-Tools, our prototype implementing these requirements and report on a preliminary user study with a dozen developers.

Chapter 6 presents our approach to automated testing. We give a motivating example that highlights the challenges in UI testing a cross-device application. We introduce the concept of a device scenario, a combination of devices, that can be used to parametrise a test case so that it can be executed repeatedly with different combinations of devices. The concepts of our domain specific language (DSL) are introduced as well as the flow visualiser for application screenshot. In a case study, we demonstrate how we used the concepts from the DSL to write test cases for existing cross-device applications and the show the results in the flow visualiser.

We address the usage analysis for deployed cross-device applications in Chapter 7. We introduce a set of metrics that are of interest in a cross-device context. We explain how we track these metrics in our reference implementation XD-Analytics and report on a case where we used the system in an application with over 3000 users.

Finally, we conclude in Chapter 8 with a summary of our contributions for each stage in the development process. We discuss limitations and some directions for future work.

2 Background

User interfaces that are distributed over multiple devices have been explored extensively in prior work. In this chapter, we provide an overview of the field. We start with an examination of the devices people typically own and how they use them in various settings such as home and work environments. This analysis identifies issues in how devices are used in combination with state-of-the-art mainstream software and highlights potential for cross-device applications. We then present prior work in cross-device systems where we focus on interaction techniques as well as scenarios and applications in Section 2.2. In this part, our perspective resembles that of a user of these systems. We demonstrate what the systems do, how they can be used and the scenarios that they address. In Section 2.3, we switch over to the perspective of developers and designers of such systems. We describe how the state-of-the-art in research and industry supports the development process from design to usage analysis. We close with concluding remarks in Section 2.4.

2.1 Devices People Own and Use

Many of the older multi-device systems were built for very specific device configurations and were often equipped with hardware bought for that purpose, for example digital whiteboards or tabletops [85, 180, 181]. This setup allowed researchers to explore new ways of working and novel interaction techniques. We take a slightly different approach: We analyse the set of devices users have access to. So rather than introducing new devices into users' lives, we investigate how we can extend the usage of the existing devices. We start the investigation by answering the following questions based on prior work: What devices do people have access to and use? How are they used? Are there any issues, in particular, when multiple devices are used?

2.1.1 Quantity and Types of Devices

The set of devices at a user's disposal has changed in the last 10 years. The introduction of the first iPhone in 2007 has led to a rapid dissemination of smartphones. Three years later, the process repeated with tablets. More recently, smartwatches and internet-connected TVs are becoming more common. Outside of private homes, advertising space is increasingly digital as billboards are replaced with public screens. In offices and schools, wall projectors are part of the standard equipment and digital whiteboards appear here and there. There are consumer versions of digital tabletops, however, these devices have seen more limited adoption. As the set of devices that we use is evolving over time, studies need to be carried out repeatedly to assess the new situation. Another dimension that needs to be taken into account is the setting. A user does not always have access to the same fixed set of devices. At work, at home, or on the go are situations where the set of devices available to the user may vary significantly [149]. A number of studies have been carried out that either focus on a specific setting in detail or assess multiple settings.

In 2008, Dearman and Pierce interviewed 27 persons from academia and the IT industry and found among these participants device collections with 3 to 11 devices, averaging at nearly 6 devices per person. The authors also included portable media such as USB disks in this device count, even though these are not interactive by themselves. Devices were categorised into work/school, home, and intermediary. Participants had, on average, one computer (laptop or desktop) at home and another one at work/school. They had a mobile phone and at least one other portable device, mostly cameras and music players. More than half of participants also had a laptop that was intermediary, thus moving between home and work. Only two participants had more than one phone and stated transitioning to a new device as the reason. Another study from 2008 [89] came to the same number of phones per person

A couple of smartphone generations later, it is nowadays more common for people to own multiple phones. In a survey that we did in 2015 [39], we found that 47% of our participants owned more than one phone. The most common explanation given was that they kept their old phone when they bought a new one. These older devices could be used in specialised roles [87], for example to increase screen real-estate when using the current phone. A study by Santosa et al. [161] from 2013 also found increased numbers of devices, 10.7 on average per person. These included 2.7 smartphones, tablets, or e-readers. As these mobile devices continue to spread, they increasingly supersede desktop and laptop computers as main computing devices in the home [95]. The latter are mainly used for specialised tasks such as working from home or playing certain games, whereas mobile devices are used for web browsing, communication and watching videos.

Even though users now own or have access to almost a dozen devices, they are usually not all available at any point in time. For example, when walking in the hallway at work or during lunch, people have reported to just carry a phone [149]. In our survey [39], we found that roughly one in three does not even carry a phone in their local work area, a number which goes up to one in two when only looking at women. We found that almost every participant carried a smartphone during commutes, however, with almost half of our respondents reporting that they also carry a laptop. To describe the devices available to a user Oulasvirta et al. [149] have introduced the term

device configuration and defined it as follows: *"the set of devices and non-computational support artefacts used in a situation, differentiating the active subset (those used at the moment) and the passive subset (those available in the room but not used)."* To the best of our knowledge, no one has investigated yet the number of public or semi-public displays encountered by a user throughout their day. Most of these are not interactive yet and it is unclear whether they should thus be added to a user's device configuration. Regardless, analysing the studies introduced so far, we observe that device configurations vary a lot. Even though these studies report average values, they also emphasise the variation between participants. There is no such thing as a standard device configuration that a designer could use as a basis when building a cross-device application. Even when looking at a single user, the configuration changes as they move between settings such as home and work.

If we also consider devices available for multi-user cross-device applications the situation becomes even more complex. Can we simply create the union of the device configurations of all users to obtain the device configuration of a group? Or are there devices that are too private to be used in multi-user cross-device applications? While a TV in a living room is typically perceived as a communal device, phones are considered more personal and private devices. However, studies have shown that mobile phones are also often shared [92, 119]. Not surprisingly, trust plays in important role and sharing occurs most often with a significant other and in the family [119]. Another factor influencing the willingness to share a mobile phone was presence [92]. People felt more comfortable sharing their phone if it remained within sight, an effect that was stronger when the person borrowing the phone was not in the closest circle of trust. These findings are an indication that phones could be used in multi-user cross-device applications, especially when the owner does not lose sight of it. However, as suggested by Karlson et al. in [92], current all-or-nothing security models do not work very well for phone sharing. As a possible solution, the authors suggest guest profiles with reduced capabilities that can be accessed quickly. Similarly, one could think of a shared cross-device mode with increased privacy. For example, notifications could be suppressed or only show limited information when a device is part of a multi-user cross-device application.

2.1.2 Usage Patterns

When a user has multiple devices at their disposal, they have to choose which one (or ones) to use for a task. Oulasvirta et al. have identified several factors in [149] that impact the decision: Input and output modalities are important. A user mentioned checking the weather on their phone during lunch but then reading the newspaper on their desktop as it did not display well on their phone at that time. Devices with hardware keyboards are often preferred over touch-screen keyboards for writing [87, 95]. Form factors can play a role as well: Another user mentioned strategies to cope with privacy issues when using a laptop on an aeroplane. Due to its size, its micro-mobility is limited and a fellow passenger's gaze cannot be avoided easily. Smaller devices such as tablets and phones are often chosen due to their portability [93]. In the home, they are used in a lightweight manner in parallel to other activities such as cleaning or watching TV [95]. Phones and tablets are typically always on and have a small setup overhead unlike laptops or desktops that may require unpacking and booting up or a

complicated login procedure [93, 95, 149]. The latter are typically preferred for longer and more substantive tasks [149] that amortise the setup time, for example writing a longer email or working from home [95]. Smartphone use is sometimes perceived as more socially acceptable compared to laptops, for example for checking emails during a talk [149]. In the home, tablets and smartphones are used as communal devices that are passed around between users, for example when video-calling [95]. Sometimes devices are chosen based on application features. Some applications offer different functionality on different platforms [87] or software may only be available on a certain device due to licensing [37].

Rather than choosing a single device and sticking with it for a task, users have been observed using multiple devices instead. Several patterns of multi-device use have been identified. There are serial or sequential patterns [87, 161] where a task is started on one device and continued on another. For example, looking up a phone number on a tablet and calling it with a phone [87] or looking for a video on a tablet and then watching it on a TV [95]. Santosa and Wigdor observed a serial pattern that they labelled *producer-consumer* where data is produced on one device and consumed on another one, for example authoring a document on a PC and reviewing it on a tablet [161]. However, the roles of producer and consumer were not tightly coupled with the device type as also the phone was found in the producer role, for example for collecting data in a meeting which would later be transferred to a PC. Such an assignment of roles to devices for a task was also found by Dearman and Pierce [37] who, for example, interviewed a user who had a computer dedicated to writing code and another one for testing it. Reasons for switching devices include changes in the character of a task that render another device more suitable (moving from looking up a phone number to making a call) or in context (reading news in the bus and continuing in the office). In other instances, the chosen device turns out to be not ideal for the task at hand and the user switches for improved efficiency [87].

In addition to these sequential patterns, there are also patterns of parallel usage of multiple devices [37, 87, 161]. Based on a diary study [87], Jokela et al. describe three different types of parallel multi-device use: resource lending, unrelated parallel use, and related parallel use. In resource lending a primary device borrows a resource such as a screen or network connectivity from another device, for example a laptop is connected to a TV to watch a film or a phone is used to provide Internet access to a computer while travelling. Unrelated parallel use happens when a user does several tasks in parallel using multiple devices. Each device is dedicated to a separate task. Often, a primary task, for example doing homework on a computer, was accompanied by a secondary activity, such as listening to music on the phone. Unrelated parallel use was reported in fewer instances than related parallel use, however, the authors also presumed that the participants were not always aware of its occurrence. In related parallel use, multiple devices are used for the same task. A typical pattern is the use of a secondary device to look up additional information for an activity on a primary device, for example looking up an actor or translating a word with a dictionary with a phone while watching a film on a TV. Figure 2.1 shows another example observed by the author of this thesis where a phone is combined with a tablet to shop for wine on a train. This *performer-informer* pattern [161] has also been observed in work settings, where tablets have been used to display documentation while the desktop is used for a primary programming task. Related to the above is the *performer-informer-helper*

Figure 2.1: While shopping for wine on the phone on a train, a tablet is used to display additional information.

pattern where a device, typically a phone, takes over the role of a helper within a task, for example for calculations [161]. Sometimes a device is used to control another device in what has been labelled the *controller-viewer/analyser* pattern [161] or *remote control* pattern [138]. This pattern has been observed when a phone is used to control a music system or a slide show presentation.

2.1.3 Issues in Multi-Device Use

Even though people already use multiple devices sequentially or in parallel, there are a number of issues that impair the user experience. A lot of these problems stem from the lack of communication between the devices and the lack of awareness of the devices of each other. To transfer information from one device to another, several strategies have been observed: Portable media, such as USB sticks, are used to copy files from one device to another [37]. This approach is suitable mainly for PCs rather than mobile devices and can only be used for information that is represented as a file, which is typically not the case for application state, for example. Users also often email information to themselves [37, 91]. While this approach avoids the use of external hardware and will work on any device with an email client, it requires the user to switch from the current activity to the email client on both the sending and receiving device. Furthermore, the user needs to consciously share the data. If, for example, they forget to send a file from the office computer to their travel laptop and only realise once they are at a conference, they may not easily access that file from a distance. To avoid such problems, users have come up with elaborate synchronisation strategies [149].

To some extent, the introduction of cloud storage such as iCloud, Google Drive, and

Dropbox provides a remedy to these problems by synchronising files on all connected devices automatically in the background. However, a network connection is required and access to cloud storage is sometimes perceived as slow and unreliable [87]. If a user expects to have no or limited network access for example during a flight, they need to plan ahead for future data needs [161]. Some of these services still require a context switch and offer limited control, in particular in collaborative use cases [161]. Furthermore, they only operate on files, but this focus on files misses important information, for example interaction histories [37]. A need for synchronised browsing histories has also been recognised by Kane et al. [91]. Nowadays, modern browsers can be configured to synchronise browsing histories and stored passwords if the user logs in with their account on all devices. Another task that has been found to work well across multiple devices is emailing [94]. As most of the state is stored on the server, an email can be read on one device and answered on another device. A draft could be started on a mobile phone and continued on a PC. However, the same mobility between devices is still not supported in many other applications, for example social networking where a status update needs be completed on a single device, leading to higher frustration [94]. Furthermore, data is often trapped inside applications and can only be accessed through the application [161], in particular for web-based and mobile applications. Exporting and sharing this data is only possible if the application provides such a service. In summary, despite the increased number of synchronisation services, there are still issues in interoperability across different platforms and with incompatible content formats [87].

While synchronisation is crucial for multi-device use [25], more issues remain, in particular in related parallel use. Above, we have described how multiple devices are often assigned distinct roles in a task. However, devices typically have no awareness of their role within a task as well as the other devices currently in use and their respective roles [37]. Such awareness of connected devices is required for improved function coordination, for example in the remote control pattern [138, 161]. Information on proximity, location, and orientation of other devices enables intuitive interaction techniques [161] but is usually limited to Bluetooth or NFC presence or not at all available in unmodified consumer devices. Increased awareness between devices could also benefit unrelated parallel use, for example the music on a laptop could automatically be turned down when a phone call is received [87]. In sequential use, indicating from what device an application state has been transferred has also been found important [94], as, for example, an email marked as read had a different meaning on PCs than on mobiles to some users.

2.2 Cross-device Systems

In the last two decades, a multitude of cross-device systems and interaction techniques have been explored. Taxonomies and frameworks have been created to classify multi-device systems based on the number and size of devices and the number of involved users [185] as well as their interactions [176]. In this section, we discuss the interaction techniques used in existing systems before we examine single-user as well as multi-user scenarios.

2.2.1 Interaction Techniques

The combination of multiple devices into a whole system creates challenges and opportunities in the interaction design. As discussed in the previous section, transferring data from one device to another is a persisting challenge. Some researchers have explored how the user interface should be designed for cross-device interactions by adding menus and buttons for sharing information across devices [10, 24, 58, 137]. Others have explored novel input modalities for cross-device interactions. One of the earliest works is Pick-and-Drop by Rekimoto et al. [157] that builds on direct manipulation using a stylus. Pick-and-Drop is an extension to the widely used drag-and-drop interaction technique. The latter typically uses a mouse and cannot be easily applied across physical device boundaries. Instead of using a mouse, Pick-and-Drop employs a stylus: Objects can be picked up with a pen in one location and dropped in another location by tapping the device with the pen.

Similar direct manipulation approaches have been proposed using mobile phones rather than a stylus as an input device [60, 162, 163, 167]. Some techniques have been developed in particular for interactions with large surfaces such as interactive tabletops [162, 163]. As each phone is typically associated with a user, phone touches on the surface can be assigned to a user, which is generally not possible for finger touches without using special-purpose hardware [41]. The association of touches with a user identity allows personal clipboards to be implemented in multi-user systems [164] and undo operations can be executed per user. The phone's capabilities exceed those of a stylus in some respects: it comes with a touch display and contains a user's data. The touch display can be used to show personalised menus, but can also act as a more private extension of surface [163]. For example, to select a photo, a user can access their personal album on their phone rather than on a shared tabletop where everyone can see its content. Personal data, such as a contact, can easily be selected on the phone and transferred to a shared surface using a phone touch. Smartwatches have been used in a similar role in combination with tabletops: as a palette for instruments, providing personalised feedback, and as personal content repository [14].

Phones and tablets have also been used as tool palettes for whiteboard interaction [158]. Phones can even be used in an eyes-free manner, due to their smaller size, to eliminate attention switches between the devices [120] (Fig. 2.2). Whereas digital whiteboards generally have pen input and the phones and tablets are used to provide additional functionality, some larger displays support no direct input on the surface itself (such as projectors in meeting rooms or display walls) or are out of reach. Early work extends mouse and keyboard input across multiple devices [86], allowing a wall display to be controlled from a laptop using the latter's input devices. All displays are combined into a single virtual desktop where a cursor can easily move from one display to another. The system supports multiple users with multiple mice and keyboards, however is limited to a single cursor per physical device. Multiple users are supported in the Remote Commander of the Pebbles project [132] where PDAs are used to emulate mouse and keyboard inputs on a PC. Later iterations of Pebbles use PDAs as slideshow controllers [131] and to interact with screens at a distance in combination with laser pointers [134]. The laser pointer is used to specify a region of interest on the larger screen which is copied onto the handheld device where the user can interact with it using direct manipulation. More recently, similar interaction techniques have

Figure 2.2: Using a phone as an eyes-free tool palette for whiteboard interaction [120]

been explored replacing PDAs with smartphones and making use of commercial body tracking sensors such as as the Kinect. Mid-air pointing gestures have been combined with touch input on phones [13, 168]. Some systems use the phone's camera to capture the content of the remote screen and make it interactive on the phone [12], to create a virtual projection area on the remote screen showing content from the phone [6], or to transfer state between a phone and a PC [21]. Phones have also been used to navigate 3D environments on large displays through tilting, making use of the phone's gyroscope sensor [153].

While the above techniques capitalise on the different capabilities of heterogeneous devices, the combination of a more homogeneous set of devices has also been explored. Multiple tablets, phones, or screens can be combined into one large surface for increased screen real estate [2, 111, 146, 155]. Several interaction techniques have been explored to establish a connection between the devices. Using a stylus, the user can stitch devices together by drawing a line starting from one device and ending on another [68]. Devices can also be paired with a pinch gesture involving two devices [110]. SurfaceLink can connect devices that are located on a shared surface through gestures on the surface rather than the devices themselves [52]. Similarly, AirLink can can be used to connect devices with a in-air wave gesture between the devices [23]. All of these techniques not only connect the devices, but also extract their relative positions. If no position information is required, a whole range of pairing interaction techniques is available [32, 188]. Recently, camera based techniques have also been explored for device association [36, 196].

Rädle et al. have demonstrated that most people prefer spatially aware interaction techniques over spatially agnostic ones in a study involving multiple tablets [156].

However, a more recent study found that the preference for spatially aware interactions disappeared in some device configurations and physical alignments [137]. Rich spatial information enables proxemic interaction techniques that take into account distance, orientation, movements, and identity of devices and users [3]. One proxemic pattern, gradual engagement, is to reveal more information on a screen as the user comes closer and to make the screen interactive once the user is in close proximity [116]. Information on the relative position of people in a room can be used to determine group memberships, which enables interactions within the group [117]. For example, a gesture can be used to share information with people from the group but not with those who are in close proximity but not part of the group. Within a group, gestures using micro-mobility, small movements of devices, enable the sharing of data across devices. For example, a device can be tilted towards another device to open a portal. After a file has been dragged through the portal on the sending device, it will appear on the recipient.

Emerging form factors have also been explored as part of cross-device systems. Smartwatches can be used in combination with phones, serving either as an additional sensor in the background or assuming a role in the foreground as input device or extended display [26]. MultiFi [55] combines bodyworn devices, such as a watch, with a headworn display for seamless interaction on the go. A headworn device was also used as a mediator between devices in a user's environment in the Gluey project [169] and eye-tracking devices have been used to enable gaze-based interaction across multiple devices [104, 187].

2.2.2 Single-User Scenarios

In this section, we inspect single user cross-device scenarios found in the literature and outside of research. While some applications can be used by a single as well as multiple users, we classify them as single-user scenarios if collaboration is not explicitly addressed and multi-user otherwise.

Quite frequent are distributed media players. For video players, the video itself is shown on the largest screen and smaller, ready-to-hand devices such as tablets can be used as controllers [45, 197]. When more than two devices are available, these can be used to show additional content such as a search panel or related videos [197]. Google offers a commercial product, the Chromecast[1], to control a TV or an audio system from any device in the same Wifi network. These applications fall into the category of the *remote control* pattern and make use of the different devices properties: while phones and tablets are ready to hand, TVs have larger screens better suited for showing a video. A slightly different scenario is proposed in Sammi [63] where a spatially-tracked smartwatch is used both to navigate a video timeline and to show a preview of the video that is being played on a mobile phone.

Others make use of the combined screen real estate of multiple devices. The video itself can be split into multiple tiles which can be played synchronously distributed over multiple screens [45, 165]. Extra devices are also used to show additional content related to the video that is being played. Nielsen found in 2010 that nearly 60% of people watching TV were using the internet at the same time [53] and launched an

[1] `https://www.google.com/intl/de_ch/chromecast/?utm_source=chromecast.com` Accessed on 13. 01. 2017

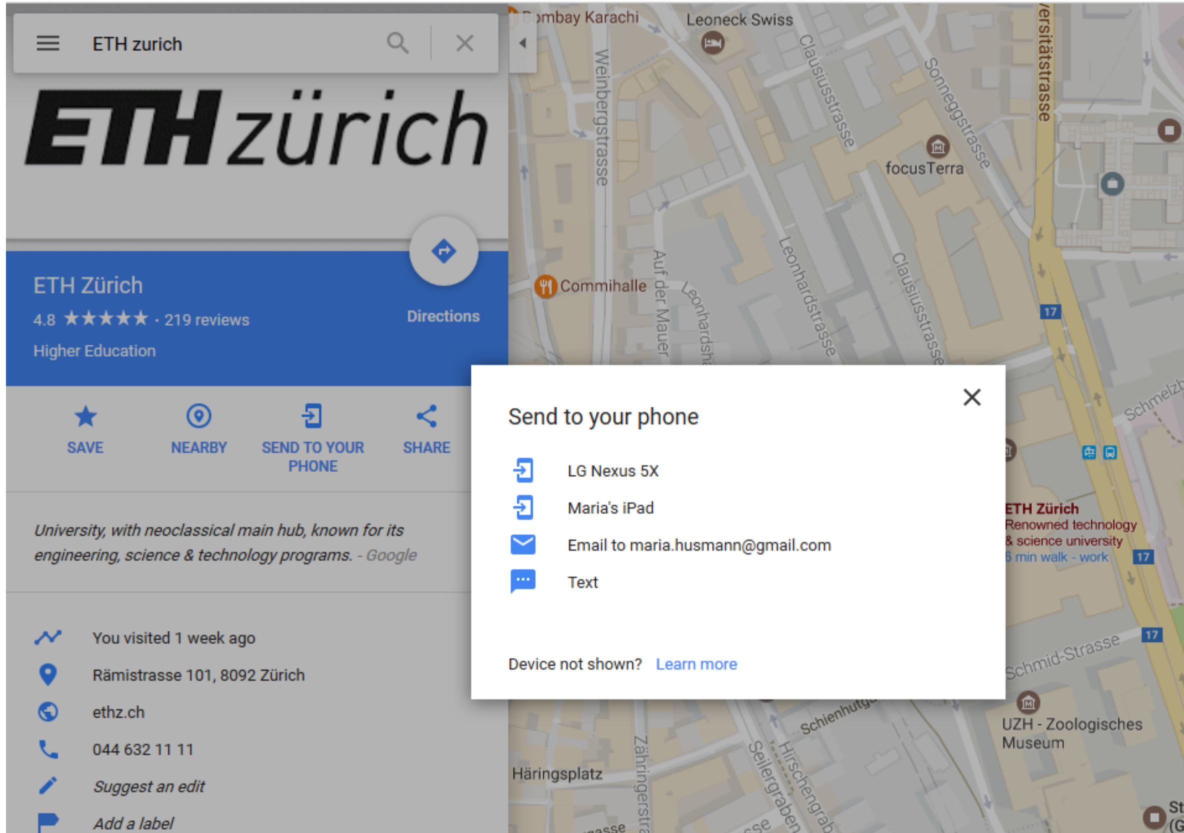


Figure 2.3: Sending a location from a PC to a phone using Google Maps.

app that shows companion content synchronised to the TV program. The term *second screen* has been used to describe these experiences and applications [69, 135, 136]. The content displayed on the second screen can be pulled from existing sources such as Wikipedia or Google Maps [103].

Another common scenario for cross-device use are map applications. Multiple devices can be coordinated to show different views of the same map location (for example satellite images or street maps) or to show different zoom levels [197]. Similar to videos, maps can be tiled across multiple devices and navigated synchronously [62]. Secondary devices can also be used to show information about markers selected on a primary device [63]. Multiple devices can not only be combined to increase screen real estate, but also to combine interaction modalities. MultiFi displays a larger map on a headmounted display and enables direct touch interaction by integrating a smartwatch [55]. By default, Google Maps only supports sequential use of multiple devices by offering a menu option to send the current location from a PC to a phone[2] (Fig. 2.3).

The *remote control* pattern has also been applied to slideshow presentations. Microsoft's PowerPoint provides a companion Office Remote[3] app that allows the speaker to control the presentation and to consult notes on a mobile device. Apple offers similar

[2] `https://support.google.com/maps/answer/6081481?hl=en` Accessed on 13. 01. 2017

[3] `https://www.microsoft.com/en-us/research/project/office-remote/` Accessed on 13. 01. 2017

functionality with its Keynote[4] software.

The increased screen real estate offered by multiple tablets has been exploited for active reading [24, 25] and sensemaking tasks [58]. Hamilton and Wigdor observed that participants assigned different roles to the devices [58]. Typically, a central device was used for notetaking while additional devices were used to store important documents or look up keywords. A similar, more recent study in a collaborative setting found relatively little use of more than one device per user [154]. The finding was attributed to a legacy bias. A study carried out with XDBrowser investigated how users would distribute a web application across a phone and a small tablet and found patterns that were common between users [138].

Commercial providers have recently started to explore better support for cross-device device usage as well. A notable example is Apple's continuity system[5]. It allows users to answer phone calls on their Mac instead of their iPhone and provides support for sequential multi-device use via a cross-device clipboard and the Handoff system. The system simplifies sharing a phone's cell network connection with a computer over Wifi.

In summary, most single user scenarios aim at either increasing screen real estate by combining multiple devices or implement the remote control pattern with a smaller ready-to-hand device and a larger out of reach screen. Most commercial systems focus on sequential use cases and resource sharing and provide little support for parallel patterns.

2.2.3 Multi-User Scenarios

A common scenario for multi-user cross-device systems are smart rooms [7, 57, 85, 181, 193]. These rooms are equipped with digital devices, such as wall displays and digital tabletops, to enhance team meetings and collaboration. In addition to the fixed equipment, people can integrate their own devices (laptops and handheld device) into the system (Fig. 2.4). The iRoom, for example, allows meeting participants to bring material on their laptop and present it on a primary screen in the room [85]. The larger, shared devices offer a public platform for discussion, whereas the personal devices offer more privacy, for example to search for content within personal files [57, 158]. The iRoom comes with a room controller application that visualises the devices in the room based on their location and allows a user to choose, for example, where a presentation should be shown.

Some systems have been built specifically for brainstorming and content creation. CurationSpace enables instrumental interaction for creating and curating historical artefacts [14] . Instruments can be selected on a smartwatch and applied to objects on a shared digital tabletop. The results can be stored and carried away on the watch. The NICE discussion room integrates digital pens and paper into the multi-device environment [57]. Bragdon et al. have aimed to support a more democratic process than the one that is supported in traditional meeting rooms that are equipped with a projector connected to a single computer [13]. Skeleton tracking sensors in the room enable every participant to use pointing gestures and their mobile phones to control the content on

[4] `https://support.apple.com/en-us/HT204378` Accessed on 13. 01. 2017

[5] `https://support.apple.com/en-us/HT204681` Accessed on 13. 01. 2017

Figure 2.4: A multi-user, multi-device scenario [140]

the projector. The system has been demonstrated in the context of software developer meetings. A similar goal was shared by the UD Co-Spaces project that demonstrated how a multi-surface system can increase engagement and improve collaboration in the setting of participatory urban design [112].

Other smart room projects have focused on data visualisation and exploration, using large screens or display walls in combination with tabletops or smaller devices such as laptops to display large datasets [7, 193]. Other collaborative scenarios with a large amount of data have been explored in the context of emergency response planning [31] as well as oil and gas exploration [170].

Data visualisation and collaborative sensemaking scenarios have also been explored in multi-device settings outside of smart rooms. Polychrome is a flexible cross-device framework that can apply different display space configurations to (existing) data visualisation applications [2]. RAMPARTS is a spatially aware sensemaking system using tablets [195], whereas others have used tablets in combination with a digital tabletop and found that it performs better than a tablet-only solution [191]. In contrast, Zagermann et al. found that sensemaking performance did not decrease when the tabletop was replaced with a much smaller shared tablet, however they found that communication styles were changed to adapt to the situation [198]. A whole range of multi-surface systems have been developed for geo-spatial data (see [172] for an overview). For example, CozyMaps targets multiple tablets combined with a larger display [27]. The larger display shows an overview and provides a space for sharing whereas the tablets allow individual users to interact with the map. Interactions on the tablets are reflected on the large display to increase awareness. While some of these systems operate in the browser but are targeted to a specific domain, general purpose solutions for collaborative browsing across multiple devices have been proposed as well [59, 109, 194].

Content creation scenarios have also been explored in ad-hoc settings as opposed to smart rooms with fixed equipment. Mobile phones have become ubiquitous and there are multiple systems that have been built for a group of users whose phones are combined into a collaborative system, for example for brainstorming [110]. The same approach has also been used to share photos [111, 146].

Multi-surface systems have been used for giving presentations [103, 105, 159, 186] and in classrooms [18]. The presentation itself can be distributed across multiple devices [103, 105] and the devices of the audience can be used to add interactivity such as polls and quizzes [159, 186].

Bardram has explored a hospital scenario [4, 5], an environment where work is fast-paced, mobile, as well as collaborative and often multiple tasks have to be executed in parallel. To tackle these challenges, he proposes the ABC system which allows activities to move between multiple devices.

A combination of multiple phones and a single tablet has been used for card games, where the player's hand of cards is shown on the private phone screen and the tablet is used as a shared surface [174].

Similar to the single-user scenarios, multi-device systems are used to increase screen real estate, which is particularly beneficial in situations with a large amount of data. In addition to the increased screen real estate, multi-device systems can allow multiple participants to interact at the same time, facilitating collaboration and fostering democratic processes. In contrast to single-user systems, multi-user systems have a higher need to provide awareness cues about the various interactions that occur. To avoid conflicts, many systems rely on social protocol, but explicit conflict management solutions have also been implemented [121].

When it comes to the sets of devices that are used, two general directions can be observed. The first assumes a relatively fixed set of devices, for example, systems for specially equipped rooms. The second kind of scenario is more flexible and targets whatever devices are available within a group of users. The first kind of system often includes a heterogeneous set of devices, ranging from digital tabletops, wall-sized displays, and laptops to tablets and mobile phones. Within the second kind, we find systems that are completely independent of device types and others that are targeted towards more homogeneous device sets, for example multiple mobile phones.

2.3 The Development Process

In this section, we investigate how applications described in the scenarios of the previous section could be built. We analyse tools and frameworks for the whole development process starting from design and prototyping to testing and usage analysis. Some solutions can be used across multiple stages of the process and those will be featured in all relevant subsections. The design of software is typically iterative [61] and software developers often cycle through short implementation, testing, and debugging phases [106]. Consequently, the stages presented here are not completely independent. Instead, a result from the analytics phase could trigger another iteration in the design or a failed test could lead to debugging and changes to the implementation. Our focus is on tools that have been built specifically for cross-device applications but we also consider conventional tools where appropriate.

2.3.1 Design and Prototyping

The first phase of the development process is characterised by experimentation and exploration. The designer or developer decides on features, the interaction techniques, and the distribution of the different parts of the application across devices, as well as the visual design. At this stage, fast iterations are important and it should be possible to try out an idea in a quick and easy manner [61]. Some of the tools that we introduce in this section have not been built with designers and developers in mind, but are targeted at non-technical end-users. We still include them in our analysis as they allow experimentation and could be used to explore alternative ideas.

A wide range of tools have been developed to support the design and prototyping stage. Some of the earlier work has addressed designing applications that can run on and adapt to many different types of devices, often coined **multi-device applications**. In contrast to cross-device applications, these run on a single device, but have been designed to cope with different platforms and device types, for example, they will execute on a PC as well as a mobile phone. There are two main issues that need to be addressed. First, there are different platforms that often come with their own style guides and design patterns [42]. Second, different screen sizes and interaction modalities may require different designs. For example, a button may be large enough to control with a mouse on a PC, but too small on a mobile phone with touch input. Building an individual prototype or design for each platform and device type requires a lot of effort that the research community has tried to reduce. The interface builder Gummy addresses the need for interfaces that run on multiple platforms by allowing the developer to create a graphical user interface (GUI) based on a GUI that was created for another platform [124]. The system maintains a platform-independent representation of the GUI. In a later iteration, the different versions were linked and changes on one platform could be propagated to another platform [125]. The same problem can also be tackled through programming by demonstration: Macros can be recorded in D-Macs for one device and replayed on other devices [126]. For example, when a label needs to be updated in multiple designs, the designer can record the change in one design and automate the change in the other designs. Another approach uses a library of design patterns which have been optimised for different device types in conjunction with layers which separate UI elements that are used on multiple devices from the ones that are specific to a given type [108].

While adapting a UI to different platforms and devices is necessary in cross-device applications, it is not sufficient. In multi-device applications the whole interface is transferred in some form to a device, but, in cross-device applications, different parts of the UI can end up on different devices. This **distribution of the UI** across devices and the interaction between the devices also needs to be designed and specified. Different mechanisms for designing and specifying distributed user interfaces (DUIs) have been explored. One concept is model-based engineering, where the user interface and its distribution are described in a modelling language [47, 113, 123, 151]. Properties of the devices, users, and environment can be modelled as well [123, 118]. These systems generate executable code based on the models that have been authored. Some projects include a visual tool that assists the designer in building the model rather than having to author it in a domain specific language (DSL) [113].

Instead of implementing a cross-device application from scratch, **existing applica-**

tions or services could be used as a basis. Some of the earlier work building on existing user interfaces targets UI migration [49, 118, 175, 190], addressing sequential use cases. The state of the application is retained as the application is moved from one device to another. Ghiani et al. provide a visual migration client which can either push an application from a source device to a target device or, on a target device, pull the application from the source [49]. The user can choose to migrate the whole application or only a part of it. For parallel use cases, our GUI builder XD-Studio, also takes existing UIs as a basis [141]. A developer defines distribution profiles based on the device types and user roles that are expected to use the application. The developer can then assign parts from the existing UI to the different profiles either by dragging and dropping UI elements from the source interface or by loading the whole UI and removing unnecessary elements. Both Polychrome [2] and HydraScope [62] can also be used to distribute and synchronise existing applications across multiple devices. In contrast to XD-Studio, the focus of these two projects is on the synchronisation of state rather than the visual design, however, they do offer a platform for experimentation.

Reusing existing UIs, components, and services is the core idea behind mashups. While the majority of works on mashups target the single-device use case, solutions for cross-device usage have also been explored [35, 101]. DireWolf allows small UI components (widgets) to be distributed across multiple devices using a visual DUI manager [101].

The **granularity** of the UI distribution or migration can be used to classify cross-device tools and frameworks [150]. While some only allow the interface as whole to be moved between devices [175], others operate at the level of widgets [101], arbitrary UI elements (such as buttons, labels or containers) [141], or even allow arbitrary window regions to be selected [138, 183].

The solutions that we have discussed so far are designed for UIs that adhere to the WIMP paradigm (window, icon, menu, pointing device). New **UI paradigms** specifically for cross-device applications have been explored. Squidy focuses on modalities other than mouse and keyboard and provides an interaction library as well as a visual tool for rapid prototyping [99]. ZOIL introduces design principles for zoomable object-oriented information landscapes [83]. Multiple toolkits have been built to facilitate prototyping interaction techniques that rely on sensor data, such as proxemic interactions [70, 115, 171]. Some of the toolkits provide a visualiser for tracked devices, persons, and their relationships [115, 171]. Sensor data is abstracted into higher-level entities and events [70, 115] and data from multiple sensors are fused [70, 171]. Some support experimentation not only at the software level, but also with hardware, by allowing additional sensors to be added in an easy manner [70, 171].

Different approaches can be observed in terms of **devices needing to be present** during the design. Some systems, mainly those focusing on end-user-development, do not differentiate between design and run-time [49, 62, 101, 118, 138]. In these cases, the application can only be distributed across devices that are present. Others, in particular model-based approaches, implement a two step process separating the design-time from the run-time [47, 113, 123, 141, 151]. At design-time, the devices need not be present. XDStudio [141], however, supports the design on actual devices, if they are present (Fig. 2.5). Otherwise, the designer can work on a main development device where other devices are either emulated [70, 141] or not required for specifying the behaviour of a cross-device application [47, 113, 123, 151]. To check the designs at run-time and iterate

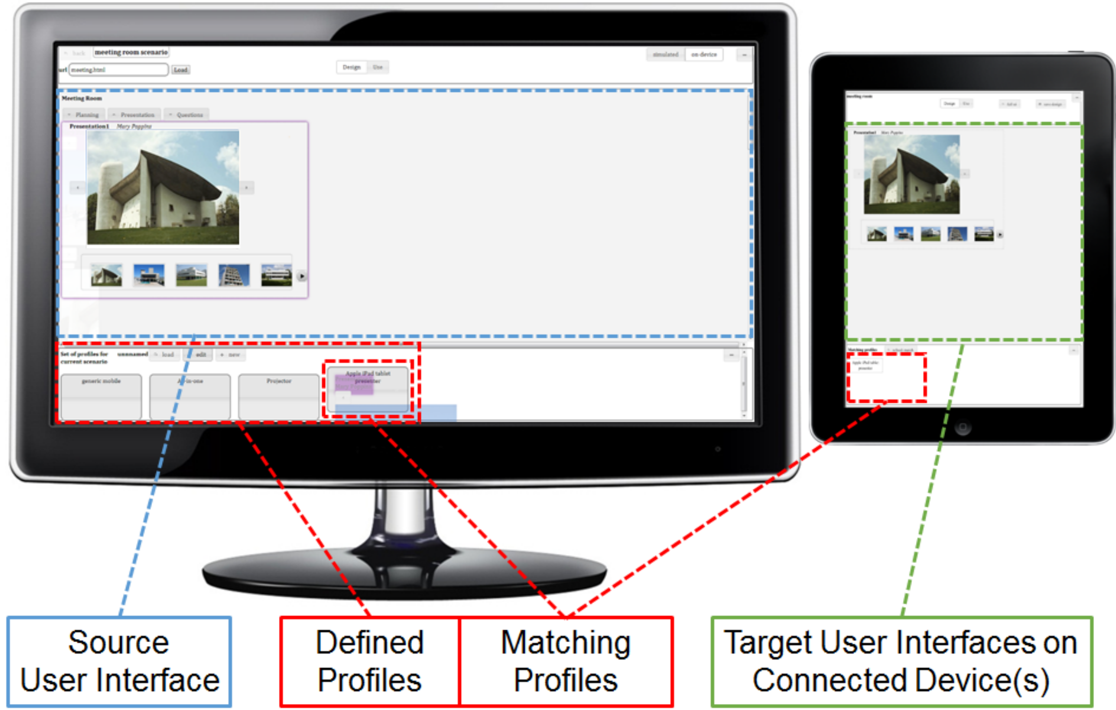


Figure 2.5: With XDStudio, design is supported on a developer machine as well as on actual devices [141].

over them, some tools integrate device emulators [70, 124, 141]. Device emulation will be discussed in more depth in the debugging section.

A number of **theoretical frameworks** map out the design space, provide design guidelines, and facilitate the requirements analysis [150, 176, 190]. The 4C framework [176] focuses on the *interactions* in multi-user, multi-device ecosystems. The framework identifies 4 themes of interactions: communality, continuity, collaboration, and complementarity. Communality refers to devices that are used by multiple users sequentially, for example a ticket machine. The sequential usage pattern that we have observed previously involving a single user and multiple devices is analysed using the term continuity. Collaboration describes how multiple users share a single device, for example a tabletop. Finally, complementarity refers to parallel usage patterns of a single user. Continuity is also part of a user experience framework by Wäljas et al. [190]. It is complemented by composition and consistency. Composition refers to the roles that are allocated to devices, the distribution of functionality, and how systems adapt if a device with a certain functionality is not available in a certain situation. Consistency refers to shared look-and-feel, semantics, and interactions across all devices. The framework of Paternò and Santoro [150] mainly refers to software design (for example the communication architecture), but includes some relevant dimensions for designing user interaction.

Design patterns can also inform the design process. Two studies have collected empirical data about user preferences regarding UI distributions of existing applications [82, 138]. Both observed a remote control pattern where a phone is used to control another device. A related pattern was labelled view+input by Nebeling et

al. [138], where one device is chosen to provide input to the system due to its superior input capabilities. The same authors also observed mirror and extend patterns. In the first case, the whole interface is duplicated and synchronised across multiple devices. In the second case, only part of the interface from a first device is duplicated on a second device. Hutchings and Pierce observed an influence of the environment – public, semi-public, or private – on the chosen distributions [82]. Both studies advocate against fully automated UI distributions and are in favour of giving some control to the user.

The main challenges in designing cross-device applications are introduced by the immense design space. Supporting multiple platforms and device types has been a challenge even for single device applications and the problem is exacerbated in cross-device development [42]. A designer may not know exactly what devices are going to be used and may not have that exact device configuration at hand. Attempting to cover every possible configuration in the design phase with a concrete design is hardly feasible [82]. Instead, the design phase could account for some expected configurations and be combined with a flexible implementation that can adapt to unexpected ones, similar to responsive web design [114]. Furthermore, the user could be given some control over the assignment of roles and UI components to devices.

2.3.2 Architectures and Implementation

The implementation step transforms the design from the previous phase into a working system. Typically this is done by writing some form of source code. As in the previous phase, a lot of decisions need to be made: Technologies, architectures, and frameworks, need to be chosen. There are various frameworks and libraries that reduce the low-level work required for cross-device applications, in particular related to the distribution of the UI and the synchronisation of state across multiple devices.

A criterion to classify a cross-device system is the **platforms** that it supports and the mechanisms for cross-platform support, if any, that it provides. Some systems target only a single platform, for example Windows [10, 83] or Android [58]. These systems have been built either for a homogeneous set of devices (for example multiple desktop computers [10] or tablets and phones [58]) or they have been used in controlled settings where the hardware can be coordinated to match the given platform [83], as is the case in smart rooms. In contrast, this thesis targets a more flexible class of applications that adapt to the devices at hand. Hence, the hardware and platforms cannot be assumed to meet a specific requirement. Rather, interoperability has been identified as a crucial requirement for cross-device applications [71].

There are several approaches to supporting multiple platforms. One approach is to implement the same functionality in different programming languages and use a platform-independent communication protocol between the devices [1, 22, 171]. While this approach allows a heterogeneous set of devices to be used in combination, the interoperability comes at the cost of higher development effort. Even though a library provides building blocks in multiple languages and facilitates the implementation, the client application itself still needs to be implemented separately in each language. This duplication of code complicates not only the development but also the maintenance of the applications as multiple code bases need to be synchronised. A benefit of the approach is that it allows each client to be tailored to the specific platform, for example in terms of look-and-feel or by using functionality that is not offered everywhere.

The development effort can be lowered by building on platform-independent toolkits or frameworks, such as Mono[6] or Tcl/Tk[7] [71, 122]. Using such frameworks, code can be written once and will be executed on multiple platforms. A similar strategy is used in model-based approaches where a concrete user interface can be specified independent of the platform and executable applications will be generated for multiple platforms [123].

The same write-once-run-everywhere effect can be achieved with **web technologies**. While differences in browser implementations remain and the rate at which browsers implement new standards vary, applications written in web technologies can be executed on a wide range of hardware and platforms. At the same time, the web has evolved from being centred on documents to supporting fully-fledged applications. New APIs provide access to sensor data such as the geolocation or device motion, enabling interactions that had been previously possible only with native technologies. The newly introduced protocols WebSocket and WebRTC enable bi-directional communication between client and server and between multiple clients respectively, a crucial building block for applications that span multiple devices. Web applications come with the additional benefits of requiring no installation step, thus enabling lightweight, ad-hoc interaction in contrast to native applications that first need to be installed. Furthermore, the web community has already tackled the problem of adapting to different device form factors and developed approaches such as responsive web design [114]. Web standards have been introduced to address this challenge, for example CSS media queries and flexbox layouts. The research community has recognised this potential and a number of web-based cross-device frameworks and libraries have been developed [2, 28, 48, 64, 97, 129, 155, 165, 197].

Another aspect that needs to be considered is the **system architecture** of a cross-device application. One approach is to have a central server that coordinates the interaction between the client devices [28, 58, 64, 83, 142, 143, 147, 155, 171, 197]. This has been the predominant approach. On the other end of the spectrum are peer-to-peer solutions where the devices communicate with each other directly [45, 100, 122]. Fisher et al. have argued that a peer-to-peer architecture has benefits over a client-server architecture for distributed user interfaces, mainly due to robustness, scalability and security [45]. Latency can be lowered when messages are directly passed between clients instead of going through a server [100]. On the other hand, a powerful central server can facilitate synchronisation and conflict resolution [147] and relieve mobile devices with little computational power [150]. In 2012, Paternò and Santoro [150] noted that the space of peer-to-peer DUI applications is underexplored. This was largely due to the lack of peer-to-peer communication protocols in the browser at that time. The introduction of WebRTC has since created an opportunity to experiment with peer-to-peer systems [2, 48, 100, 103], however, not all browser vendors have implemented the protocol to date. Apart from pure client-server and peer-to-peer architectures, there are also hybrid or flexible systems [71]. Badam et al. use peer-to-peer connections to synchronise data between clients and a server for persistence [2]. Others eliminate the need for a remote sever by having one of the clients dynamically take over the role of the server or coordinator [33, 47, 72, 165]. This role can even be transferred between devices at run time, which could be relevant when the current server device is removed [46, 72]. This approach enables functional applications even when there is

[6] `http://www.mono-project.com/` Accessed on 06. 02. 2017
[7] `https://www.tcl.tk/software/tcltk/platforms.html` Accessed on 06. 02. 2017

no internet connection (a requirement in most client-server systems) as Bluetooth and local networks can be used to communicate [165].

Another step in the implementation of a cross-device application is the **specification of the UI distribution**. In this step, the developer specifies which part of a UI should be allocated to which device. Several frameworks provide mechanisms for UI distribution using different approaches. Some use an *event-based* approach [28, 50, 123, 155, 165] where an event, such as a device joining or leaving, a user interaction, or a change in the context, triggers an update to the user interface and its distribution. Each of these frameworks provides commands for manipulating the UI of a given device in an imperative manner, for example showing or hiding a UI element or moving it from one device to another. Panelrama used a *declarative* approach based on affinity scores [197]. UI components that belong together are wrapped in panels which can be distributed across devices. The developer assigns affinity scores to each panel describing its suitability to be allocated to a device based on its characteristics. For example, in a video player application consisting of a controller panel and a video playback panel, the controller panel would have have a high *ready to hand* score because it requires user input whereas the video playback panel would have a high *physical size* score to show the video. The system automatically calculates an optimal distribution of the panels based on the properties of the available devices. If a TV and a phone were available in our example, the controller would be assigned to the phone and the playback panel to the TV. A similar automatic distribution of the UI can be achieved in Weave with the *combine* operator that groups multiple devices into one virtual device [28]. Similar to Panelrama, developers using Weave can address devices matching high-level input and output characteristics rather than specific device types. Weave's successor DemoScript infers these device selectors automatically based on concrete example devices that a developer can pick in a visual tool [29]. This abstraction allows developers to implement applications that are, to some degree, independent of the specific devices that will use them. As discussed in the previous sections, users assign roles to their devices, which can in some cases be independent of device characteristics, for example when a set of homogeneous devices is used [58]. Thus, basing the distribution of the UI solely on device characteristics could fail to meet users' needs. Most frameworks have no explicit concept for roles. With those that do, the roles are associated with the user rather than the device [46, 71]. For example, an application can be adapted to a user who has admin rights or is a guide (as opposed to a tourist), essentially implementing a role concept similar to the one that we have seen in the design phase with XDStudio [141].

A distributed application comes with an inherent challenge of keeping the state consistent across all devices. Most cross-device frameworks provide a mechanism for **state synchronisation**. The synchronisation can be handled *explicitly* or *implicitly* [2, 197]. In the explicit case, the developer has to trigger the sharing manually. Panelrama differentiates three different explicit sharing functions: state can be *pushed* from a device to a global shared state representation, it can be *pulled* from the global representation to overwrite the local state, and it can be *merged*, where application-specific logic combines the local and the global state [197]. In the implicit case, the developer marks state variables that need to be shared and the system tracks changes and shares them to the other devices whenever they change without any need for further intervention from the developer. Polychrome also proposes *unilateral* sharing where a single leader device

can write changes while the other devices can only listen to changes [2]. Fine grained control over data sharing can be obtained with our session concept [143]. A session associates data, users, and devices. Devices can join multiple sessions and changes to data within a session are propagated to all devices in the session. This approach is targeted towards *multi-user* cross-device applications where some data might be private and only shared with a user's devices whereas other data is shared with other users. In mashup and component-based approaches, a loosely-coupled *publish-subscribe* approach is predominant where UI components such as widgets publish state changes as events and other components register to be notified of these events, regardless of the devices on which the sender and the listener are situated [48, 100, 103].

Two approaches can be observed for describing state changes: a *snapshot* of the new state can be transmitted [4, 100, 197] or an *operation* that will transform the old state into the new state [2, 65, 97]. The second approach can ensure consistency and conflict resolution through operation transformation (OT) when multiple devices send updates simultaneously. In contrast, the snapshot approach facilitates the addition of newcomer devices to a system. The new device only needs to be sent the latest version of the shared state [4] whereas, in the OT approach, a (potentially long) history of operations needs to be applied. In either approach, changes can be captured at the level of the *model* [48, 65, 100, 103, 143, 197] or its representation in the *UI* [2, 97]. For example, in a video application with a play/pause toggle button, a change of state can be described as a change from the play state to the pause state or as a click of the button. In Polychrome, these two approaches are labelled data-centric and interaction-centric [2]. However, the second approach does not necessarily track only interactions such as clicks but can also track changes in the structure of the UI which might be a result of interactions but could also be triggered by the system. This is the approach used by Webstrates where changes in the DOM are synchronised across devices [97]. In this UI-centric approach, no semantic knowledge on the data is required and it is thus also suited to legacy applications [2]. On the other hand, in the model-centric case, semantic information could allow more advanced conflict resolution or merging and it enables different representations of the same model on different device types.

Conventional WIMP UIs are usually built with some variation of the model-view-controller (MVC or MV* to denote variations) **software architecture pattern** [102], separating the model data from its visual representation, the view. A myriad of MV* frameworks have been developed in particular for web applications[8]. The MV* pattern has also been used for distributed applications where the model is kept in sync across multiple devices and the views are updated accordingly [1, 54, 83]. Other software architectures have been explored in the context of new interaction paradigms such as instrumental interaction [51, 72, 71, 96]: VIGO (Views, Instruments, Objects, Governors) is a pattern where instruments can manipulate objects (represented in views) mediated by governors [96]. Shared substance builds on the same concept of instrumental interaction and promotes a data-oriented approach [51]. In contrast to object-oriented design, this approach decouples data and operations that can be executed on the data. Activity-centric approaches focus on activities as first class objects [4] which are composed of actions, users, and meta-data [71, 72].

Some frameworks provide explicit support for **device management** ranging from

[8]See `http://todomvc.com/` for a comprehensive list

built-in pairing methods [58, 197] to programming APIs for accessing information about the devices in the system [152]. Discovering devices and pairing them into a cross-device system has been declared an integral requirement for cross-device infrastructure [71]. Devices in physical proximity can be detected and paired automatically through Bluetooth [165]. Other techniques that have been implemented in cross-device frameworks include QR codes [58, 197], NFC [58], and gestures such as bumping [58]. Some web-based systems implement pairing based on URLs which are shared between devices [97, 197]. Devices can also be associated based on their owner [152]. The infrastructure of Pierce and Nichols has turned device ownership into a first class property and a list of users and the devices they own is maintained on the server [152]. The association of devices and users allows a device to send messages to other devices belonging to the same user. Devices are also typically a first class property in frameworks with event-based distribution approaches [28, 165] and in model-based systems [50, 123], allowing a device to react to the presence (or absence) of another device. On the other hand, in the declarative approach of Panelrama, the distribution of the UI is handled by the server and there is no mechanism for a device to query and directly interact with other devices in the system [197].

Novel cross-device interaction styles such as proxemic interaction rely on **sensor** data to provide information on the position of other devices and users. Most position-tracking sensors provide streams of low level data and often multiple data-sources need to be integrated. Some cross-device frameworks and toolkits provide higher-level APIs and facilitate the integration of multiple sensors. Both the SoD [171] and the Proximity Toolkit [115] generate high level events for changes in the relationship among devices and between devices and users, for example distance or orientation changes. Similarly, WatchConnect [70] provides an event-based API for gestures sensed by a smartwatch, for example a mid-air swipe, whereas XDKinect [142] enables web-based cross-device applications with gesture and speech interaction through a JavaScript event API. The SoD Toolkit [171] can fuse data from multiple sensors into location events, freeing the developer from having to coordinate them. Alternatively, the developer can choose a sensor based on confidence levels provided by the toolkit. HuddleLamp [155] can be used to abstract away the sensing completely: It creates and manages a virtual workspace in which devices positioned on a flat surface are tracked. The developer specifies the content of the workspace while the system ensures that each device renders the right excerpt of the workspace. Tracko [84] eliminates the need for external sensing by leveraging built-in sensors of phones, namely Bluetooth, the microphone, and inertial sensors, to track the presence and relative position of devices and to enable gestures in 3D space.

This section illustrates the vast amount of related work that has been done to support the implementation of cross-device applications. As in the design phase, it remains a challenge to cover all possible device configurations with the implementation. With event-based approaches, the order of the events, for example devices joining the system, could potentially impact the state of the UI, as each event can trigger change. Frequently, sample applications that are used to illustrate these approaches use only two devices or omit any explanation of how the application adapts to varying configurations [28, 50]. It is unclear, if a developer would manage to correctly predict how the UI behaves with a given device configuration, if it was one they did not have in mind during the implementation. With declarative approaches such as Panelrama's

affinity scores, the order in which devices are added to the system is not expected to have an impact, however, developers still struggled to predict how the UI would distribute across devices without executing the applications [197]. While developer tools that provide previews or execute the application are discussed in the following section, another approach worth investigating could be to base implementations on cross-device patterns [137, 138] and to introduce a concept for device roles.

2.3.3 Debugging

We use the term debugging to describe the next phase in the development cycle. We will use the term in a broad sense not only focusing on fixing bugs, but including informal checks verifying if the software behaves as expected and adapting the implementation if it does not. This cycle of writing code and checking the result is how developers typically work during code composition tasks [173]. Developers also dedicate time to code comprehension which is the understanding of existing code [173], a prerequisite for debugging. Finally, there is the actual debugging task [173] where the developer attempts to eliminate a bug from the code. A crucial first step is reproducing the bug [9]. Next, the developer will typically determine why something did or did not happen, for example by stepping through the code with a debugger [98]. In this section, we analyse how existing cross-device tools support checking the implementation, code comprehension, and debugging. Since this thesis focuses on web-based cross-device applications, we also examine how these tasks are supported in conventional web development.

Two approaches can be observed for checking the implementation. Either the code can be executed on **emulated devices** [28, 29, 141, 143] or on **real devices** [28, 141, 143]. Emulated devices offer the benefit of needing less (potentially expensive) hardware. Adding a new device can be as easy as clicking a few buttons and it is typically faster to load a new version of the code onto a device emulated on the developer machine than deploying and loading it onto external hardware. For these reasons, device emulators have also been used to verify multi-device applications [125]. On the other hand, emulation cannot cover every aspect of a real device [42], for example haptics and form factors which could be central to the user experience. Weave [28], XDSession [143], and XDStudio [141] offer support for executing the application on both emulated or real devices or a mix thereof. However, none of these tools couples the emulation with the code. Weave provides a logging mechanism but no debugger integration. This approach mainly facilitates checking if an application works or looks as expected, but in cases where it does not, little support is offered to track down the source of the problem. Thus the tools operate at a higher level of abstraction and mainly help to test and debug the design and the interaction rather than the implementation. DemoScript [29], a follow-up on Weave, connects the code with the emulated devices, allowing the developer to step through the code line by line and showing the result on emulated devices. A sequence of steps in the code is visualised as a storyboard that not only shows the current state (as is typical for device emulation) but also previous states corresponding to each line in the code.

Checking an implementation is particularly challenging when **sensors** are involved and may require physical manipulations [42]. This is the case in particular for applications with proxemic interaction techniques [115]. Device emulators have also been used for this scenario. WatchConnect [70] provides a hardware emulator for smartwatches

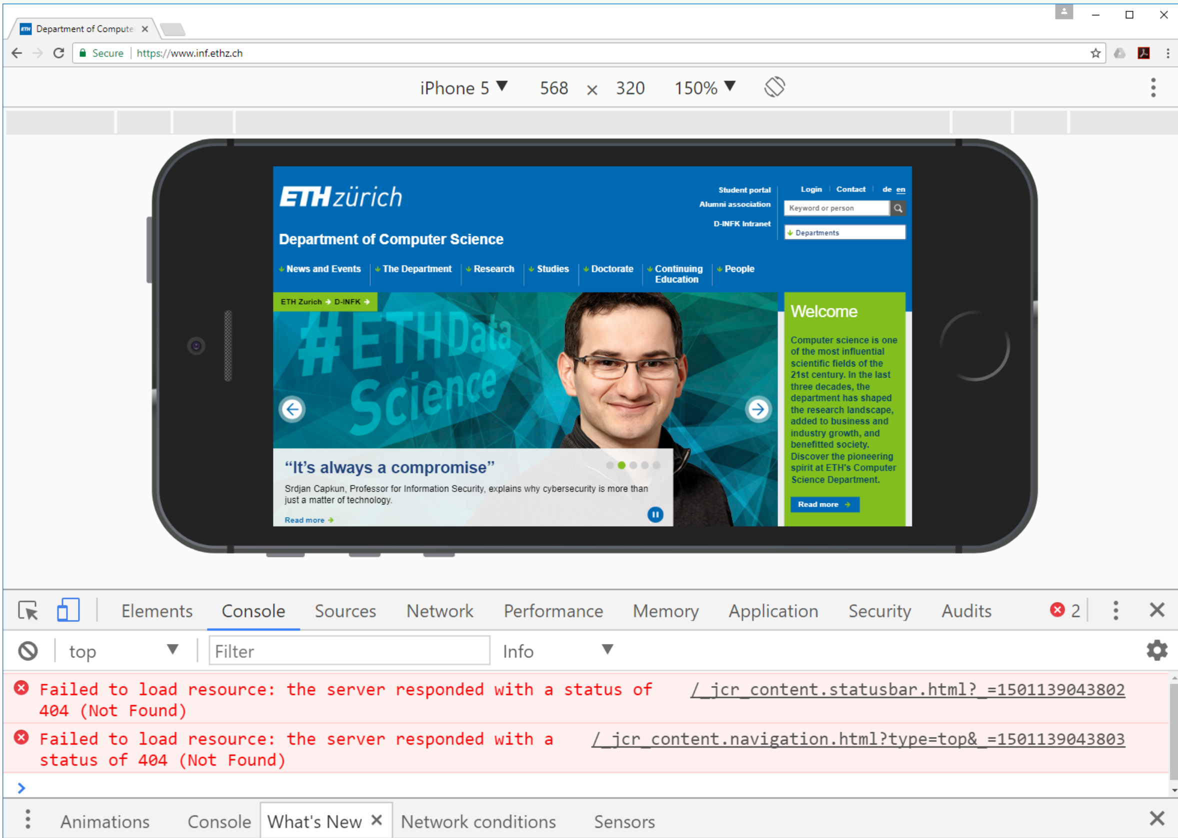


Figure 2.6: An emulated phone in the Chrome browser.

and focuses on testing the sensor data and machine learning algorithms. The Proximity Toolkit [115] provides no emulation, but visualises tracked entities and their relationship which can help a developer's understanding of the data model. Furthermore, a developer can **record** sensor data with the Proximity Toolkit and **replay** it later. This functionality eliminates the need to re-enact proxemic interactions physically every time a change to the implementation needs to be checked. A similar approach could be used for Kinect-based cross-device interactions [142] and the tool Kinect Analysis [144], however the latter offers no cross-device specific support.

Record and replay functionality is also offered in XDSession [143]. The functionality is implemented at the level of session manipulations as opposed to user interactions, that involve the creation, deletion, and changes of data. As a result, the tool can only be used with applications that use XDSession as an implementation framework. In addition to session manipulations, meta-data is recorded including the device and user that triggered the manipulation. The data is visualised on a timeline in the Session Inspector along with information on devices joining and leaving sessions. A recording can be transferred from one set of devices to be replayed on another, for example from a phone and tablet to a tablet and a TV, allowing the developer to compare and contrast the results. A Session Controller provides a means to manage devices, sessions, and users through a visual interface.

Modern **web applications** are normally designed to work on a multitude of devices, ranging from large screens to small smartphones. However, developers usually use

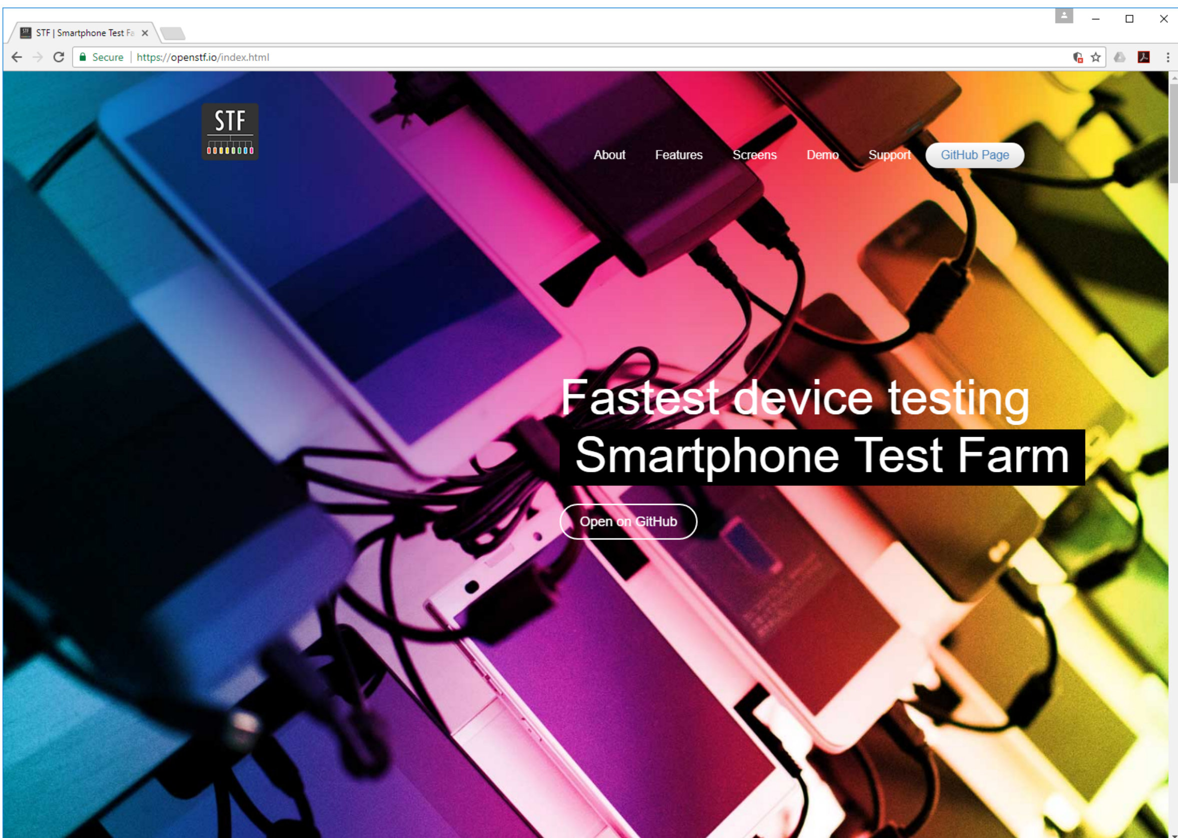


Figure 2.7: Smartphone Test Farm offers products for managing device inventories for testing and debugging.

powerful desktop or laptop machines with keyboards and large screens to work efficiently. As with cross-device applications, there is the need to test and debug these applications on devices other than the developer machine. Furthermore, implementations of web standards vary across different browser vendors and operating systems, so testing with just one of them may not unearth all of the issues. A number of tools have been proposed to address these issues and again we can distinguish approaches based on device emulation from those using real devices. Modern browsers have support for device emulation built in. In Chrome, for example, common device models can be chosen from a list or custom ones created dynamically. For preconfigured devices, the system stores screen dimensions and pixel density and newer versions of the browser even include the device frame so that it can be captured in screenshots (Fig. 2.6). Chrome allows the network to be throttled to lower speed to emulate bad network conditions, however, no emulation of device computing power is offered. Basic emulation for sensor data including location, orientation, and touch is available, but the input of the data is static and not suitable for gestures. For example, the sensors can be set to emulate a portrait pose, but not a dynamic shaking gesture. The emulators are tightly coupled with the debugging tools of the browser allowing the developer to inspect and manipulate the program state at run-time and to step through the code.

Given the limitations of device emulation, support has also been provided for testing and debugging web applications on real devices. On smaller devices such as phones

there is little room for displaying debugging tools and relying on touch interaction could make it difficult to navigate through the code. The problem has been addressed with remote debugging tools that display the debugging interface on a developer machine and communicate with the device under test. Early approaches were browser agnostic and required a debug script to be injected into the client application that would communicate with a debug server over HTTP [127, 130]. Since then browser vendors have integrated remote debugging into the desktop and mobile versions and usually require that the mobile device is connected via USB. The cable introduces a scalability issue when many different devices need to be tested. Products such as Smartphone Test Farm[9] (Fig 2.7) or Device Lab[10] have been developed to address these issues with software for managing device inventories and physical components such as stands. Services such as BrowserStack[11] and CrossBrowserTesting[12] eliminate the need to have physical (or virtual) device inventories and offer testing on real devices as a service. Screenshots can be generated in both services and CrossBrowserTesting offers access to debugging tools.

Whenever the source code of a web application changes, the browser needs to reload the application both on real and emulated devices. This can become tedious, in particular when many devices need to be tested in parallel. Various tools (for example BrowserSync[13] or Ghostlab[14]) have automated this process by observing the source files for changes and automatically refreshing the browser. In addition, both tools can synchronise user interactions such as clicks, scrolling or text input across multiple windows, allowing a developer to verify multiple devices in parallel. Cross-device applications typically include a pairing step where devices are associated with each other before the actual interaction. That pairing step would also need to be repeated on every reload and there is currently no tool support for this specific task, so it has to be executed manually by the developer.

Capture and replay of events is another established technique for debugging web applications. Mugshot [128] captures events such as user interaction and non-determinism on end-user machines and allows a developer to replay the program execution on a developer machine to track down program failures. FireCrystal [148] uses capture and replay to increase code comprehension of dynamic behaviour. The developer can record an interaction and analyse on timeline how the DOM changed and what part of the HTML, CSS, and JavaScript were involved. An interactive timeline is also offered by Timelapse [17] which integrates with existing debugging tools such as breakpoints, the call stack, and source code that can be inspected and stepped through. The tool proved to be most useful to experts whereas less-skilled developers were distracted by it. All three systems supported recording and replaying on a single device and thus do not cover cross-device use cases where interactions can take place on multiple devices in parallel. Replaying the same interaction on each device would not suit the characteristics of cross-device applications, as each device could be displaying different UI elements for interaction. Many cross-device applications are built for collaborative scenarios. A

[9] `http://openstf.io/index.html` Accessed on 14.02.2017
[10] `https://www.vanamco.com/devicelab/` Accessed on 20.04. 2017
[11] `https://www.browserstack.com/` Accessed on 14.02.2017
[12] `https://crossbrowsertesting.com/` Accessed on 14.02.2017
[13] `https://browsersync.io/` Accessed on 14. 02. 2017
[14] `https://www.vanamco.com/ghostlab/` Accessed on 14. 02. 2017

single developer may struggle to accurately provide input to all devices in a realistic manner without help. A replay mechanism that coordinates multiple devices and takes into account their different roles could alleviate the problem.

2.3.4 Testing

In the previous section, we have examined how applications are frequently checked as the developer is writing code. These informal tests mainly serve the purpose of seeing if the feature that was just implemented works or looks as expected. However, changes to the code could introduce unexpected side effects that may not be covered in the informal test. Testing all aspects of an application manually after every small change in the code would require a lot of time and would be tedious due to the repetitive nature of the task. On the other hand, omitting such tests can lead to regressions. For this reason, software tests have been automated and testing has become an integral part of software engineering practice.

Testing can be done at various **levels of abstraction**. On one side of the spectrum are *unit tests* with high granularity that verify program behaviour at the level of individual classes or functions [15]. At the other end of the spectrum, there is *system testing* where a whole, integrated system is verified. UI testing, also referred to as end-to-end testing, is a form of system testing where user interactions are mimicked on the interface, for example a button click could be triggered and then a label would be checked to see if it shows updated information.

Testing can also be analysed in terms of the **coupling** with the underlying code. The aim of unit tests is to check every possible state and interaction between system components [15]. Consequently, they require knowledge of the internal structure of the system and are highly coupled with the code. Such an approach is labelled *white-box* testing. In contrast, UI testing focuses on the expected input/output behaviour of the system and has less coupling with the code. It is is thus a *black-box* testing approach.

As unit tests operate in isolation and at a very high granularity, the impact of the fragmentation and distribution of cross-device applications on the testing is limited. The behaviour under test would typically be executed on a single device. In contrast, UI testing is highly impacted by the distribution and fragmentation. Multiple devices need to be coordinated during a test. For example, clicking a button on one device, could update a label on another device. This makes UI testing a particular challenge for cross-device developers and we will focus on that approach in this thesis.

Two main approaches for UI testing can be observed: Interactions with the UI can be *recorded and replayed* or they can be *scripted* by a tester [107]. Tools like Selenium[15] allow a tester to demonstrate interactions to the system which will be recorded and can be replayed repeatedly. While this requires little knowledge of the structure of the UI, it can fail when significant changes are introduced. The scripting approach can take longer initially to carefully craft interactions, but has been shown to be easier to adapt to changes [16]. A study has shown that the scripting approach can become less expensive than record and replay after one to three releases [107]. The reduced maintenance costs of the scripts amortise the higher initial costs of writing the scripts.

To the best of our knowledge, there is no specific support for testing cross-device

[15] `http://www.seleniumhq.org/` Accessed on 15. 02. 2017

applications. However, a subset of the issues in testing cross-device applications are addressed by existing work. Most conventional applications are developed on desktop machines, but many will run on mobile devices. Testdroid [90] enables tests to be executed in parallel on multiple mobile devices. It executes test scripts that can either be written manually or derived from recorded interactions. Testing novel forms of interactions can be a particular challenge as most testing systems are limited to conventional mouse and keyboard input. Hesenius et al. have extended an existing test automation framework with support for gesture-based interactions [66]. Neither approach handles the coordination of multiple devices with interaction between each other. Furthermore, the uncertainty of what devices will actually use the application is not accounted for.

2.3.5 Usage Analysis

The previous sections illustrate the extensive interest of the research community in cross-device applications and interaction. However, little is known about how cross-device applications are used outside of studies and lab conditions. Under these controlled conditions users are typically equipped with a specific set of devices and instructed how to use the application. Once an application has been deployed, it is difficult to assess how it is used in the wild as current cross-device frameworks have no built-in support for collecting usage data. Knowing how an application is used could inform a future iteration of the design or enable data-driven design [38]. For example, information on observed device configurations could prompt an optimisation of that specific configuration. Furthermore, given that cross-device applications are still rare outside of research labs, users may not realise that an application can be used with multiple devices in parallel [154]. If such a legacy bias is detected, efforts could be made to educate the user about how other available devices could be integrated.

The research community has investigated how people use their set of devices. However, these studies were rarely at the level of a single application but rather focused on general use. Most work is based on self-reported data that is obtained through interviews, surveys, diary studies, activity logs or a combination thereof [37, 87, 95, 149, 161]. Other work focuses on tracking activities on a single device. Church et al. provide a comprehensive summary of challenges and techniques of tracking users on mobile devices [34]. A logger needs to be installed on the devices or integrated into an application. Tracked data includes application launches, sensor data (for example device motion), and screen state (on or off). While most work focuses on analysis of a single device, ENGAGE [88] measures engagement in multi-device settings. It combines camera data from desktop and laptops with information on the current application and screen state on mobile devices to infer user engagement across the set of devices. These tracking approaches produce a large amount of interesting data, however, they can also raise privacy concerns [34]. Most studies are thus conducted with a smaller set of participants and a limited duration (a notable exception being [189] with 12'500 participants tracked over two years).

In contrast, more lightweight approaches are well-established in the domain of web applications [20, 178]. Given that many cross-device applications are built with web technologies, these approaches could be extended for cross-device applications. To track a web application, typically a script is injected into the client-side code, tracking user interactions such as visited URLs and information about the device. No installation of

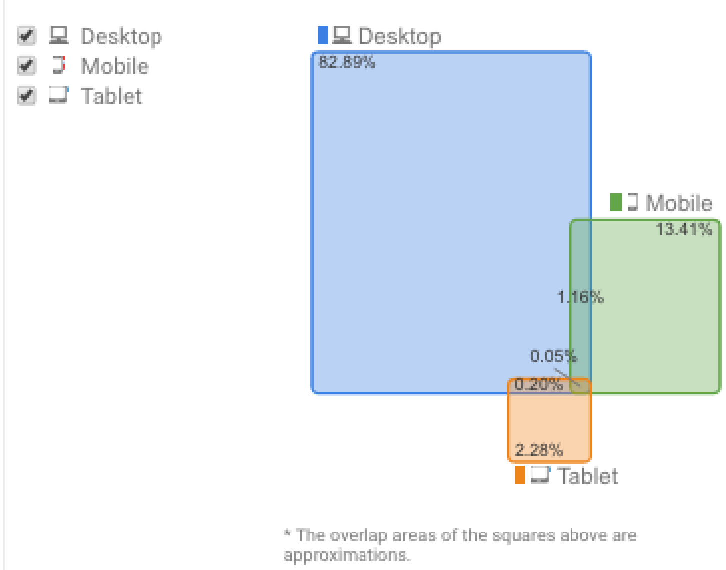


Figure 2.8: Google Analytics showing device overlap.

software by the user is required. As a trade off, compared to the heavyweight methods discussed above, less information is obtained, for example measuring user engagement based on camera data is not possible. On the other hand, this approach introduces fewer privacy issues. The behaviour is typically tracked at the level of a single application rather than general device usage. Commercial systems such as Google Analytics[16], Alexa[17], and Web-Stat[18], provide such tracking services that can be integrated into any website. They generate reports and visualisations of the tracked data that can be used a basis to improve a website or measure business goals. For example, if an online shop measures only few sales on tablets, this finding could prompt an evaluation of the tablet user interface and the checkout procedure. The focus of these services is on single-device use. While Google Analytics offers basic cross-device reports[19] based on devices associated with the same users, the tracked metrics are either very general or only target sequential use cases: *Device overlap* tracks the device types and numbers thereof that are used to access an application (Fig. 2.8). There is no information that specifies if any of these accesses were in parallel or on the order of the devices used. However, it provides information on the kind of devices that are generally used to access an application. The order of the devices used is tracked with *device paths*, which show the last 5 devices used before a conversion, for example a purchase. There is no support for tracking parallel usage of multiple devices, which would be of particular interest in cross-device applications.

[16] `http://www.google.com/analytics/` Accessed on 20. 02. 2017
[17] `http://www.alexa.com/tools` Accessed on 20. 02. 2017
[18] `http://www.web-stat.com/` Accessed on 20. 02. 2017
[19] `https://support.google.com/analytics/answer/3234673` Accessed on 20. 02. 2017

2.4 Conclusion

Most prior work specific to cross-device applications targets the early phases in the process, namely design and implementation. This may be sufficient to build prototypes to study in research labs, but does not address the challenges of building products that are used outside of such controlled environments. Products need to be robust to endure usage deviating from the main envisioned scenarios. Their code base needs to be maintained, often over years and by a changing team. In conventional development, for example in web development, a host of tools exist to support testing, debugging, and usage analysis. In this thesis, we use them as a basis for our investigation into better support for cross-device application development. We introduce tools for all stages in the process, but the main contributions are in debugging, testing, and usage analysis. We focus on flexible cross-device applications that adapt to the set of devices at hand and implement parallel usage patterns. This flexibility introduces the particular challenge that many different sets of devices could be used which leads to a large design space. Hutchings and Pierce have argued that it may not be possible to design an application that can react to every possible situation and that the user should be given the option to adapt the distribution of an application [82]. Dong et al. have also identified uncertainty of a user's intention as a challenge in the design phase [42]. The implementation needs to be capable of handling such flexibility. Our analysis showed that current implementation frameworks provide no concept of assigning roles to devices rather than users, even though it has been shown that this would better reflect the user's mental model [37, 161]. For informal checks and debugging, we can learn from responsive web design tools that allow a developer to easily switch between many different device types. However, these tools do not account for interdependency between the devices, which is an integral part of cross-device applications [42]. A crucial property of these tools is the tight coupling with the source code, which is missing in most cross-device tools. Even though it may not be possible to design for every imaginable device configuration, testing should cover a wide range of possible configurations to ensure robustness. Without automation this kind of testing would be very time consuming and probably skipped in many cases. The stages from implementation to usage analysis also suffer from the fragmentation of the application logic across multiple devices. The lack of tools for testing cross-device applications has been identified as a major challenge [42].

This thesis focuses on the engineering process of flexible, distributed graphical user interfaces. We acknowledge social challenges and those challenges specific to new device form factors such as glasses and watches, as well as new forms of interactions such as proxemics [56]. However, these are out of the scope of this thesis.

3 MultiMasher: Design and Prototyping

This chapter[1] presents visual tools and architectures for cross-device mashups. The goal is to support experimentation and the exploration of ideas using existing applications. Instead of building a new cross-device prototype from scratch, existing applications built for single device use can be mashed up and distributed across multiple devices. The resulting prototypes are functional and could be used to receive a first round of feedback from end-users. We present concepts for cross-device mashups and requirements for a mashup tool that we implemented in our MultiMasher prototype. MultiMasher provides visual tools for creating web-based cross-device mashups using direct manipulation and only requires limited technical knowledge of the underlying applications. We analyse two different architectures that we explored with MultiMasher and discuss trade-offs. Finally we evaluate MultiMasher along the dimensions of a logical framework [150] that we extended for cross-device mashups and in terms of its technical compatibility with 50 top websites. This chapter was developed in [179].

3.1 Concepts and Requirements

We define a cross-device mashup as a web application that reuses content, presentation, and functionality provided by other web pages and that is distributed among multiple cooperating devices. The following scenario (Fig. 3.1) illustrates how cross-device mashups could be used in the design process.

> *Ted, a designer, has the goal of building a travel planning cross-device application. It targets users that are on the go and should offer information on nearby places. Ted expects potential users to each have their smartphone, but they could also want to make use of the larger, digital TV in their accommodation. A first step, Ted chooses Wikipedia as an information source. As Wikipedia provides good background information, but does not give a good*

[1]Earlier versions of parts of this chapter were originally published as Husmann et al. [74, 75].

Figure 3.1: A cross-device mashup composed of two websites and three devices.

visual impression of a place, he also wants to simultaneously add Google Image Search results for each place users look up. Ted uses MultiMasher to quickly mash up articles from Wikipedia and images from Google for the same place on the large screen. He envisions that each user could enter new locations with their smartphone. However, they should be able to use a single input field to update both the article and the images on the large screen rather than searching separately on each page. Ted demonstrates his mashup to two stakeholders who collaboratively explore a couple of places using their smartphones while sitting on the sofa. One of them mentions that he will typically also have his laptop with him when travelling and asks whether that device could be integrated. Using MultiMasher, Ted can easily migrate the mashup to the laptop while preserving the current state and demonstrate how he could adapt the design to that particular device.

We introduce the following three entities that are central to cross-device mashups: *components*, *mashups*, and *devices*. Components extracted from existing websites are composed into a set of inter-connected mashups which are accessed simultaneously by multiple devices (Fig. 3.2). We define a component as a subset of HTML elements that can range from a single input field to a nested container, such as the whole HTML body. A mashup contains a specific set of components. All devices accessing the same mashup receive the same set of components with the same state. To obtain a different view or an independent state, a new mashup must be created, however, the same component can be used in multiple mashups. Finally, a cross-device mashup is a set of multiple mashups accessed by multiple devices with communication between the mashups. The relation between user and device may be one-to-one, many-to-one or one-to-many as a user could use multiple devices at a time, for example a phone and a tablet, while a larger device such as a smart TV may be shared by multiple users.

We have identified the following dimensions of operations that a cross-device mashup tool should support: *mashing up*, *distributing* and *co-browsing* (see Table 3.1).

Mashing up refers to operations that are also used in conventional, single-device mashup development. *Component creation* can be supported programmatically or by direct manipulation of the UI. *Component manipulation and adaptation* includes moving, resizing, copying, and deleting components to design a mashup visually. For the

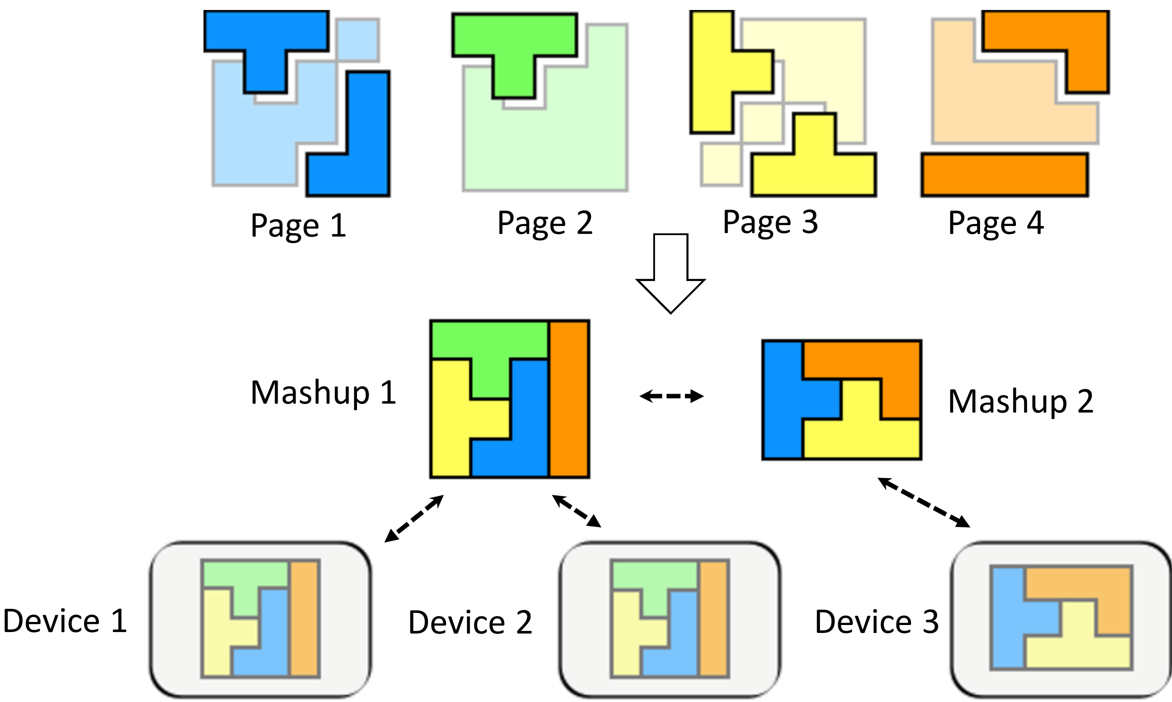


Figure 3.2: Overview of a cross-device mashup

design of functionality, *inter-component communication* is required so that the manipulation on one component, for example entering a text into an input field, triggers an update to another component, for example showing image results for that text. We distinguish explicit and implicit inter-component communication (Fig. 3.3). Components originating from the same website are expected to communicate implicitly, without any need for specification by a developer. For example, a component with the input from the Google image search is expected to communicate the search term to an image result component. This is the default behaviour of the website and it should persist despite the website's division into components. On the other hand, components stemming from two different websites must be explicitly connected by the developer to create interaction. Introducing a third component that contains the result of a search on Wikipedia into our example, it must be explicitly connected to the Google search component, if it is to update upon the input of a keyword into the Google search component.

In the distributing dimension, we group operations that are related to components being distributed across multiple mashups and devices. Component *migration* from one mashup to another (and consequently from one device to another) while maintaining its state and configuration, such as connections to other components, would support experimentation and quick iterations on the design. For example, when a new device joins an existing cross-device mashup, components could quickly be reshuffled to account for the new device. *Inter-component communication* is also relevant in the distributing dimension as the communication should function independent of components being on the same or two different mashups or devices. Note that in our model of cross-device mashups, the same mashup could also be accessed by multiple devices, requiring communication across devices even when components are located in a single mashup. The developer of the mashup should not have to think about the location of

Dimension	Operation	Details
Mashing up	Component creation	Programmatic Direct manipulaton
	Component manipulation and adaptation	Moving Resizing Copying Deleting
	Inter-component communication	Explicit Implicit
Distributing	Migration of components	Across devices
	Inter-component communication	Across devices
Co-browsing	Raising awareness for interactions	Between multiple users

Table 3.1: Operations in a cross-device mashup tool

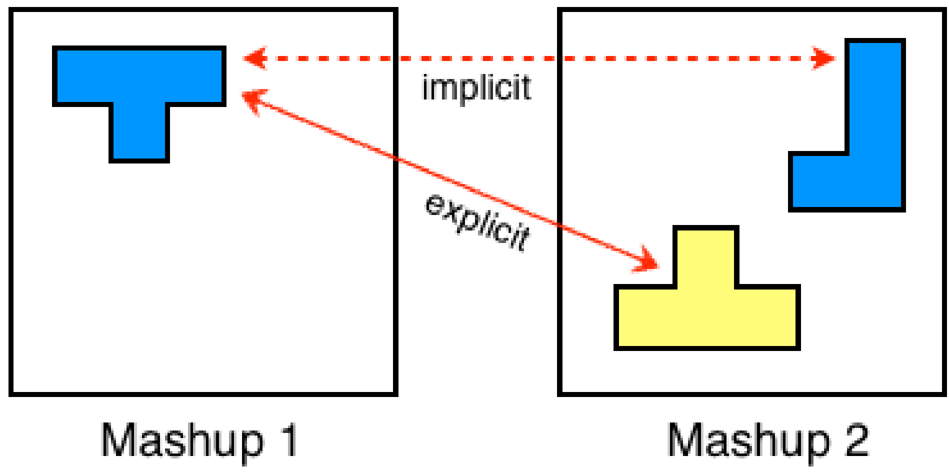


Figure 3.3: Implicit and explicit inter-component communication. Components from the same website communicate implicitly whereas components from different websites communicate explicitly. Both types of communication can span mashup and device boundaries.

the components and their movement across devices. Rather they should be able specify the communication channels between components, while the mashup tool handles all effects introduced by the fragmentation.

The co-browsing dimension covers the aspect of multiple users accessing a cross-device mashup simultaneously. As a user interacts with a cross-device mashup, updates to another user's view of the mashup are likely to occur. In our scenario, a person searching on the phone would update the shared TV. To avoid confusion, a mechanism for *raising awareness for interactions* on other devices should be employed.

3.2 MultiMasher

We have implemented the operations and concepts from the previous section in our MultiMasher prototype. MultiMasher provides visual tools for creating cross-device

mashups as well as a run-time platform. MultiMasher does not differentiate between design- and run-time. Any changes to a mashup take effect immediately, allowing a designer to quickly try out ideas. The MultiMasher client runs in a modern browser and does not require any installation. A new device can easily be added to a cross-device mashup by opening a URL and selecting the mashup to be displayed. We designed MultiMasher on the principle of direct manipulation, thus it requires no programming experience and was built with both technical and non-technical users in mind. MultiMasher consists of two main views: The *global view* provides an overview of the complete cross-device mashup, all the mashups of which it is composed, and all devices currently using the system. Upon the selection of a mashup, the *mashup view* is opened where the mashup can be edited and used.

3.2.1 Global View

The global view shows an overview of the whole cross-device mashup and all devices currently connected to the tool (Fig. 3.4). The user starts with an initially empty cross-device mashup and can start by creating a mashup. For each mashup, there is a preview in the global view. A newly created mashup is shown as an empty canvas. Coloured rectangles represent components inside a mashup. Components originating from the same website share a background colour, for example a search bar component and an image result component both coming from Google search. To differentiate multiple components from the same website, they are marked with distinct border colours. A list below the mashups maps the colours to websites and component names. It can be used as a guide to the user to understand which component is which. Hovering over a component in the list highlights it the mashup preview. The list includes information on explicit inter-component communication.

Connected devices are shown in another list and each device is assigned a colour. In the mashup preview, a small rectangle at the bottom right shows which devices are currently accessing a mashup. Basic information about the device such as the browser and the operating system is displayed to differentiate the devices. This information could be simplified into device types or supplemented with user provided device names in the future.

The mashup previews are interactive. Components can be resized and moved from one mashup to another using drag-and-drop. This direct manipulation provides an easy means to distribute components across devices. As all interactions take immediate effect, a component can be moved from one device to another simply by dragging it from one mashup to another in the global view. Any device connected to the affected mashup will update immediately.

3.2.2 Mashup View

A newly created mashup is represented as an empty canvas. By clicking and dragging the mouse over the canvas, the user can create a new component with the dimensions of the selected area (Fig. 3.5). A dialog prompts the user to provide a name and a URL for the website from where the component should be extracted. Alternatively, the website can be selected from a list with websites that have been used before. MultiMasher then loads the full website inside component boundaries. As a next step in *configuration*

Figure 3.4: The global view with two mashup previews (Mobile and Large) at the top. Components and devices (clients) are listed below the previews.

mode, the user can select UI elements that will be extracted as the component (Fig. 3.6). When in *element selection mode*, hovering over an HTML element adds a coloured border to that element to provide feedback on the current selection. The selection can be confirmed with a click and the user is prompted to provide a name (for example *search*). Subsequently, only the selected element and its descendants are displayed.

The *linking* mode provides a means to link components from different source websites to enable inter-component communication. Whereas components from the same website (for example article and the search box from Wikipedia) are linked implicitly, components from different sources need to be linked explicitly. MultiMasher supports these explicit links via tags than can be attached to change, submit and click events of an HTML element. Subsequently, all elements with identical tags for the same event type will be connected and an event triggered on one element will trigger the same event on all connected elements. For example, to ensure that a search on Wikipedia also triggers results on Google image search, the following approach needs to be taken. A search interaction consists of a user typing a text into an input and then either clicking a search button or hitting the enter key to submit the query. At the HTML level, this results in three events that need to be mapped. A `change` event for the input element that is triggered as the user types into the field, a `click` event when the button is pressed, and a `submit` even when instead the query is sent using the enter key. For each of these events, a label needs to be attached both on Wikipedia and Google image search. MultiMasher will then mirror all events that occur on the first website on the second one.

As an alternative to creating a component from scratch, an existing component can be cloned to produce a new component from the same website sharing the created

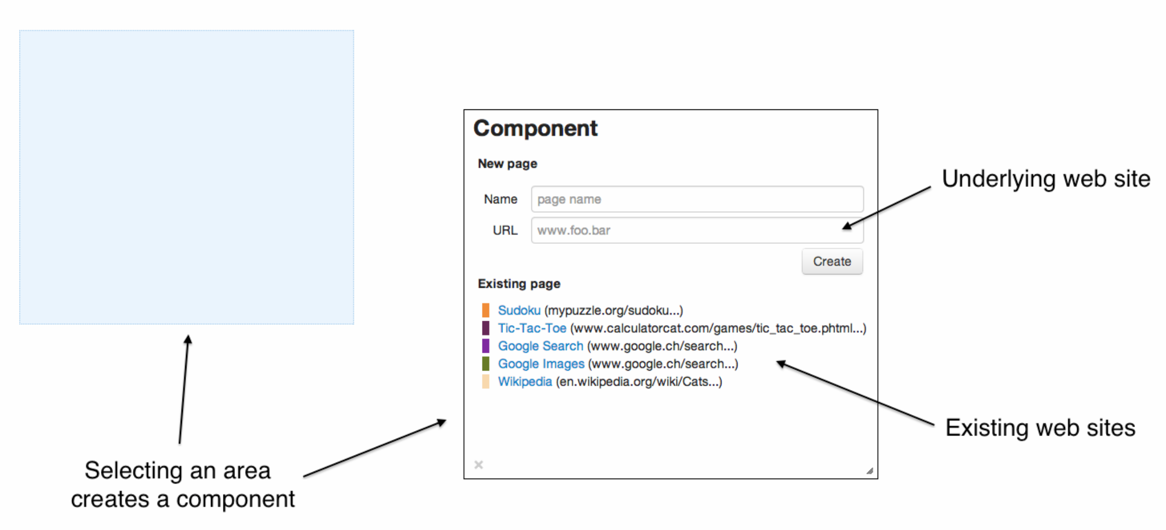


Figure 3.5: Dragging over an empty part of the canvas creates a new component.

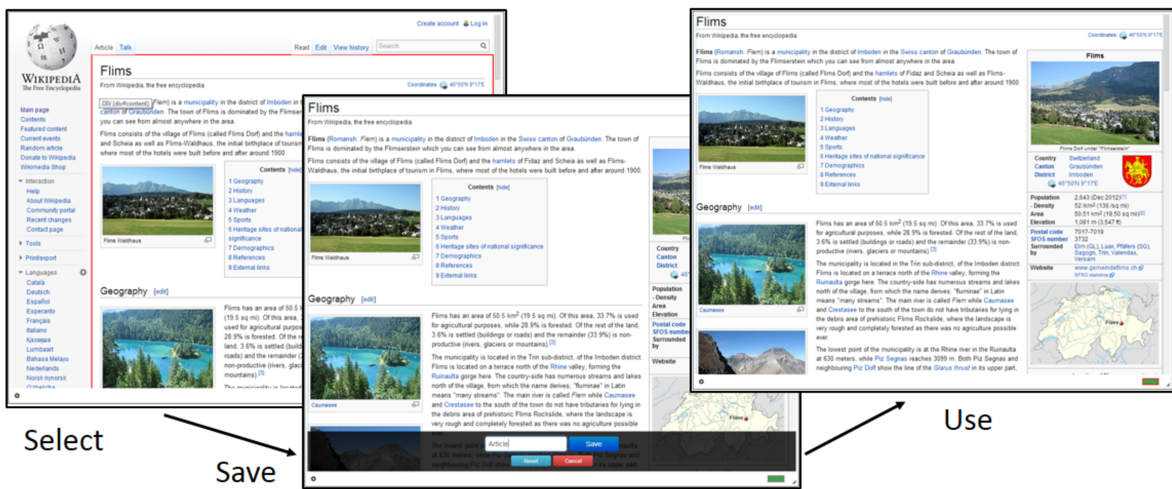


Figure 3.6: In selection mode, the currently selected elements are highlighted with a red border. After confirming the selection, only the selected element and its descendants are displayed.

event tags. Initially, the cloned component is an exact copy of the existing component. The designer can then go into the configuration mode where the cloned component will display the whole underlying website again. Then, the designer can choose a different set of HTML elements to be displayed. The event tags will be retained from the original component, saving the developer the effort of having to create them for each component coming from the same website.

Components can be resized and positioned inside the mashup using direct manipulation. As in the global view, all changes are immediately synchronised to all devices accessing the mashup. To move a component to a different mashup (and device), the global view needs to be used.

Once the designer is done configuring a component, it can be switched from *configuration mode* into *execution mode*. Now the component is ready for interaction, for example a user could enter a query into the search box. However, the designer can

switch back into *configuration mode* at any time to tweak the component.

3.2.3 Co-browsing Feedback

A user manipulating a mashup while others are accessing it can cause confusion. For example, a component that is being moved from one mashup to another would disappear from one device and emerge on another. MultiMasher provides an explanation to such events by highlighting components that are being manipulated with the colour of the client device that triggers the manipulation (Fig. 3.7). A toast message describing the manipulation slides in from the bottom of the screen and is shown for a couple of seconds.

Similarly, events that are triggered due to explicit component-linking are displayed at the bottom of each component. The event name is shown as well as the source page that triggered the original event (Fig. 3.8).

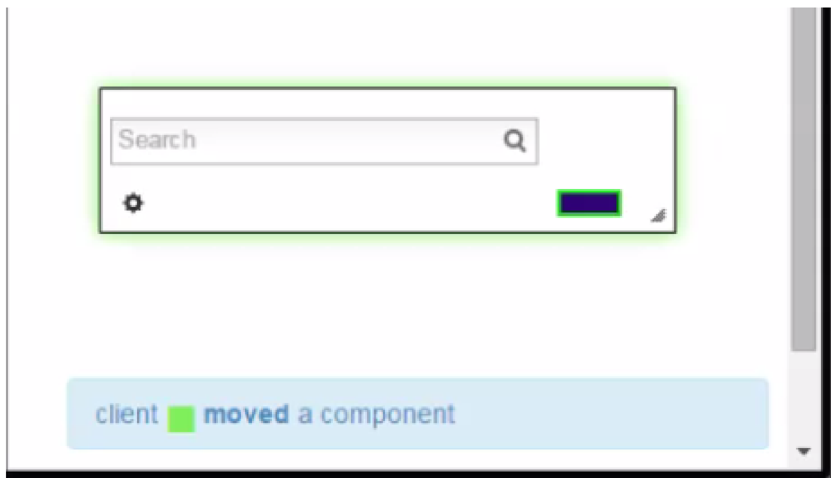


Figure 3.7: If a user moves a component, it is highlighted in the colour of the used device. In addition, a toast messages announces the manipulation to increase awareness.

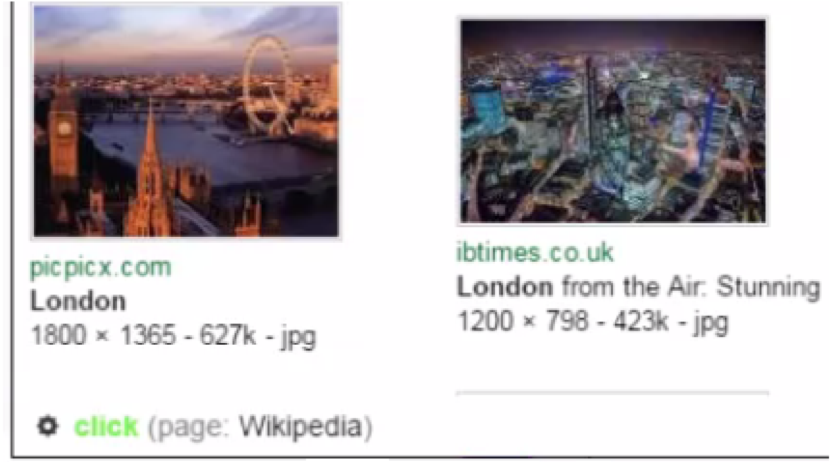


Figure 3.8: A component based on Google image search has been updated due to a click on the Wikipedia page. Awareness information is shown at the bottom left of the component.

3.3 Architecture and Implementation

In this section, we discuss the architecture and implementation of MultiMasher. We start by discussing two alternative architectures that we explored. Then we provide details on the implementation.

3.3.1 Alternative Architectures

One major challenge in cross-device applications is to keep the state synchronised across multiple devices. We have experimented with two different architectures for MultiMasher. The architectures are based on the two approaches Lowet and Goergen describe in [109]: JavaScript engine *input synchronisation* and *output synchronisation*. The first approach synchronises UI events, for example clicks. The synchronisation thus happens before JavaScript is executed. In contrast, output synchronisation propagates the changes to the DOM that have been triggered by an event and subsequent execution of JavaScript. Lowet and Goergen favour input synchronisation arguing with better scalability and user experience. This is the approach that we used in the first MultiMasher prototype [74]. In this version, events are tracked on all client devices. When an event occurs, it is forwarded to a server which propagates it to all clients participating in the cross-device mashup (Fig. 3.9 left). On each device, the MultiMasher event engine triggers the event, invoking event handlers that were installed by the original web application. The event handlers are also invoked on the device from which the event originated. For example, when the Wikipedia search button is clicked on one device, the MultiMasher event engine replicates that click on all connected devices. Consequently, the search is repeated on all devices and they all send a search request to Wikipedia, resulting in the same article being displayed on all devices.

Our experimentations with this architecture has revealed two main drawbacks. First, repeating requests to the server can be problematic, in particular when the request updates the server. The HTTP standard[2] describes some requests (in particular GET) as *safe*, meaning that they only serve to retrieve information and should not change the state of the server. Similarly, the DELETE and PUT requests should be *idempotent* so that multiple invocations of the same request have the same effect as a single invocation. Safe and idempotent requests pose no problem with input synchronisation. In contrast, POST is neither safe nor idempotent. Consequently, sending the same POST request from multiple devices due to the synchronisation of events could have unwanted side effects. For example, if a mashup contained an online shop, the input synchronisation could lead to the *buy* button being pressed multiple times (once per device). This in turn could lead to multiple orders. The second problem with input synchronisation is that the server might send different content to different devices, causing the devices to run out of sync. For example, a server may answer a search query differently based on the search history on each device or show personalised results.

Since MultiMasher is targeted to be used with arbitrary websites whose servers are out of our control, we opted for a centralised approach based on output synchronisation in the second prototype. Instead of synchronising the event itself, the changes to a website after the event has been triggered are synchronised. Each event is handled only

[2] `https://tools.ietf.org/html/rfc2616#section-9` Accessed on 13.03.2017

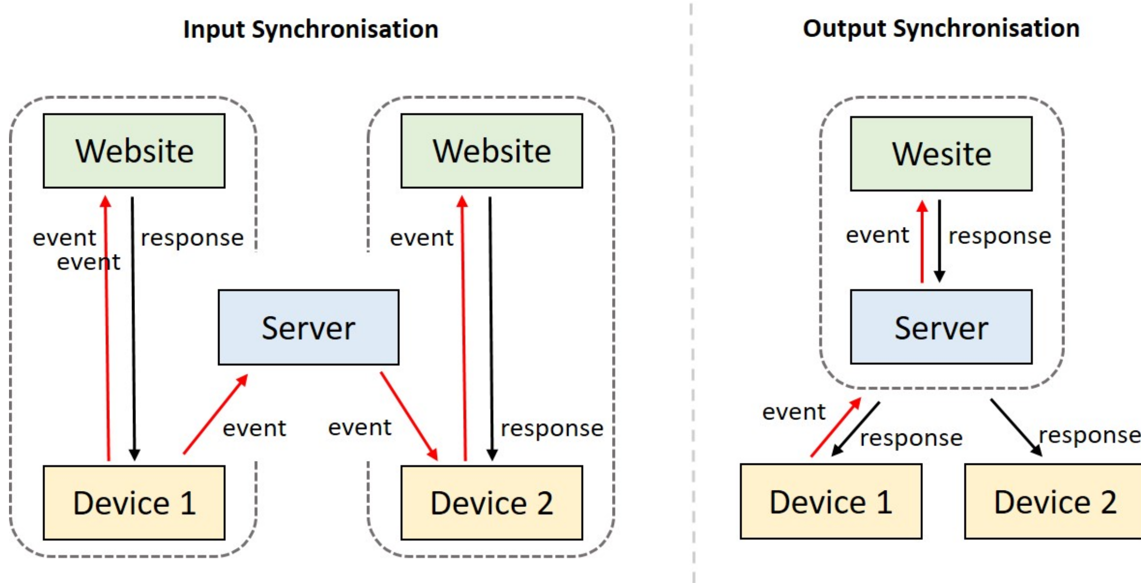


Figure 3.9: The input synchronisation MultiMasher architecture on the left and the output synchronisation architecture on the right.

once, eliminating duplicate requests to external servers. To simplify the state management, there is exactly one reference browser that applies all events and communicates the changes to the connected devices. We opted for a remote control solution similar to one used in Highlight [145] and use a browser on a server as a proxy (Fig. 3.9 right). Events on the devices are intercepted and forwarded to the proxy browser on the server. Only the proxy browser executes the attached event handlers. The event handler could, depending on the implemented functionality of the website, trigger local JavaScript updating the DOM or send requests to the server of the website to retrieve new data. The resulting website is sent to all connected devices, including the one that triggered the event. Whereas in the first approach, each device maintains a session with the website server, there is only a single session in the second approach, namely between the proxy browser and the website server.

3.3.2 Implementation

Figure 3.10 illustrates the MultiMasher implementation which constists of a client and a server component. The server is running on the Node.js[3] platform with MongoDB[4] for persistence. It uses the PhantomJS[5] headless browser as the proxy that handles the events. PhantomJS offers a GUI-less browser that can be controlled through a JavaScript API. Events are captured on the client, sent to the Node.js server and replayed in the PhantomJS proxy. The server then reads the updated DOM from the proxy and sends the changes back to the clients. The communication between the server and the clients has been realised with the Socket.io[6] WebSocket library. The client-side

[3]`https://nodejs.org/` Accessed on 13.03.2017
[4]`https://www.mongodb.com/` Accessed on 13.03.2017
[5]`http://phantomjs.org/` Accessed on 13.03.2017
[6]`https://socket.io/` Accessed on 13.03.2017

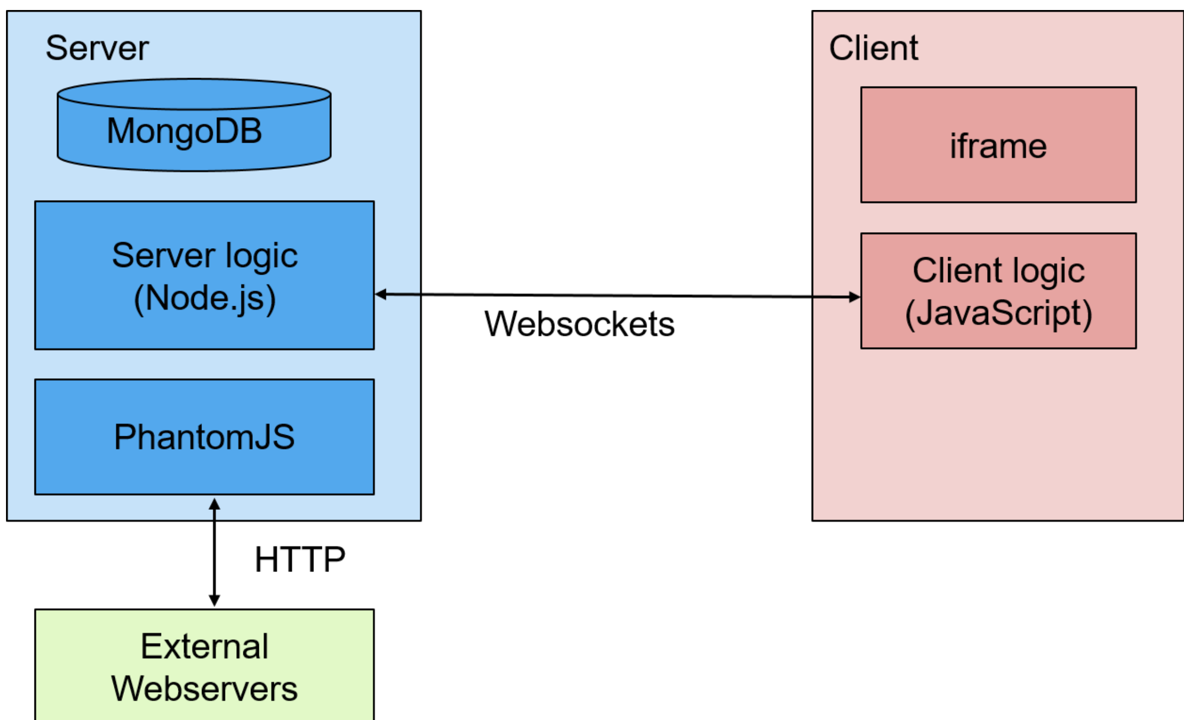


Figure 3.10: The MultiMasher implementation.

logic was implemented in JavaScript while the UI uses the MV* framework AngularJS[7] and interface components from Bootstrap[8]. In the MultiMasher client, each mashup component is loaded inside an iframe. The JavaScript of the original website is disabled in the client-side components to avoid that the state on the different devices can run out of sync. Instead all events, for example clicks, are captured by the MultiMasher client-side logic and replayed on the proxy browser.

3.4 Evaluation

We have carried out two evaluations of MultiMasher. First, we present the results of a conceptual evaluation along the dimensions of an established framework, then we discuss a technical evaluation that analysed the comparability with 50 top websites.

3.4.1 Conceptual Evaluation

We have evaluated MultiMasher along the dimensions of a logical framework for multi-device user interfaces [150]. The framework describes design dimensions that are relevant in cross-device frameworks and applications. However, it does not take into account dimensions relating to mashups. To account for the mashup aspect in our work, we have added the following dimensions.

Inter-Component Communication describes how distributed UI elements can be set up to communicate with each other. This might be *implicit*, if no configuration

[7] `https://angular.io/` Accessed on 30.08.2017
[8] `http://getbootstrap.com/` Accessed on 13.03

by the user is required; *explicit*, if the connection has to be manually established; and *mixed* if both cases are possible. MultiMasher provides a mixed approach. Implicit communication is only possible for components that originate from the same source, otherwise the communication has to be set up explicitly via tagging.

Synchronisation Consistency defines how complex it is to maintain synchronisation across devices and websites. In a *consistent* system, resynchronisation is easy to achieve; while in an *inconsistent* system, resynchronisation is hard to achieve. An example of similar observations may be found in [109]. As MultiMasher stores the state of a mashup centrally on the server, a device that is out of sync simply needs to reconnect to the server in order to synchronise.

Type of Components analyses the type of components used in the distribution. These can be *widgets*, which are small, pre-built, self-contained web applications. Another option is to support arbitrary *HTML elements*, which is the case in MultiMasher.

Component Creation defines the mechanisms that can be supported for creating components. Components may be *pre-built*, i.e. widgets; *scripted* at design-time, i.e. by developers working on the cross-device-mashup; or generated by *direct-manipulation* of websites by the user as in MultiMasher.

Flexibility to Changes analyses flexibility to changes in the source websites, e.g. because of evolving website structure of the re-authored pages. It ranges from *high* if components can adapt well to the new source to *low* if no adaptation is provided. In MultiMasher, an evolving website can interrupt inter-component communication. If the DOM elements for a component can no longer be found, it only affects that component. The rest of the mashup remains stable and the user may adjust the component to any changes in the source website.

Table 3.2 provides a summary of our analysis of MultiMasher with respect to the existing and additional framework dimensions. We justify the values chosen in the existing dimensions as follows. Distribution is *dynamic* as it can be updated at run-time. Migration is supported at the level of *UI elements*, as mashup components can be moved between devices while preserving state. The granularity that is supported in MultiMasher ranges from the *entire UI* down to *components of UI elements*. The smallest granularity that can be chosen is the HTML element, however, since components can contain nested elements the entire UI could be chosen by selecting the `body` element. The trigger dimension describes who can cause changes to a cross-device interface. In MultiMasher the system does not automatically redistribute components across devices. Instead all changes have to be made by the *user*. In terms of sharing devices, MultiMasher provides support for *moving information*. Any user can move components to a shared device. *Sharing by interacting* is also possible, as is illustrated in the scenario where two users can control a shared TV with their mobile devices. MultiMasher supports *mixed* timing. A mashup can be migrated *immediately* from one device to another. As the mashup state is stored centrally on the server, it is not required that both devices are present and migration could also happen in a *deferred* manner when a second device joins after the first device has left the system for a while. The interaction

Multi-Device Dimensions	MultiMasher
Distribution	Dynamic
Migration	UI elements
Granularity	Entire UI to components of UI elements
Trigger	User
Sharing	Moving information, sharing by interacting
Timing	Mixed
Modalities	Multi-modal
Generation	Run-time
Adaptation	Resizing
Architecture	Client-Server
Mashup Dimensions	MultiMasher
Inter-Component Communication	Mixed
Synchronisation Consistency	High
Type of Comp.	HTML element
Flexibility	Medium
Creation	Direct manipulation

Table 3.2: Conceptual evaluation for multi-device UI dimensions (left) and mashup dimensions (right)

modalities that can be used with MultiMasher are dependent on the ones supported by the browser and implemented by the underlying websites. As some browser can be used with touch, mouse, or speech, we rate this dimension as *multi-modal*. MultiMasher does not differentiate between design- and run-time, so all UI generation happens at *run-time*. The most essential support for UI adaptation is provided in that components can be moved and *resized* to accommodate different device characteristics. As outline above, the system has been built with a *client-server* architecture.

3.4.2 Technical Evaluation

The technical evaluation used a methodology similar to [139] and had the goal to analyse compatibility with existing websites. MultiMasher was used with 50 top websites, ranked according to popularity and traffic by Alexa[9]. We selected the first 5 websites from 10 categories: Arts, Business, Games, Health, Home, News, Science, Shopping, Society, and Sports. The evaluation was conducted in February/March 2014.

Setup

To asses MultiMasher with these top websites, we developed a simple but representative cross-device scenario (Fig. 3.11). The scenario was chosen because it can be constructed using almost any type of website. Two devices are used in the scenario, each accessing a different mashup constructed from two websites. The two websites are separated into components, which are distributed across the devices and tested. We have identified

[9] `http://www.alexa.com/topsites` Accessed on 15.05.2017

Browsing	Page elements are loaded and displayed correctly. Page elements behave as expected. User interactions are handled correctly.
Distribution	Page elements can be extracted and distributed as components. Distributed elements are displayed correctly. Distributed elements behave as expected. Distributed elements are synchronised with the underlying web page.
Mashing up	Elements of a web page can be linked with elements belonging to another page. Click, change, and submit events are correctly replayed in linked elements.

Table 3.3: Criteria of the technical evaluation.

three types of components which can be found on most websites: menu, search bar, and main content. Only one of the two websites is changed at each iteration of the scenario, while the other is fixed. The changing website is the one under test. The purpose of the fixed website is exclusively to test the mashup capabilities – in particular the inter-component communication across devices – and it should not impact the evaluation. We selected Wikipedia for this purpose, as it had proved to work very well in MultiMasher.

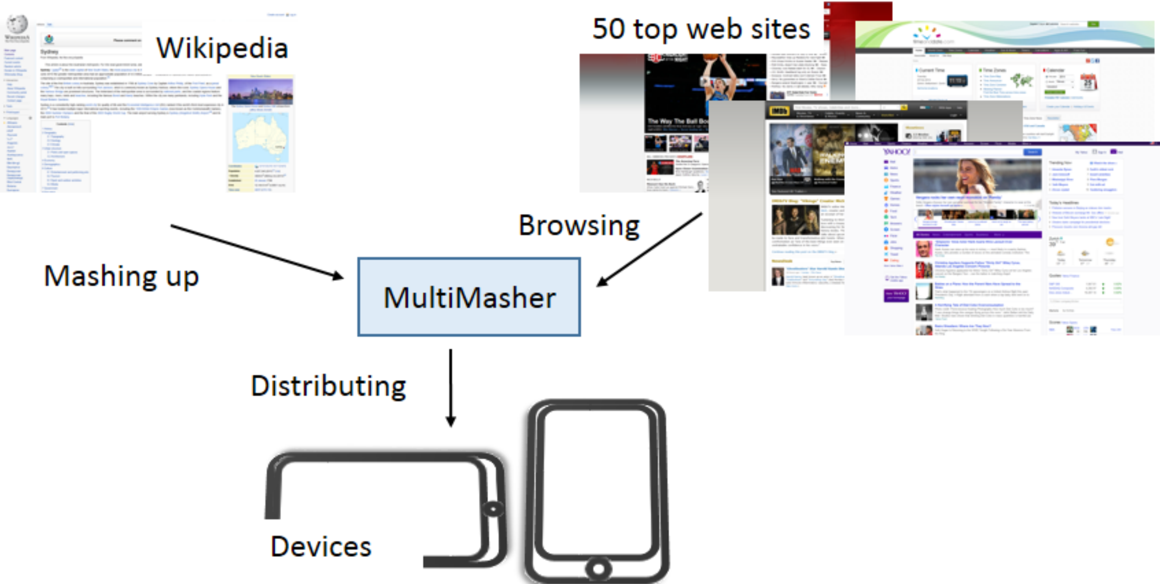


Figure 3.11: The evaluation scenario.

We have defined criteria in three categories to assess MultiMasher (Table 3.3). *Browsing* criteria assess whether a website can be loaded in MultiMasher, displaying and behaving correctly. *Distribution* criteria refer to component extraction. Web page elements should display and behave correctly when distributed as components and remain synchronised with the underlying website (implicit inter-component communication). Finally, the *mashing up* criteria are used to check if components can be linked with elements from another website (explicit inter-component communication) and if events are propagated correctly. For each criterion, a value between 1 and 5 was

assigned using the following scale:

1. very poor, major issues (e.g. page will not load)
2. poor, only specific elements are working
3. fair, most critical elements are working
4. good, only specific elements are not working
5. very good, full support

The evaluation and assignment of values was carried out by Stefano Pongelli as part of his Master thesis project [179].

Results

Overall, MultiMasher demonstrated good compatibility with many websites from different domains. 43 of the 50 tested websites (86%) received an average rating of 4 or higher when aggregating over all criteria. However, the browsing criteria take precedence over the other categories and can be understood to be an upper bound on the compatibility. A low score in terms of browsing implies a low compatibility with MultiMasher overall, despite possibly higher scores in distributing and mashing up. Considering the browsing criteria in isolation, the number of websites that scored an average of 4 or higher drops to 31 (60%). This score implies that they had either no issues or only small issues affecting non-critical elements, for example ads in separate iframes. Figure 3.12 provides a summary of the browsing results. 48 sites (96%) scored a 3 or higher, which implies that the majority of critical elements were working as expected.

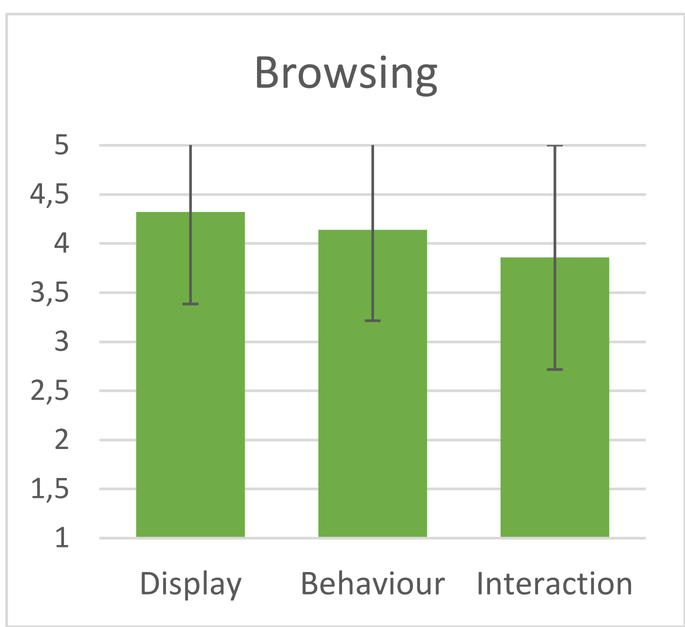


Figure 3.12: The results for the browsing criteria.

The compatibility with the distributing and mashing up criteria was high. For the distribution criteria, 48 sites (96%) scored a 4 or higher and 17 sites were rated a 5

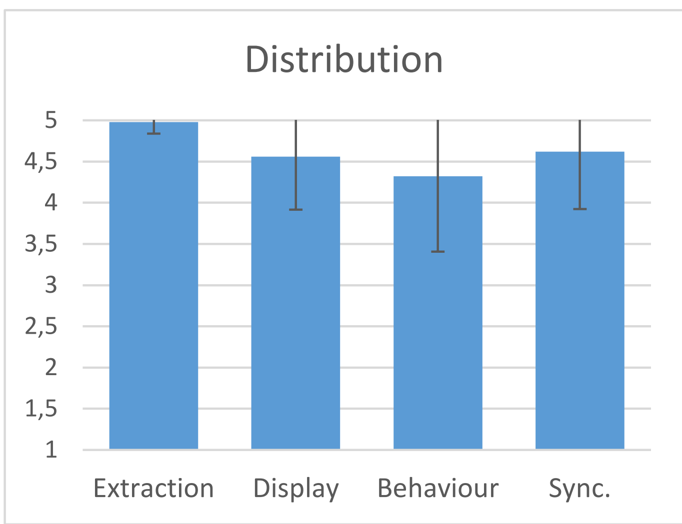


Figure 3.13: The results for the distributing criteria.

(Fig. 3.13). Similarly, for the mashing up criteria, 44 sites (88%) reached at least a 4 and 39 sites (78%) obtained the maximum rating of a 5 (Fig. 3.14).

Most issues we observed are related to the following challenges.

- *Heavy use of JavaScript.* To prevent duplicate updates and clients running out of sync, we have blocked JavaScript execution on client-side components. Instead, JavaScript is executed in the proxy browser and the results are sent to the clients. In most cases, this did not affect the normal behaviour of a website. However, it does limit cases where dynamic behaviour is essential, for example for drag-and-drop interactions or panning a map. Such operations trigger mouse move events, which are not synchronised to the server for performance reasons. A single interaction may produce hundreds of events. It would be possible to throttle the events and only execute a reduced number on the server, however, the usability of such interaction techniques heavily depends on short reaction times. The latency introduced by a round-trip to the server would be problematic.

- *CSS extraction.* We encountered some instances where the extracted element was not positioned correctly inside the component or where the page background was shown out of place. We manipulate the CSS of the extracted element so that it is positioned at the top left corner of its component. In some cases this manipulation conflict with existing CSS rules. Currently, efforts are made to introduce a component standard to the web[10]. While web components are not yet widely spread, their goal of encapsulation of functionality and style could mitigate the problem.

- *DOM evolution.* MultiMasher identifies the HTML elements making up a component with a CSS path. Some websites change their DOM continuously to prevent spam, for example by randomly changing IDs of form elements. This behaviour interferes with our path-based approach and causes elements to be missed or wrong elements to be matched. DOM re-matching techniques, such as the ones introduced in PageTailor [11], could be used to alleviate this issue.

[10]`https://www.w3.org/standards/techs/components` Accessed on 17.03.2017

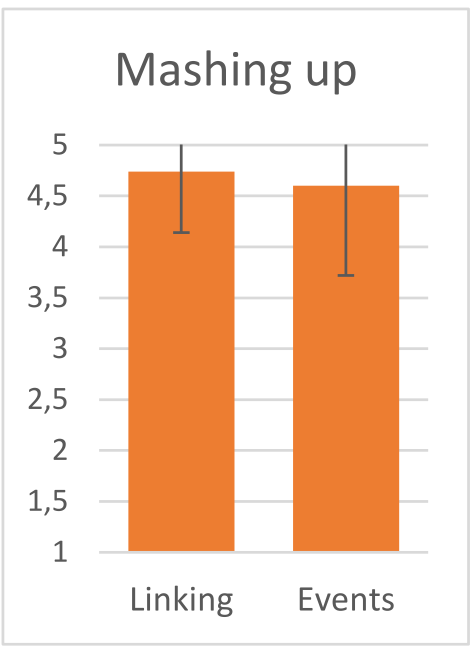


Figure 3.14: The results for the mashing up criteria.

- *Nested iframes.* Certain websites load content inside iframes. We have no control over this nested content due to the same origin policy[11] and cannot forward events from the client to the proxy in the server. Furthermore, the iframes can be used to inject JavaScript to the parent website causing the client-side copy to behave differently from the one in the proxy browser.

3.5 Discussion

With MultiMasher, we have presented a platform for experimentation with cross-device applications. Cross-device prototypes can be built with the visual tools and little technical knowledge is required. The reuse of existing applications enables a fast process and the direct drag-and-drop manipulations allow the user to quickly adapt to new devices joining a mashup. Since the produced prototypes are functional, they can be evaluated with potential users and even changed on the fly during an evaluation to take into account a user's feedback.

While we have not formally evaluated MultiMasher with end-users, it was demonstrated to a wide range of visitors and in classes. Up to 30 students have joined the same cross-device mashup with their own devices and we did not observe any issues. In particular, a Sudoku game that was turned into a multi-user application with only a few clicks proved to be very popular among students.

While the produced cross-device mashups can be used on any device with a browser, the visual tools for creating and manipulating them have been optimised for mouse interaction and somewhat larger screens. Mashups are thus best created at a laptop or a PC. However, we do expect most designers to have access to such equipment. Disabling JavaScript in the components on the client-side can hurt user experience to some extent. Dynamic behaviour that relies on JavaScript such as certain animations

[11]`https://www.w3.org/Security/wiki/Same_Origin_Policy` Accessed on 17.03.2017

or drag-and-drop interaction is not supported. However, we consider this limitation acceptable for the prototyping stage.

4 XD-MVC: Implementation

This chapter[1] presents a web-based library for implementing cross-device applications. While the web platform offers primitives that enable cross-device applications, working with the platform directly requires every cross-device developer to tackle issues such as UI distribution, state synchronisation, or device pairing individually and manually. On the other hand, existing cross-device frameworks often follow an all-in approach and restrict the choice of technologies that can be used, for example in terms of UI frameworks. With the XD-MVC library, we investigate how we can provide abstractions for common cross-device development tasks while allowing the developer to work with familiar tools and technologies. We propose a layered architecture that gives developers a choice in the level of support they need, with the goal of achieving a low threshold and a high ceiling [133].

We first introduce the structure and concepts of XD-MVC and explain how they support cross-device development. We then discuss the architecture and implementation of the library with particular focus on the communication architecture. XD-MVC can support both client-server and peer-to-peer communication to enable low latency in browsers that support peer-to-peer communication and a client-server fallback solution for those that do not. To demonstrate the usefulness of XD-MVC, we describe a set of sample applications that have been implemented using the library. The library has been made available to the public on Github[2].

4.1 Concepts

XD-MVC provides support for the development of cross-device applications at different levels of abstraction. We first motivate and explain the layered structure of XD-MVC. Next, we introduce the concepts that the lowest layer provides. Then we explain how

[1] Earlier versions of parts of this chapter were originally published as Husmann et al. [73, 77, 76].
[2] `https://github.com/mhusm/XD-MVC`

the library can be used with the MV* library Polymer[3] and introduce the higher level components that we provide with Polymer.

4.1.1 Layered Architecture

Given the large number of front-end frameworks that are available and used in web development we wanted to avoid locking developers into a single framework. At the same time, only offering a general purpose solution would still leave many tasks to the developer. We thus explored a layered approach (Fig. 4.1).

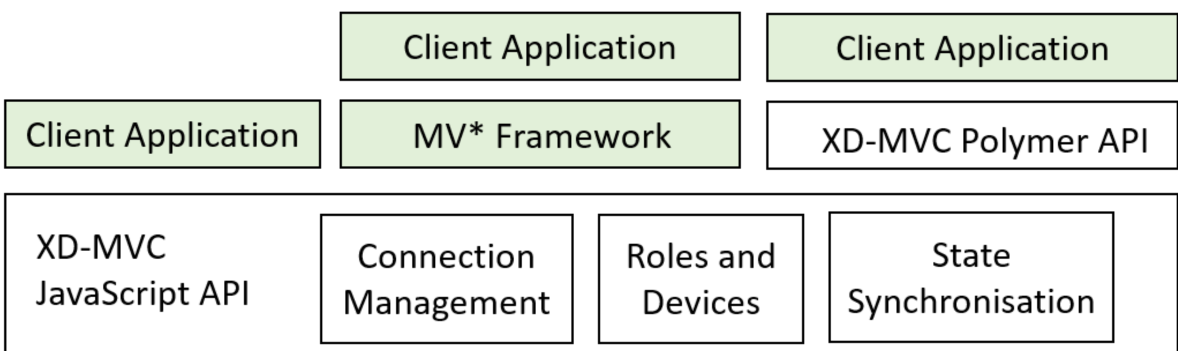


Figure 4.1: The layers of XD-MVC.

The lowest layer is implemented in pure JavaScript for a maximum of compatibility with front-end frameworks. This layer facilitates connection management, distributes information about devices and their roles in the system, and provides state synchronisation with an event-based API. A developer can choose to use the library as they wish, either by building their application directly on top of the library or with the help of an MV* framework. If an MV* framework is used, some integration may be required to map library concepts to framework concepts. As a proof-of-concept, we provide an integration with the Polymer framework. Our Polymer integration extends beyond simply providing the JavaScript API in Polymer. It includes a set of Polymer components as building blocks that handle common cross-device tasks, including preconfigured UI distributions and visual components. While these components give less freedom to the developer in comparison to the lower level API, they require less manual configuration and scripting.

4.1.2 JavaScript API

The JavaScript layer provides an event-based API for connection management, information about devices and roles, and state synchronisation. Note that XD-MVC is based on a thick-client architecture. All code examples in this section are executed on a client device. The server-side is discussed in the architecture and implementation section. An early version of the JavaScript API was developed in [182].

[3] `https://www.polymer-project.org/` Accessed on 23.03.2017

Connection Management

In a cross-device application, devices that participate together in an application need to be associated somehow. XD-MVC provides an API so that devices can be paired and unpaired in an ad-hoc manner. As applications built with XD-MVC are web-based, an application is started by loading a URL in the browser. The XD-MVC library will register the device with a server upon initialisation. The server assigns a unique ID to each device. The ID can be made persistent so that it is kept by a device across reloading of the application or it can be set to change any time a new connection is made to the server for privacy reasons. Device A can be paired to device B by specifying B's ID in a pairing request. B will receive a pairing event so that it is aware of the connection and A will be notified of the success or failure of the request. When a third device C is to be added to the group, it is sufficient to connect C to either A or B. The library ensures that all devices within a group are connected to each other by building up a complete graph of connections. Listing 4.1 illustrates the API for connecting devices.

```
1 XDmvc.connectTo('someDeviceId');
2
3 XDmvc.on('XDconnection', function(device){
4    console.log('now connected to ' +device.id).
5 });
```

Listing 4.1: Connecting to a device by its ID and receiving an event for an incoming connection.

Devices and Roles

XD-MVC shares information about devices connected to each other in a group with each device. This allows the developer to distribute and update the UI accordingly. For example, when a TV and a phone are paired, the developer could display the controller of a video player on the phone and the video itself on the TV. On the other hand, if they only detect a single device, both components could be displayed on the same device. The event-based API informs the developer of devices joining and leaving. In addition, the developer can query the connected devices at any time. XD-MVC groups the devices into categories based on screen size (extra large to extra small), but also provides the exact dimension of the application window in pixels. An event is triggered when a device changes the size of the application window, for example when a browser is resized (Lst. 4.2).

```
1 XDmvc.on("XDdevice", , function(device){
2    console.log('device has changed ' +device.width +' ' +device.height);
3 });
4
5 if (XDmvc.othersDevices.small > 0) {
6   console.log('there is a small device connected to this device');
7 }
```

Listing 4.2: Listening for changes in the devices and querying devices by type.

XD-MVC integrates a concept of device roles. A developer can define arbitrary roles and associate them with the devices. While other frameworks associate roles with users, in our analysis of related work on device use we found that associating roles with devices better fits the mental model of users. However, a developer is free to map a user role to a device role. Essentially, a role in our model is simply a piece of information associated with a device. It is up to the developer to define the semantics of a role. A device can have multiple roles and a role can be taken by multiple devices. The JavaScript API provides functions to add and remove roles from a device (Lst. 4.3). Similar to the device information described above, XD-MVC allows the developer to register for events in role changes and query the roles at any time. The library informs the developer about role changes on any connected device. The video player example could also be realised with a *controller* and a *player* role that are dynamically assigned to the devices.

The developer determines how the roles are allocated to a given device. For example, they could do it based on device information. Alternatively, the assignment can be passed on to the user through the UI, enabling end-user customisation of the distribution. Any device with the *controller* role could then be configured to show the video controls whereas *player* devices would be set to show the video.

```
1  XDmvc.addRole('controller');
2  XDmvc.removeRole('viewer');
3
4  XDmvc.on('XDroles', function(device){
5     console.log('device ' +device.id +' has changed roles' +device.roles).
6  });
7
8  if (XDmvc.otherHasRole('controller')) {
9    console.log('At least one of the other devices has the controller
        role');
10 } else {
11   console.log('None of the other devices has the controller role');
12 }
```

Listing 4.3: Adding and remove roles to the current device. Listening for changes to roles on connected devices and querying roles.

State Synchronisation

In a cross-device application, multiple devices must be kept in sync. XD-MVC operates on the concept of a distributed MVC architecture [54]. Instead of synchronising a rendered UI across devices – as is the case for image based solutions (for example [12, 183]) – only the model is kept in sync. When the model changes, each device needs to re-render its view (Fig. 4.2). In contrast to the image-based approaches, smaller amounts of data need to be transmitted and dynamic behaviour, such as animations, can be handled locally. As a disadvantage, the MVC-based approach requires developer intervention whereas the image-based ones can handle legacy applications more easily.

XD-MVC allows the developer to specify models to be synchronised across devices. The specified models must be of type `Object` or `Array`. Models of simple types such

as `String` or `Number` can be wrapped in an `Object`. To register for synchronisation, the developer needs to specify the model object, an optional callback function that is triggered whenever another device manipulates the model, and a label that identifies the model (Lst. 4.4, line 2). The callback function (line 4) receives the model label, the new value, and the ID of the device that caused the change. It is the developer's task to initiate a change of the view.

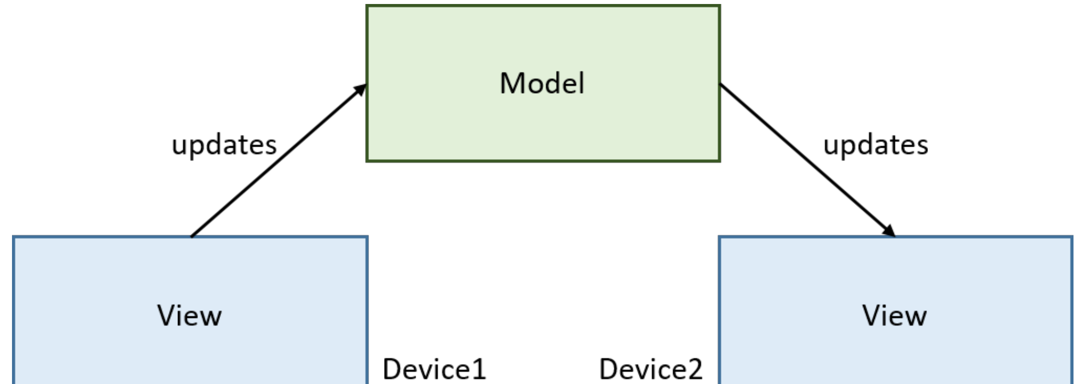


Figure 4.2: State synchronisation based on a shared model in XD-MVC.

```
1 var zoom = { level : 1};
2 XDmvc.synchronize(zoom, setZoom ,"zoom");
3
4 var setZoom = function(label, newZoom, deviceId){
5   console.log('The device ' +deviceId + 'has changed the model ' +label
      +'. The new value is ' +newZoom);
6 }
```

Listing 4.4: Synchronising a model object and listening for changes with a callback function.

4.1.3 Polymer Components

XD-MVC provides a set of web components built with the Polymer library. On the one hand, these serve as a demonstration of how XD-MVC can be integrated with an MV* framework. On the other hand, they provide additional functionality that is not available in the JavaScript API, including visual components. Polymer is a library for building web components with a declarative API. A component provides functionality and optionally a visual UI. Components built with Polymer can be integrated into any web application. At the same time, Polymer can be used as an MV* framework and it implements established concepts such as two-way data-binding. In line with this component-based philosophy, we have built a set of components for XD-MVC that can be used in isolation or combination.

State Synchronisation with Data-Binding

Many MV* frameworks offer a data-binding mechanism to map a model to a view. Polymer includes two-way data-binding: changes in the model update the view and

manipulations in the view (for example entering text into an input) can directly update the model. We hook into this mechanism for state synchronisation with the `xdmvc-synchronised` component. The component has a declarative API for binding an object (Lst. 4.5). The bound object needs to be a map of `String` labels to values of type `Object` or `Array` to enable a mapping to the JavaScript API described above (Lst. 4.6). The synchronised objects can then be bound to any other component. Changes in the state will be propagated to all connected devices. The two-way data-binding ensures that the view updates to reflect changes received from another device.

```
1 <xdmvc-synchronised objects="{{synced}}"></xdmvc-synchronised>
2 <video-controller state="{{synced.video.state}}">
3 </video-controller>
4 <video-player state="{{synced.video.state}}"
     title="{{synced.video.title}}">
5 </video-player>
```

Listing 4.5: Synchronising a model object with Polymer and two-way data-binding.

```
1  properties: {
2      synced: {
3          type: Object,
4          value: function(){ return
5              {"video": {"state": "play",
6                         "title": "Cape Epic Summary"},
7               "playlist": [1,5,7],
8              }
9          },
10         notify: true
11     }
12 }
```

Listing 4.6: Declaring a synchronisation map containing an object (video) and an array (playlist) in Polymer.

Declarative UI Distribution with Roles and Devices

XD-MVC provides declarative access to the device and roles API with the `xdmvc-roles` and `xdmvc-devices` components. Combined with Polymer's conditional templates, these elements allow the developer to specify a UI distribution in a declarative manner. The `dom-if` templates in Polymer display or hide content based on a Boolean condition. Listing 4.7 illustrates an example. To enable declarative access to roles the corresponding XD-MVC component is added. Using data-binding, the developer can query if a given role is currently selected. In the example, the video controller is only displayed if the *controller* role is selected and similarly the video player requires the *player* role. The same could be implemented by querying the device size instead of using roles.

```
<xdmvc-roles roles="{{roles}}"></xdmvc-roles>

<template is="dom-if" if="{{roles.isselected.controller}}">
   <video-controller state="{{synced.video.state}}">
   </video-controller>
</template>
<template is="dom-if" if="{{roles.isselected.player}}">
   <video-player state="{{synced.video.state}}">
   </video-player>
</template>
```

Listing 4.7: Declarative specification of a UI distribution using XD-MVC and Polymer's conditional templates.

Distributed Layout Templates

While the information about devices and their roles enables the developer to distribute the UI in many ways, it does require some effort on the part of the developer. In contrast, systems that distribute the UI automatically lower that effort at the price of control. We have opted for a middle ground by providing a small template library for distribution patterns. Related work has identified patterns in end-user customised distributions [138, 137]. The patterns reduce the amount of code that needs to be written to achieve a distributed UI. As a proof of concept, we have implemented the following three templates:

- The *controller* template distributes content between controller and viewer devices and implements the most popular *remote-control* pattern observed in [138]. Listing 4.8 illustrates the pattern in a code example and Fig. 4.3 shows the resulting application loaded onto 3 devices.

- The *pages* template distributes content paginated content across devices. The content can be navigated in a synchronised manner. Adding additional devices allows more content to be viewed simultaneously. Listing 4.9 illustrates the pattern in a code example and Fig. 4.4 shows the resulting application loaded onto 3 devices.

- The *lazy pages* template is similar to the *pages* template, however content is loaded lazily. It is thus better suited to situations where the list of pages is very long.

Note that the *mirror* pattern from [138] is achieved by default if the appropriate state synchronisation is set up. No further configuration is needed. Templates can be nested and combined. For example, a *pages* template could be nested inside the viewer part of the *controller* template.

Pairing

Device association is a recurring task in cross-device development. A whole range of interaction techniques and technologies have been explored in related work. We have

```
1  <controller-layout>
2      <div class="controller">
3        <paper-button on-click="prev" raised>Previous</paper-button>
4        <paper-button on-click="next" raised>Next</paper-button>
5      </div>
6      <div class="viewer">
7          <img on-tap="next" src="{{getSource(synced.gallery.selected,
              albums)}}"/>
8      </div>
9  </controller-layout>
```

Listing 4.8: Using the *controller* template. Content wrapped in the `controller-layout` tag is automatically distributed across devices. The content must be structered into a *controller* and *viewer* section using the `class` attribute.

```
1  <page-layout current="{{current}}" selected="{{selected}}">
2      <div>page 0</div>
3      <div>page 1</div>
4      <div>page 2</div>
5      <div>page 3</div>
6      <div>page 4</div>
7      <div>page 5</div>
8  </page-layout>
9  <paper-button on-click="prev" raised>Previous</paper-button>
10 <paper-button on-click="next" raised>Next</paper-button>
11 <div>Global current page: <span>{{current}}</span></div>
12 <div>Selected page for this device: <span>{{selected}}</span></div>
```

Listing 4.9: Using the *pages* template. Content wrapped in the `page-layout` tag is automatically distributed across devices. The other content is replicated.

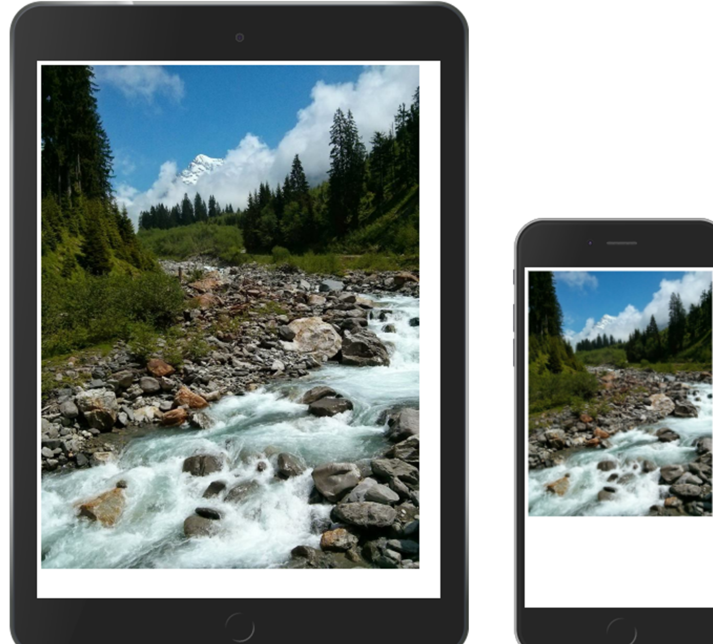


Figure 4.3: The controller example from Lst. 4.8 when loaded onto 3 devices. The first device has the controller role and shows buttons to navigate a gallery. The other two devices are viewers and display the selected image

investigated two approaches in XD-MVC. The first is based on URL sharing which is an established technique in web-based cross-device frameworks. The latter is more experimental and explores the space of social and physical proximity. We will discuss each in turn.

URLs and QR codes XD-MVC provides a component for connecting devices based on a shared URL. When an application is first loaded, it modifies its URL to include the device ID. That URL can be shared to another device, for example with a messenger application or NFC. When the second device opens the URL, it will automatically be paired to the first device. The component can be configured to assign roles to the devices based on their role in the pairing process (Lst. 4.10). In cases where copying and sharing a URL via a service is not easily possible, a URL can easily be represented as a QR code. Thus, the user can scan the code using a camera-enabled device instead of having to type complicated and long URLs into the browser. XD-MVC provides a component that generates such a QR code. Furthermore, it includes a visual component consisting of the QR code and a list of all connected devices to provide feedback on the current state to the end-user (Fig. 4.5).

```
1 <xdmvc-url-pairing connector="controller" connectee="viewer"/>
```

Listing 4.10: Pairing with shared URLs. If device A loads the URL of device B, device A will be assigned the *controller* role while device B will be a *viewer*.

Social and Physical Proximity While a number of interaction techniques have explored device association based on physical proximity, we propose that social proximity

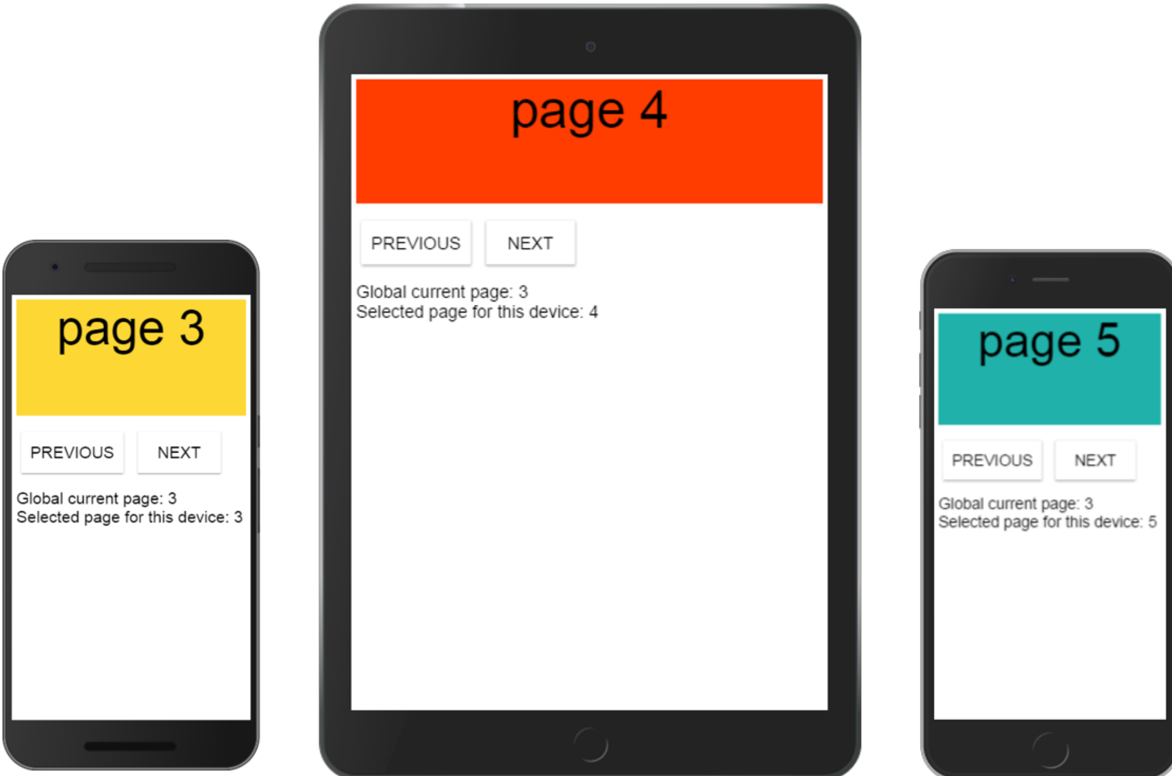


Figure 4.4: The pages example from Lst. 4.9 when loaded onto 3 devices. Each device shows a page, the other controls are replicated on each device. Stepping through the pages is synchronised across devices.

should also be taken into account. Physical proximity does not necessarily imply social proximity, as we illustrate with four scenarios in Fig. 4.6. Existing techniques enable the pairing of devices that are physically close. However if physical proximity is the only criterion, they can present a security risk, as is illustrated by the case of a woman who received unwanted pictures from a stranger on a train [8]. While some systems do allow the user to restrict access to people on the contact list, there are cases where sharing with physically close strangers is wanted and the user has to remember to switch the settings to the appropriate mode. We introduce three pairing modes that balance user control and the amount of user intervention (Fig. 4.7).

- *Auto-Pairing* automatically pairs two devices and requires no user interaction.
- *Req-Pairing* requires a pairing request from the initiating device. No interaction is required from the receiving device.
- *Ack-Pairing* requires a pairing request from the initiating device and the receiver has to accept the request in order to establish the connection.

With *auto-pairing* the user has no control over the pairing but also does not need to interact. At the other end of the spectrum, there is *ack-pairing* where on both the sending and receiving device, a user interaction is required. We propose that the appropriate mode could be chosen automatically depending on both the physical and social proximity of the devices and their owner. For example, two physically close

Figure 4.5: A visual component for connecting devices via QR code or URL. The button at the top shows the number of current connections and toggles the QR code and the connection information. At the bottom, currently connected devices are shown together with information on device type.

devices belonging to the same user could be automatically paired. To avoid that, for example, a user's smartphone at work is paired with the TV at home, *req-pairing* could be used when the distances between the devices passes a threshold. Consequently, the devices would only be paired when the user explicitly requests it. Finally, for people on the user's contact list *ack-pairing* could be used to ensure that both parties agree with the pairing. To pair devices of two strangers, existing methods based on shared information such as a QR code or a PIN code could be used. We have implemented a prototype of this pairing model as an extension to XD-MVC. The extension is available as set of Polymer components including visual elements (Fig. 4.8). The concept of pairing based on social and physical proximity was developed in [30].

4.2 Architecture and Implementation

In this section, we discuss interesting aspects of the architecture and implementation of the XD-MVC library. We start by analysing different communication architectures between devices and motivate our choice for a hybrid architecture. We then describe the overall architecture of the library and discuss its implementation.

4.2.1 Device-to-Device Communication

Communication between devices is essential in cross-device applications. It is a requirement for state synchronisation. As we have discussed in Chapter 2, client-server

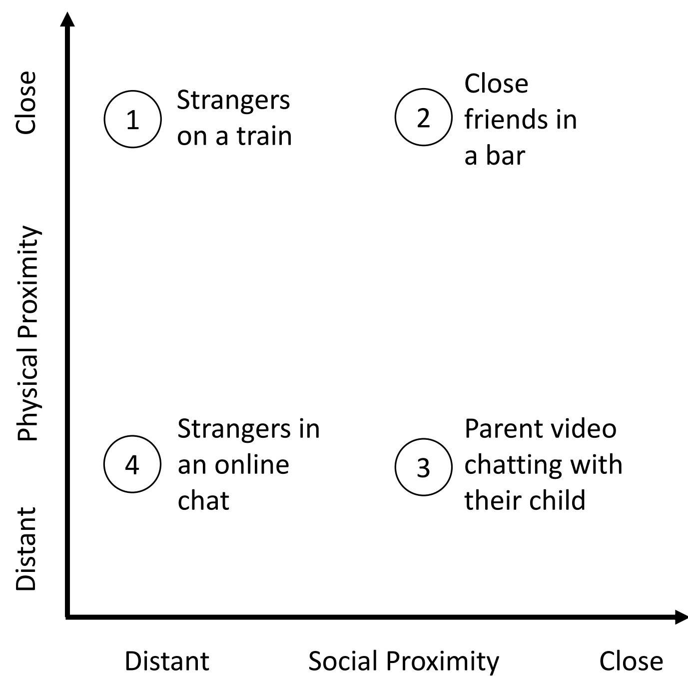


Figure 4.6: Scenarios of social and physical proximity.

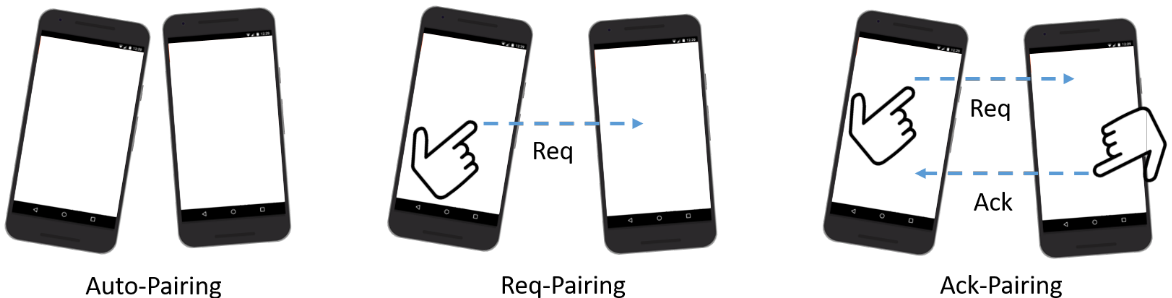


Figure 4.7: The three pairing modes with varying user control.

architectures have been predominant in cross-device systems. In client-server architectures, communication between clients is mediated by the server. However, peer-to-peer architectures have been proposed as a better fit by some [45]. For systems built with pure web technologies and running in unmodified browsers, peer-to-peer communication has become an option only recently with the introduction of the WebRTC standard. However, the standard has not yet been implemented by all major browser. Most notably, Safari is missing support as of Summer 2017, but has committed to the integration of the standard later in the year.

Moving from a client-server to a peer-to-peer architecture can drastically reduce latency. This is the case, in particular, when the server is remote and the clients are co-located. Incidentally, that is a common setup for web-based cross-device applications. The application is typically served by some web server which could be anywhere in the world. The devices used together in a cross-device application are usually in the same room, even though it could be possible to integrate remote devices. We have measured round-trip times between two clients communicating over a server. The clients were

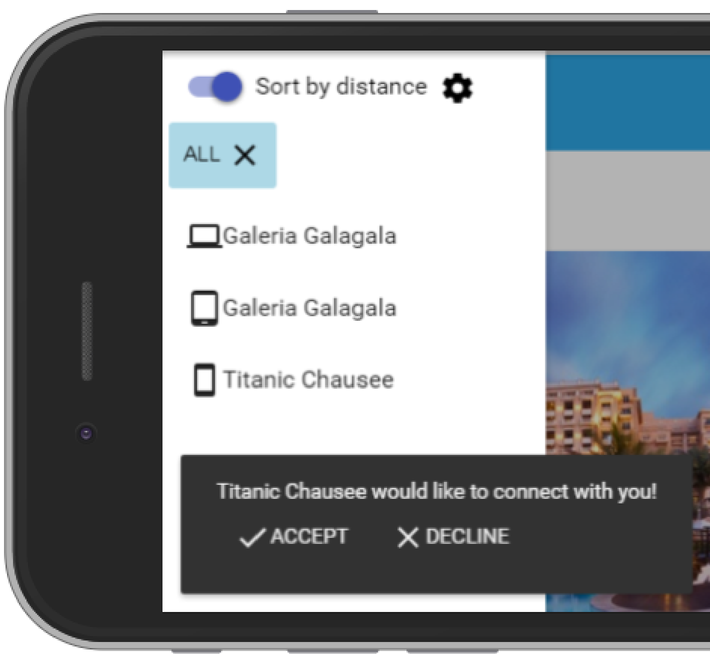


Figure 4.8: A visual component for pairing devices based on the user's contact list. Devices can be sorted by distance to show physically close ones first. Selecting a device sends a pairing request. Since the *ack-pairing* mode is used, each request needs to be confirmed.

in Zurich, Switzerland and the server was in New York, USA. We measured the time for message from device A to device B and back to A. Resulting times were at around 200ms on average. Direct peer-to-peer communication over WiFi was a whole order of magnitude faster, measured at 20ms. Humans perceive reactions by a system that occur within less than 100ms as instantaneous [19]. When we halve the round-trip time to obtain an estimate for the time required to send a single message, we already reach that limit with the client-server architecture. Furthermore, the measured time does not include any updating of the UI which would add more time. Consequently, cross-device applications built with client-server architectures may struggle to meet optimal response times for human perception. This could, in particular, be problematic in applications with continuous interaction, for example panning maps that are distributed over multiple devices.

While the improved latency is a clear argument in favour of a peer-to-peer architecture, the lack of support for WebRTC in Safari would exclude a large set of devices (all iPads and iPhones). To address this problem, we propose a hybrid communication architecture: Where available, device communicate directly over WebRTC. Devices with no support fall back to communication mitigated by a server (Fig. 4.9). The performance measures were collected in [43] where this hybrid architecture was developed as well.

4.2.2 System Architecture

XD-MVC is split into a client- and a server-side part. Even when only peer-to-peer communication is used, a server is still needed to establish the communication between the devices. Once the devices have been paired, the server could in theory be discon-

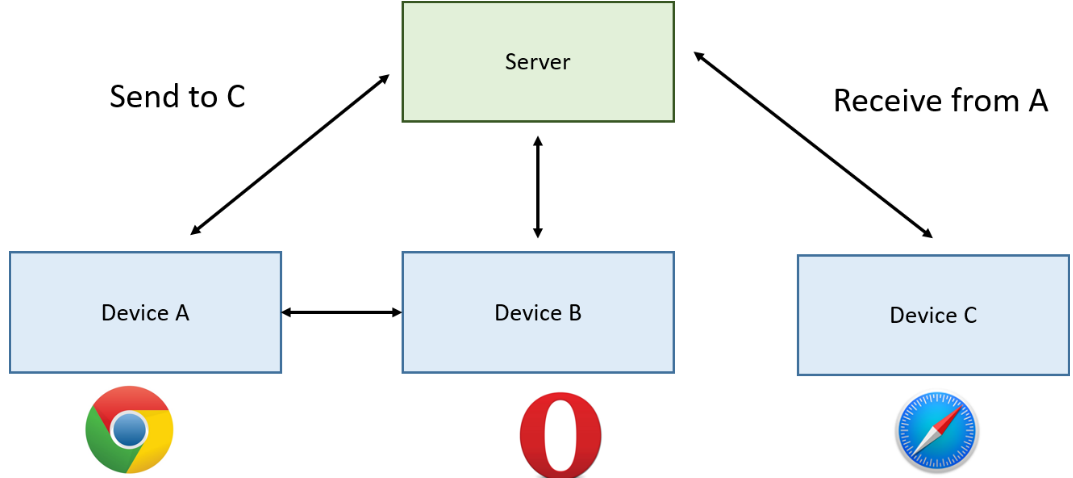


Figure 4.9: Devices with WebRTC support can communicate directly (A and B). Devices without support can use the fallback via the server (C).

nected, however, no new connections could then be established. Figure 4.10 provides an overview of a generic application built with XD-MVC. As with any web application, there is the need for a web server that serves the client HTML and other resources. The client integrates the XD-MVC library. The library communicates with the XD-MVC server component. The server component can either be integrated into the web server, if it is written in Node.js, or executed as a stand-alone server. The XD-MVC server maintains a list of all devices that are connected to it. For devices that support WebRTC, it establishes the peer-to-peer connection upon a pairing request. In addition, a WebSocket connection is maintained to each device for the fallback communication for devices without WebRTC support.

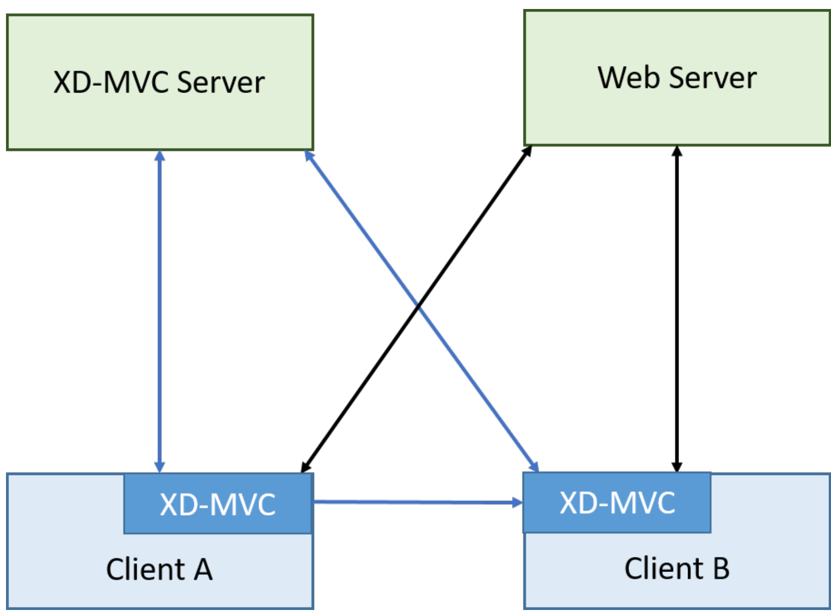


Figure 4.10: The architecture of an XD-MVC application.

4.2.3 Implementation

The XD-MVC server is implemented in JavaScript as a Node.JS module. For the WebSocket communication, it uses Socket.io. The WebRTC communication was implemented with PeerJS[4]. The server has the responsibility of enabling and coordinating communication among devices. Most of the logic is implemented in the client-side of the library.

To enable the envisioned flexible use of the framework, the client is implemented in a number of layers and modules. The lowest layers are all implemented in JavaScript. The first layer implements device-to-device communication. Using this layer, a developer can enable direct communication between devices. The system automatically routes the communication via the server if needed. This process is transparent to the developer. The next layer adds the concept of roles and devices and provides state synchronisation. To minimise the amount of data that needs to be transferred, only a delta that represents the change is transmitted. For example, when a new item is added to an array, only that item is transmitted along with its position in the array rather than a snapshot of the whole array. The library computes the delta on the sender side and applies it on the receiver side.

On top of this JavaScript layer, there are the Polymer components that provide the same functionality with a declarative API. The Polymer layer was initially implemented with Polymer 0.5, but was migrated later to Polymer 1.0. Optional Polymer components provide functionality such as the pairing via URLs and QR codes as well as the layout templates.

4.3 Evaluation

We have evaluated the XD-MVC library by creating a number of demonstrator applications. These applications demonstrate the breadth of applications that can be built using the library. In this section, we provide an overview of a selection of applications developed with XD-MVC. The applications were built while the library was still evolving. Consequently, some were developed when not all concepts were implemented yet. For example, the layout patterns and the social pairing were added last, while state synchronisation was available from the beginning. As a result, not all applications use all features of the library. Table 4.1 provides an overview of all applications and the main features each uses. However, this varied use is in line with the modular concept of the library and the applications serve a means to demonstrate the flexibility of XD-MVC. At the same time, the development of these demonstrator applications inspired concepts that could be added to library and unearthed issues in other phases of the workflow, in particular the insufficient support in testing and debugging. A number of people were involved in building the applications. While some were built by the author of this thesis and by students contributing to the library, others were built by students not involved in the development of the library.

[4] `https://github.com/peers/peerjs` Accessed on 30.30.2017

Application	JavaScript API	Polymer Components	QR Pairing	Social Pairing	Layout Templates
XD-Bike	x	x			
Hotel Booking	x	x		x	x
Webcam Viewer		x	x		x
Maps	x				
Voting	x				

Table 4.1: A summary of the main XD-MVC features that each application uses.

4.3.1 XD-Bike

XD-Bike is an application for planning mountain bike trips. It was inspired by existing route repositories such as GPS-Tracks.com[5]. Existing repositories offer a large number of routes to choose from; GPS-Tracks.com has over 1300 for Switzerland alone. To enable an informed choice, each route is accompanied by details such as length, technical difficulty, and a map. Due to the large amount of data, the sites are best accessed on devices with larger screens. Even though some have started to implement a responsive design to better cater to mobile devices, the user experience clearly suffers on small screens. As cross-device applications can provide increased screen real-estate, we considered this an interesting scenario to explore. We had in mind that multiple users could pair their mobile devices while planning a route on the go, for example on the train. On the other hand, we also wanted to cater to scenarios where a larger device might be available, for example a TV in a hotel room.

These considerations have resulted in XD-Bike, a flexible cross-device mountain bike route repository. XD-Bike is built as a number of components that can either be viewed on a single device or distributed across multiple devices. The distribution is by default done automatically by the system based on the space requirements of each component and the size of the devices (Fig. 4.1). A map takes up more space than a route summary, for example. Large devices can show multiple components simultaneously while smaller one only show a single component. The user can however select a different component for each device if they wish so.

XD-Bike was implemented by three students during a semester as part of the Information Systems Lab where students earn 10 ECTS by working in teams on a project during one semester (14 weeks). The project uses XD-MVC and Polymer (version 0.5). Most of the time was spent on the general implementation and only a small portion was spent on implementing the cross-device support with XD-MVC. Only an index for the current route needed to be configured for state synchronisation. The automatic distribution of the UI was implemented based on the device information provided by the system. Pairing was implemented with URLs which were shared using NFC during demos with Android devices. The URL sharing component was not yet available in XD-MVC and the implementation was done manually. As we expected this to be a

[5] `http://www.gps-tracks.com/` Accessed on 31.03.2017

recurring developer task, we later added that component to XD-MVC. XD-Bike experimented with the role concept. The device initiating the cross-device interaction was given a controller role. Only the controller device can choose routes while visitor devices can only view routes.

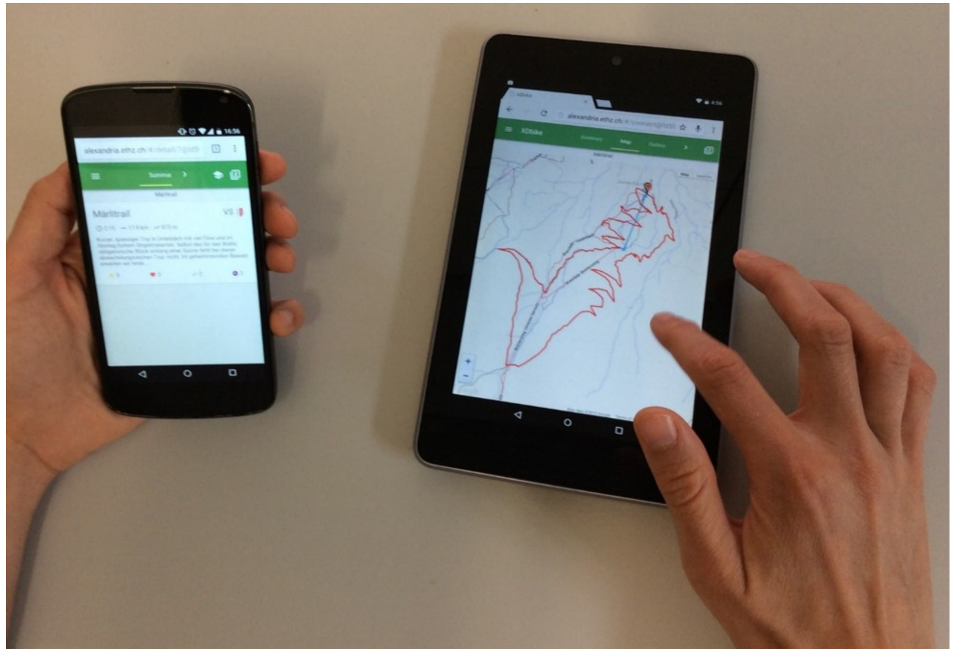


Figure 4.11: XD-Bike used on two devices. A map is shown on the larger device. The smaller device shows a summary of the selected bike route.

4.3.2 Hotel Booking

The cross-device hotel booking application was developed as a demonstrator for the concept of pairing based on physical social proximity. It was inspired by hotel booking applications such as Booking.com[6]. Similar to the bike route repositories, hotel booking sites present the user with a large amount of data: descriptions, maps, reviews and more. Our cross-device booking application distributes that information across available devices. One device lists available hotels. When a hotel is chosen, a map and a detail view are available that can each be viewed on separate devices (Fig. 4.12). Alternatively all components can be loaded onto a single device. However, we paid particular attention to users who are on the go and have access to mobile devices only. When a single user accesses the applications on two devices (for example a phone a tablet), the devices are paired automatically. To support scenarios where the user is travelling with friends or a partner, the devices can be paired based on this social connection. The application lists contacts from the address book, showing physically close ones first. This would allow friends in a cafe or on a train to quickly pair devices while avoiding pairing with strangers. The application was implemented by Sivaranjini Chithambaram as part of her Bachelor thesis project [30]. It uses XD-MVC with the Polymer integration, in particular it integrates the extension for pairing based on social and physical proximity. It uses a controller and a pages layout template. The list of results serves as a controller component where as the map and hotel details have been implemented as two pages inside the viewer part.

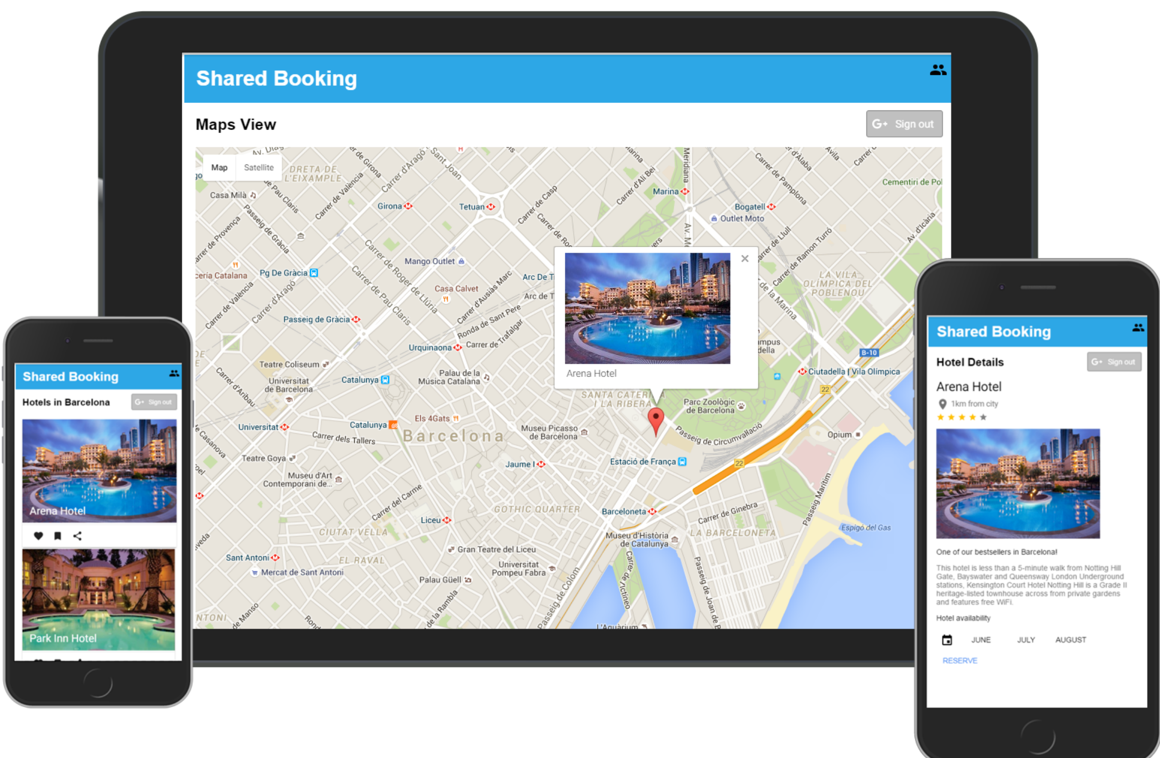


Figure 4.12: A cross-device hotel booking application.

[6] http://www.booking.com/ Accessed on 31.03.2017

4.3.3 Webcam Viewer

The webcam viewer application is composed of two main parts: a webcam viewer and a controller interface. The viewer displays a webcam image and is meant to be loaded onto large screens, for example a public display or a TV. The controller has two modes: in portrait orientation, the user can choose a webcam location from a list (Fig. 4.13), in landscape they can rotate the camera to see a different part of the view and they can navigate in time. The application was implemented with XD-MVC and the Polymer integration. It uses live images obtained from 360 high-resolution Roundshot cameras[7] in Switzerland. When an application is loaded by entering a URL in the browser, it automatically assumes the viewer role, making use of the role concept provided by the library. It displays a QR code for pairing, using another library component. Devices that scan the code are automatically assigned the controller role and will show the corresponding UI. The application was implemented in less than 300 lines of code, not counting the dependencies.

Figure 4.13: A cross-device webcam viewer that can be controlled with a phone.

[7]http://www.roundshot.com Accessed on 31.03.2017

4.3.4 Maps

We have experimented with maps applications as they are a common usage scenario in related work. We have created a maps application where devices can assume various roles. *Mirrored* devices have their centres synchronised. If the map is panned on one device, the other device updates accordingly. This mechanism allows the same region to be inspected with different views, for example street maps and satellite maps. *Overview* devices show the view ports of *mirrored* and *viewer* devices on an map (Fig. 4.14). *Viewer* devices are independent of other devices, but show up on the overview. The user can choose a role for each device in the UI.

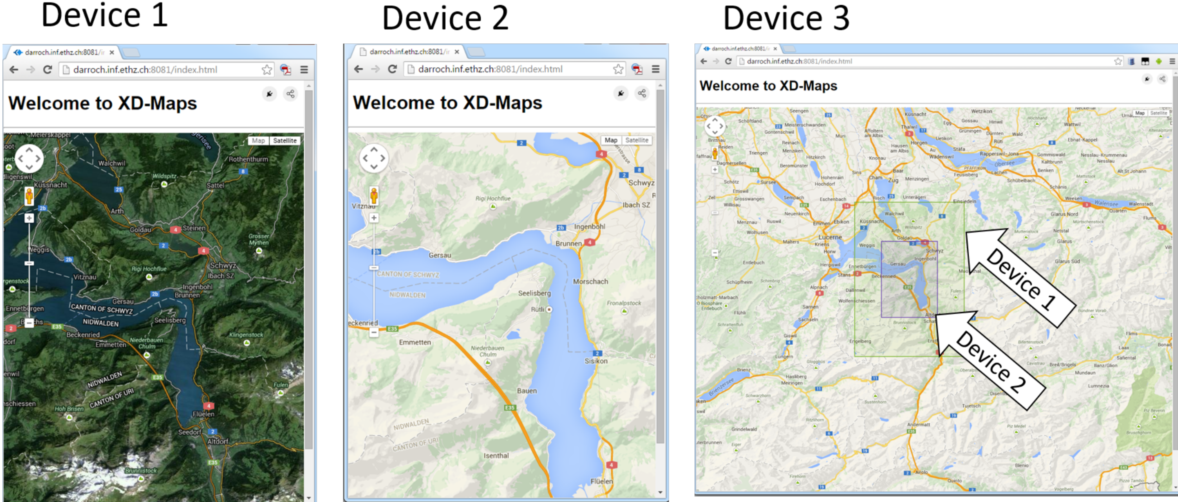


Figure 4.14: Device 1 and 2 are *mirrored* devices and share a synchronised centre, but have independent zoom and layers. Device 3 has the *overview* role and shows the view ports of device 1 and 2 as coloured rectangles.

This application was built on top of the JavaScript API of XD-MVC. The maps were realised with Google Maps and the application thus serves as a demonstrator for integration with external components. The application uses the role concept, connection management, and state synchronisation provided by the library. The benefits of the peer-to-peer architecture have become apparent with this application in informal tests. While panning is smooth on devices supporting WebRTC there is a notable lag with those that do not.

4.3.5 Voting

To demonstrate the usage of the library with an MV* framework other than Polymer, we implemented an application with the React framework. It builds on the XD-MVC JavaScript API. The application provides a voting mechanism. It could be used in group meetings or deployed on a (semi-)public screen. There are two main components: The results view shows questions and corresponding answers with the numbers of votes they have received. The voter view allows a user to vote on existing questions or to add new questions. Figure 4.15 illustrates the applications used with two phones and a tablet. The tablet shows the results view. The phone is used to vote for a question, while a new question is being added with the phone on the right. Devices can be paired with shared URLs and QR codes. These pairing options are not available in the XD-MVC JavaScript API and thus needed to be reimplemented in React. The application including the React integration was implemented by Aryaman Fasciati as part of his Bachelor thesis project [44].

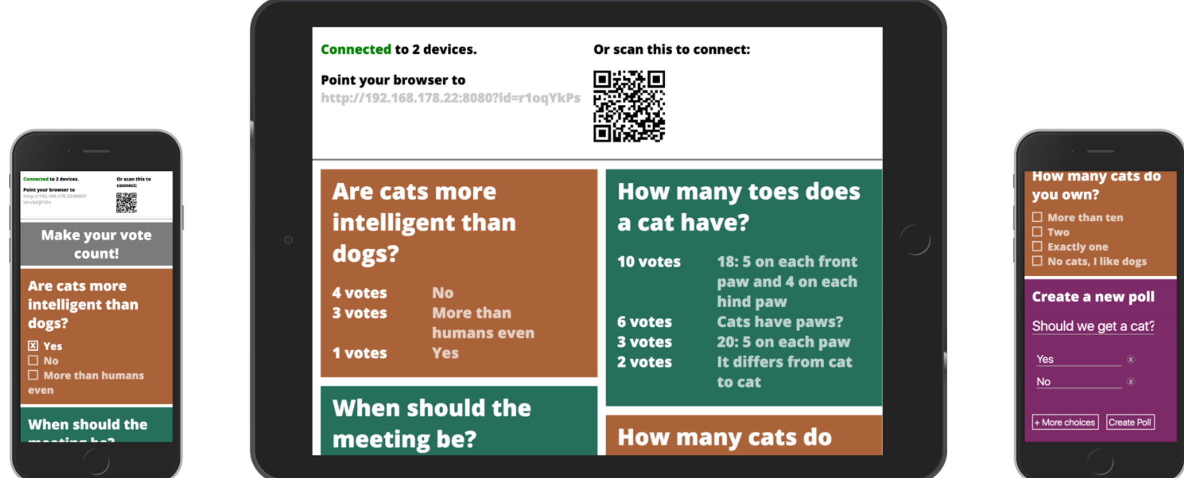


Figure 4.15: A cross-device voting application. Devices can either be used to vote or to display the results.

4.4 Discussion

In this chapter, we have introduced the cross-device library XD-MVC. Our goal with the library was to provide flexible support for developers. The layered architecture allows developers to choose the level of support they need, ranging from simple state synchronisation to visual pairing components and distributed layout templates. We also contribute a hybrid architecture for device-to-device communication. Transparent to the developer, the library chooses a peer-to-peer or a client-server connection depending on the technologies supported by the devices. We have demonstrated the flexibility and usefulness of the library in a range of sample applications.

Some related work advocates against classic client-server architectures where an internet connection is needed to reach a fixed server [122, 165]. Instead they suggest the use of ad-hoc networks and peer-to-peer connections to support situations without an

internet connection. Even though our library uses peer-to-peer connections, it requires access to a server to be fully functional. The server is needed to setup the connection between devices. Ad-hoc interaction in offline scenarios is thus not supported. In our approach, the benefit of the peer-to-peer communication is the reduced latency.

When developing the sample applications, we noticed that, despite the usage of a cross-device library, some tasks in the development process are poorly supported by current tools. All applications are web-based and run in the browser. To test and debug these applications we used conventional browser tools. However, these do not cater for the parallel use of multiple devices in a single application. While there are workarounds, for example using multiple browser windows or user profiles, there is a lot of manual labour and repetition. In the following chapter, we investigate how these processes could be better supported.

5 XD-Tools: Debugging

This chapter[1] presents an integrated set of tools for debugging cross-device applications. While there are many frameworks and toolkits that target the design and implementation phase of cross-device applications, there is only limited support for testing and debugging. Checking whether a piece of code that was just written behaves as expected and, if it does not, finding out why, are crucial parts of the developer workflow [173]. For single-device development, a wide range of tools support this stage of the process. However, these conventional tools have not been built with cross-device applications in mind and consequently offer only limited support for such scenarios. The large set of possible device combinations that can be used increases the space of solutions that need to the checked. Furthermore, conventional tools do not account for the fragmentation of a cross-device application across multiple devices. Devices are handled in isolation and interdependencies between devices are not supported conceptually. As a result, there is more manual labour required by the developer for coordinating devices and aggregating debugging information that is distributed across devices.

In this chapter, we analyse how cross-device development differs from single-device development with respect to informally verifying code and debugging. Based on these differences, we identify a set of requirements for debugging support. Next we present XD-Tools, a proof-of-concept implementation of these requirements. We report on a qualitative evaluation of XD-Tools in a user study. This chapter was developed in [67].

5.1 Developer Tasks

Shneiderman and Mayer [173] have identified five basic programming tasks: composition, comprehension, debugging, modification, and learning. We discuss the challenges cross-device development introduces with respect to these tasks. We omit learning as it is out of scope and describe no specific modification tasks as modification "requires skills gained in composition, comprehension, and debugging". To illustrate these tasks,

[1] Earlier versions of parts of this chapter were originally published as Husmann et al. [78].

we use a running example of a cross-device video player (like the application presented in [197]). The video player is composed of the video stream, playback controls, and a search interface. The video player adapts to the device configuration at hand by distributing these components across the devices.

5.1.1 Composition

Composition entails understanding the problem, devising and implementing a plan to solve it, and finally checking the solution [173]. In cross-device development, the first three steps are supported by tools for design, prototyping, and implementation such as the ones described in the two previous chapters. For checking the solution, we distinguish four kinds of checks: Does the program distribute across devices as expected? Does it behave as it should? Does it look as it should? Is it fast enough? While the last three questions are not specific to cross-device applications, they can be more challenging to answer when multiple devices are involved.

Distribution Checks

Most cross-device design and implementation frameworks allow the developer to specify how an application should be distributed across devices, for example using event-based [28] or declarative approaches [197]. Independent of the approach, the developer should verify that the application distributes components as expected for a given set of devices. The request for preview tools mentioned in [197] illustrates that it can be challenging for developers to predict the distribution solely by looking at the code. In our video example, the playback stream should go to devices with larger screens, while smaller devices such as phones show the playback controls. A developer could check this assumption by testing the application with a few device configurations. In addition, they could verify that the application also distributes correctly in corner cases. For example when two phones of equal size are used, and no other devices, the application should still be functional and display all components. However, there is a trade-off between testing a lot of device configurations which would likely result in better chances that bugs are caught and the time and effort spent on these repetitive tests.

Functional Checks

In functional checks, the developer verifies that the application specific behaviour is as expected. For example, when the play button is pressed, the video should start playing on whichever device has the playback stream. This could be the same device or another device. The potential distribution adds complexity to the process.

Visual Checks

The challenge of making a UI look good on a variety of devices is exacerbated in cross-device applications where the possible combinations of devices increase the design space and result in even more configurations that need to be checked. For example, when only a tablet is used, the device shows the video stream as well as the playback controls. When a mobile is connected, the controls are moved there. While in a conventional application, a developer might check whether they implemented the design correctly

for a phone, a tablet, and a desktop, a cross-device application can support various combinations of these (and other) devices. So the developer might check with a phone *and* tablet, *only* a phone, *only* a tablet and other configurations. As with the distribution checks, there is significant effort required to repeat visual checks on many different configurations.

Performance Checks

While we do not address rigorous, quantified performance tests in this chapter, we do expect a developer to informally check performance and react to obvious issues. The distributed architecture of a cross-device application introduces a communication and synchronisation overhead. This overhead can impact performance and the perceived responsiveness of an application. When the application is tested on a single device, performance issues may not be presented, for example, due to shorter communication paths and lower latencies. Furthermore, the developer should take into account the diverse nature of devices that may run the application. Not all of them may be as powerful as the usual development machine. Finally, if an application supports multiple users, simultaneous interaction and a large number of users can also impact performance by increasing the synchronisation overhead.

5.1.2 Comprehension

When a developer has to work with software written by another person, for example to fix a bug or add functionality, they need to have some understanding of the code. The developer can acquire this knowledge by reading the code, by stepping through it with a debugger, or by executing the code and examining the program output [106]. The distribution of a cross-device application adds complexity. It may be unclear from the code what part will be executed on which device. On the other hand, stepping through the code with a debugger on a single device may not have the same result as doing so on multiple devices. However, using the debugger on multiple devices manually requires coordination and the same process might need to be repeated with different device configurations.

5.1.3 Debugging

Any of the above checks can reveal an issue or a bug could be reported by a user or an automated test. A crucial first step in debugging is to reproduce the bug [9]. The flexibility of cross-device applications can introduce challenges in reproducing bugs. It is possible that a bug only manifests itself in a certain device configuration but not in others. If the bug report omits such details, it can be challenging to reproduce the bug. For example, the video player application has a button to toggle play and pause. A bug report could state that the state of the button does not match the state of the video. When the developer checks this using a single device that shows playback controls, they do not observe the bug and the button behaves correctly. Only when they add a second device with controls, do they realise that the state of the button is inconsistent across multiple devices with playback controls because it is not properly synchronised.

Once the bug has been reproduced, the developer will attempt to find the cause in the code and fix it. During this process the developer will typically try to answer why something did or did not happen [98]. This can be done by stepping through the code with the debugger and examining the run-time state. This results in the same issue of having to coordinate multiple debuggers on multiple devices that we have discussed above. If it is not obvious which device is affected, breakpoints have to be repeatedly set on each device and the developer needs to monitor each device to see which one hits the breakpoint.

5.2 Requirements

Based on the developer tasks, related work in cross-device and web development, and our own experiences in implementing cross-device applications, we have identified the following requirements for tool support.

5.2.1 Emulation of Multiple Devices

Using emulated devices rather than real devices can speed up the iterations of writing code and checking the solution significantly. It also allows the developer to test on devices that are not physically available. However, emulated devices in current browsers have been tailored to sequential rather than parallel use. For example, test first on an (emulated) smartphone and then on a tablet. In contrast, in cross-device scenarios devices are used in parallel and it would be desirable to be able to emulate multiple devices simultaneously. Existing tools have not been built with such scenarios in mind. While it is possible to emulate multiple devices on the same developer machine, manual coordination and workarounds need to be employed. For example, the developer can use multiple browser windows to emulate multiple devices. However, devices in the same browser instance share session data and do not accurately represent independent devices. This issue can be worked around, at least in Chrome, with multiple user profiles. This comes at the cost of having to configure a new user profile when a new device needs to be added. In summary, a tool for testing and debugging cross-device application should support the emulation of multiple devices.

5.2.2 Integration of Real Devices

While we consider device emulation a crucial requirement, emulated devices cannot replace testing with real devices. Device emulation cannot cover all aspects of a real device. In particular, characteristics such as haptics and form factors cannot be emulated. While performance emulation is possible in theory, it is often limited to network throttling in practice. For these reasons, an application should be tested, at least occasionally, on real devices. When issues are found during these checks, it is desirable that debugging can be started right away incorporating these devices. Existing tools in browsers provide support for integrating real devices, but it is rather static (for example requires a USB connection) and not a good fit for rapid changes in device configurations which are common in cross-device scenarios.

While current tool support treats device emulation and the integration of real devices as two independent tools, we suggest there could be benefits in combining the two. The following scenario illustrates a possible use case. In our cross-device video player, expected device configurations include a single laptop PC, a TV with a laptop, and a phone with a TV. When the phone interface is being tested, using a real phone occasionally ensures that the UI looks and feels good when using touch input. However, a TV may not be readily available and, when the focus is on the phone, an emulated TV could be used in the test.

5.2.3 Switching of Device Configurations

As illustrated in the video scenario, a developer may want to test and debug an application with a range of different device configurations. While the state of the art allows a developer to quickly switch from one device to another, for example from a phone to a tablet, it does not consider groups of devices. However, a developer may have a number of device configuration that they want to test in succession, for example a phone and tablet, two phones and a TV, as well as just a tablet. In addition to enabling a way to quickly switch between predefined device configurations, a tool should also provide a means to add new devices to an existing configuration so that the developer can test how the application adapts to dynamic changes at run-time. For example, an application might update its UI distribution when a tablet is connected. In the case of the video streaming application, it will move playback controls to the tablet if one is available.

5.2.4 Integration of Debugging Tools

Modern browsers integrate a wide range of debugging tools. These allow the source code of the application to be inspected. In addition, network communication can be tracked and failures highlighted. The console allows the developer to output debugging information and to execute code in an ad-hoc fashion. With the debugger, the developer can step through the code and inspect program state. With CSS tools, the developer can see which rules are applied to a given element and add, remove, or update rules at run-time. Such tools are very useful, but they do not cater to the parallel use of multiple devices. Consequently, the developer has to coordinate these tools on all devices, for example by inserting a CSS rule separately on each device or by setting breakpoints repeatedly.

5.2.5 Automatic Connection Management

In web development, the code editing and testing cycle can be described as follows. The developer edits a file containing code for the client application. The application is executed by loading a URL in a browser. Every time the developer makes a change to the client and wants to verify it, the URL needs to be reloaded. When iterations are short and happen in a quick succession, manually reloading the application is tedious and tools such as BrowserSync have automated this process. However, in cross-device applications, there is typically an additional pairing step that connects the devices. When the browser is refreshed, the connections between devices are lost and this pairing

steps needs to be repeated every time the application changes. Since the issue is specific to cross-device applications, current tools for responsive web applications provide no support. A tool specific to cross-device development could alleviate the problem by reconnecting previously paired devices automatically when the application is reloaded.

5.2.6 Coordinated Record and Replay

Testing and debugging applications typically includes repetition. A small change to the code requires the application to be checked to see if the change had the desired effect, for example if it eliminates a bug or changes the UI or logic as expected. Often, the changed code does not affect the initial state that the application is in upon loading, but fist requires user interaction. A test of a responsive application could, for example, include entering text in a field and then clicking a submit button. Recording and replaying such interactions lowers the amount of repetitive work that a developer has to do manually. State of the art tools allow tests to be recorded on one device, for example a phone, and replayed on another, for example a tablet, so that the results can be compared. Existing tools target single-device scenarios where all interactions occur on the same device. In contrast, in cross-device scenarios interactions can occur on multiple devices. The interaction on these devices would need to be coordinated. In the above example, the input field and the submit button could be located on two different devices and the click on the submit button should not be triggered before the input has been entered on the other device. Furthermore, some cross-device scenarios target multiple users and interactions can occur simultaneously. It is not practical for a developer to gather several people each time they need to test an application. Thus, record and replay functionality should be tailored towards multiple devices and should support simulating multiple users.

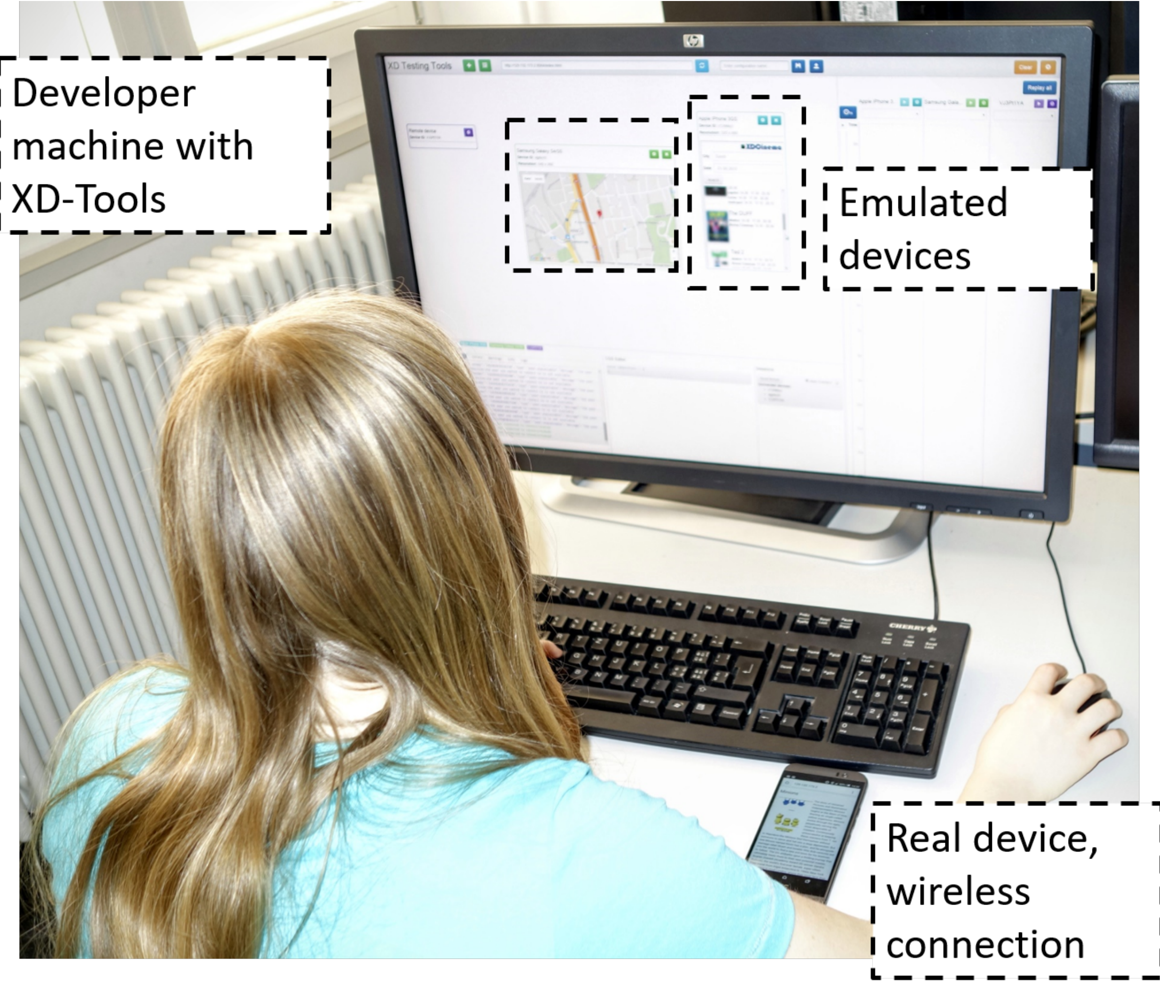


Figure 5.1: A developer using XD-Tools. Two devices are emulated in the main XD-Tools application on the developer screen. A real device is also connected to the system.

5.3 XD-Tools

Based on the requirements, we have designed and implemented an integrated set of tools, called XD-Tools. XD-Tools runs as a proof-of-concept application inside a browser, however, we imagine that the tools could become part of a browser's own developer tools in the future. The high-level goals of XD-Tools are twofold: First, we aim to facilitate the navigation of the large space of possible device combinations by introducing the concept of device configurations that allow devices to be grouped together and to be saved and loaded. Second, we mitigate the impact of the fragmentation of the application across devices by aggregating information and the control of multiple devices centrally. At the same time, XD-Tools builds on established concepts and tools in testing and debugging single-device applications.

XD-Tools consists of two main parts. A main application was designed to be loaded onto the developer machine and is optimised for larger screens. It is complemented by a helper application for the integration of real devices. Figure 5.1 illustrates the setup that a developer could use when working with XD-Tools. An overview of the main application is shown in Figure 5.2. It is centred around a set of emulated devices ⓐ that take up most of the available screen real estate. At the top, a toolbar provides

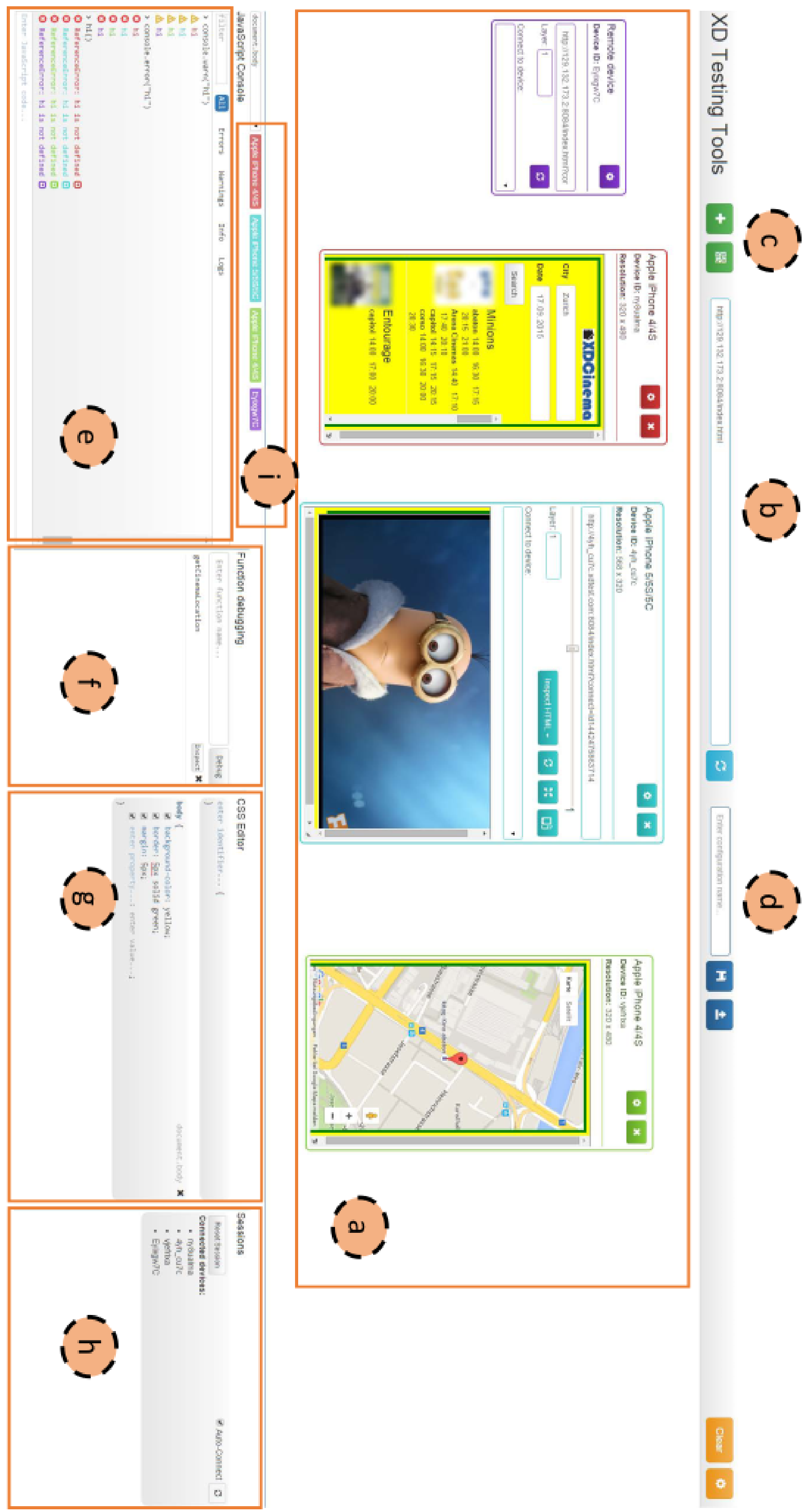


Figure 5.2: The main application of XD-Tools with one real device connected and three emulated devices. A cinema application has been loaded into the system.

functionality for specifying and reloading the application under test ⓑ, adding new emulated devices ⓒ as well as saving and loading device configurations ⓓ. The bottom is reserved for debugging tools, such as the JavaScript console ⓔ, function debugging ⓕ, the CSS editor ⓖ and connection management ⓗ. Devices (emulated and real ones) are assigned a colour for easier identification and each device can be enabled or disabled for debugging ⓘ. Not visible in this screenshot is the record and replay tool which would reside next to the emulated devices on the right side of the screen. To prevent visual overload and to cater to a developer's preferences and the current stage in the workflow, the UI can be customised. The tools can be configured in size or hidden if not required. Next, we describe how XD-Tools addresses each requirement.

5.3.1 Emulation of Multiple Devices

XD-Tools supports emulation of multiple devices at the same time. Devices can be chosen from a list of predefined, common device models (Fig. 5.3) or customised manually. Each emulated device is represented by a coloured rectangle that comes with a toggled menu at the top and the device view port containing the application that is being tested or debugged (Fig. 5.4). The emulated devices can be freely arranged on the screen using direct manipulation. If many devices are used simultaneously or if devices with larger screen resolutions are emulated, the emulated devices can be scaled down. With the press of a button, the device can be flipped from landscape to portrait mode and vice-versa. At a glance, the developer can see an entire configuration of devices and, when interacting with a device, immediately sees the reaction of the other devices.

In contrast to emulating multiple devices with conventional browser tools in multiple tabs or windows, XD-Tools ensures that each device has its own separate resources, such as local and session storage. For example, an application with user login would require the user to log in on every device in XD-Tools, while they would be automatically logged in on all devices after logging in on a single device with the conventional tools. Our approach is thus a more accurate representation of using multiple devices.

5.3.2 Integration of Real Devices

We decided against using USB cables to integrate real devices. Cables do not scale beyond a few devices unless specialised hardware is used. Furthermore, not all devices can be easily connected with a USB cable. While it is simple to connect a tablet or phone (assuming the correct type of cable is at hand), integrating a TV or an interactive whiteboard by cable may not be straightforward. Instead, XD-Tools offers a mechanism to add devices by either scanning a QR code that is displayed on the developer machine or loading a URL manually on the device to be integrated. This step will load a helper application on the device that communicates with the main XD-Tools application over the network. The helper application does not contain any UI elements in order not to interfere with the application under test and to provide a realistic testing environment. Each connected real device is represented in the XD-Tools main application by a coloured proxy similar to the emulated devices but without the viewport. This allows the developer to see at a glance how many devices are connected. Real devices can be used for debugging, record and replay, and connection management alongside the emulated devices.

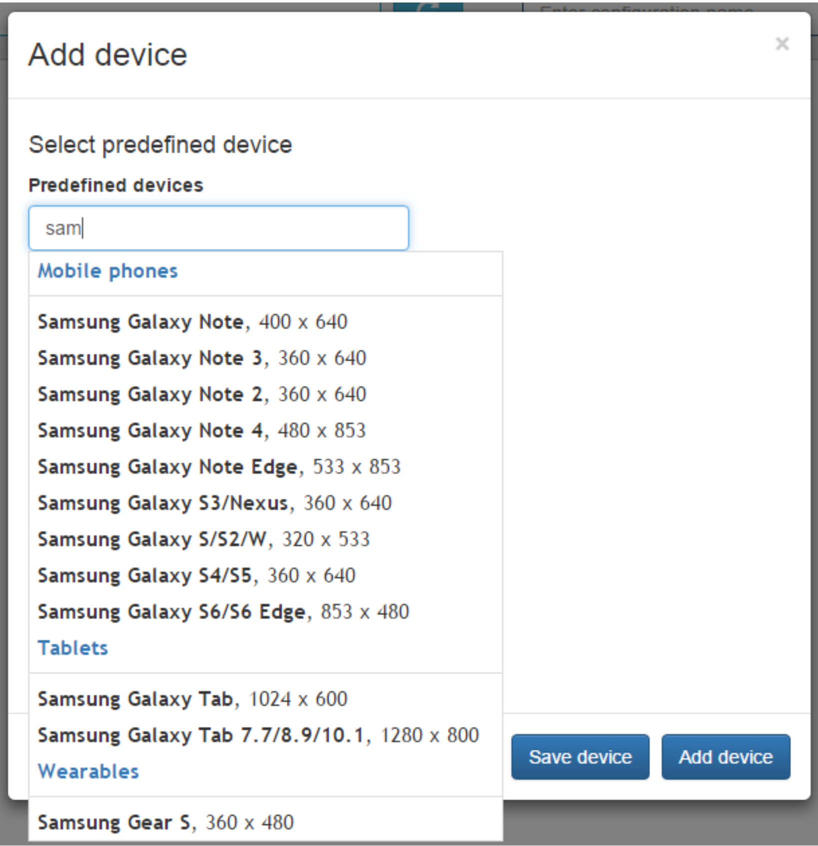


Figure 5.3: Selecting an device to be emulated from a list of preconfigured device models.

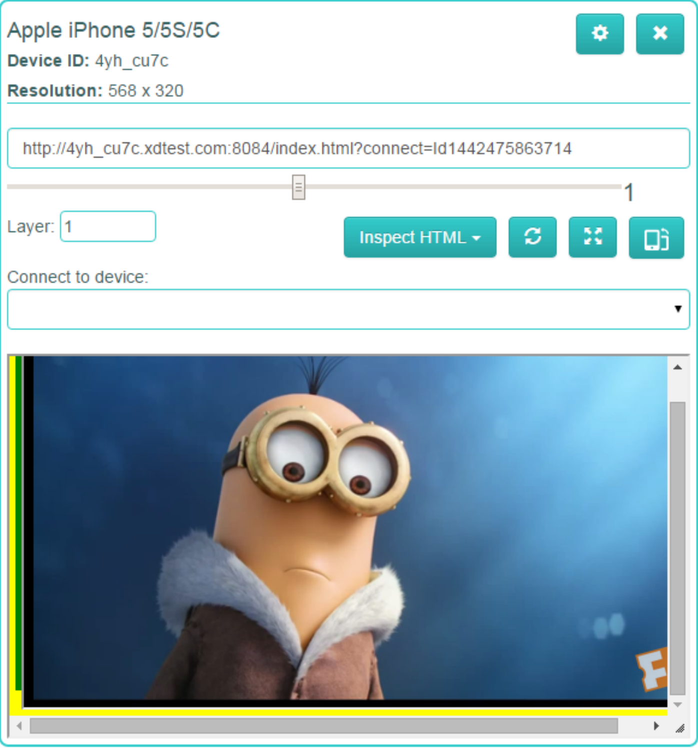


Figure 5.4: An emulated device in XD-Tools.

JavaScript Console

```
filter   All   Errors   Warnings   Info   Logs
> getCinemaLocatio("Zurich", "abaton")
ReferenceError: getCinemaLocatio is not defined
ReferenceError: getCinemaLocatio is not defined
ReferenceError: getCinemaLocatio is not defined
ReferenceError: getCinemaLocatio is not defined
> getCinemaLocation("Zurich", "abaton")
<- {"lat":47.388953,"long":8.52118}
<- {"lat":47.388953,"long":8.52118}
<- {"lat":47.388953,"long":8.52118}
<- {"lat":47.388953,"long":8.52118}
> XDmvc.roles
<- ["sync-all","general"]
<- ["sync-all","extra"]
<- ["sync-all","location"]
<- ["sync-all","movie"]
Enter JavaScript code...
```

Figure 5.5: The aggregated JavaScript console showing logs from four devices.

5.3.3 Switching of Device Configurations

To enable fast switching between device configurations, XD-Tools allows the developer to name and store the current device configuration on the screen. The system saves the arrangement of the devices on the screen and allows the developer to restore it at a later time. New devices can be added quickly from the list of predefined devices. The developer can extend that list with their own device types if needed.

5.3.4 Integration of Debugging Tools

We have either re-implemented or integrated some of the existing debugging tools commonly offered by browsers as they have proven to be useful. These tools have been built for single device use cases. We have extended them to provide integrated support for multiple devices. By default, all devices are available in the tools, however, a device can easily be deselected with a click on a button that matches its assigned colour.

JavaScript Console

The console is a commonly used tool for debugging. Logs show errors produced by the browser, for example if a resource is not found, as well as any information that the developer chooses to log from the code using the `console` API. In XD-Tools, we aggregate the logs from all devices and show them centrally in one place so that the developer does not have to check an individual console for each device. The logs are colour coded in the device colours to facilitate the identification of the device that is responsible for printing the log.

Imitating the behaviour of a typical browser console, our console can also be used to

execute JavaScript at run-time on all selected devices, including connected real devices. This allows the developer to quickly test functionality, inspect the state of the application by printing variables, or experiment with the implementation by injecting or overwriting functions without reloading the whole application. Figure 5.5 shows example usage of the aggregated console. First, a function is called that is not defined and an error message is shown. When the correct function name is used, each device prints the output (a location object) to the console. Here the developer can see that all devices print the same location, which could indicate that the synchronisation works as expected. The application under test in the example uses different roles for devices and, when the roles are printed, the developer can inspect the roles assigned to each device. Without XD-Tools, the developer would need to execute the function on each device separately and aggregate the output manually. If a function needs to be executed on a single device only, the developer can deselect the other devices by clicking their corresponding toggle buttons.

CSS Editor

XD-Tools includes a simple CSS editor that allows new style rules to be added without reloading the application, facilitating fast experimentation with the visual design or finding the correct rules that produce a desired result. As with the console, the developer can select the devices that should apply the rules from the CSS editor. For example, the developer could create a rule while a phone is selected. After the rule has been applied to the phone, the developer could select the TV and add a different rule for that device.

Source Code Inspection and Function Debugging

The source code can be inspected for emulated devices. A button in the menu of each device takes the developer to its source code. It is also possible to debug functions. The developer can specify a function name globally to set a breakpoint. When any device hits the function, the execution is stopped and the developer can step through the code with the debugger. The device that hit the breakpoint is highlighted (Fig. 5.6). XD-Tools integrates the debugger provided by the browser, so the developer has access to the usual tools for inspecting the state of variables or the call stack. Without XD-Tools, the developer would need to set a separate breakpoint on each device and constantly monitor the devices to realise when one hits a breakpoint.

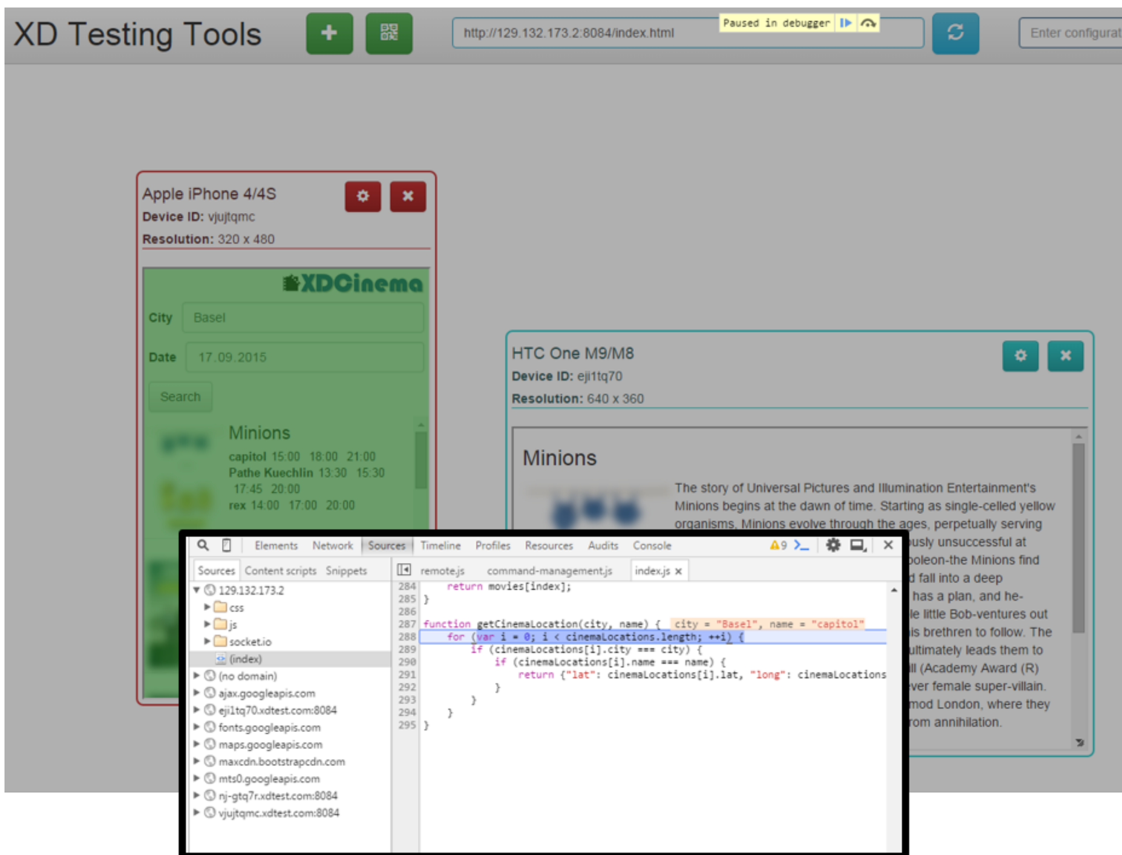


Figure 5.6: Debugging with XD-Tools. The device hitting the breakpoint is highlighted in green.

5.3.5 Automatic Connection Management

To speed up iterations, XD-Tools provides a button that reloads all devices, emulated or real, simultaneously. Furthermore, it provides an option to reconnect previously connected devices automatically, eliminating the time consuming step of re-pairing devices manually at each reload. Additionally, the developer can configure XD-Tools to automatically pair newly added devices to the existing device configuration. In our experience, it is a common use case that a new device is added and immediately paired to the other devices in the system. On the other hand, there are situations where this behaviour might not be desired, for example when testing the pairing process itself.

5.3.6 Coordinated Record and Replay

User interaction can be recorded on emulated devices and replayed both on emulated and real devices with XD-Tools. Figure. 5.7 illustrates the record and replay interface with a configuration of three devices. A column represents a timeline for each device. After the record button has been pressed, the system records all user interactions. Once that process has been stopped, XD-Tools represents the recording on the timeline. As many events can happen within a short period, for example a sequence of mouse move events, the events occurring close together are collapsed into a single label. The developer can cut a longer sequence into subsequences and describe them with labels. For example, in the video player, the user could first look for a video by entering a term into the search box and then choose a video from the results to play. This interaction could be recorded in one session, but later split into a searching and a playing part. The recorded sequences can be moved for replay on any device. In our example, the developer could record the whole sequence on a single tablet. Later they could add a phone to the device configuration and choose to replay the search sequence on the phone while keeping the play sequence on the tablet to mirror the UI distribution.

Sequences can be copied so that multiple devices replay the same sequence. The developer can either replay sequences individually on each device or globally. In the case of global replay, the system ensures a basic coordination between the devices so that a sequence on one device will start when another one has finished on another device. Replaying sequences simultaneously on multiple devices allows the developer to simulate multiple users using the system. The replay can be stopped with breakpoints so that the developer can inspect the system at any time during the interaction.

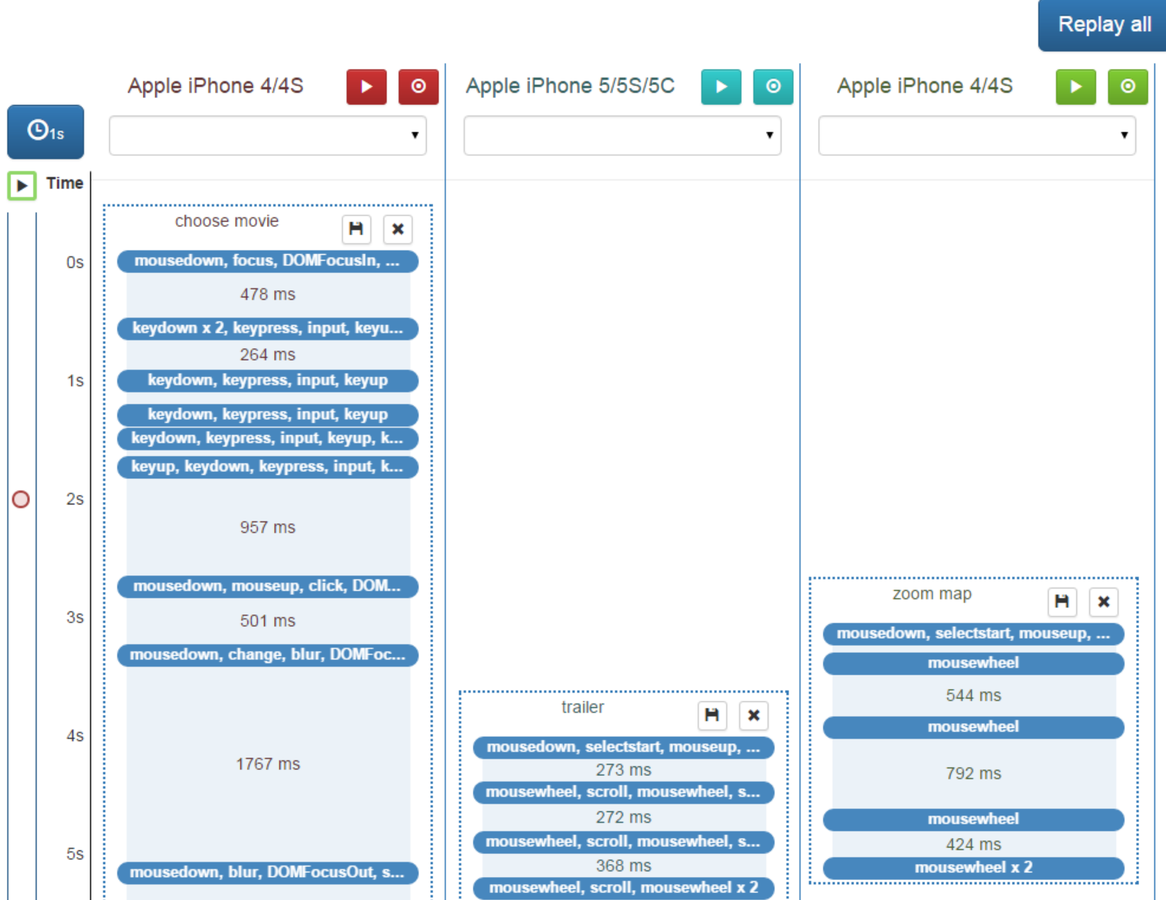


Figure 5.7: Record and replay on three devices in parallel.

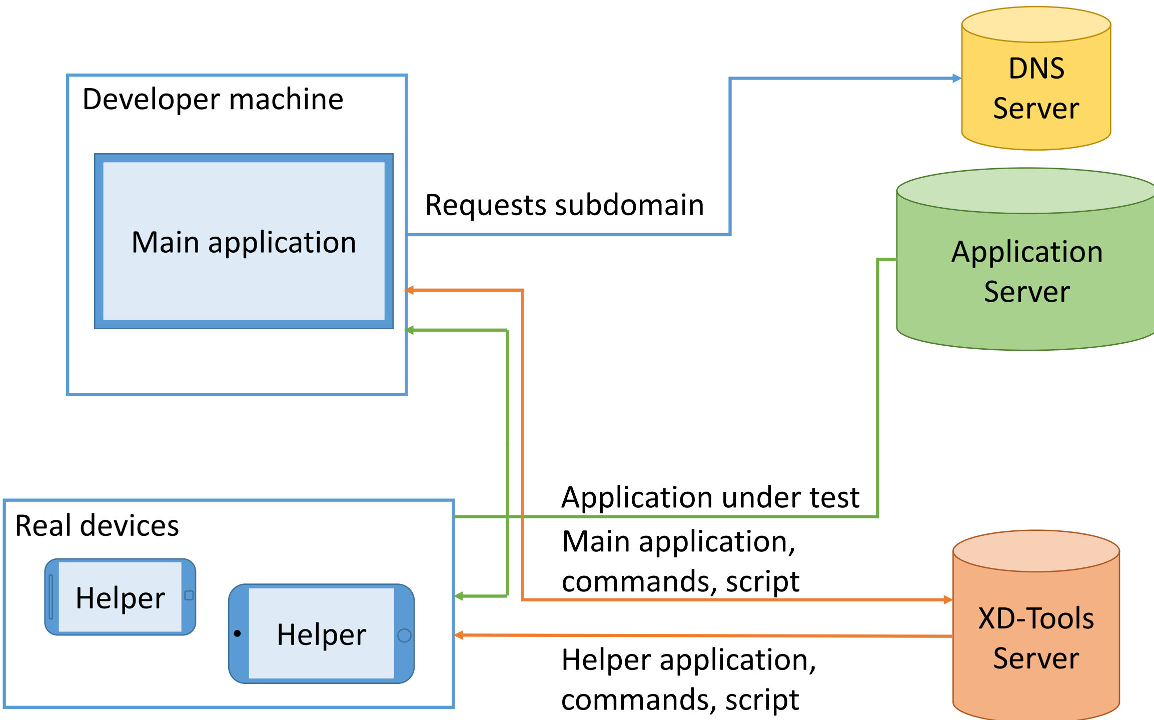


Figure 5.8: XD-Tools architecture.

5.4 Architecture and Implementation

Figure 5.8 illustrates the architecture of XD-Tools. The system consists of several parts. There is a main application that provides the UI of XD-Tools. The main application was designed to be loaded onto a developer machine with a large screen. It is served by the XD-Tools server. Inside the main application, emulated devices load the application under test from the application server. There is no restriction on the application server other than it must permit the application to be loaded into iframes. A small script must be injected into the application under test. The script is hosted on the XD-Tools server. The real devices load a helper application from the XD-Tools server. The helper application has no UI component. Instead it is responsible for reloading the application under test, replaying events, and handling the automatic connection management. The communication between the helper application and the main application is routed via the XD-Tools server and was implemented with Socket.io.

Both the helper and the main application have been implemented with pure web technologies and do thus not depend on a specific browser vendor. However, to achieve a tight integration with an existing debugger, we created a Chrome extension. While the rest of tools can be used in any modern browser, the source code inspection and function debugging are only available through the extension in Chrome. As Chrome extensions are only supported on the desktop version of Chrome and not in the mobile version, the real devices do not offer this functionality.

In contrast, the JavaScript console and the CSS editor are re-implementations of the existing browser tools and are not part of the browser extension. Real devices are affected by changes made through these tools via the injected script. The script overwrites the default logging function, forwarding all logs to the main application

where the content is shown in the aggregated console. The script will also execute all JavaScript commands that are typed into the console.

Device emulation has been implemented with iframes. The application content of each emulated device is loaded into a separate iframe. With a naive implementation this approach would result in shared content across multiple emulated devices. The default behaviour of all major browsers is to share session and persistent data (localstorage) that originate from the same host (for example myapplication.com) across all windows, tabs, or iframes. We have observed developers using different browser profiles to workaround this issue. To better replicate the behaviour of separate devices, we trick the browser into assuming that each emulated device loads the application from a different host. This is achieved with the introduction of a DNS server. For each emulated device, we generate a new host name (for example myapplication2.com) that will be resolved by the DNS and point to the application server. To the browser each emulated device loads a different application, while in reality they all load the same one. As a result, each emulated device has its own session and persistent storage.

For the coordinated record and replay functionality, the event handlers are registered for all events that are tracked by XD-Tools. It is crucial that XD-Tools is the first event handler to be called. Otherwise other event handlers could modify the state of the document or even stop the propagation of the event before the XD-Tools handler is called. In the latter case, XD-Tools would miss the event. To avoid these problems, the XD-Tools script should be inserted at the top of the application's main HTML so that it is the first script to execute. Furthermore, the script uses the event capturing[2] on the `document` node rather than the default bubbling. When replaying, devices schedule all events up to the first breakpoint using `setTimeout`. Once the breakpoint is reached, the main application is informed and the next events will be scheduled when the user decides to resume the execution.

XD-Tools is not coupled with any cross-device framework. The requirements are independent of the specific implementation of a cross-device framework. There is one exception: The automatic connection management clearly depends on the connection mechanisms used in the cross-device application or the framework with which it was built. We have implemented default auto-connections for applications that use URLs for pairing. To cater for different mechanisms, there is an option to disable the URL-based approach and configure a script-based approach. When using the latter, developers have to provide the implementation for two functions (List. 5.1). XD-Tools selects one device as the main device to which all other devices will connect. The main device can be changed in the UI. The function `getConnectionParam` will first be called on the main device, giving it a chance to execute any necessary code and to provide a parameter (for example a device identifier) that will be passed on to all other devices in `connectWithParam`. In that second function, these can then execute the necessary steps to connect to the main device. Note that this approach also allows for all devices to join a predefined session or room if such a concept is used. Listing 5.1 shows the implementation of these two functions when the JavaScript API of XD-MVC is used for pairing rather than URLs.

[2] `https://www.w3.org/TR/DOM-Level-3-Events/#event-flow` Accessed on 10.04.2017

```
function getConnectionParam() {
  // Return a parameter that is required
  // to connect to this device.
  return XDmvc.deviceId;
}

function connectWithParam(param) {
  // Establish a connection to a device.
  // Device information is given in param.
  XDmvc.connectTo(param);
}
```

Listing 5.1: Configuring automatic connection management with a script.

5.5 Evaluation

Cross-device development is an active topic in academia, but still relatively rare outside. The lack of experienced cross-device developers limits the form of evaluation studies that can be carried out and precluded a long-term study in the wild. Instead, we evaluated XD-Tools in a small qualitative study with 12 developers. We designed the experiment around two conditions: In the baseline condition, *browser-tools*, participants only had access to the Chrome Developer tools (including device emulation and remote debugging), while, in the second condition, they also had access to *XD-Tools*. In both cases, they had the option to connect real devices.

5.5.1 Participants and Setup

We recruited 12 participants (10 males, 2 females) aged 23 to 33 (median=26) from our computer science department. At least basic skills in web programming (JavaScript, HTML and CSS) were required to participate in the study, however, we relied on self-assessment. Two participants reported more than 5 years of experience in web development, five 2 – 5 years and five less than 2 years. Nine participants had experience in developing responsive applications and eight had at least some experience developing cross-device applications. Among those eight participants, five reported using browser tools for emulating devices and six using real devices for testing. Other strategies mentioned were using multiple browsers, multiple browser profiles, and private windows (to avoid shared data). Ten participants had experience with the Chrome Developer tools that we used as a baseline.

The participants were provided with a 30 inch screen (2560x1600 pixels) and a desktop computer as a main device (Figure 5.9). A Nexus 7 tablet and an HTC M9 Android phone were available for testing. The main device had five browser profiles preconfigured that could be used for emulating devices.

5.5.2 Tasks and Procedure

A different cross-device application was used for each condition to reduce learning effects: The first was a video player that consists of controller devices that can be

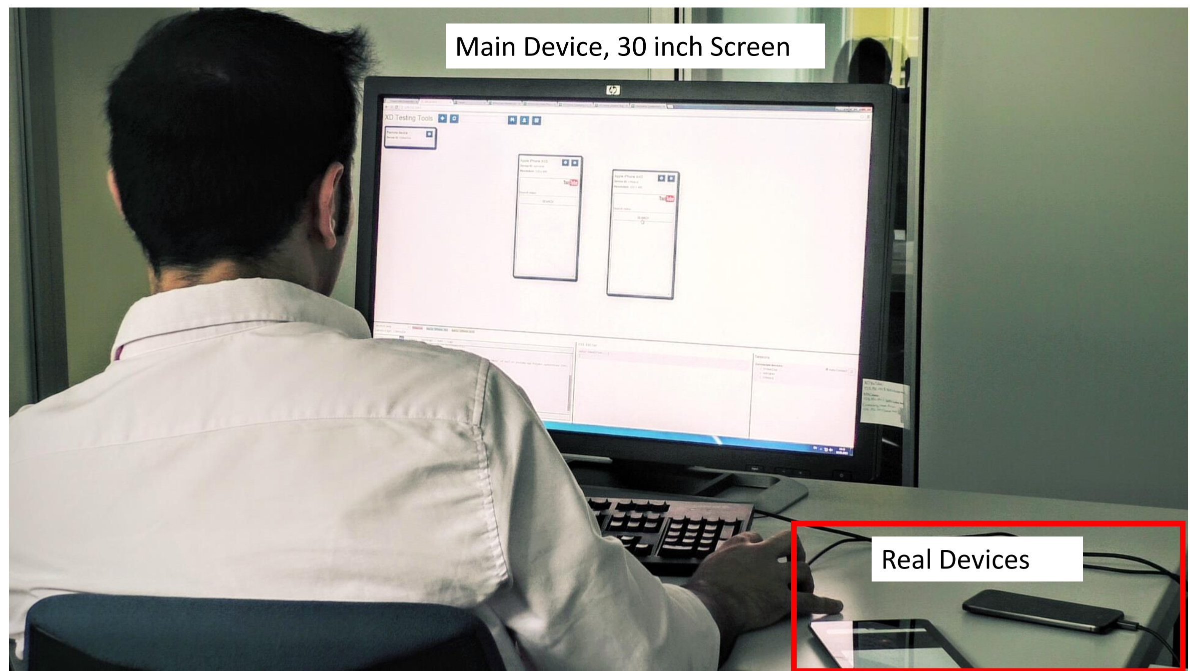


Figure 5.9: Study Setup.

used to search for videos and create playlists and player devices that show the videos. The second was a cinema application that shows information such as location, a film summary, and show times on the set of available devices. Both applications were implemented with the XD-MVC framework. For each application, participants were given two tasks: a *debugging* and an *implementation* task.

The conditions and applications were rotated and counterbalanced. However, participants always started with a debugging task followed by an implementation task. The video debugging task was designed to require at least two devices to reproduce the bug in the UI. The cinema debugging task was optimised for two devices as well, however it was possible to solve it using a single device in which case it required more user interaction to reproduce the bug. For both tasks, it was also possible to spot the bug in the source code. Participants were asked to fix the bug once they found it. Similarly, for the implementation tasks, at least two devices were required in the video application and three devices in the cinema application to verify the correctness. The following tasks had to be completed.

- *cin_bug*: A wrong variable name in a loop caused cinemas to be displayed in incorrect map locations.
- *cin_impl*: When a film is chosen, the prices of all theatres showing it in the selected city had to be displayed. In a next step, the price for a selected theatre had to be highlighted and the style of the highlighting had to be improved using CSS. The selecting of location and theatre was done on other devices.
- *vid_bug*: Access to an empty video queue stopped the playlist from working in some use cases.

- *vid_impl*: Functionality of a remote control button had to be implemented on the controller device to play and pause the video on the player device. The button should only be functional if a player device is currently showing a video. The state of the video should be reflected in the button (play, pause). All controller devices in the system should reflect the correct state. The style of the button was given as a mockup.

Participants were given skeletons for the implementation and were allowed to test their implementation at any time, debugging it if necessary. In all tasks, participants were encouraged to test if their implementations and bug fixes worked correctly.

Participants were introduced to the Chrome Developer Tools, XD-Tools, and the applications. The record and replay feature of XD-Tools was hidden and not available in the study in order to decrease the study duration. Debugging tasks were stopped after 15 minutes and implementation tasks after 30 minutes if the participants were not close to reaching a solution. The whole study took participants 90 to 120 minutes to finish, including completing questionnaires at the beginning, after each task, and at the end of the study. The study was captured on video in order to analyse devices and tools that were used.

5.5.3 Results

Nine participants managed to complete all tasks within the given time. Three participants did not complete five tasks in total (2 *cin_bug*, 2 *cin_impl*, 1 *vid_impl*). The participants who had difficulties with the tasks generally struggled with web technologies (JavaScript in particular) despite assessing their skills at 3 on a 5-point scale (1=basic, 5=proficient). As can be seen in Figure 5.10, developers were not faster using XD-Tools compared to the baseline. As most developers (10 out of 12) were familiar with the baseline tools and but not with XD-Tools, more exposure to XD-Tools could potentially result in performance gains. One participant (P5) commented that *"using Chrome devtools happens automatically. It takes time to get used to different tools"*.

Overall, only 3 participants connected real devices. In the *browser-tools* condition, one user connected two devices in the implementation task and another user connected one device in both tasks. In *XD-Tools*, one user connected one real device in each task. Figure 5.11 shows the average number of emulated devices per user grouped by task. Three participants commented that they like to see everything (including emulated devices) in one window or one screen. On the other hand, one participant requested separate windows for emulated devices which our system currently does not support.

We received very positive feedback. All participants agreed or strongly agreed that they would use XD-Tools for implementing cross-device applications and all but one participant strongly agreed that they would use XD-Tools for debugging (Figure 5.12). P3 said *"I think it is very useful for cross-device applications and I wish I had access to it when I was working on a cross-device application last year."* In particular, the auto-connection option was very well received. Also the integration of the debugging tools was praised. Another participant (P8) stated that *"it was very convenient to use. I especially liked the CSS editor and connection features. It really simplifies the process of cross-device application development when one has everything visible on one screen and does not have to switch to other devices, which might be a bit distracting."*

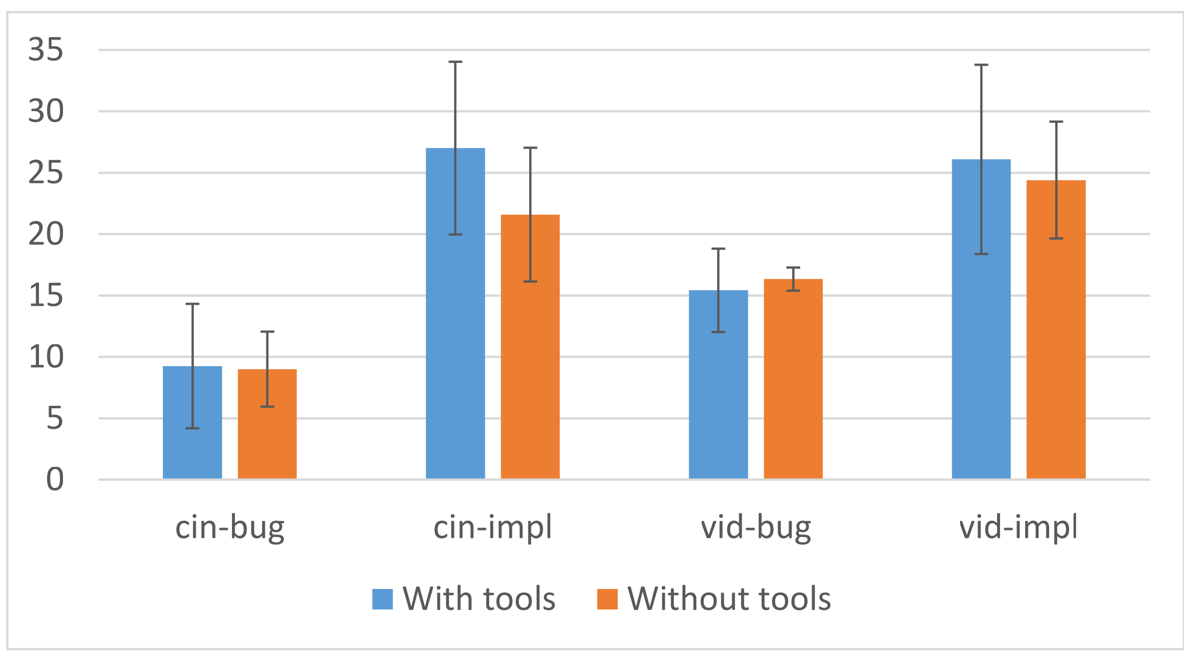


Figure 5.10: Average time per task. Only completed tasks are shown. Error bars show standard deviation.

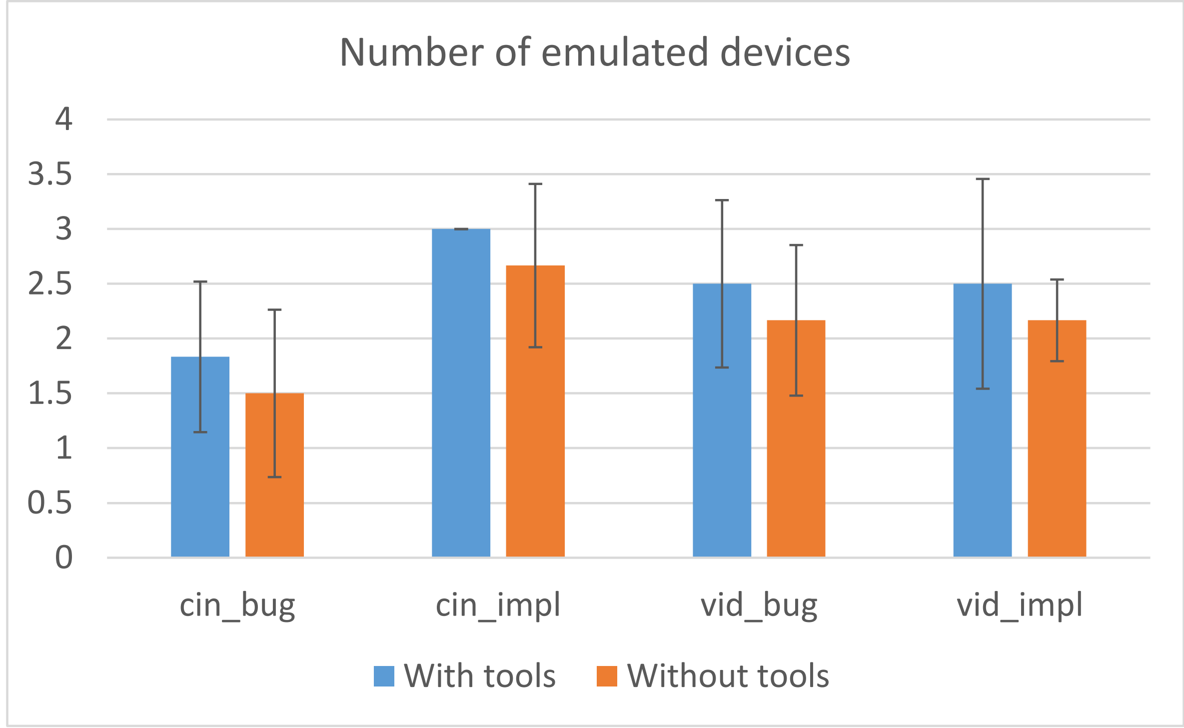


Figure 5.11: Average number of emulated devices per user. Error bars show standard deviation.

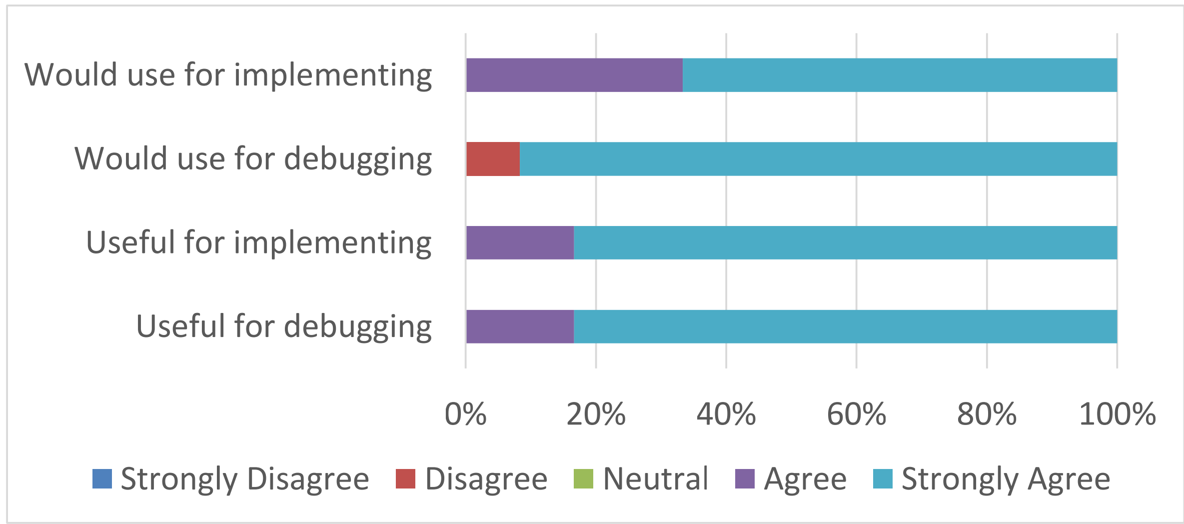


Figure 5.12: Results from the questionnaire.

Participants requested an even tighter integration with current developer tools. Some of the debugging tools were reimplemented and not all functionality of existing tools was available. For example, breakpoints for the debugger could only be set at the beginning of a function and not in arbitrary locations. As a result, some developers missed some of the features that they were used to. Similarly, auto-complete of variable names and values in the CSS editor was requested

5.6 Discussion

The feedback in our study showed that the support for *emulation of multiple devices* and *automatic connection management* enable fast cycles of authoring or modifying code and checking the result. As they remove tedious reloading and pairing steps, they reduce the overhead for the developer required to check even small changes in the code. This could lead the developer to execute their code more frequently and discover flaws in the design or implementation earlier.

We did not evaluate the save and reload functionality for *switching of device-configurations*. It would have been necessary to add additional tasks or make the existing tasks more complex for participants to really appreciate this feature. For example, we could have asked participants to implement two different designs for smartphone and tablet as opposed to tablet and TV. At roughly two hours duration, our study was already quite long and we did not want to increase the time further. However, we assume that it would also contribute to shorter implementation and testing cycles in combination with the *automatic connection management* by allowing the developer to test different device configuration at a lower cost compared to manually managing configurations. Due to the long study duration, we also decided against evaluating *coordinated record and replay*.

We observed only one instance of a participant using the *integration of real devices* with XD-Tools in our study. We attribute this to the somewhat artificial setting of the study where all tasks could be solved using only emulated devices. However, we

assume that developers would test on real devices at least occasionally during longer development phases. For example, even when touch is emulated, it does not accurately convey what it feels like to use an application on a smartphone. In our questionnaire, six participants mentioned using real devices when developing cross-device applications.

Based on the participants' feedback, we consider the *integration of existing debugging tools* crucial. In our prototype we re-implemented the CSS editor and the JavaScript console and provided basic access to the debugger. Our version of these tools are not quite as capable as the original ones and participants expressed interest in a tighter integration. In our vision, the support we provide with XD-Tools would be implemented by the browsers themselves in the future, eliminating the need for a separate set of tools.

The reduced functionality of these re-implementations is a limitation of XD-Tools. While the CSS editor in the browser usually allows the inspection and manipulation of existing CSS rules, our version is limited to the addition of new rules. Rules added through the editor can be changed or removed, but not those added in the code. The device emulation is centred around screen size and modalities such as touch are not emulated. Neither are network conditions or geographic location. While these settings could be emulated using the tools provided by the browser, the settings would apply to all devices (and the main application itself) rather than to an individual device.

We have used XD-tools with applications written with XD-MVC and with one application that did not use a cross-device framework. Although no other frameworks have been tested, no issues should arise with purely web-based frameworks. While we took care to be framework-agnostic, we also see benefits in tailoring debugging tools to a specific framework. For example, the data-synchronisation could be offered for inspection. While it is possible to inspect arbitrary state with the JavaScript console, a visualisation tailored to the specific mechanism in use could be helpful. Similarly, the specification of the UI distribution could be visualised. However, to create such visualisation knowledge of the inner workings of an application is needed.

The setup of XD-Tools is somewhat complicated as it requires the installation of a DNS server and the configuration of the operating system on the developer machine. These steps are required for the emulation of multiple devices with separate storage. If these mechanism were directly implemented in the browser tools, the DNS would no longer be needed and setup would be significantly less complex.

The main limitation of the study is the short amount of time that developers had to work with XD-Tools. As a participant commented, it takes some time to become familiar with new tools. It would be interesting to collect data of a long-term field study with XD-Tools outside of the artificial setting of a lab study. In the future, cross-device development may become more prevalent which would also increase the number of cross-device developers available to such studies.

The UI of XD-Tools has been optimised for a single large screen. However, many developers use more than a single screen and it would interesting to explore if XD-Tools itself could be distributed across multiple screens or devices.

While we provide some automation for testing applications with the *record and replay* functionality, XD-Tools has been built for manual testing and debugging. It is the developer's responsibility to execute a test and decide if the result is acceptable or not. Formally specified tests can be automated and integrated into the development process, for example in continuous integration. In the next chapter, we explore automated testing in the context of cross-device applications.

6 XD-Testing: Automated Testing

In this chapter[1], we analyse testing in the context of cross-device applications. Even though testing is an established step in the software engineering process, there is little support for testing cross-device applications and it has been identified as a particular challenge [42]. Flexible cross-device applications that adapt to the set of devices at hand could, in theory, be used by an unlimited number of different device configurations. In practice, some configurations are more likely to occur than others, but writing a separate test for each configuration would still be time-consuming and repetitive. The fragmentation of the application logic across devices exacerbates the problem as the devices need to be coordinated. Existing tools have been built for single-device use cases and offer little support to the challenges of testing cross-device applications.

In the following section, we introduce a motivating example that highlights the shortcomings of existing tools and the requirements specific to cross-device testing. As motivated in the Background chapter, we focus on UI testing as it is impacted by the fragmentation of cross-device applications more than unit testing. Next, we introduce our testing library XD-Testing and discuss how it addresses these shortcomings and requirements. We start with parametrised tests that can be repeated across multiple configurations of devices. Then, we introduce a domain specific language for cross-device tests that allows a developer to address devices explicitly and implicitly in a testing script. The final component of XD-Testing is a mechanism to record the program flow on multiple devices with screenshots. The result can be viewed in a visualiser that allows the developer to compare and contrast the execution of the program on different device configurations. We then discuss the architecture and implementation of XD-Testing and explain how it integrates with existing testing libraries. XD-Testing was evaluated in a case study with a sample cross-device application. We report on the test cases written and the results of using the library to test code written by students as part of a web engineering class. Finally, we discuss limitations of our approach and directions for possible extensions. This chapter was developed in [177].

[1]Earlier versions of parts of this chapter were originally published as Husmann et al. [81].

6.1 Motivating Example

To illustrate the challenges in testing cross-device applications, we will again use the video player application introduced in the previous chapter. Listing 6.1 presents a simple test case that checks if a video, when selected from a list of results with a click, will be displayed in the player by asserting that the correct title is shown. This example test case will work fine, if the player and the list of search results are both on the same device. However, it not suitable for a distributed UI. To address that situation, the tester could write a test case with two devices. They would need to pair the devices first. Then, they would need to know which device shows the list and which device the player so that they can execute each command on the correct device. Identifying the correct device may not be trivial, as this may depend on the capabilities of all devices, such as the screen size, or the context in general. Furthermore, the tester would need to ensure that the title element is checked after the video has been clicked and allow time for the change to propagate across devices. While writing such a test case requires increased effort compared to the single-device test case, it also just covers one particular combination of devices. However, selecting a video in the search list should *always* display that video, regardless of the device configuration used and the distribution of the UI. Existing testing libraries offer no constructs for writing tests that are *implicitly* assumed to be distributed across multiple devices and that could be executed on different device configurations.

On the other hand, addressing devices *explicitly* could allow the developer to verify if the UI has been distributed correctly for a given configuration. For example for a device configuration with a phone and a TV, a test could be written that checks if the controls are on the phone and the video on the TV.

```
client
  .click('li=My Video')
  .getText('#title')
  .then(function(value) {
    assert(value === 'My Video'); // true
  });
```

Listing 6.1: A simple test case that checks if a user clicks on a video in a list that the title of the video is displayed.

6.2 Parametrised Tests with Device Templates and Scenarios

To better support automated testing of cross-device applications, we introduce the library XD-Testing. The library offers a mechanism to parametrise a test with a set of devices so that the same test can be executed on different device configurations. To this end, we introduce *device templates* and *device scenarios*. A device template represents a reusable description of a specific device (for example an iPhone 5 or an Nexus 6). We define a device scenario as a set of devices that interact with each other. A device scenario can be composed by instantiating devices from templates. Table 6.1 lists all

Template property	Supported values
name	Strings
type	*"watch"*, *"phone"*, *"tablet"*, *"desktop"*
size	*"xsmall"*, *"small"*, *"medium"*, *"large"*
width	Positive integer
height	Positive integer
desiredCapabilities	Accepted by Selenium

Table 6.1: Properties and supported values of device templates.

properties and accepted values of a device template, whereas Listing 6.2 provides an example template. XD-Testing includes 8 predefined templates as examples. These examples include a desktop PC, different smartphone and tablet models, and a smartwatch. Additional templates can be added by the developer. As new device models are released by manufacturers, new templates can be created to reflect these additions. Templates can be shared among testers and reused.

```
1 {
2     name: "Nexus 5",
3     type: "phone",
4     size: "small",
5     width: 360,
6     height: 640,
7     desiredCapabilities: {browserName: "chrome"}
8 }
```

Listing 6.2: An example device template for a Nexus 5 phone.

Listing 6.3 shows how a scenario can be composed directly in JavaScript when writing a test case. Alternatively, the scenario can be loaded from a configuration file, enabling re-use of scenarios across test cases and projects. To better support the testing of unexpected device combinations, XD-Testing provides a command line tool for generating device scenarios randomly. The tester can restrict the randomness to some extent by limiting the maximum number of devices in the scenario and only allowing certain template properties (for example only *phone* and *tablet* as types). A set of scenarios can be iterated over so that the same test can be executed on each scenario. In the next section, we describe how individual devices can be addressed within a scenario and how tests can be composed that are independent of the scenario.

6.3 A Domain Specific Language for Cross-Device Tests

To enable the authoring of tests similar to the one in the motivating example but with support for cross-device configurations, we have developed a domain specific language. The new language constructs enable easier specification of cross-device application tests. Our language was implemented as an extension to WebdriverIO[2] which provides programmatic commands to remote control browsers with Selenium. WebdriverIO provides

[2] `http://webdriver.io/` Accessed on 04.05.2017

```
1  var scenario = {
2    ctr: xdt.templates.devices.nexus4(),
3    dA: xdt.templates.large.nexus10(),
4    dB: xdt.templates.desktops.chrome()
5  };
6
7  xdTesting.multiremote(scenario).init()
8      // Load url on all devices
9      .url("http://myapp.com")
10     .click("button");
```

Listing 6.3: Creating a device scenario ad-hoc through JavaScript: Templates can be accessed from a general set (line 2) or by characterisation such as size (line 3) or type (line 4). A test is parametrised with the scenario (line 7) so that all following steps (line 9 and 10) are executed on the chosen devices.

no easy way to coordinate multiple browsers. While it does allow the instantiation of multiple browsers, coordinating them is limited to either broadcasting commands to all browsers or sending a command to a single one that must be selected using an identifier. We introduce novel device selectors that allow the developer to either let the system choose devices implicitly, or explicitly specify the device set for a command based on device characteristics such as the size or type of the device or accessible application elements. Matching devices are passed to a callback function which executes clearly separated from the initial device set. Commands within the callback context are only executed on the selected device subset. Selectors can be nested for a step-wise narrowing of the device specifications, enabling the combination of multiple criteria. For example, a tester could first select devices by element and then restrict the resulting set to tablets only. The DSL supports method chaining which means that each method returns an object that again can receive a method call. Commands chained after a selection callback are executed on the original device set and are executed after the commands inside the callback. XD-Testing handles the asynchronism that is inherent to cross-device testing and waits for each browser to finish a command before executing the next method in the chain.

6.3.1 Explicit Device Selection

Explicit device selection allows the tester to specify exactly what devices should execute a set of commands. Devices can be chosen based on static criteria, such as the characteristics specified in the device templates, or with dynamic properties that are evaluated at run-time. Table 6.2 shows an overview of the explicit device selectors provided by XD-Testing. Devices can be selected by id, size, or type as specified in a scenario. These criteria are static and do not change at run-time. Alternatively, devices can be selected based on the UI elements that they display: The `selectByElement` selector will return all devices where a given HTML element is visible. The selection criterion needs to be a valid CSS selector. All of these device selectors return a set of zero or more matching devices. In addition, there is a `selectAny` selector that limits the set

to a single arbitrary device or throws an error if the set is empty. This selector allows the developer to ensure that a command is executed on a single device only. When it is nested in another selection, it can be used to choose a single device from a set of devices that already match some criterion. Listing 6.4 illustrates such a use case. First, all tablets are selected with a `selectByType` selector and a button is clicked on each device. Then a single device is selected among the tablets with the `selectAny` selector and another click command is triggered.

Command	Criteria Category	Criterion
selectById	Device characteristic	One or multiple device Ids
selectBySize		One or multiple screen sizes
selectByType		One or multiple device types
selectByElement	Dynamic	A element selector
selectAny		None, arbitrarily chosen

Table 6.2: Explicit device selection commands.

```
1 .selectByType("tablet", tablets => tablets
2   .click("button")
3   .selectAny(device => device
4     .click("button")
5   )
6 );
```

Listing 6.4: First, all tablets are selected with a *type* selector (line 1), then the selection is constrained to a single device with the *any* selector (line 3).

Explicit device selectors accept a second optional callback to address the *complementary device set*. This set includes all devices *not* matching the selector. The original device set can thus be split into two sets with a selector. The set in the first callback includes all devices matching the selector while the second callback includes the original set minus these devices. The complementary callback can be useful in combination with the `selectAny` selector. It allows the tester to trigger a command on one device and check results on all other devices. Listing 6.5 illustrates such a use case with the video player application. When it is used with multiple devices that act as controllers, the state of the playback buttons should be synchronised across all controllers. The test case selects all devices that act as controllers based on the elements that are displayed. Then a single device is chosen with the `selectAny` selector and a click of the play button is triggered. Using the complementary callback, all other devices are checked to see if the button is correctly showing the *pause* state. The test will fail if any of the devices does not comply within the given timeout.

```
.selectByElement("#controls", controller => controller
  .selectAny(
    one => one
      .click("#play"),
    rest => rest
      .waitForVisible("#play=Pause", 2000)
  )
);
```

Listing 6.5: Example usage of the complementary device set.

```
.implicitAny(device => device
  .click('li=My Video')
  .getText('#title')
  .then(function(value) {
    assert(value === 'My Video');
  }));
```

Listing 6.6: A simple test case using the `implicitAny` selector. Each command is executed on exactly one device.

6.3.2 Implicit Device Selection

Implicit device selection removes the need to know about the device configuration that is used in a specific test. Instead, it allows the tester to specify a test independent of the devices used and lets the system choose appropriate devices for each command. The system analyses each command and finds the set of devices that are able to execute it based on the availability of the UI elements required to execute the command. For example, if the play button should be clicked, the system finds all devices that show the play button and only triggers a click on these devices. We define two implicit device selectors and define their semantics as follows.

- `implicitAll`: The system ensures that each command is executed on *at least one* device. Each command is guaranteed to be executed on all devices that match the selector given in a command.
- `implicitAny`: The system ensures that each command is executed on *exactly one* device. While `selectAny` chooses a device and then tries to execute all commands on the same device, `implicitAny` chooses a suitable device for each command.

For both selectors, the device will throw an error if it does not find a matching device for a given command. Listing 6.6 and 6.7 illustrate the two selectors. Using `implicitAny` the first listing triggers a click on a list item with the content *My Video* on an arbitrary device that has that element. Then, the value of the title UI element is checked whether it displays that same content. The check is also executed on an arbitrary device granted that it displays that element. In contrast, the same test written using `implictAll` triggers the click on all devices with that list item and the check is repeated on all devices with the title item. As a result, we cannot assume a single return

```
.implicitAll(device => device
  .click('li=My Video')
  .getText('#title')
  .then(function() {
    var values = Array.prototype.slice.call(arguments);
    values.forEach(value => assert(value === 'My Video'))
  }));
```

Listing 6.7: A simple test case using the `implicitAll` selector. Each command is executed on one or more devices.

value for commands that provide a result (such as `getText`) within a `implicitAll` selector. Each text could potentially display a different value and each value needs to be checked individually, resulting in slightly more complex code. The two selectors could also be combined to first click the list item on exactly one device and then verify that the title is correct on all devices that have the title element.

6.3.3 Device Set Commands

XD-Testing provides additional commands to control a device set.

- `getCount` returns the number of devices contained in the device set.
- `forEach` provides access to each individual device in a set and accepts a callback with two parameters. The first parameter provides an individual device and allows the tester to execute commands on it. The second parameter gives a consecutive index for each device, i.e. the first device is identified by 0, the second by 1, etcetera.

`getCount` could be used to verify that the correct number of devices show a given UI element for a given device scenario. With `forEach` the tester has fine-grained control over all devices and can execute arbitrary commands.

6.4 Flow Recording and Visualisation

In addition to the formal tests that can be specified with the DSL, we also provide tool support for more informal checks based on visual properties. We record screenshots of the application during the test execution at points of interest marked by the tester. The recorded screenshots are presented in a visualiser so that the tester can verify the application behaviour visually. We define a *flow* as the compound assembly containing information about the participating devices, the executed commands, and recorded screenshots.

6.4.1 Recording

Each test execution generates a JSON representation of the flow which is stored in a `.json` file. Since the file contains all necessary information for the visualisation, it can be transferred and shared with other developers. We provide the following commands for recording flows.

- `name(name)` allows the developer to specify a unique and human-readable name for a flow recording. It is used to identify a flow in the visualiser tool.
- `checkpoint(name)` marks interesting checkpoints in the test execution. At each checkpoint, the recording generates a screenshot that is associated with the checkpoint name. Checkpoint names for a single device need to be unique, but the same name can be used across multiple devices. Screenshots originating from different devices with the same checkpoint name are associated in the visualiser for comparison.
- `addErrorCheckpoint()` marks a flow as failed. Failures in a flow are highlighted in the visualiser so that they can be spotted easily. In addition to marking failures individually, this command also allows a developer to add error checkpoints globally by creating a test hook that checks after each test if a failure has occurred (Lst. 6.8).

```
1 afterEach(function() {
2   if (this.currentTest.state == 'failed') {
3     return devices
4       .addErrorCheckpoint()
5   }
6 });
```

Listing 6.8: Adding error checkpoints globally.

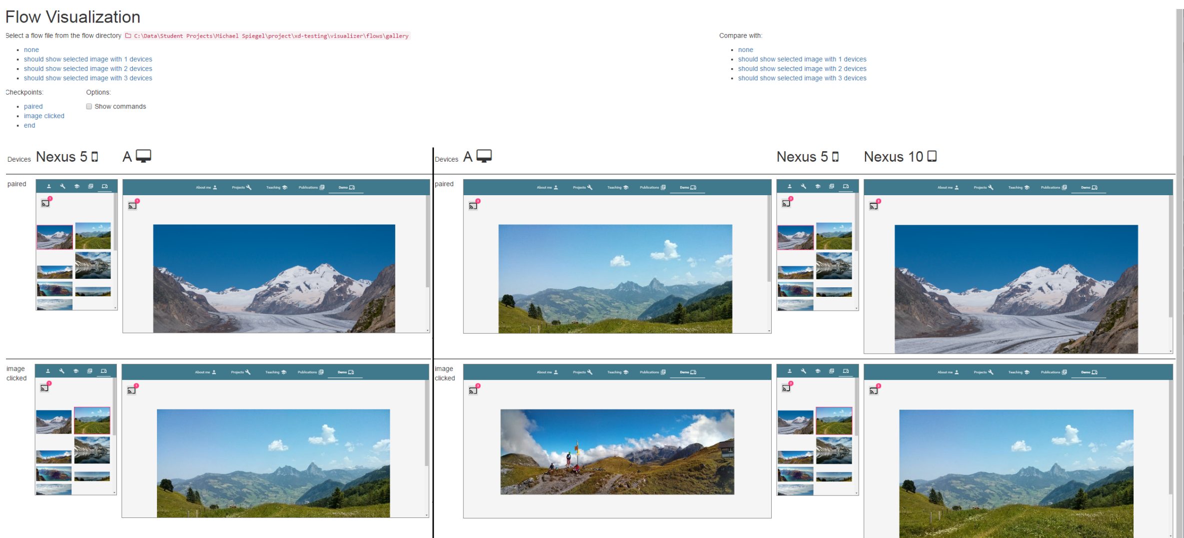


Figure 6.1: Two flows being compared with the visualiser: The one on the left has been generated with a scenario consisting of a phone and a larger screen, the one on the right additionally contains a tablet.

6.4.2 Visualisation

XD-Testing comes with a basic visualiser for recorded flows (Fig. 6.1). The tool lists all flows that are available in its directory. Each device that is part of a flow is represented by a column. A row is added for each checkpoint, showing the checkpoint name and the captured screenshot. Optionally, all executed commands can be shown, which could be of particular interest for failed flows. Failed flows are marked in the flow list so that they stand out. Note that a flow may hit no failure checkpoint but still be considered failed due to its visual properties. In that case, it will not be marked and it is up to the tester to notice the problem. Multiple flows can be loaded into the visualiser and be compared side by side. This is of particular interest when they represent the same test executed on different device scenarios. For example, this allows a flow recorded on two phones to be contrasted with a flow recorded on a phone and a tablet. Alternatively, it could support finding regressions when the same scenario is used with different versions of the source code.

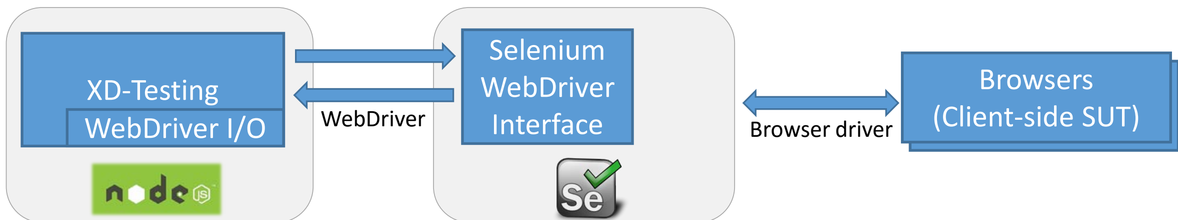


Figure 6.2: The architecture of XD-Testing: XD-Testing is run as Node.js application (left). It communicates with a Selenium server (middle) using the WebDriver protocol. Selenium controls the browsers that run the system under test (right).

6.5 Architecture and Implementation

XD-Testing has been built for UI testing of web-based cross-device applications. Hence, its architecture (Fig 6.2) has been designed to enable XD-Testing to control multiple browser instances simultaneously which run the client-side portion of the system under test (SUT). XD-Testing has been implemented on top of WebdriverIO and is executed in the Node.js runtime. WebdriverIO communicates with a Selenium server over the WebDriver protocol[3]. The Selenium server can control any number of browser instances and is able to execute operations on the browser instances, for example triggering a click command or reading a label. Selenium is not restricted to a particular browser vendor, however, it requires a separate browser driver for each vendor. Thus, XD-Testing can support any browser in its scenarios that Selenium can handle. XD-Testing loads a browser instance for every device specified in the scenario.

XD-Testing is an extension to WebdriverIO and can act as a replacement for it. It is backwards compatible, so that tests written for WebdriverIO can also be executed in XD-Testing. This supports adding cross-device support to legacy single-device applications with existing tests. Existing tests cases can be kept and executed with XD-Testing while new ones specific to the cross-device usage can be added later. XD-Testing adds cross-device abstractions to WebdriverIO and introduces some additional tracking to enable the implicit selectors and screenshot generation.

XD-Testing can be used in combination with existing test automation frameworks. Such frameworks can be used for development practices where testing is a central component, for example with continuous integration. In continuous integration, a commit of a code change to a central repository automatically triggers the execution of a test suite. This approach can lead to regressions being caught earlier. XD-Testing itself was built with continuous integration and used the Mocha[4] test runner, which was also used for the example test cases in the case study.

We have designed XD-Testing to be agnostic of any specific cross-device application framework. At the same time, we see merit in giving the tester access to certain features of a cross-device framework. For example, it may be useful to query the number of connected devices, information usually contained in the internal state of the cross-device framework used for development. For this reason, XD-Testing can integrate facades that provide access to cross-device frameworks. This mechanism makes internal state of these frameworks available in test cases. Functionality that is common to

[3]`https://w3c.github.io/webdriver/webdriver-spec.html` Accessed on 05.05.2017
[4]`http://mochajs.org/` Accessed on 05.05.2017

multiple frameworks can even be abstracted so that the implementation framework can be dynamically switched without the need to update the test code. We have implemented a facade for XD-MVC that provides access to various functionalities, such as device pairing, querying connected devices, or handling events defined in the cross-device framework itself.

6.6 Case Study

We have conducted a case study to evaluate XD-Testing. We have written tests for a cross-device gallery application to analyse how XD-Testing can be used to cover the needed tests. The case study consisted of two main parts. First, we worked with an existing code base of a gallery application written by the thesis author which uses the XD-MVC framework. We highlight some sample test cases that were written for this version. In a next phase, we evaluated a gallery application with a similar, but slightly different feature set that was implemented by students as part of our web engineering lecture. This version was written in plain JavaScript, using no cross-device framework.

6.6.1 Gallery Application and Sample Test Cases

The gallery application in the first phase implemented the following behaviour. The gallery displays a number of photos. The application can be used with a number of viewers and one controller device. Viewer devices show one photo each in large, while the controller can be used to select photos to be shown from a list. If more than one viewer device is paired to the same controller, it shows subsequent photos from the list. The controller interface is dynamically assigned to the smallest device among all paired devices. If only a single device is present, it will assume both the controller and viewer roles so that the application is fully functional at all times.

We have written test cases that verify this behaviour and present two examples here. The first test (Lst. 6.9) checks that the controller is displayed on the smallest device. This is done by instantiating a scenario with expected devices, in the example a phone and a desktop computer (lines 2 to 4). The scenario is loaded (line 6) and the devices are paired using the cross-device framework facade (line 7). Then, an explicit selector is used to find the smaller device of the two (line 8). Finally, the test waits for five seconds for the controller UI component to become visible (line 9). If it does not, the test will fail.

This test checks one expected device combination where the devices and their characteristics (for example the expect size) are known to the tester. The test benefits from the device templates that make it easy to instantiate a device with certain characteristics and compose multiple devices into a scenario. The test demonstrates the use of explicit device selectors that eliminate the need to keep a reference to each device. Instead, the devices are managed by XD-Testing. A device could be swapped out or added easily without having to replace the actual test (lines 6 to 12) and adding a new device only requires another line of code.

The second test makes no assumption about the device configurations that will use the application. It uses an implicit selector to test application behaviour independent of the devices used. The application should always display the photo that is selected

```
it('should show controller on the smallest device', () => {
  let deviceSet = {
    A: xdTesting.templates.xsmall.nexus4(),
    B: xdTesting.templates.desktops.chrome()
  };
  return devices = xdTesting.multiremote(deviceSet).init()
    .app().pairDevicesViaURL(baseUrl)
    .selectBySize('xsmall', small => small
      .waitForVisible('.controller', 5000)
    )
    .end()
});
```

Listing 6.9: This test creates a scenario with a phone and a desktop device. It checks that the controller is displayed on the smallest device.

```
xdTesting.loadScenarios().forEach(scenario => {
  it('should show selected image', () => {
    return devices = xdTesting.multiremote(scenario.devices).init()
      .app().pairDevicesViaURL(baseUrl)
      .checkpoint('paired')
      .implicitAny(device => device
        .waitForVisible('.controller', 5000)
        .click('.controller img:nth-of-type(2)')
        .waitForVisible('.viewer img[src="img/album/large/02.jpg"]', 3000)
        .checkpoint('image clicked')
      )
  })
});
```

Listing 6.10: This tests loads scenarios from a configuration file. For each scenario, it checks if a photo selected on a controller is displayed on a viewer device.

on the controller device, independent of the number and type of devices that are used. Listing 6.10 verifies this behaviour. For this test, random scenarios have been generated using the command line tool. The scenarios are loaded from the generated file and the following test code is repeated for each scenario (line 1). As before, each scenario is instantiated (line 3) and the devices are paired (line 4). After pairing, a checkpoint is set (line 5) so that the initial state is captured in a screenshot for the flow visualiser. Then, an implicit block is started with the `implicitAny` selector (line 6) which ensures that the following commands are executed on exactly one device. The test waits for the controller UI component to become visible (line 7) and, once it is visible, triggers a click on the second photo in the list (line 8). Then, the test waits for any viewer device to show the photo (line 9). Finally, another checkpoint captures the final state in a screenshot (line 10).

This second test would be even more challenging to write without XD-Testing than the first one. Without the `implicitAny` command, the tester would need to find the

```
$ ./node_modules/.bin/mocha test/galleryTest.js

  GalleryRemote
    √ should show the first image after connecting a screen (9386ms)
    √ should clear a screen after disconnection (9598ms)
    √ should show next image on additional screen (12002ms)
    1) should redistribute images after unpairing

  3 passing (44s)
  1 failing

  1) GalleryRemote should redistribute images after unpairing:

      Uncaught undefined: [ 'http://localhost:7070/images/1.jpg' ] deepEqual [ 'http://localhost:7070/images/0.jpg' ]
      + expected - actual

       [
      -  "http://localhost:7070/images/1.jpg"
      +  "http://localhost:7070/images/0.jpg"
       ]
```

Figure 6.3: A test suite with four tests, 3 passing and one failing.

appropriate device for each command in lines 7 to 10 before the command can be executed. Furthermore, the commands would need to be properly synchronised. This would make the code longer, more complex, and harder to read. With XD-Testing, the tester can focus on the application behaviour while the library handles the identification and synchronisation of the devices. Both test cases use the framework integration for pairing devices, resulting in a single line of code for this common step.

6.6.2 Evaluation of Web Engineering Exercise

As part of our web engineering class, students were asked to implement a simplified version of the gallery application. In contrast to the previous version, the viewer and controller roles were not assigned dynamically, instead each interface could be loaded with a different URL. The application did not use any cross-device framework and students were free in their implementation, as long as it met the specifications in the exercise description. However, they all started from the same code skeleton. We created 9 test cases verifying behaviour described in the specification. The tests were executed on the reference solution that was implemented by the thesis author. All tests passed, demonstrating that there were no false positives in the tests and that the reference solution indeed meets the requirements.

After the exercise had been graded, we asked students to voluntarily send us their code, in particular if they had any issues. We tested three different solutions that we received and that were reported to have issues. While they all covered the basic functionality, we were able to confirm the issues with the tests. The issues were edge cases and concerned either the re-distribution of images when viewer devices connected or the incorrect handling of more than one session of paired viewers and controllers. Figure 6.3 illustrates the output of a test suite of four tests. The first three tests have passed, while the last one has failed. The test output indicates that there is an issue with an image source not being equal to the expected one. In cases like this one, the visualiser can provide further help. Figure 6.4 shows the visualiser for a failed test. Failed tests are marked in the overview. In the first row, two viewers (*ScreenA* and *ScreenB*) are paired with a controller (*RemoteA*) and the photos are distributed correctly: *ScreenA* shows the selected photo and *ScreenB* the next one in the list. When *ScreenA* is disconnect in the second row, *ScreenB* is expected to update and show the

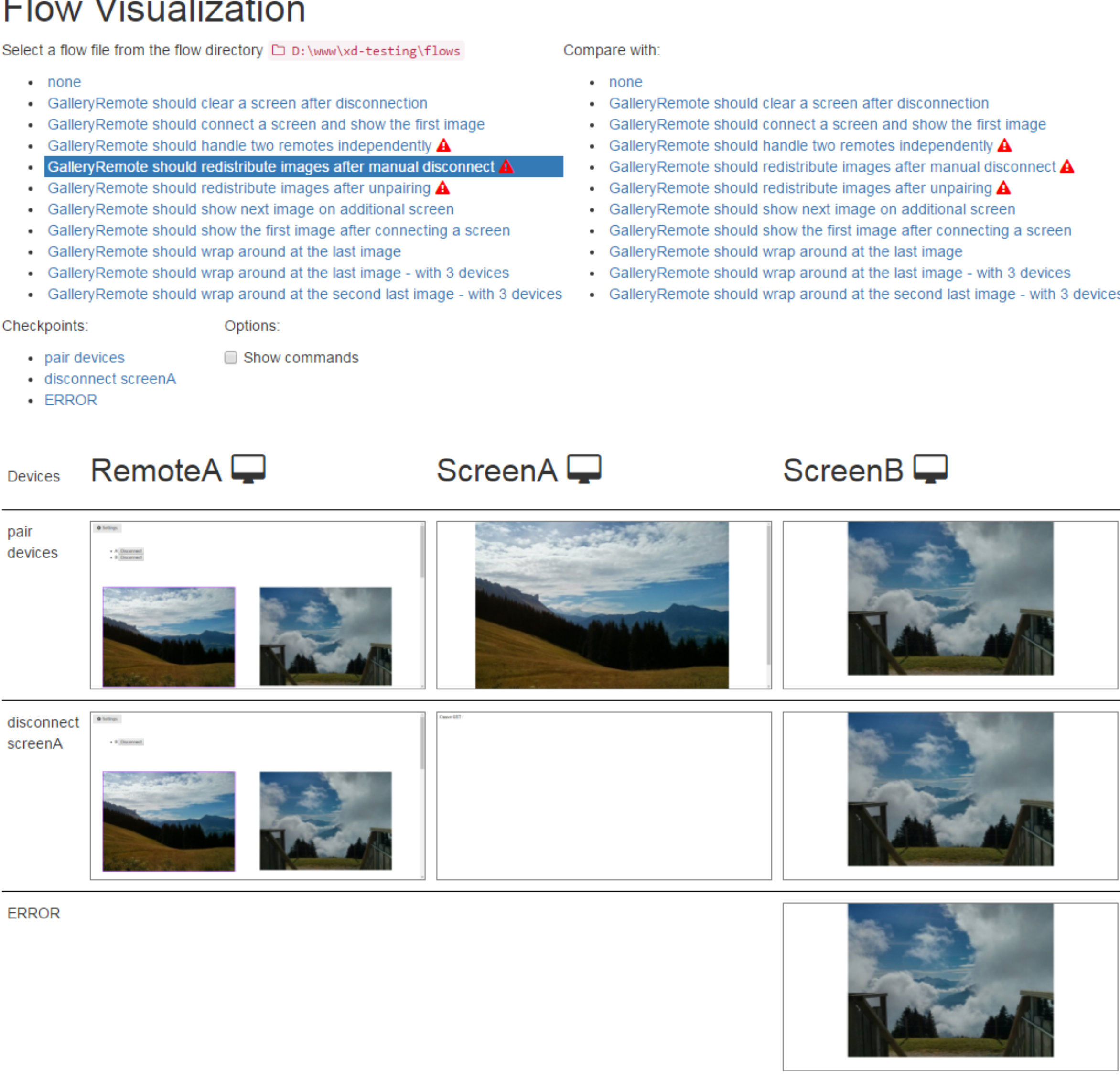


Figure 6.4: The visualiser displaying a failed test.

selected photo. However, it remains unchanged and the tests thus fails. The failure is represented with an *ERROR* checkpoint that is automatically created and represented in the third row. While the test output indicated that there is a problem, the visual representation aids the comprehension of the underlying cause of the problem.

As XD-Testing is a library for end-to-end testing, little knowledge of the implementation is required to write test cases. This was confirmed by the fact that the same test cases could be re-used across the reference solution and the three student solutions with small tweaks. Mainly CSS selectors needed to be adapted to identify the correct UI elements which may vary slightly between implementations.

6.7 Discussion

With XD-Testing, we had the goal of reducing the effects of the fragmentation and the many possible device combinations that we have identified as challenges in cross-device testing. The case study has confirmed that XD-Testing facilitates verifying cross-device applications. We were able to compose test cases that check complex cross-device-specific application behaviour. Tests that were hard to express with existing tools became more concise and easy to understand. The explicit selectors allow the application to be verified with expected device combinations, for example to check whether the UI distributes correctly across the devices. The implicit selectors enable tests that check functionality independent of the devices used. When combined with the device scenarios that can be randomly generated by XD-Testing, implicit selectors allow the tester to cover a wide range of possible device combinations, even unexpected ones.

In the case study, we also found directions for future extensions of XD-Testing. First, the explicit device selectors are limited to a small set of characteristics and to one characteristic at a time. While nesting allows the combination of multiple selectors to a certain degree, the code can become complex and hard to understand if the nesting is too deep. Furthermore, some selections cannot be expressed with nesting, for example the selection of the smallest device in a scenario. Such selections could be enabled with a more sophisticated query language that could allow queries such as *find the smallest device with a touch screen*.

Our case study was carried out with emulated devices. Device types were differentiated mainly based on different screen sizes. However, there are other factors that could impact a test, for example computing power that affect wait times or interactions that rely on sensor only available on some device types. Thus, we also see benefits in using real devices with differing hardware for tests. As XD-Testing builds on the standardised WebDriver protocol, the tests could be executed on any device that support the protocol. The selectors are independent of a device being emulated or real. However, the device templates have been built for emulated devices and the composition of the scenarios would need to be adapted to take into account available real devices.

The visualiser has been built with the goal of providing additional information for failed tests and to allow the tester to inspect the application visually. This could, for example, allow them to compare the application in different scenarios and to find issues that were not covered in any test. The visualiser was optimised to compare at most two scenarios at a time. While more scenarios could technically be displayed simultaneously, we suspect that this could lead to visual overload. Other means to compare a larger number of scenarios visually could be explored in the future.

7 XD-Analytics: Usage Analysis

After an application has been deployed, monitoring its usage can provide interesting insights. Usage analysis products such as Google Analytics promise insights into user experience and behaviour and allow business goals to be measured. These can be used to shape a product in a next iteration or to help choose between different designs in A/B testing. Existing analytics solutions take into account the different device types, as user behaviour may vary between them, however they have not been built with cross-device applications in mind. In this chapter[1], we explore how analytics tools need to be adapted to provide insights tailored to cross-device applications. We illustrate how a cross-device analytics tool might be used by listing interesting questions one might ask about cross-device usage of an application that cannot easily be answered with conventional tools. Next, we introduce a set of metrics for cross-device analytics. We have implemented these metrics in the tracking and visualiser tool XD-Analytics. We give a short overview of the tool and explain how it can be used to answer the questions from the use cases. Then we present the architecture of the tool before we report on our experiences using it in a case study with educational software in the wild. Note that some of the screenshots in this chapter have been edited to give better readability. This chapter was developed in [160].

7.1 Cross-Device Analytics Use Cases

In the following list, we present use cases for cross-device analytics. We list interesting questions that could inform a next iteration in the design and development of a cross-device application that cannot easily be answered with conventional analytics tools.

U1 – Detecting cross-device usage Is there any cross-device usage of the application? How much time is spent online with cross-device usage and how much with single device usage? If a cross-device application is only used with a single device, efforts could be made to better educate the user about cross-device functionality.

[1] Earlier versions of parts of this chapter were originally published as Husmann et al. [79, 80].

U2 – Detecting device combinations What devices are typically involved in parallel usage? Are there any unexpected combinations (e.g. three smartphones) that the application could be optimised for? Device combinations that are detected could be optimised for in the design. Furthermore, this information could also be fed back into the testing phase where these combinations could specifically be tested.

U3 – Localising cross-device usage within an application Which part of the application is commonly used with multiple devices? When transitioning from a single-device to a cross-device application, it is possible that cross-device support has only been enabled for certain features of the application. Localising the cross-device usage could help to confirm that the application is used as intended. Furthermore, it could be the case that only certain features are suited to cross-device usage whereas others might be used mostly on a single device. Data on usage could inform a next design iteration of the application.

U4 – Finding patterns by device combination What do users who combine a lot of devices do? What do users with specific combinations of devices (e.g. two tablets) do? Answers to these questions could help tailor specific functionality to specific device combinations.

U5 – Preparing for the move from a single-device to a cross-device application What devices in general are used to access the application? For an application without cross-device support, can we find indications where it could benefit from introducing cross-device functionality? For example, if we can detect the parallel use of an application without cross-device support, this could be an indication that the user might benefit from adding such support and provide some first insights into what kind of cross-device features might be useful.

7.2 Metrics

To answer the questions from the previous section, we propose that the following metrics are tracked in a cross-device application.

- **Devices, users, and sessions** The two most important entities to be recorded are devices and users. To identify that the same user is using multiple devices, a user concept is required. As soon as a user is logged in, the device can be associated with the user. Additionally, we propose that open tabs are tracked as well in what we call *sessions*. An application opened in multiple tabs on the same device could indicate a need for more screen real estate. In summary, a user can have multiple devices which can have multiple sessions.

- **Device types** Device types provide information on the kind of devices that use an application. Variations of this metric are also tracked in conventional analytics tools where devices are often classified as phones, tablets, or desktop computers. In our prototype, we classify devices into five categories (xs, sm, md, lg, xl) according to their screen size.

- **Device combinations** When multiple devices are used in parallel, we can record the combination of device types that are used, for example two small devices and a large device.

- **Device cardinalities** The device cardinalities count the number of devices in a combination regardless of the device type.
- **Location combinations** When devices are used in combination, the URL locations that are viewed on each device can be associated. The locations are typically an indication of the content, state, or functionality of the application that a user is accessing.
- **Combined views ratio** The combined views ratio is calculated by dividing the number of views of a location as part of device combination by the total number of views. This will result in a high number for locations that are mostly used in a cross-device setting and a low number for those that are mostly accessed from a single device.

7.3 XD-Analytics

We have implemented these metrics in a cross-device analytics tool. The proof-of-concept tool consists of a tracker that needs to be integrated into the application under inspection and a visual UI for the analysis of the collected data. We first give an overview of the tool and explain how it addresses the use cases introduced above. Then we provide some details on the architecture and implementation of the tool.

7.3.1 Overview

Figure 7.1 provides an overview of the UI of XD-Analytics. Bar charts, time line charts, and ordered lists can be used to view the data in the upper part of the UI. Below, the analyst can filter the data and drill down into aspects of interest. The interaction has been designed on principles from faceted search. Figure 7.2 illustrates a facet for device types. When a value in a facet (e.g. a small device) is selected, the other facets update to only reflect values that also match the selected facet (e.g. locations that were visited with small devices). XD-Analytics can provide answers to the questions in the following way.

U1 – Detecting cross-device usage Looking at device cardinalities (Fig. 7.3) gives a first indication. Any cardinality higher than one denotes cross-device usage. A chart can be created that displays aggregated time online for specific device cardinalities (Fig. 7.4). A similar chart can be created with average time online which could be used to determine if users with multiple devices spend more or less time with the application than single device users. Time spent online could be used as an indication of engagement.

U2 – Detecting device combinations As this directly corresponds to a metric, questions regarding device combinations can easily be answered by inspecting the data in either the list or bar chart (Fig. 7.5) format.

U3 – Localising cross-device usage within an application The analyst can investigate this by inspecting location combinations (Fig 7.6). The data is presented in descending order so that the most frequent location combinations are shown first. The combined views ratio highlights locations that are more frequently used with multiple devices than with a single device. Furthermore, the analyst can enter location patterns

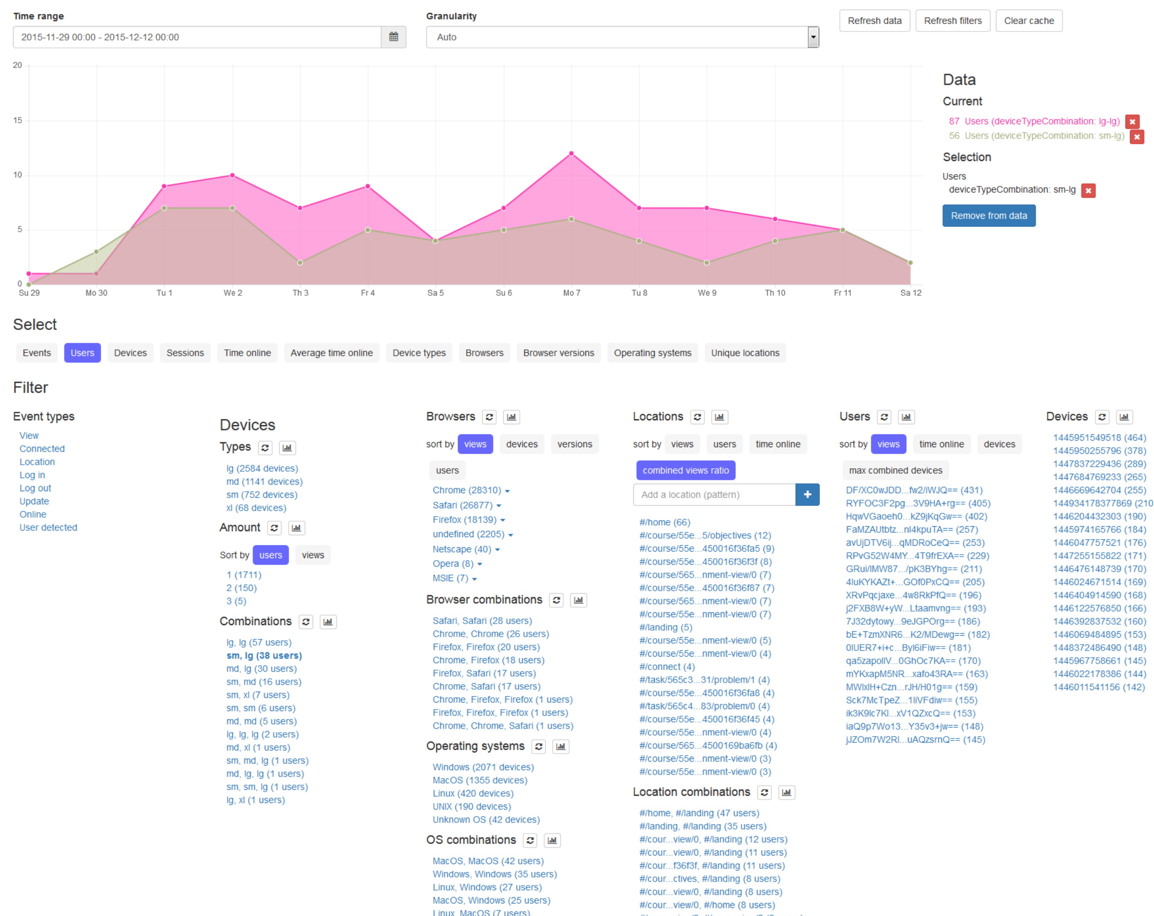


Figure 7.1: An overview of XD-Analytics.

Devices

Types

lg (2584 devices)

md (1141 devices)

sm (752 devices)

xl (68 devices)

Figure 7.2: A facet in XD-Analytics.

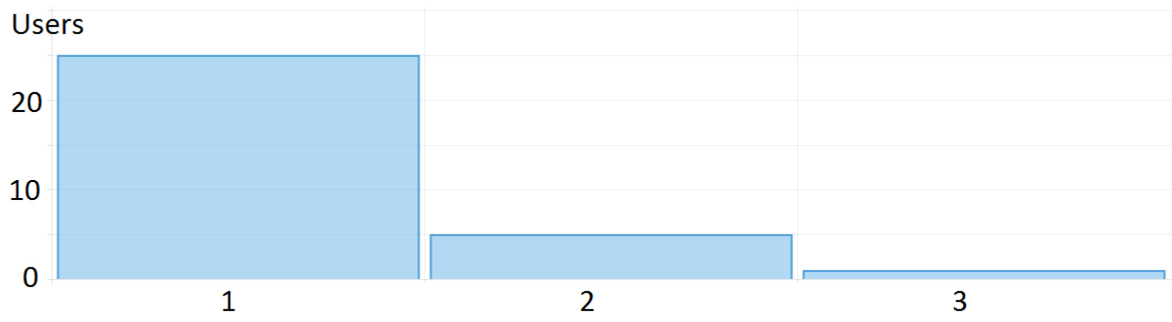


Figure 7.3: Device cardinalities

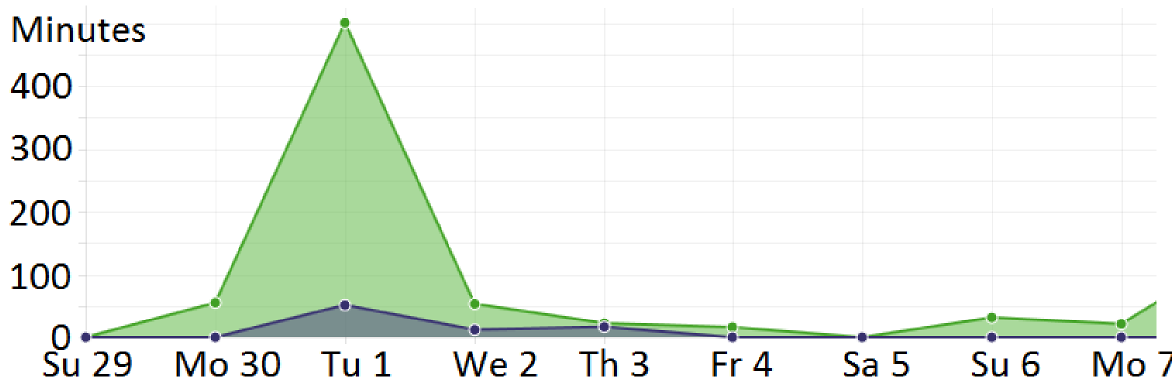


Figure 7.4: Time online for single device usage (green) and for combinations of two devices (dark grey)

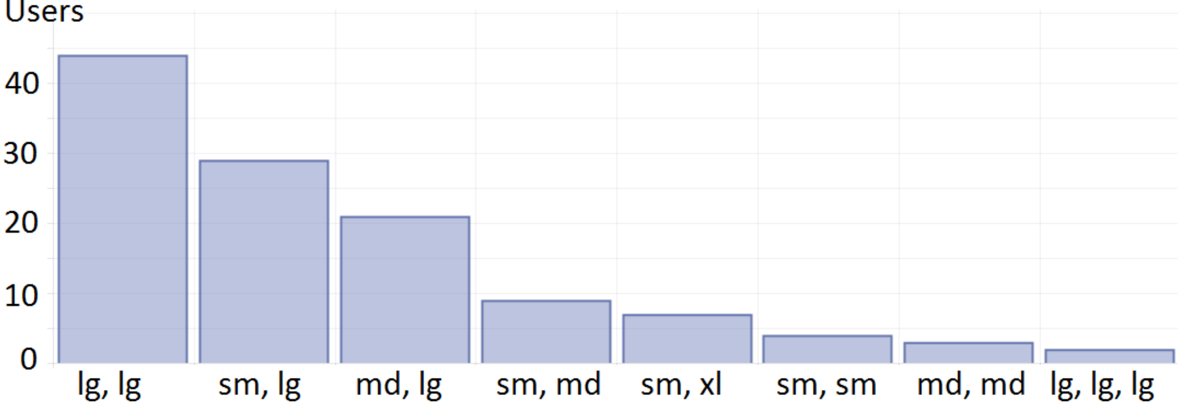


Figure 7.5: Device combinations

Location combinations

#/home, #/landing (47 users)
#/landing, #/landing (35 users)
#/cour...view/0, #/landing (12 users)
#/cour...view/0, #/landing (11 users)
#/cour...f36f3f, #/landing (11 users)

Figure 7.6: Location combinations, showing 47 users who accessed *home* and *landing* simultaneously.

using regular expressions and filter for these patterns. This restricts the analysis to certain locations within the application.

U4 – Finding patterns by device combination These questions can be answered by filtering the device combinations or cardinalities and then inspecting the locations. For example, the device cardinality could be set to the highest occurring number and then the locations and their combinations could be inspected. This would give an indication of what the user is doing with their devices. Likewise, a given combination (md, md) could be fixed and the process repeated.

U5 – Preparing for the move from a single-device to a cross-device application General information about the devices can be obtained by inspecting the device types, information on browsers, their versions, and operating systems used. If an application has no specific cross-device support, but parallel usage is detected (using the same methods as for U1), this could be an indication that they would benefit from added cross-device support. Similarly, if there are more sessions than devices and thus users have multiple tabs open with the same application, it could be worth investigating if users would benefit from moving the additional tabs to a second device and coordinating interaction with the first one. Inspecting the location combination could provide pointers to what functionality is used in parallel.

7.3.2 Architecture and Implementation

XD-Analytics consists of two main parts: a script that tracks the user in the client application and a server that collects the data and presents it to the analyst through a visual analytics interface. Figure 7.7 provides an overview of this architecture. The tracker is a JavaScript file that needs to be included in the client application. It tracks device characteristics, user interactions, and application state based on URL locations. The device type is derived from the screen size in CSS pixels[2]. While not entirely accurate for all devices due to differences among vendors, informal tests produced good results with mobile phones (Nexus 4, Samsung Galaxy Note 4) classified as small, a tablet (iPad Air) as medium, a notebook as large, and a 30inch screen as extra large. The data collected in the client is sent to the server in batch uploads. Every 30 seconds, the tracker checks if the user is still using the application and an event is generated to track the duration of the session. To associate devices with users, the application should

[2] `https://www.w3.org/TR/css3-values/#reference-pixel` Accessed on 12.05.2017

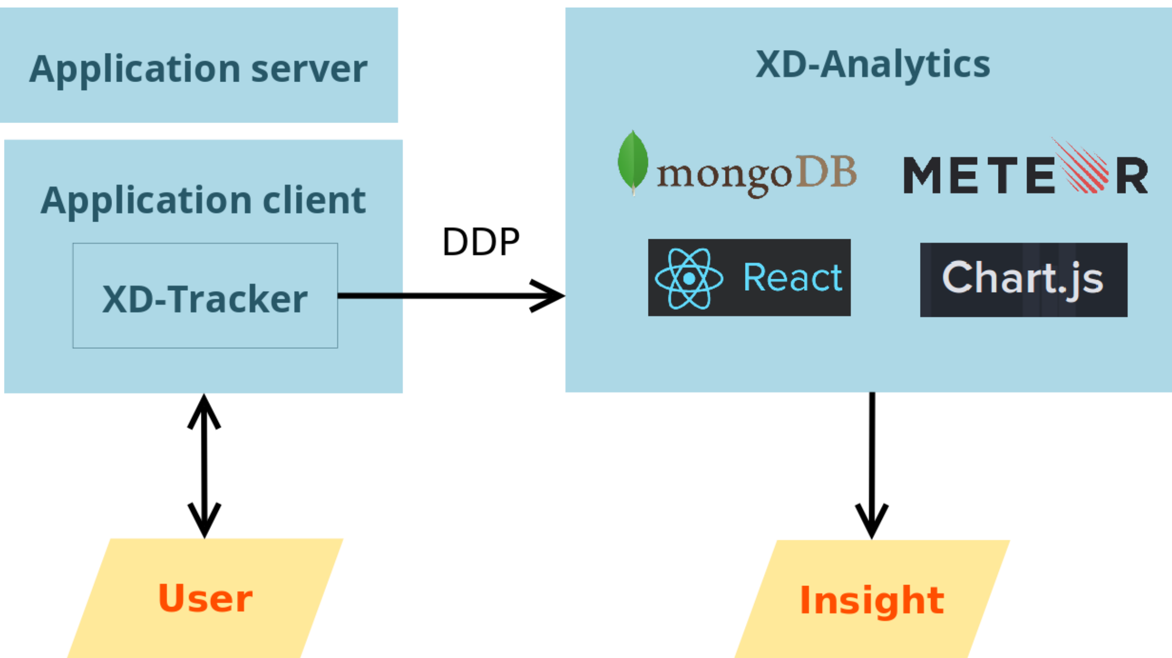


Figure 7.7: XD-Analytics architecture: A tracker integrated into the client application collects data and sends it to the analytics server.

supply a user ID on each device. This could be derived from any login mechanism that is used and it should be chosen so that personal identification of the user is not possible to protect their privacy. For example, instead of an email address a hash derived from the address could be used. The tracker assigns a unique, persistent ID to each device so that devices can be distinguished from each other and repeated visits with the same device can be recognised.

On the server, the data is preprocessed. The event logs received from the tracker are aggregated into 10-minute intervals for performance reasons. These metrics are aggregated by device and user, before the cross-device metrics are calculated. XD-Analytics has been implemented using the Meteor[3] platform. Both the tracker and the analytics client communicate with the server over websockets using the JSON-based Distributed Data Protocol (DDP) that is part of the Meteor platform. The data is stored persistently on the server using MongoDB. The analytics client has been implemented using React and Chart.js[4] components.

7.4 Case Study

We have used XD-Analytics to track the introduction of a cross-device feature into an application with an existing user base. The application had no cross-device support at the start of the case study. Over the course of 9 weeks, we recorded more than 3 million log entries. The case study was conducted with the Taskbase[5] application.

[3]`https://www.meteor.com/` Accessed on 12.05.2017
[4]`http://www.chartjs.org/` Accessed on 12.05.2017
[5]`https://www.taskbase.com/` Accessed on 12.05.2017

Taskbase is an educational platform that allows teachers to upload course material, such as exercise sheets or lecture slides. Students can give feedback on the exercises and rate their difficulty. At the time of the study, the software was used at Swiss schools and universities, including ETH Zurich with more than 3500 users from the mathematics department. ETH Zurich has its own installation[6] which was used in the work described here.

7.4.1 Cross-Device Extension

We used XD-Analytics to track the use of Taskbase before the introduction of the cross-device feature to get an impression of how the platform is used. Even though exercise sheets can be printed, instructors observed that students often access them digitally during exercise sessions using notebooks or tablets. This was reflected in our logs when we analysed access patterns. We also noticed that more than 2700 users accessed the platform with two different devices (not necessarily in parallel), while fewer than 400 used only a single device. Fewer than 100 users used three or more devices. In our own teaching, we have experienced students handing in hand-written exercises for marking digitally, by either scanning or photographing them. Based on this experience and the analysis of Taskbase usage, we decided to implement a feature that allows students to submit hand-written solutions by photographing them with their phone or tablet. The feature was designed for the following scenario. The student displays the exercise instructions on their laptop or tablet while they solve the exercise on paper. Once they are ready to submit the solution, they use their phone to scan a QR code that is displayed alongside the instructions (Fig. 7.8). This step takes them to an upload page where they are prompted to photograph their solution (Fig. 7.9). The solution is associated with the correct exercise automatically. The solution is then made available to the instructor for marking through the platform.

The feature was implemented in Taskbase and introduced in a class that the author of this thesis was teaching as an assistant. The class had previously not been using Taskbase. The platform was introduced and the feature was demonstrated in an exercise session. Students were encouraged to use it, but were still allowed to hand in their solutions in person or by email as they had done previously. In total 44 students were enrolled in that class and roughly 30 students typically attended the exercise sessions. Handing in assignments was voluntary (a master solution was provided) and generally done by 6 to 10 students to get personalised feedback.

7.4.2 Results

In the section, we present the data and insights that we have gathered in our case study using XD-Analytics. Note that all data presented here was filtered to only show users from that specific course. The filtering was done using regular expressions on the URL locations, a feature that is provided by XD-Analytics.

Figure 7.10 shows the morning when the feature was introduced in the course. From 8.15 to 10 students were in the lecture. They had received an email stating that the course material was available on the platform. During the lecture, there was some

[6] `https://e-lectures.ethz.ch` Accessed on 12.05.2017

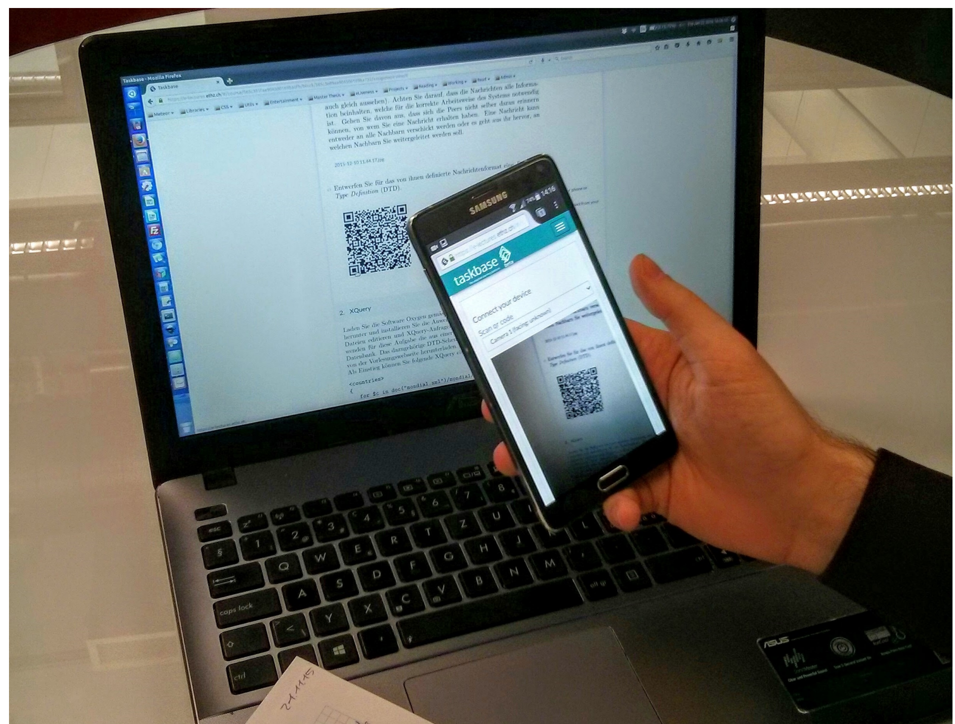


Figure 7.8: A Taskbase user connects their phone by scanning a QR code on the notebook.

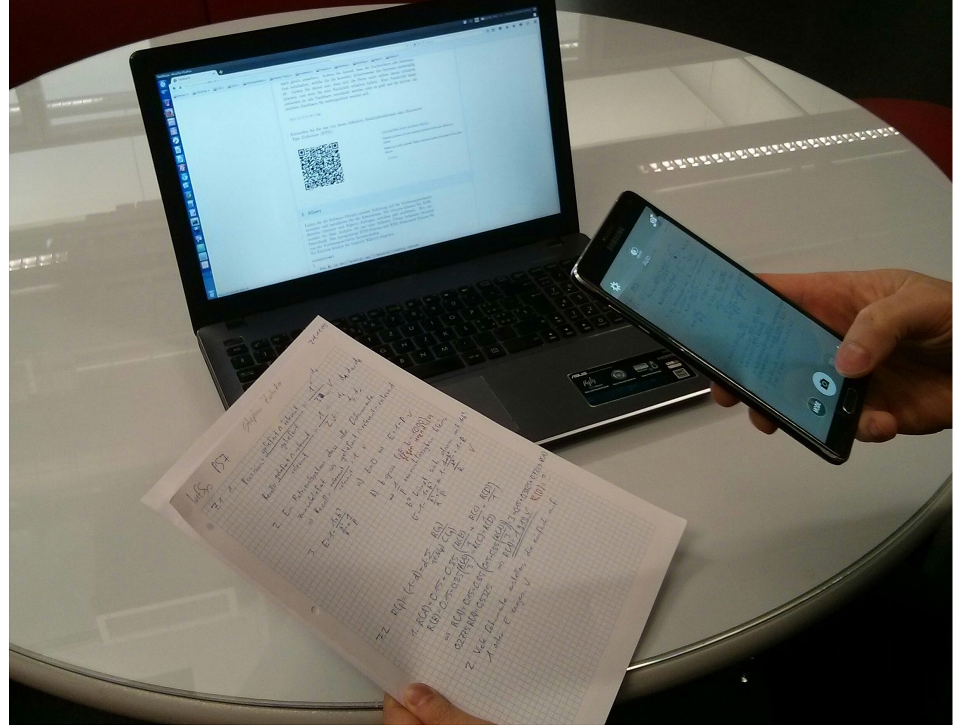


Figure 7.9: A Taskbase user submits their solution by taking a picture with their phone.

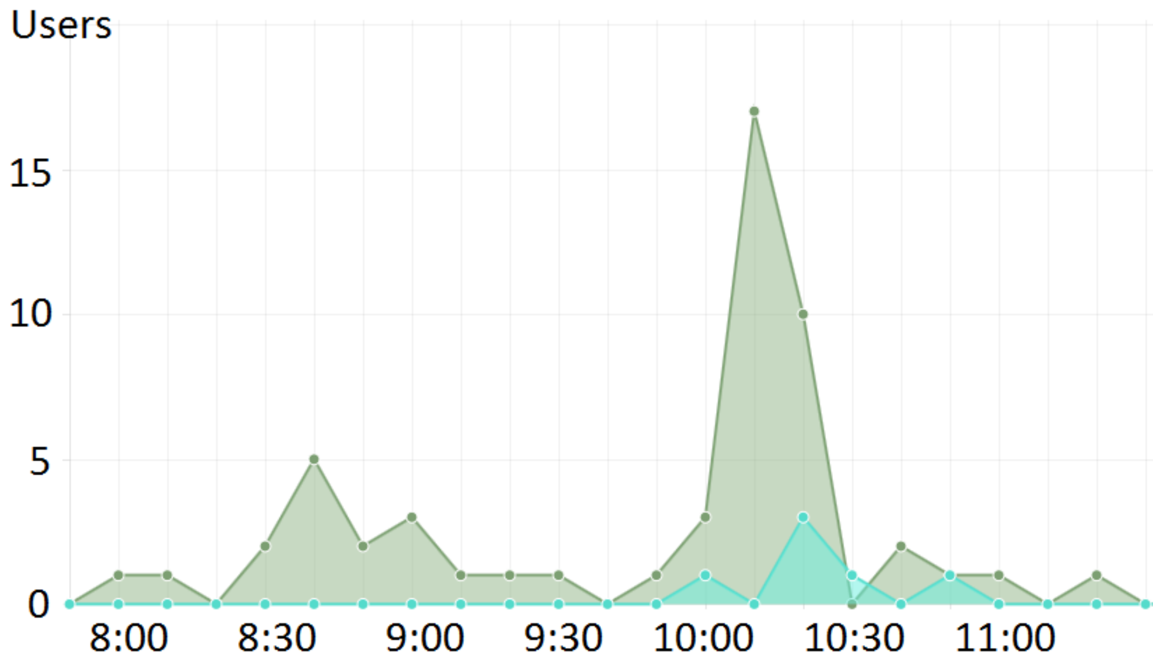


Figure 7.10: Device cardinalities for the morning of the feature introduction. The dark green line shows single device users, the light green one users with two devices.

activity on the platform. The activity spikes after 10. That is when the new cross-device feature was demonstrated in the exercise session that took place from 10.15 to 11. The dark green line denotes access with a single device, whereas the lighter green line shows device cardinalities of two. The data shows that most students used only a single device to access the platform, but a few used two devices in parallel. The data is line with our observation that only a few students tried the cross-device feature in the exercise session.

The students then had two weeks time to solve the exercise and hand in the solution. Figure 7.11 shows a day when two students handed in their solution using the added cross-device feature. The chart is filtered to only show device cardinalities of two to limit the data to users who used two devices in parallel. One handed in around 11 and the other one around five o'clock. This data matches the number of solutions that we received.

Figure 7.12 lists the device combinations that were tracked. The most common combination is a small and large device, matching the intended use with a phone and a laptop. When we removed the filter for the specific course and inspected the combined view ratio, we found that locations related to our course were listed first (Fig 7.13). Given that other courses had much higher student numbers (up to 10 times more) and as a result much higher activity in general, this observation shows that the metric is suited to locate cross-device activity within an application.

XD-Analytics helped us inform the design of the cross-device feature and allowed us to observe its introduction in the wild. The system let us quantify cross-device usage and track it over time. It showed that the feature was indeed used, but only by a small number of students. Partly, this can be explained by the fact that the study was carried out as part of a Master thesis and had to be done within tight time constraints.

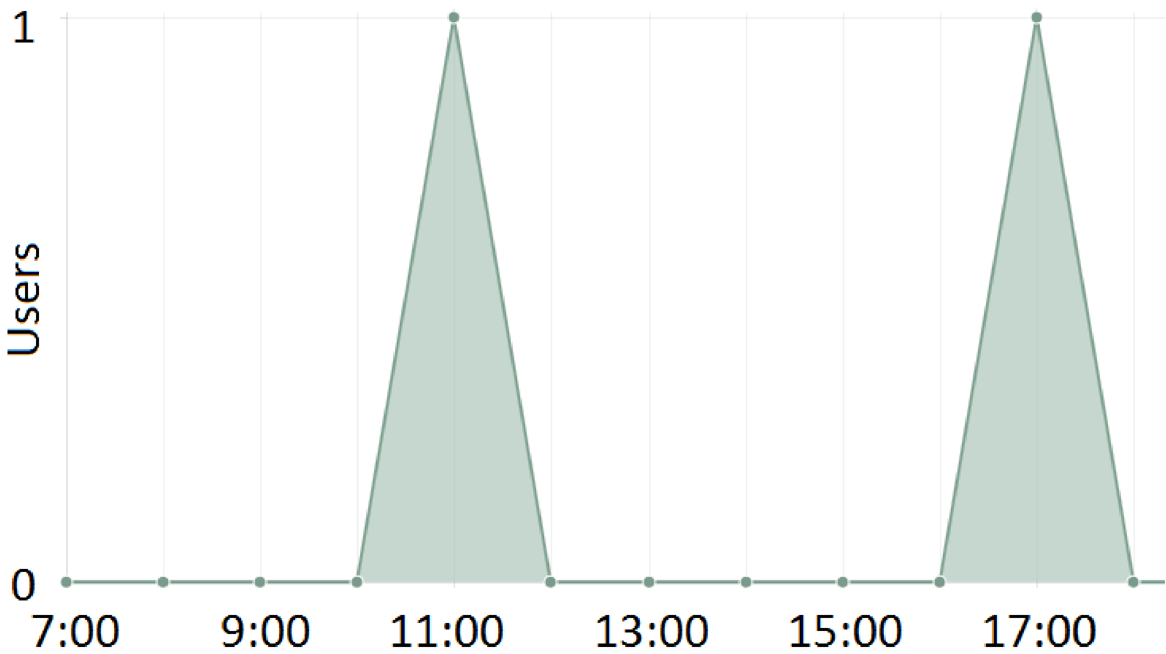


Figure 7.11: Cross-device usage for hand in. One student used the cross-device feature at 11.00, the other at 17.00 to hand in their solutions.

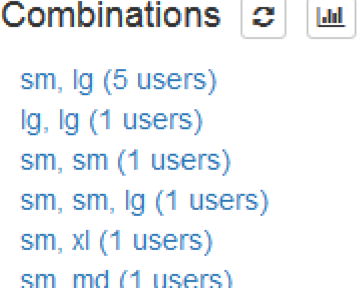


Figure 7.12: The device combinations observed in the class.

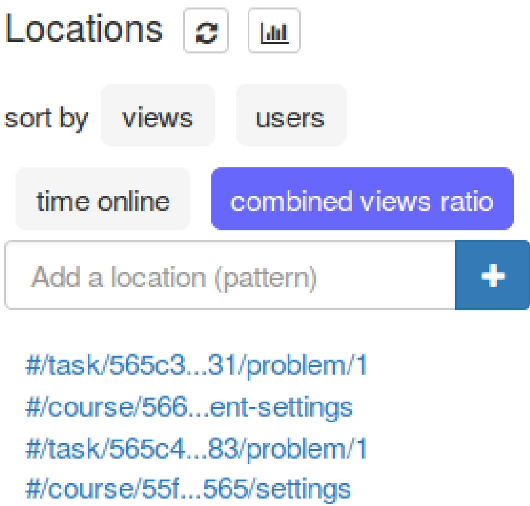


Figure 7.13: Locations sorted by the combined view ratio. Locations with high cross-device activity in relation to total activity are listed first.

While the Taskbase platform does have many users, we failed to find a professor who was already using the system to agree to take part in the study. At that time of study, the semester was almost over and they were reluctant to introduce a change so late in the semester. Hence, we decided to test the system with a class of this thesis' author. However, since only a few students had been handing in their exercises previously, as it was not mandatory, the low usage numbers were not surprising.

7.5 Discussion

In this chapter, we have introduced a set of metrics together with use cases for cross-device analytics. With our prototype implementation XD-Analytics and its use in a case study we have shown what we can learn about cross-device use in the wild. We have also identified some limitations of our implementation. First, we aim at detecting parallel usage of multiple devices based on a user concept. However, some applications do not include one. In its current state, XD-Analytics cannot identify parallel usage in such applications. Furthermore, if an application includes a user concept but applies a separate pairing mechanism, the explicit pairing will also not be tracked. That is, if the user is logged into the same application on two devices simultaneously, XD-Analytics will record it as parallel usage, whether the devices are paired or not. Similarly, cross-device usage between multiple users are currently not detected. A benefit of our approach is that is independent of any cross-device framework and pairing mechanism. It also allows applications with no cross-device support to be tracked. Furthermore, it is in line with the intuition behind what we understand by parallel usage: the same person using multiple devices simultaneously. However, if explicit pairing was needed, XD-Analytics could be extended with a pairing event to be sent to the server along with the rest of the logs. The calculation of the device combination could then be adapted to take pairing into account.

Tighter integration with existing cross-device frameworks could allow further ana-

lyses. User or device specific roles could be tracked as well. Other frameworks make use of the relative position of devices and/or users. Collecting and analysing such information could help us understand how the applications are used when deployed outside of lab environments.

We have classified the devices according to their screen size. Other categorisations could be used, for example phone or tablet. However, the device landscape is quite diverse. Some notebooks come with removable keyboard and can be used as tablets. Large phones are approaching small tablets in size. Rather than putting devices into these narrow categories, additional device characteristics could be recorded to provide more information on the type of device that is used. For example, information on input and output modalities could be recorded, as these characteristics are sometimes used to determine the distribution of the UI.

8 Conclusion & Outlook

Motivated by the increasing number of devices users have at their disposal, we investigated how applications can be built that make use of multiple devices in parallel. Our analysis of related work showed that tools for building cross-device applications focus on the design and prototyping stage and the implementation stage. Later stages in the development process, including testing and debugging, have received little attention in prior work. As a result, existing tools are suitable for building experimental prototypes but offer little support for building well-tested products. In this thesis, we filled that gap by analysing the different stages of the development process and providing appropriate tool support for each stage. Let us revisit the research questions from the Introduction.

- **RQ1** How well do existing tools support cross-device development across the whole development lifecycle?
- **RQ2** What are the requirements of tools specific for cross-device development?
- **RQ3** Can we provide better tools for cross-device development, in particular for testing, debugging, and usage analysis?

We identified inadequacies of existing tools that have been built with single-device applications in mind, answering research question RQ1. Essentially, these issues boil down to two major challenges that developers face in cross-device development: The many possible device combinations lead to a huge design space and make it difficult to develop, debug, and test the applications. The fragmentation of the logic across devices introduces challenges in particular during testing and debugging.

Next, we summarise our contributions for each stage in the development process. These address research questions RQ2 and RQ3. For **design and prototyping**, we introduced MultiMasher, a visual tool for creating functional cross-device prototypes from existing websites. The tool only requires limited technical knowledge as most interactions build on direct manipulation of the underlying website. Components can be

extracted from a website and combined with other components from the same or a different website. MultiMasher handles the inter-component communication across device boundaries. The tool enables experimentation by allowing the designer to easily add new devices and migrate components from one device to another using drag-and-drop. We also used MultiMasher as an opportunity to experiment with different architectures for state synchronisation.

For the **implementation** phase, we contributed the JavaScript framework XD-MVC. Using a wide range of sample applications that were built with the framework, we demonstrated its flexibility and utility. Its layered architecture allows developers to choose the level of support they need while maintaining freedom to work with familiar technology. With the Polymer components we provided building blocks for common tasks such as pairing and predefined layout components implement common distribution patterns.

With XD-Tools we contributed a set of tools to the **debugging** phase. Based on common developer tasks, we analysed challenges that are particular to cross-device debugging and informal testing and derived requirements for tool support. With XD-Tools, we presented a prototype that illustrates how browsers could support cross-device debugging in the future. XD-Tools enables the inclusion of both real and emulated devices for testing and debugging. It eliminates some of the challenges introduced by the fragmentation of the logic across devices by grouping devices into device configurations that can be inspected and manipulated as a whole. Furthermore, repetitive tasks can be automated, for example the pairing of devices or by recording and replaying interactions on multiple devices. In a qualitative user study, developers welcomed the cross-device support introduced by XD-Tools.

Based on a motivating example, we demonstrated the challenges of **testing** cross-device applications. We found in our analysis that UI tests are more problematic in cross-device applications than unit tests which are more isolated and do not suffer from the fragmentation of the logic across devices. To enable repeatable automated UI tests across a range of device configurations, we introduced device templates and scenarios that can be used to parametrise tests. Our explicit device selectors allow a tester to verify specific device configurations whereas the implicit selectors enabled device-independent tests and handle the fragmentation of logic across devices. In addition to these concepts that we implemented in the library XD-Testing, we also contributed a tool for visually verifying test execution based on screenshots. In a case study, we showcased the kind of tests that can be written with the library and how the screenshots could help to find the cause of a failed test.

Our final contribution was XD-Analytics, a tool for cross-device **usage analysis**. Based on five cross-device analytics use cases, we introduced a set of metrics of interest. We track these metrics in our prototype implementation XD-Analytics and discussed how they address the use cases. XD-Analytics was evaluated in a in-the-wild study with an educational application. It provided insights into how a cross-device feature was adopted by the users when it was introduced during the study.

8.1 Analysis

Our original vision was to foster communication between the devices in our everyday lives. We approached the problem by analysing state-of-the-art tool support for developing cross-device applications and filled the gaps where we found them. A common theme in all the tools that we have presented in the thesis is that we have a concept of a set of connected devices. This is in contrast to conventional development tools that treat each device individually. The concept of connected devices allows us to aggregate information, for example for debugging or analytics, and to address the set as whole, for example when testing.

Our hope is that with improved tool support, we will spark the creation of more and better cross-device applications and bring us a step closer to Mark Weiser's vision of the seamless use of devices. While we see better tool support as a necessary condition, it may not be sufficient. Researchers have lamented that a *killer* cross-device application is still missing. Furthermore, while our analysis of related work showed that people do use multiple devices in parallel and do ask for better support, it is unclear whether that implies that every application should support parallel use or whether it will remain limited to certain domains or use cases.

We have focused on the development of applications that distribute across devices. However, we also see a lot of potential in better communication among different applications across devices (like the example presented in [58]). We imagine that the best experience could be achieved with cross-device support at the operating system level. However, finding solutions that work across different platforms, each with its own user and cloud systems, would be a big challenge to tackle and would require cooperation from the platform owners.

Our web-based approach is more lightweight and works across most platforms. However, it comes at the cost of reduced access to platform specific APIs, for example a user's contacts or the file system. Furthermore, we have focused on somewhat conventional devices with GUIs. In a smart home, we may encounter a range of other devices (for example personal assistants such as Amazon Alex or Google Home) or sensors that could add to the cross-device experience. We have not considered these devices in our work. While our tools and concepts are applicable to smartwatches, we have not included these in any of our evaluations as at the time of writing they have not been tailored to support modern web browsers. However, we see a lot of potential for cross-device interaction of watches with other devices (in particular with phones [26]).

We have focused our work mostly on somewhat conventional interactions with the devices using touch or mouse and keyboard. However, new interaction modalities might be used with cross-device applications (for example proxemics [117] or gestures [40]). At the same time, we observe a trend towards conversational, speech-based interfaces with the personal assistant devices mentioned above. Our work has not considered these modalities. While we would argue that the main concepts can also be applied to these modalities, namely that devices should be handled as a set, further support would need to be explored, in particular for testing and debugging, as our tools have been tailored to the visual part of the applications.

We have approached the problem by analysing existing workflows and tools and adapting these to cross-device applications. This is an incremental approach. Starting from a blank slate one could opt for a more radical approach. The benefit of our

approach is that the tools we built are familiar to the developers to some extent, which should increase learnability. We also expect that familiar tools are more likely to be adopted by developers. A discussion of the individual tools including their limitations and possible extensions can be found at the end of each respective chapter.

8.2 Outlook

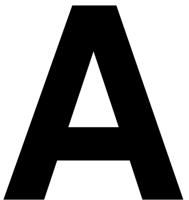
We can see many opportunities for further research. Our implementation library XD-MVC has been kept general enough to enable both automatic and user-driven distribution of an UI. A more opinionated approach could be explored along with specific tool support. Others have explored giving users complete freedom in defining UI distributions [138] or relied on common patterns [137]. This **user-driven UI distribution** is an interesting approach which eliminates the burden of attempting to design and implement a suitable UI distribution for every possible device combination. On the other hand, this approach could introduce new challenges for debugging, testing, and usage analysis. If the UI distribution is defined by the user, the developer has to deal with even more uncertainty. In addition to not knowing the devices that might be used, the UI distribution is also in the hands of the user and not the developer. This can make testing and debugging especially hard, where it is vital to know as much as possible about the setting to reproduce issues and to test.

Another direction of research would be to investigate **cross-device interaction across application boundaries**. The application framework Conductor [58] enables communication across applications and devices. For example, a map application can be combined with a contacts application to display nearby friends. The focus of Conductor is on interactions rather than the development process. However, we also expect challenges in the development process that would need further investigation. Allowing arbitrary applications to be combined would require clear communication protocols and APIs. A developer would need to test their own application carefully to ensure that it really behaves according to specification. As the input from another application is out of the developer's control, they would need to ensure that their own application is robust in all situations. Debugging may require that the setup with the other application is replicated or that at least the communication can be simulated, whereas usage analysis may need coordination of multiple applications which may originate from different developers or companies. Furthermore, support at the operating systems level could be investigated.

As we have discussed above, we have focused on the graphical user interface and mostly conventional input based on touch, mouse, and keyboard. **New input modalities**, such as proxemic interaction, gestures, or conversational UIs require further investigation. These modalities come with higher degrees of freedom than the conventional input modalities and we expect that they are more challenging to debug and test for that reason. Mechanism such as recording and replaying interactions may be even more important with these modalities, but it may not be enough to use the data of a single test user to achieve good coverage. Instead, larger data sets may be required, making usage analysis and data collection even more important. The combination of multiple modalities also merits further research.

Finally, despite the many research prototypes that have been built, we have seen

little **adoption of cross-device applications** outside labs, apart from a few very specific scenarios (for example using the phone to control the TV). This can be explained to some extent with a legacy bias [154]. Can we overcome this bias with better applications? How can we teach the user that they can use more than single device at a time using our application? Or are they simply overwhelmed with the complexity of using multiple devices? These are important questions that should be investigated.



Student Contributions

A number of students have contributed to the technical realisation of the tools and showcase applications presented in this thesis. The following students have made a significant contribution as part of their Master's, Bachelor's or lab project. All students have been personally supervised by this thesis' author.

- **Stefano Pongelli** has implemented the second MultiMasher prototype in [179].
- **Fabian Stutz** has implemented a first version of the state synchronisation in XD-MVC in [182].
- **Silvan Egli** has implemented the hybrid communication architecture of XD-MVC in [43].
- **Dhivyabharathi Ramasamy, Alexander Richter and Marko Zivkovic** have developed the XD-Bike application.
- **Madelin Schumacher** has experimented with the distributed layout patterns in [166].
- **Sivaranjini Chithambaram** has developed the pairing based on social and physical proximity in XD-MVC and implemented the hotel booking showcase application in [30].
- **Aryaman Fasciati** has developed the XD-MVC integration with React and the voting showcase application in [44].
- **Nina Heyder** has developed XD-Tools and executed the associated user study in [67].
- **Michael Spiegel** has implemented XD-Testing and written the tests in the associated case study in [177].

- **Nicola Marcacci Rossi** has developed XD-Analytics and conducted the case study with Taskbase in [160].

Bibliography

[1] R. Arthur and D. R. Olsen, Jr. XICE Windowing Toolkit: Seamless Display Annexation. *ACM Trans. Comput.-Hum. Interact.*, 18(3):14:1–14:46, Aug. 2011. ISSN 1073-0516. doi: 10.1145/1993060.1993064. URL `http://doi.acm.org/10.1145/1993060.1993064`. Cited on pages 27 and 30.

[2] S. K. Badam and N. Elmqvist. PolyChrome: A Cross-Device Framework for Collaborative Web Visualization. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces*, ITS '14, pages 109–118, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2587-5. doi: 10.1145/2669485.2669518. URL `http://doi.acm.org/10.1145/2669485.2669518`. Cited on pages 18, 22, 25, 28, 29, and 30.

[3] T. Ballendat, N. Marquardt, and S. Greenberg. Proxemic Interaction: Designing for a Proximity and Orientation-aware Environment. In *ACM International Conference on Interactive Tabletops and Surfaces*, ITS '10, pages 121–130, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0399-6. doi: 10.1145/1936652.1936676. URL `http://doi.acm.org/10.1145/1936652.1936676`. Cited on pages 3 and 19.

[4] J. E. Bardram. Activity-based computing: support for mobility and collaboration in ubiquitous computing. *Personal and Ubiquitous Computing*, 9(5):312–322, 2005. ISSN 1617-4917. doi: 10.1007/s00779-004-0335-2. URL `http://dx.doi.org/10.1007/s00779-004-0335-2`. Cited on pages 23 and 30.

[5] J. E. Bardram. Activity-based Computing for Medical Work in Hospitals. *ACM Trans. Comput.-Hum. Interact.*, 16(2):10:1–10:36, June 2009. ISSN 1073-0516. doi: 10.1145/1534903.1534907. URL `http://doi.acm.org/10.1145/1534903.1534907`. Cited on page 23.

[6] D. Baur, S. Boring, and S. Feiner. Virtual Projection: Exploring Optical Projection As a Metaphor for Multi-device Interaction. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '12, pages 1693–1702, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1015-4. doi: 10.1145/2207676.2208297. URL `http://doi.acm.org/10.1145/2207676.2208297`. Cited on page 18.

[7] M. Beaudouin-Lafon, S. Huot, M. Nancel, W. E. Mackay, E. Pietriga, R. Primet, J. Wagner, O. Chapuis, C. Pillias, J. Eagan, T. Gjerlufsen, and C. N. Klokmose. Multisurface Interaction in the WILD Room. *IEEE Computer*, 45(4):48–56, 2012.

doi: 10.1109/MC.2012.110. URL http://dx.doi.org/10.1109/MC.2012.110. Cited on pages 21 and 22.

[8] S. Bell. Police investigate 'first cyber-flashing' case. BBC News, August 2015. Retrieved September 14, 2016 from http://www.bbc.com/news/technology-33889225. Cited on page 68.

[9] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What Makes a Good Bug Report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 308–318, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-995-1. doi: 10.1145/1453101.1453146. URL http://doi.acm.org/10.1145/1453101.1453146. Cited on pages 32 and 83.

[10] J. T. Biehl, W. T. Baker, B. P. Bailey, D. S. Tan, K. M. Inkpen, and M. Czerwinski. Impromptu: A New Interaction Framework for Supporting Collaboration in Multiple Display Environments and Its Field Evaluation for Co-located Software Development. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 939–948, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-011-1. doi: 10.1145/1357054.1357200. URL http://doi.acm.org/10.1145/1357054.1357200. Cited on pages 17 and 27.

[11] N. Bila, T. Ronda, I. Mohomed, K. N. Truong, and E. de Lara. PageTailor: Reusable End-user Customization for the Mobile Web. In *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, MobiSys '07, pages 16–29, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-614-1. doi: 10.1145/1247660.1247666. URL http://doi.acm.org/10.1145/1247660.1247666. Cited on page 56.

[12] S. Boring, D. Baur, A. Butz, S. Gustafson, and P. Baudisch. Touch Projector: Mobile Interaction Through Video. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 2287–2296, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-929-9. doi: 10.1145/1753326.1753671. URL http://doi.acm.org/10.1145/1753326.1753671. Cited on pages 18 and 62.

[13] A. Bragdon, R. DeLine, K. Hinckley, and M. R. Morris. Code Space: Touch + Air Gesture Hybrid Interactions for Supporting Developer Meetings. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*, ITS '11, pages 212–221, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0871-7. doi: 10.1145/2076354.2076393. URL http://doi.acm.org/10.1145/2076354.2076393. Cited on pages 18 and 21.

[14] F. Brudy, S. Houben, N. Marquardt, and Y. Rogers. CurationSpace: Cross-Device Content Curation Using Instrumental Interaction. In *Proceedings of the 2016 ACM on Interactive Surfaces and Spaces*, ISS '16, pages 159–168, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4248-3. doi: 10.1145/2992154.2992175. URL http://doi.acm.org/10.1145/2992154.2992175. Cited on pages 17 and 21.

[15] B. Bruegge and A. H. Dutoit. *Object-Oriented Software Engineering Using UML, Patterns and Java*. Prentice Hall, 2004. Cited on page 36.

[16] A. Bruns, A. Kornstadt, and D. Wichmann. Web Application Tests with Selenium. *IEEE Software*, 26(5), 2009. ISSN 07407459. doi: 10.1109/MS.2009.144. Cited on page 36.

[17] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive Record/Replay for Web Application Debugging. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pages 473–484, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2268-3. URL `http://doi.acm.org/10.1145/2501988.2502050`. Cited on page 35.

[18] B. Cao, M. Esponda, and R. Rojas. The Use of a Multi-Display System in University Classroom Lectures. In *Proceedings of the 2016 ACM on Interactive Surfaces and Spaces*, ISS '16, pages 427–432, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4248-3. doi: 10.1145/2992154.2996793. URL `http://doi.acm.org/10.1145/2992154.2996793`. Cited on page 23.

[19] S. K. Card, G. G. Robertson, and J. D. Mackinlay. The Information Visualizer, an Information Workspace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '91, pages 181–186, New York, NY, USA, 1991. ACM. ISBN 0-89791-383-3. doi: 10.1145/108844.108874. URL `http://doi.acm.org/10.1145/108844.108874`. Cited on page 71.

[20] L. D. Catledge and J. E. Pitkow. Characterizing Browsing Strategies in the World-Wide Web. *Computer Networks and {ISDN} Systems*, 27(6): 1065 – 1073, 1995. ISSN 0169-7552. doi: http://dx.doi.org/10.1016/0169-7552(95)00043-7. URL `http://www.sciencedirect.com/science/article/pii/0169755295000437`. Cited on page 37.

[21] T.-H. Chang and Y. Li. Deep Shot: A Framework for Migrating Tasks Across Devices Using Mobile Phone Cameras. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2163–2172, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0228-9. doi: 10.1145/1978942.1979257. URL `http://doi.acm.org/10.1145/1978942.1979257`. Cited on page 18.

[22] O. Chapuis, A. Bezerianos, and S. Frantzeskakis. Smarties: An Input System for Wall Display Development. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, CHI '14, pages 2763–2772, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2473-1. doi: 10.1145/2556288.2556956. URL `http://doi.acm.org/10.1145/2556288.2556956`. Cited on page 27.

[23] K.-Y. Chen, D. Ashbrook, M. Goel, S.-H. Lee, and S. Patel. AirLink: Sharing Files Between Multiple Devices Using In-air Gestures. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '14, pages 565–569, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2968-2. doi: 10.1145/2632048.2632090. URL `http://doi.acm.org/10.1145/2632048.2632090`. Cited on page 18.

[24] N. Chen, F. Guimbretiere, and A. Sellen. Designing a Multi-slate Reading Environment to Support Active Reading Activities. *ACM Trans. Comput.-Hum. Interact.*, 19(3):18:1–18:35, Oct. 2012. ISSN 1073-0516. doi: 10.1145/2362364.2362366.

URL http://doi.acm.org/10.1145/2362364.2362366. Cited on pages 17 and 21.

[25] N. Chen, F. Guimbretière, and A. Sellen. Graduate Student Use of a Multi-slate Reading System. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 1799–1808, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1899-0. doi: 10.1145/2470654.2466237. URL http://doi.acm.org/10.1145/2470654.2466237. Cited on pages 16 and 21.

[26] X. A. Chen, T. Grossman, D. J. Wigdor, and G. Fitzmaurice. Duet: Exploring Joint Interactions on a Smart Phone and a Smart Watch. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 159–168, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2473-1. doi: 10.1145/2556288.2556955. URL http://doi.acm.org/10.1145/2556288.2556955. Cited on pages 19 and 137.

[27] K. Cheng, L. He, X. Meng, D. A. Shamma, D. Nguyen, and A. Thangapalam. CozyMaps: Real-time Collaboration on a Shared Map with Multiple Displays. In *Proceedings of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services*, MobileHCI '15, pages 46–51, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3652-9. doi: 10.1145/2785830.2785851. URL http://doi.acm.org/10.1145/2785830.2785851. Cited on page 22.

[28] P.-Y. P. Chi and Y. Li. Weave: Scripting Cross-Device Wearable Interaction. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 3923–3932, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3145-6. doi: 10.1145/2702123.2702451. URL http://doi.acm.org/10.1145/2702123.2702451. Cited on pages 28, 29, 31, 32, and 82.

[29] P.-Y. P. Chi, Y. Li, and B. Hartmann. Enhancing Cross-Device Interaction Scripting with Interactive Illustrations. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 5482–5493, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3362-7. doi: 10.1145/2858036.2858382. URL http://doi.acm.org/10.1145/2858036.2858382. Cited on pages 29 and 32.

[30] S. Chithambaram. A User Concept for Cross-Device Applications. Bachelor's thesis, ETH Zurich, 2016. Cited on pages 69, 76, and 141.

[31] A. Chokshi, T. Seyed, F. Marinho Rodrigues, and F. Maurer. ePlan Multi-Surface: A Multi-Surface Environment for Emergency Response Planning Exercises. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces*, ITS '14, pages 219–228, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2587-5. doi: 10.1145/2669485.2669520. URL http://doi.acm.org/10.1145/2669485.2669520. Cited on page 22.

[32] M. K. Chong, R. Mayrhofer, and H. Gellersen. A Survey of User Interaction for Spontaneous Device Association. *ACM Comput. Surv.*, 47(1):8:1–8:40, May 2014. ISSN 0360-0300. doi: 10.1145/2597768. URL http://doi.acm.org/10.1145/2597768. Cited on page 18.

[33] D. Chu, Z. Zhang, A. Wolman, and N. Lane. Prime: A Framework for Co-located Multi-device Apps. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 203–214, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3574-4. doi: 10.1145/2750858.2806062. URL http://doi.acm.org/10.1145/2750858.2806062. Cited on page 28.

[34] K. Church, D. Ferreira, N. Banovic, and K. Lyons. Understanding the Challenges of Mobile Phone Usage Data. In *Proceedings of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services*, MobileHCI '15, pages 504–514, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3652-9. doi: 10.1145/2785830.2785891. URL http://doi.acm.org/10.1145/2785830.2785891. Cited on page 37.

[35] F. Daniel, S. Soi, S. Tranquillini, F. Casati, C. Heng, and L. Yan. Distributed Orchestration of User Interfaces. *Information Systems*, 37(6):539–556, 2012. doi: 10.1016/j.is.2011.08.001. URL http://dx.doi.org/10.1016/j.is.2011.08.001. Cited on page 25.

[36] A. A. de Freitas, M. Nebeling, X. A. Chen, J. Yang, A. S. K. Karthikeyan Ranithangam, and A. K. Dey. Snap-To-It: A User-Inspired Platform for Opportunistic Device Interactions. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 5909–5920, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3362-7. doi: 10.1145/2858036.2858177. URL http://doi.acm.org/10.1145/2858036.2858177. Cited on page 18.

[37] D. Dearman and J. S. Pierce. It's on My Other Computer!: Computing with Multiple Devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 767–776, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-011-1. doi: 10.1145/1357054.1357177. URL http://doi.acm.org/10.1145/1357054.1357177. Cited on pages 2, 14, 15, 16, 37, and 39.

[38] B. Deka, Z. Huang, and R. Kumar. ERICA: Interaction Mining Mobile Apps. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, UIST '16, pages 767–776, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4189-9. doi: 10.1145/2984511.2984581. URL http://doi.acm.org/10.1145/2984511.2984581. Cited on page 37.

[39] L. Di Geronimo, M. Husmann, and M. C. Norrie. Surveying Personal Device Ecosystems with Cross-Device Applications in Mind. In *Proceedings of the 5th ACM International Symposium on Pervasive Displays*, PerDis '16, New York, NY, USA, 2016. ACM. Cited on page 12.

[40] L. Di Geronimo, M. Husmann, A. Patel, C. Tuerk, and M. C. Norrie. CTAT: Tilt-and-Tap Across Devices. In *Web Engineering - 16th International Conference, ICWE 2016, Lugano, Switzerland, June 6-9, 2016. Proceedings*, pages 96–113, 2016. doi: 10.1007/978-3-319-38791-8_6. URL https://doi.org/10.1007/978-3-319-38791-8_6. Cited on page 137.

[41] P. Dietz and D. Leigh. DiamondTouch: A Multi-user Touch Technology. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology*, UIST '01, pages 219–226, New York, NY, USA, 2001. ACM. ISBN 1-58113-438-X. doi: 10.1145/502348.502389. URL `http://doi.acm.org/10.1145/502348.502389`. Cited on page 17.

[42] T. Dong, E. F. Churchill, and J. Nichols. Understanding the Challenges of Designing and Developing Multi-Device Experiences. In *Proceedings of the 2016 ACM Conference on Designing Interactive Systems*, DIS '16, pages 62–72, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4031-1. doi: 10.1145/2901790.2901851. URL `http://doi.acm.org/10.1145/2901790.2901851`. Cited on pages 24, 27, 32, 39, and 105.

[43] S. Egli. Exploring Communication Architectures for XD-MVC. Bachelor's thesis, ETH Zurich, 2015. Cited on pages 71 and 141.

[44] A. Fasciati. One-Way Data Flow for XD-MVC. Bachelor's thesis, ETH Zurich, 2016. Cited on pages 79 and 141.

[45] E. R. Fisher, S. K. Badam, and N. Elmqvist. Designing Peer-to-peer Distributed User Interfaces: Case Studies on Building Distributed Applications. *International Journal of Human-Computer Studies*, 72(1):100–110, Jan. 2014. ISSN 1071-5819. doi: 10.1016/j.ijhcs.2013.08.011. URL `http://dx.doi.org/10.1016/j.ijhcs.2013.08.011`. Cited on pages 19, 28, and 70.

[46] L. Frosini and F. Paternò. User Interface Distribution in Multi-device and Multi-user Environments with Dynamically Migrating Engines. In *Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '14, pages 55–64, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2725-1. doi: 10.1145/2607023.2607032. URL `http://doi.acm.org/10.1145/2607023.2607032`. Cited on pages 28 and 29.

[47] L. Frosini, M. Manca, and F. Paternò. A Framework for the Development of Distributed Interactive Applications. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13, pages 249–254, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2138-9. doi: 10.1145/2494603.2480328. URL `http://doi.acm.org/10.1145/2494603.2480328`. Cited on pages 24, 25, and 28.

[48] A. Gallidabino and C. Pautasso. Deploying Stateful Web Components on Multiple Devices with Liquid.js for Polymer. In *2016 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering)*, CBSE '16, pages 85–90, 2016. doi: 10.1109/CBSE.2016.11. URL `http://dx.doi.org/10.1109/CBSE.2016.11`. Cited on pages 28 and 30.

[49] G. Ghiani, F. Paternò, and C. Santoro. Push and Pull of Web User Interfaces in Multi-device Environments. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, AVI '12, pages 10–17, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1287-5. doi: 10.1145/2254556.2254563. URL `http://doi.acm.org/10.1145/2254556.2254563`. Cited on page 25.

[50] G. Ghiani, M. Manca, and F. Paternò. Authoring Context-dependent Cross-device User Interfaces Based on Trigger/Action Rules. In *Proceedings of the 14th International Conference on Mobile and Ubiquitous Multimedia*, MUM '15, pages 313–322, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3605-5. doi: 10.1145/2836041.2836073. URL http://doi.acm.org/10.1145/2836041.2836073. Cited on pages 29 and 31.

[51] T. Gjerlufsen, C. N. Klokmose, J. Eagan, C. Pillias, and M. Beaudouin-Lafon. Shared Substance: Developing Flexible Multi-surface Applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 3383–3392, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0228-9. doi: 10.1145/1978942.1979446. URL http://doi.acm.org/10.1145/1978942.1979446. Cited on page 30.

[52] M. Goel, B. Lee, M. T. Islam Aumi, S. Patel, G. Borriello, S. Hibino, and B. Begole. SurfaceLink: Using Inertial and Acoustic Sensing to Enable Multi-device Interaction on a Surface. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 1387–1396, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2473-1. doi: 10.1145/2556288.2557120. URL http://doi.acm.org/10.1145/2556288.2557120. Cited on page 18.

[53] S. Gorham. Nielsen and ABC's Innovative iPad App Connects New Generation of Viewers, 2010. http://www.nielsen.com/us/en/insights/news/2010/nielsen-and-abcs-innovative-ipad-app-connects-new-generation-of-viewers.html Accessed on 13. 01. 2017. Cited on page 19.

[54] T. C. N. Graham, T. Urnes, and R. Nejabi. Efficient Distributed Implementation of Semi-replicated Synchronous Groupware. In *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology*, UIST '96, pages 1–10, New York, NY, USA, 1996. ACM. ISBN 0-89791-798-7. doi: 10.1145/237091.237092. URL http://doi.acm.org/10.1145/237091.237092. Cited on pages 30 and 62.

[55] J. Grubert, M. Heinisch, A. Quigley, and D. Schmalstieg. MultiFi: Multi Fidelity Interaction with Displays On and Around the Body. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 3933–3942, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3145-6. doi: 10.1145/2702123.2702331. URL http://doi.acm.org/10.1145/2702123.2702331. Cited on pages 19 and 20.

[56] J. Grubert, M. Kranz, and A. Quigley. Challenges in Mobile Multi-Device Ecosystems. *mUX: The Journal of Mobile User Experience*, 5(1):5, 2016. ISSN 2196-873X. doi: 10.1186/s13678-016-0007-y. URL http://dx.doi.org/10.1186/s13678-016-0007-y. Cited on page 39.

[57] M. Haller, J. Leitner, T. Seifried, J. R. Wallace, S. D. Scott, C. Richter, P. Brandl, A. Gokcezade, and S. Hunter. The NiCE Discussion Room: Integrating Paper and Digital Media to Support Co-Located Group Meetings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages

609–618, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-929-9. doi: 10.1145/1753326.1753418. URL http://doi.acm.org/10.1145/1753326.1753418. Cited on page 21.

[58] P. Hamilton and D. J. Wigdor. Conductor: Enabling and Understanding Cross-Device Interaction. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2773–2782, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2473-1. URL http://doi.acm.org/10.1145/2556288.2557170. Cited on pages 17, 21, 27, 28, 29, 31, 137, and 138.

[59] R. Han, V. Perret, and M. Naghshineh. WebSplitter: A Unified XML Framework for Multi-device Collaborative Web Browsing. In *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*, CSCW '00, pages 221–230, New York, NY, USA, 2000. ACM. ISBN 1-58113-222-0. doi: 10.1145/358916.358993. URL http://doi.acm.org/10.1145/358916.358993. Cited on page 22.

[60] R. Hardy and E. Rukzio. Touch & Interact: Touch-based Interaction of Mobile Phones with Displays. In *Proceedings of the 10th International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '08, pages 245–254, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-952-4. doi: 10.1145/1409240.1409267. URL http://doi.acm.org/10.1145/1409240.1409267. Cited on page 17.

[61] B. Hartmann, S. R. Klemmer, M. Bernstein, L. Abdulla, B. Burr, A. Robinson-Mosher, and J. Gee. Reflective Physical Prototyping Through Integrated Design, Test, and Analysis. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology*, UIST '06, pages 299–308, New York, NY, USA, 2006. ACM. ISBN 1-59593-313-1. doi: 10.1145/1166253.1166300. URL http://doi.acm.org/10.1145/1166253.1166300. Cited on pages 23 and 24.

[62] B. Hartmann, M. Beaudouin-Lafon, and W. E. Mackay. HydraScope: Creating Multi-surface Meta-applications Through View Synchronization and Input Multiplexing. In *Proceedings of the 2Nd ACM International Symposium on Pervasive Displays*, PerDis '13, pages 43–48, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2096-2. doi: 10.1145/2491568.2491578. URL http://doi.acm.org/10.1145/2491568.2491578. Cited on pages 20 and 25.

[63] K. Hasan, D. Ahlström, and P. Irani. SAMMI: A Spatially-Aware Multi-Mobile Interface for Analytic Map Navigation Tasks. In *Proceedings of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services*, MobileHCI '15, pages 36–45, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3652-9. doi: 10.1145/2785830.2785850. URL http://doi.acm.org/10.1145/2785830.2785850. Cited on pages 19 and 20.

[64] T. Heikkinen, J. Goncalves, V. Kostakos, I. Elhart, and T. Ojala. Tandem Browsing Toolkit: Distributed Multi-Display Interfaces with Web Technologies. In *Proceedings of The International Symposium on Pervasive Displays*, PerDis '14, pages 142:142–142:147, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2952-1.

doi: 10.1145/2611009.2611026. URL http://doi.acm.org/10.1145/2611009.2611026. Cited on page 28.

[65] M. Heinrich, F. J. Grüneberger, T. Springer, and M. Gaedke. Exploiting Annotations for the Rapid Development of Collaborative Web Applications. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 551–560, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2035-1. doi: 10.1145/2488388.2488437. URL http://doi.acm.org/10.1145/2488388.2488437. Cited on page 30.

[66] M. Hesenius, T. Griebe, S. Gries, and V. Gruhn. Automating UI Tests for Mobile Applications with Formal Gesture Descriptions. In *Proceedings of the 16th International Conference on Human-computer Interaction with Mobile Devices & Services*, MobileHCI '14, pages 213–222, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3004-6. doi: 10.1145/2628363.2628391. URL http://doi.acm.org/10.1145/2628363.2628391. Cited on page 37.

[67] N. Heyder. XDTools: Testing and Debugging Cross-Device Applications. Master's thesis, ETH Zurich, 2015. Cited on pages 81 and 141.

[68] K. Hinckley, G. Ramos, F. Guimbretiere, P. Baudisch, and M. Smith. Stitching: Pen Gestures That Span Multiple Displays. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '04, pages 23–31, New York, NY, USA, 2004. ACM. ISBN 1-58113-867-9. doi: 10.1145/989863.989866. URL http://doi.acm.org/10.1145/989863.989866. Cited on page 18.

[69] M. E. Holmes, S. Josephson, and R. E. Carney. Visual Attention to Television Programs with a Second-screen Application. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, ETRA '12, pages 397–400, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1221-9. doi: 10.1145/2168556.2168646. URL http://doi.acm.org/10.1145/2168556.2168646. Cited on page 20.

[70] S. Houben and N. Marquardt. WatchConnect: A Toolkit for Prototyping Smartwatch-Centric Cross-Device Applications. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 1247–1256, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3145-6. URL http://doi.acm.org/10.1145/2702123.2702215. Cited on pages 25, 26, 31, and 32.

[71] S. Houben, S. Nielsen, M. Esbensen, and J. E. Bardram. NooSphere: An Activity-centric Infrastructure for Distributed Interaction. In *Proceedings of the 12th International Conference on Mobile and Ubiquitous Multimedia*, MUM '13, pages 13:1–13:10, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2648-3. doi: 10.1145/2541831.2541856. URL http://doi.acm.org/10.1145/2541831.2541856. Cited on pages 27, 28, 29, 30, and 31.

[72] S. Houben, P. Tell, and J. E. Bardram. ActivitySpace: Managing Device Ecologies in an Activity-Centric Configuration Space. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces*, ITS '14, pages

119–128, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2587-5. doi: 10.1145/2669485.2669493. URL http://doi.acm.org/10.1145/2669485.2669493. Cited on pages 28 and 30.

[73] M. Husmann and M. C. Norrie. XD-MVC: Support for Cross-Device Development. In *Proceedings of the 1st Intl. Workshop on Interacting with Multi-Device Ecologies in the Wild*, Cross-Surface '15, 2015. Cited on page 59.

[74] M. Husmann, M. Nebeling, and M. C. Norrie. MultiMasher: A Visual Tool for Multi-device Mashups. In *Current Trends in Web Engineering - ICWE 2013 International Workshops ComposableWeb*, pages 27–38, 2013. doi: 10.1007/978-3-319-04244-2_4. URL http://dx.doi.org/10.1007/978-3-319-04244-2_4. Cited on pages 41 and 49.

[75] M. Husmann, M. Nebeling, S. Pongelli, and M. C. Norrie. MultiMasher: Providing Architectural Support and Visual Tools for Multi-Device Mashups. In *Web Information Systems Engineering*, WISE'14, pages 199–214. Springer International Publishing, 2014. ISBN 978-3-319-11745-4. URL http://dx.doi.org/10.1007/978-3-319-11746-1_15. Cited on page 41.

[76] M. Husmann, S. Chithambaram, and M. C. Norrie. Combining Physical and Social Proximity for Device Pairing. In *Proceedings of the 3rd Intl. Workshop on Interacting with Multi-Device Ecologies in the Wild*, Cross-Surface '16 V2, 2016. Cited on page 59.

[77] M. Husmann, L. D. Geronimo, and M. C. Norrie. XD-Bike: A Cross-Device Repository of Mountain Biking Routes. In *Current Trends in Web Engineering - ICWE 2016 International Workshops, DUI, TELERISE, SoWeMine, and Liquid Web, Lugano, Switzerland, June 6-9, 2016, Revised Selected Papers*, pages 107–113, 2016. doi: 10.1007/978-3-319-46963-8_9. URL http://dx.doi.org/10.1007/978-3-319-46963-8_9. Cited on page 59.

[78] M. Husmann, N. Heyder, and M. C. Norrie. Is a Framework Enough?: Cross-device Testing and Debugging. In *Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '16, pages 251–262, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4322-0. doi: 10.1145/2933242.2933249. URL http://doi.acm.org/10.1145/2933242.2933249. Cited on page 81.

[79] M. Husmann, N. M. Rossi, and M. C. Norrie. Extending a Learning Platform with Cross-Device Functionality. In *Proceedings of the 2nd Intl. Workshop on Interacting with Multi-Device Ecologies in the Wild*, Cross-Surface '16 V1, 2016. Cited on page 121.

[80] M. Husmann, N. M. Rossi, and M. C. Norrie. Usage analysis of cross-device web applications. In *Proceedings of the 5th ACM International Symposium on Pervasive Displays*, PerDis '16, pages 212–219, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4366-4. doi: 10.1145/2914920.2915017. URL http://doi.acm.org/10.1145/2914920.2915017. Cited on page 121.

[81] M. Husmann, M. Spiegel, A. Murolo, and M. C. Norrie. UI Testing Cross-Device Applications. In *Proceedings of the 2016 ACM on Interactive Surfaces and Spaces*, ISS '16, pages 179–188, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4248-3. doi: 10.1145/2992154.2992177. URL `http://doi.acm.org/10.1145/2992154.2992177`. Cited on page 105.

[82] H. M. Hutchings and J. S. Pierce. Understanding the Whethers, Hows, and Whys of Divisible Interfaces. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '06, pages 274–277, New York, NY, USA, 2006. ACM. ISBN 1-59593-353-0. doi: 10.1145/1133265.1133320. URL `http://doi.acm.org/10.1145/1133265.1133320`. Cited on pages 26, 27, and 39.

[83] H. Jetter, M. Zöllner, J. Gerken, and H. Reiterer. Design and Implementation of Post-WIMP Distributed User Interfaces with ZOIL. *International Journal of Human-Computer Interaction*, 28(11):737–747, 2012. doi: 10.1080/10447318.2012.715539. URL `http://dx.doi.org/10.1080/10447318.2012.715539`. Cited on pages 25, 27, 28, and 30.

[84] H. Jin, C. Holz, and K. Hornbæk. Tracko: Ad-hoc Mobile 3D Tracking Using Bluetooth Low Energy and Inaudible Signals for Cross-Device Interaction. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology*, UIST '15, pages 147–156, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3779-3. doi: 10.1145/2807442.2807475. URL `http://doi.acm.org/10.1145/2807442.2807475`. Cited on page 31.

[85] B. Johanson, A. Fox, and T. Winograd. The Interactive Workspaces Project: Experiences with Ubiquitous Computing Rooms. *IEEE Pervasive Computing*, 1 (2):67–74, 2002. doi: 10.1109/MPRV.2002.1012339. URL `http://dx.doi.org/10.1109/MPRV.2002.1012339`. Cited on pages 11 and 21.

[86] B. Johanson, G. Hutchins, T. Winograd, and M. Stone. PointRight: Experience with Flexible Input Redirection in Interactive Workspaces. In *Proceedings of the 15th Annual ACM Symposium on User Interface Software and Technology*, UIST '02, pages 227–234, New York, NY, USA, 2002. ACM. ISBN 1-58113-488-6. doi: 10.1145/571985.572019. URL `http://doi.acm.org/10.1145/571985.572019`. Cited on page 17.

[87] T. Jokela, J. Ojala, and T. Olsson. A Diary Study on Combining Multiple Information Devices in Everyday Activities and Tasks. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 3903–3912, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3145-6. doi: 10.1145/2702123.2702211. URL `http://doi.acm.org/10.1145/2702123.2702211`. Cited on pages 2, 12, 13, 14, 16, and 37.

[88] R. Jones, S. Clinch, J. Alexander, N. Davies, and M. Mikusz. ENGAGE: Early Insights in Measuring Multi-Device Engagements. In *Proceedings of the 4th International Symposium on Pervasive Displays*, PerDis '15, pages 31–37, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3608-6. doi: 10.1145/2757710.2757720. URL `http://doi.acm.org/10.1145/2757710.2757720`. Cited on page 37.

[89] H. Jung, E. Stolterman, W. Ryan, T. Thompson, and M. Siegel. Toward a Framework for Ecologies of Artifacts: How Are Digital Artifacts Interconnected Within a Personal Life? In *Proceedings of the 5th Nordic Conference on Human-computer Interaction: Building Bridges*, NordiCHI '08, pages 201–210, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-704-9. doi: 10.1145/1463160.1463182. URL `http://doi.acm.org/10.1145/1463160.1463182`. Cited on page 12.

[90] J. Kaasila, D. Ferreira, V. Kostakos, and T. Ojala. Testdroid: Automated Remote UI Testing on Android. In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*, MUM '12, pages 28:1–28:4, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1815-0. doi: 10.1145/2406367.2406402. URL `http://doi.acm.org/10.1145/2406367.2406402`. Cited on page 37.

[91] S. K. Kane, A. K. Karlson, B. R. Meyers, P. Johns, A. Jacobs, and G. Smith. Exploring Cross-Device Web Use on PCs and Mobile Devices. In *Proceedings of the 12th IFIP TC 13 International Conference on Human-Computer Interaction: Part I*, INTERACT '09, pages 722–735, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03654-5. doi: 10.1007/978-3-642-03655-2_79. URL `http://dx.doi.org/10.1007/978-3-642-03655-2_79`. Cited on pages 15 and 16.

[92] A. K. Karlson, A. B. Brush, and S. Schechter. Can I Borrow Your Phone?: Understanding Concerns when Sharing Mobile Phones. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 1647–1650, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-246-7. doi: 10.1145/1518701.1518953. URL `http://doi.acm.org/10.1145/1518701.1518953`. Cited on page 13.

[93] A. K. Karlson, B. Meyers, A. Jacobs, P. Johns, and S. K. Kane. Working Overtime: Patterns of Smartphone and PC Usage in the Day of an Information Worker. In *Pervasive Computing, 7th International Conference, Pervasive 2009, Nara, Japan, May 11-14, 2009. Proceedings*, pages 398–405, 2009. doi: 10.1007/978-3-642-01516-8_27. URL `http://dx.doi.org/10.1007/978-3-642-01516-8_27`. Cited on pages 13 and 14.

[94] A. K. Karlson, S. T. Iqbal, B. Meyers, G. Ramos, K. Lee, and J. C. Tang. Mobile Taskflow in Context: A Screenshot Study of Smartphone Usage. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 2009–2018, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-929-9. doi: 10.1145/1753326.1753631. URL `http://doi.acm.org/10.1145/1753326.1753631`. Cited on page 16.

[95] F. Kawsar and A. B. Brush. Home Computing Unplugged: Why, Where and When People Use Different Connected Devices at Home. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '13, pages 627–636, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1770-2. doi: 10.1145/2493432.2493494. URL `http://doi.acm.org/10.1145/2493432.2493494`. Cited on pages 12, 13, 14, and 37.

[96] C. N. Klokmose and M. Beaudouin-Lafon. VIGO: Instrumental Interaction in Multi-surface Environments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 869–878, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-246-7. doi: 10.1145/1518701.1518833. URL http://doi.acm.org/10.1145/1518701.1518833. Cited on page 30.

[97] C. N. Klokmose, J. R. Eagan, S. Baader, W. Mackay, and M. Beaudouin-Lafon. Webstrates: Shareable Dynamic Media. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology*, UIST '15, pages 280–290, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3779-3. doi: 10.1145/2807442.2807446. URL http://doi.acm.org/10.1145/2807442.2807446. Cited on pages 28, 30, and 31.

[98] A. J. Ko and B. A. Myers. Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 151–158, New York, NY, USA, 2004. ACM. ISBN 1-58113-702-8. doi: 10.1145/985692.985712. URL http://doi.acm.org/10.1145/985692.985712. Cited on pages 32 and 84.

[99] W. A. König, R. Rädle, and H. Reiterer. Interactive Design of Multimodal User Interfaces. *Journal on Multimodal User Interfaces*, 3(3):197–213, 2010. doi: 10.1007/s12193-010-0044-2. URL http://dx.doi.org/10.1007/s12193-010-0044-2. Cited on page 25.

[100] I. Koren, J. Bavendiek, and R. Klamma. DireWolf Goes Pack Hunting: A Peer-to-Peer Approach for Secure Low Latency Widget Distribution Using WebRTC. In *Web Engineering, 14th International Conference, ICWE 2014, Toulouse, France, July 1-4, 2014. Proceedings*, pages 507–510, 2014. doi: 10.1007/978-3-319-08245-5_38. URL http://dx.doi.org/10.1007/978-3-319-08245-5_38. Cited on pages 28 and 30.

[101] D. Kovachev, D. Renzel, P. Nicolaescu, and R. Klamma. DireWolf - Distributing and Migrating User Interfaces for Widget-Based Web Applications. In *Web Engineering: 13th International Conference, ICWE 2013, Aalborg, Denmark, July 8-12, 2013. Proceedings*, ICWE'13, pages 99–113. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-39199-6. URL http://dx.doi.org/10.1007/978-3-642-39200-9_10. Cited on page 25.

[102] G. E. Krasner and S. T. Pope. A Cookbook for Using the Model-view Controller User Interface Paradigm in Smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, Aug. 1988. ISSN 0896-8438. URL http://dl.acm.org/citation.cfm?id=50757.50759. Cited on page 30.

[103] M. Krug, F. Wiedemann, and M. Gaedke. SmartComposition: A Component-Based Approach for Creating Multi-screen Mashups. In *Web Engineering, 14th International Conference, ICWE 2014, Toulouse, France, July 1-4, 2014. Proceedings*, pages 236–253, 2014. doi: 10.1007/978-3-319-08245-5_14. URL http://dx.doi.org/10.1007/978-3-319-08245-5_14. Cited on pages 20, 23, 28, and 30.

[104] C. Lander, S. Gehring, A. Krüger, S. Boring, and A. Bulling. GazeProjector: Accurate Gaze Estimation and Seamless Gaze Interaction Across Multiple Displays. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology*, UIST '15, pages 395–404, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3779-3. doi: 10.1145/2807442.2807479. URL `http://doi.acm.org/10.1145/2807442.2807479`. Cited on page 19.

[105] J. Lanir, K. S. Booth, and K. Hawkey. The Benefits of More Electronic Screen Space on Students' Retention of Material in Classroom Lectures. *Computers & Education*, 55(2):892 – 903, 2010. ISSN 0360-1315. doi: http://dx.doi.org/10.1016/j.compedu.2010.03.020. URL `http://www.sciencedirect.com/science/article/pii/S0360131510001065`. Cited on page 23.

[106] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining Mental Models: A Study of Developer Work Habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 492–501, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. URL `http://doi.acm.org/10.1145/1134285.1134355`. Cited on pages 23 and 83.

[107] M. Leotta, D. Clerissi, F. Ricca, and P. Tonellla. Capture-Replay vs. Programmable Web Testing: an Empirical Assessment During Test Case Evolution. In *Proceedings of the 2013 20th Working Conference on Reverse Engineerings*, WCRE '13, pages 272–281, 2013. doi: 10.1109/WCRE.2013.6671302. Cited on page 36.

[108] J. Lin and J. A. Landay. Employing Patterns and Layers for Early-stage Design and Prototyping of Cross-device User Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 1313–1322, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-011-1. doi: 10.1145/1357054.1357260. URL `http://doi.acm.org/10.1145/1357054.1357260`. Cited on page 24.

[109] D. Lowet and D. Goergen. Co-browsing Dynamic Web Pages. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 941–950, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-487-4. doi: 10.1145/1526709.1526836. URL `http://doi.acm.org/10.1145/1526709.1526836`. Cited on pages 22, 49, and 52.

[110] A. Lucero, J. Keränen, and H. Korhonen. Collaborative Use of Mobile Phones for Brainstorming. In *Proceedings of the 12th International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '10, pages 337–340, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-835-3. doi: 10.1145/1851600.1851659. URL `http://doi.acm.org/10.1145/1851600.1851659`. Cited on pages 18 and 23.

[111] A. Lucero, J. Holopainen, and T. Jokela. Pass-them-around: Collaborative Use of Mobile Phones for Photo Sharing. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 1787–1796, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0228-9. doi: 10.1145/1978942.1979201. URL `http://doi.acm.org/10.1145/1978942.1979201`. Cited on pages 18 and 23.

[112] N. Mahyar, K. J. Burke, J. E. Xiang, S. C. Meng, K. S. Booth, C. L. Girling, and R. W. Kellett. UD Co-Spaces: A Table-Centred Multi-Display Environment for Public Engagement in Urban Design Charrettes. In *Proceedings of the 2016 ACM on Interactive Surfaces and Spaces*, ISS '16, pages 109–118, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4248-3. doi: 10.1145/2992154.2992163. URL `http://doi.acm.org/10.1145/2992154.2992163`. Cited on page 22.

[113] M. Manca and F. Paternò. Customizable Dynamic User Interface Distribution. In *Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '16, pages 27–37, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4322-0. doi: 10.1145/2933242.2933259. URL `http://doi.acm.org/10.1145/2933242.2933259`. Cited on pages 24 and 25.

[114] E. Marcotte. Responsive Web Design. A List Apart, 2010. Retrieved February 6, 2017 from `http://alistapart.com/article/responsive-web-design`. Cited on pages 27 and 28.

[115] N. Marquardt, R. Diaz-Marino, S. Boring, and S. Greenberg. The Proximity Toolkit: Prototyping Proxemic Interactions in Ubiquitous Computing Ecologies. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 315–326, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0716-1. URL `http://doi.acm.org/10.1145/2047196.2047238`. Cited on pages 25, 31, 32, and 33.

[116] N. Marquardt, T. Ballendat, S. Boring, S. Greenberg, and K. Hinckley. Gradual Engagement: Facilitating Information Exchange Between Digital Devices As a Function of Proximity. In *Proceedings of the 2012 ACM International Conference on Interactive Tabletops and Surfaces*, ITS '12, pages 31–40, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1209-7. doi: 10.1145/2396636.2396642. URL `http://doi.acm.org/10.1145/2396636.2396642`. Cited on page 19.

[117] N. Marquardt, K. Hinckley, and S. Greenberg. Cross-device Interaction via Micromobility and F-formations. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST '12, pages 13–22, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1580-7. doi: 10.1145/2380116.2380121. URL `http://doi.acm.org/10.1145/2380116.2380121`. Cited on pages 3, 19, and 137.

[118] J. P. M. Massó, J. Vanderdonckt, and P. G. López. Direct Manipulation of User Interfaces for Migration. In *Proceedings of the 11th International Conference on Intelligent User Interfaces*, IUI '06, pages 140–147, New York, NY, USA, 2006. ACM. ISBN 1-59593-287-9. doi: 10.1145/1111449.1111483. URL `http://doi.acm.org/10.1145/1111449.1111483`. Cited on pages 24 and 25.

[119] T. Matthews, K. Liao, A. Turner, M. Berkovich, R. Reeder, and S. Consolvo. "She'll Just Grab Any Device That's Closer": A Study of Everyday Device & Account Sharing in Households. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 5921–5932, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3362-7. doi: 10.1145/2858036.2858051. URL `http://doi.acm.org/10.1145/2858036.2858051`. Cited on page 13.

[120] F. Matulic, M. Husmann, S. Walter, and M. C. Norrie. Eyes-Free Touch Command Support for Pen-Based Digital Whiteboards via Handheld Devices. In *Proceedings of the 2015 International Conference on Interactive Tabletops & Surfaces*, ITS '15, pages 141–150, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3899-8. doi: 10.1145/2817721.2817728. URL http://doi.acm.org/10.1145/2817721.2817728. Cited on pages 17 and 18.

[121] W. McGrath, B. Bowman, D. McCallum, J. D. Hincapié-Ramos, N. Elmqvist, and P. Irani. Branch-explore-merge: Facilitating Real-time Revision Control in Collaborative Visual Exploration. In *Proceedings of the 2012 ACM International Conference on Interactive Tabletops and Surfaces*, ITS '12, pages 235–244, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1209-7. doi: 10.1145/2396636.2396673. URL http://doi.acm.org/10.1145/2396636.2396673. Cited on page 23.

[122] J. Melchior, D. Grolaux, J. Vanderdonckt, and P. Van Roy. A Toolkit for Peer-to-peer Distributed User Interfaces: Concepts, Implementation, and Applications. In *Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '09, pages 69–78, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-600-7. doi: 10.1145/1570433.1570449. URL http://doi.acm.org/10.1145/1570433.1570449. Cited on pages 28 and 79.

[123] J. Melchior, J. Vanderdonckt, and P. Van Roy. A Model-based Approach for Distributed User Interfaces. In *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '11, pages 11–20, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0670-6. doi: 10.1145/1996461.1996488. URL http://doi.acm.org/10.1145/1996461.1996488. Cited on pages 24, 25, 28, 29, and 31.

[124] J. Meskens, J. Vermeulen, K. Luyten, and K. Coninx. Gummy for Multi-platform User Interface Designs: Shape Me, Multiply Me, Fix Me, Use Me. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, AVI '08, pages 233–240, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-141-5. doi: 10.1145/1385569.1385607. URL http://doi.acm.org/10.1145/1385569.1385607. Cited on pages 24 and 26.

[125] J. Meskens, K. Luyten, and K. Coninx. Jelly: A Multi-device Design Environment for Managing Consistency Across Devices. In *Proceedings of the International Conference on Advanced Visual Interfaces*, AVI '10, pages 289–296, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0076-6. doi: 10.1145/1842993.1843044. URL http://doi.acm.org/10.1145/1842993.1843044. Cited on pages 24 and 32.

[126] J. Meskens, K. Luyten, and K. Coninx. D-Macs: Building Multi-device User Interfaces by Demonstrating, Sharing and Replaying Design Actions. In *Proceedings of the 23Nd Annual ACM Symposium on User Interface Software and Technology*, UIST '10, pages 129–138, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0271-5. doi: 10.1145/1866029.1866051. URL http://doi.acm.org/10.1145/1866029.1866051. Cited on page 24.

[127] J. Mickens. Rivet: Browser-agnostic Remote Debugging for Web Applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 30–30, Berkeley, CA, USA, 2012. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2342821.2342851`. Cited on page 35.

[128] J. W. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic Capture and Replay for JavaScript Applications. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1855711.1855722`. Cited on page 35.

[129] T. Mikkonen, K. Systä, and C. Pautasso. Towards Liquid Web Applications. In *Engineering the Web in the Big Data Era - 15th International Conference, ICWE 2015, Rotterdam, The Netherlands, June 23-26, 2015, Proceedings*, pages 134–143, 2015. doi: 10.1007/978-3-319-19890-3_10. URL `http://dx.doi.org/10.1007/978-3-319-19890-3_10`. Cited on pages 3 and 28.

[130] P. Mueller. Weinre, 2011. Retrieved September 11, 2015 from `http://people.apache.org/~pmuellr/weinre/docs/latest/Home.html`. Cited on page 35.

[131] B. A. Myers. Using Handhelds and PCs Together. *Commun. ACM*, 44(11): 34–41, Nov. 2001. ISSN 0001-0782. doi: 10.1145/384150.384159. URL `http://doi.acm.org/10.1145/384150.384159`. Cited on page 17.

[132] B. A. Myers, H. Stiel, and R. Gargiulo. Collaboration Using Multiple PDAs Connected to a PC. In *Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work*, CSCW '98, pages 285–294, New York, NY, USA, 1998. ACM. ISBN 1-58113-009-0. doi: 10.1145/289444.289503. URL `http://doi.acm.org/10.1145/289444.289503`. Cited on page 17.

[133] B. A. Myers, S. E. Hudson, R. F. Pausch, B. Myers, S. E. Hudson, and R. Pausch. Past, Present, and Future of User Interface Software Tools. *ACM Trans. Comput.-Hum. Interact.*, 7(1):3–28, Mar. 2000. ISSN 1073-0516. doi: 10.1145/344949.344959. URL `http://doi.acm.org/10.1145/344949.344959`. Cited on page 59.

[134] B. A. Myers, C. H. Peck, J. Nichols, D. Kong, and R. Miller. Interacting at a Distance Using Semantic Snarfing. In *Proceedings of the 3rd International Conference on Ubiquitous Computing*, UbiComp '01, pages 305–314, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42614-0. URL `http://dl.acm.org/citation.cfm?id=647987.741332`. Cited on page 17.

[135] T. Neate, M. Jones, and M. Evans. Mediating Attention for Second Screen Companion Content. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 3103–3106, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3145-6. doi: 10.1145/2702123.2702278. URL `http://doi.acm.org/10.1145/2702123.2702278`. Cited on page 20.

[136] T. Neate, M. Evans, and M. Jones. Designing Visual Complexity for Dual-screen Media. In *Proceedings of the 2016 CHI Conference on Human Factors*

in Computing Systems, CHI '16, pages 475–486, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3362-7. doi: 10.1145/2858036.2858112. URL `http://doi.acm.org/10.1145/2858036.2858112`. Cited on page 20.

[137] M. Nebeling. XDBrowser 2.0: Semi-Automatic Generation of Cross-Device Interfaces. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 4574–4584, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4655-9. doi: 10.1145/3025453.3025547. URL `http://doi.acm.org/10.1145/3025453.3025547`. Cited on pages 17, 19, 32, 65, and 138.

[138] M. Nebeling and A. K. Dey. XDBrowser: User-Defined Cross-Device Web Page Designs. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 5494–5505, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3362-7. doi: 10.1145/2858036.2858048. URL `http://doi.acm.org/10.1145/2858036.2858048`. Cited on pages 7, 15, 16, 21, 25, 26, 27, 32, 65, and 138.

[139] M. Nebeling, M. Speicher, and M. C. Norrie. CrowdAdapt: Enabling Crowdsourced Web Page Adaptation for Individual Viewing Conditions and Preferences. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13, pages 23–32, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2138-9. doi: 10.1145/2494603.2480304. URL `http://doi.acm.org/10.1145/2494603.2480304`. Cited on page 53.

[140] M. Nebeling, C. Zimmerli, M. Husmann, D. E. Simmen, and M. C. Norrie. Information Concepts for Cross-device Applications. In *Distributed User Interfaces: Models, Methods and Tools, DUI 2013 In conjunction with ACM EICS 2013 Conference, London, UK, June 24th, 2013.*, pages 14–17, 2013. Cited on page 22.

[141] M. Nebeling, T. Mintsi, M. Husmann, and M. Norrie. Interactive Development of Cross-device User Interfaces. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, CHI '14, pages 2793–2802, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2473-1. doi: 10.1145/2556288.2556980. URL `http://doi.acm.org/10.1145/2556288.2556980`. Cited on pages 25, 26, 29, and 32.

[142] M. Nebeling, E. Teunissen, M. Husmann, and M. C. Norrie. XDKinect: Development Framework for Cross-device Interaction Using Kinect. In *Proceedings of the 2014 ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '14, pages 65–74, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2725-1. doi: 10.1145/2607023.2607024. URL `http://doi.acm.org/10.1145/2607023.2607024`. Cited on pages 28, 31, and 33.

[143] M. Nebeling, M. Husmann, C. Zimmerli, G. Valente, and M. C. Norrie. XDSession: Integrated Development and Testing of Cross-device Applications. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '15, pages 22–27, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3646-8. doi: 10.1145/2774225.2775075. URL `http://doi.acm.org/10.1145/2774225.2775075`. Cited on pages 28, 30, 32, and 33.

[144] M. Nebeling, D. Ott, and M. C. Norrie. Kinect Analysis: A System for Recording, Analysing and Sharing Multimodal Interaction Elicitation Studies. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '15, pages 142–151, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3646-8. doi: 10.1145/2774225.2774846. URL `http://doi.acm.org/10.1145/2774225.2774846`. Cited on page 33.

[145] J. Nichols, Z. Hua, and J. Barton. Highlight: A System for Creating and Deploying Mobile Web Applications. In *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology*, UIST '08, pages 249–258, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-975-3. doi: 10.1145/1449715.1449757. URL `http://doi.acm.org/10.1145/1449715.1449757`. Cited on page 50.

[146] H. S. Nielsen, M. P. Olsen, M. B. Skov, and J. Kjeldskov. JuxtaPinch: Exploring Multi-device Interaction in Collocated Photo Sharing. In *Proceedings of the 16th International Conference on Human-computer Interaction with Mobile Devices & Services*, MobileHCI '14, pages 183–192, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3004-6. doi: 10.1145/2628363.2628369. URL `http://doi.acm.org/10.1145/2628363.2628369`. Cited on pages 18 and 23.

[147] D. R. Olsen, Jr., S. T. Nielsen, and D. Parslow. Join and Capture: A Model for Nomadic Interaction. In *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology*, UIST '01, pages 131–140, New York, NY, USA, 2001. ACM. ISBN 1-58113-438-X. doi: 10.1145/502348.502367. URL `http://doi.acm.org/10.1145/502348.502367`. Cited on page 28.

[148] S. Oney and B. A. Myers. FireCrystal: Understanding Interactive Behaviors in Dynamic Web Pages. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, VL/HCC '09, pages 105–108. IEEE, 2009. URL `http://dx.doi.org/10.1109/VLHCC.2009.5295287`. Cited on page 35.

[149] A. Oulasvirta and L. Sumari. Mobile Kits and Laptop Trays: Managing Multiple Devices in Mobile Information Work. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '07, pages 1127–1136, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-593-9. doi: 10.1145/1240624.1240795. URL `http://doi.acm.org/10.1145/1240624.1240795`. Cited on pages 12, 13, 14, 15, and 37.

[150] F. Paternò and C. Santoro. A Logical Framework for Multi-device User Interfaces. In *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '12, pages 45–50, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1168-7. doi: 10.1145/2305484.2305494. URL `http://doi.acm.org/10.1145/2305484.2305494`. Cited on pages 9, 25, 26, 28, 41, and 51.

[151] F. Paterno', C. Santoro, and L. D. Spano. MARIA: A Universal, Declarative, Multiple Abstraction-level Language for Service-oriented Applications in Ubiquitous Environments. *ACM Trans. Comput.-Hum. Interact.*, 16(4):19:1–19:30, Nov. 2009. ISSN 1073-0516. doi: 10.1145/1614390.1614394. URL `http://doi.acm.org/10.1145/1614390.1614394`. Cited on pages 24 and 25.

[152] J. S. Pierce and J. Nichols. An Infrastructure for Extending Applications' User Experiences Across Multiple Personal Devices. In *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology*, UIST '08, pages 101–110, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-975-3. doi: 10.1145/1449715.1449733. URL `http://doi.acm.org/10.1145/1449715.1449733`. Cited on page 31.

[153] K. Pietroszek, J. R. Wallace, and E. Lank. Tiltcasting: 3D Interaction on Large Displays Using a Mobile Device. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software and Technology*, UIST '15, pages 57–62, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3779-3. doi: 10.1145/2807442.2807471. URL `http://doi.acm.org/10.1145/2807442.2807471`. Cited on page 18.

[154] T. Plank, H.-C. Jetter, R. Rädle, C. N. Klokmose, T. Luger, and H. Reiterer. Is Two Enough?! Studying Benefits, Barriers, and Biases of Multi-Tablet Use for Collaborative Visualization. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 4548–4560, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4655-9. doi: 10.1145/3025453.3025537. URL `http://doi.acm.org/10.1145/3025453.3025537`. Cited on pages 21, 37, and 139.

[155] R. Rädle, H.-C. Jetter, N. Marquardt, H. Reiterer, and Y. Rogers. HuddleLamp: Spatially-Aware Mobile Displays for Ad-hoc Around-the-Table Collaboration. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces*, ITS '14, pages 45–54, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2587-5. doi: 10.1145/2669485.2669500. URL `http://doi.acm.org/10.1145/2669485.2669500`. Cited on pages 18, 28, 29, and 31.

[156] R. Rädle, H.-C. Jetter, M. Schreiner, Z. Lu, H. Reiterer, and Y. Rogers. Spatially-aware or Spatially-agnostic?: Elicitation and Evaluation of User-Defined Cross-Device Interactions. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 3913–3922, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3145-6. doi: 10.1145/2702123.2702287. URL `http://doi.acm.org/10.1145/2702123.2702287`. Cited on page 18.

[157] J. Rekimoto. Pick-and-drop: A Direct Manipulation Technique for Multiple Computer Environments. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology*, UIST '97, pages 31–39, New York, NY, USA, 1997. ACM. ISBN 0-89791-881-9. doi: 10.1145/263407.263505. URL `http://doi.acm.org/10.1145/263407.263505`. Cited on page 17.

[158] J. Rekimoto. A Multiple Device Approach for Supporting Whiteboard-based Interactions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '98, pages 344–351, New York, NY, USA, 1998. ACM Press/Addison-Wesley Publishing Co. ISBN 0-201-30987-4. doi: 10.1145/274644.274692. URL `http://dx.doi.org/10.1145/274644.274692`. Cited on pages 17 and 21.

[159] R. Roels, C. Vermeylen, and B. Signer. A Unified Communication Platform for Enriching and Enhancing Presentations with Active Learning Components. In *IEEE 14th International Conference on Advanced Learning Technologies (ICALT 2014)*, pages 131–135, 2014. doi: 10.1109/ICALT.2014.46. URL `http://dx.doi.org/10.1109/ICALT.2014.46`. Cited on page 23.

[160] N. M. Rossi. Usage Analysis of Cross-Device Applications in the Wild. Master's thesis, ETH Zurich, 2016. Cited on pages 121 and 142.

[161] S. Santosa and D. Wigdor. A Field Study of Multi-Device Workflows in Distributed Workspaces. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '13, pages 63–72, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1770-2. URL `http://doi.acm.org/10.1145/2493432.2493476`. Cited on pages 2, 12, 14, 15, 16, 37, and 39.

[162] D. Schmidt, F. Chehimi, E. Rukzio, and H. Gellersen. PhoneTouch: A Technique for Direct Phone Interaction on Surfaces. In *Proceedings of the 23Nd Annual ACM Symposium on User Interface Software and Technology*, UIST '10, pages 13–16, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0271-5. doi: 10.1145/1866029.1866034. URL `http://doi.acm.org/10.1145/1866029.1866034`. Cited on page 17.

[163] D. Schmidt, J. Seifert, E. Rukzio, and H. Gellersen. A Cross-device Interaction Style for Mobiles and Surfaces. In *Proceedings of the Designing Interactive Systems Conference*, DIS '12, pages 318–327, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1210-3. doi: 10.1145/2317956.2318005. URL `http://doi.acm.org/10.1145/2317956.2318005`. Cited on page 17.

[164] D. Schmidt, C. Sas, and H. Gellersen. Personal Clipboards for Individual Copy-and-paste on Shared Multi-user Surfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 3335–3344, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1899-0. doi: 10.1145/2470654.2466457. URL `http://doi.acm.org/10.1145/2470654.2466457`. Cited on page 17.

[165] M. Schreiner, R. Rädle, H.-C. Jetter, and H. Reiterer. Connichiwa: A Framework for Cross-Device Web Applications. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '15, pages 2163–2168, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3146-3. doi: 10.1145/2702613.2732909. URL `http://doi.acm.org/10.1145/2702613.2732909`. Cited on pages 19, 28, 29, 31, and 79.

[166] M. Schumacher. Extending Existing User Interface Design Patterns for Cross-Device Applications. Bachelor's thesis, ETH Zurich, 2014. Cited on page 141.

[167] K. Seewoonauth, E. Rukzio, R. Hardy, and P. Holleis. Touch & Connect and Touch & Select: Interacting with a Computer by Touching It with a Mobile Phone. In *Proceedings of the 11th International Conference on Human-Computer Interaction with Mobile Devices and Services*, MobileHCI '09, pages 36:1–36:9, New York,

NY, USA, 2009. ACM. ISBN 978-1-60558-281-8. doi: 10.1145/1613858.1613905. URL http://doi.acm.org/10.1145/1613858.1613905. Cited on page 17.

[168] J. Seifert, A. Bayer, and E. Rukzio. PointerPhone: Using Mobile Phones for Direct Pointing Interactions with Remote Displays. In *Human-Computer Interaction - INTERACT 2013 - 14th IFIP TC 13 International Conference, Cape Town, South Africa, September 2-6, 2013, Proceedings, Part III*, pages 18–35, 2013. doi: 10.1007/978-3-642-40477-1_2. URL http://dx.doi.org/10.1007/978-3-642-40477-1_2. Cited on page 18.

[169] M. Serrano, B. Ens, X.-D. Yang, and P. Irani. Gluey: Developing a Head-Worn Display Interface to Unify the Interaction Experience in Distributed Display Environments. In *Proceedings of the 17th International Conference on Human-Computer Interaction with Mobile Devices and Services*, MobileHCI '15, pages 161–171, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3652-9. doi: 10.1145/2785830.2785838. URL http://doi.acm.org/10.1145/2785830.2785838. Cited on page 19.

[170] T. Seyed, M. Costa Sousa, F. Maurer, and A. Tang. SkyHunter: A Multi-surface Environment for Supporting Oil and Gas Exploration. In *Proceedings of the 2013 ACM International Conference on Interactive Tabletops and Surfaces*, ITS '13, pages 15–22, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2271-3. doi: 10.1145/2512349.2512798. URL http://doi.acm.org/10.1145/2512349.2512798. Cited on page 22.

[171] T. Seyed, A. Azazi, E. Chan, Y. Wang, and F. Maurer. SoD-Toolkit: A Toolkit for Interactively Prototyping and Developing Multi-Sensor, Multi-Device Environments. In *Proceedings of the 2015 International Conference on Interactive Tabletops & Surfaces*, ITS '15, pages 171–180, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3899-8. doi: 10.1145/2817721.2817750. URL http://doi.acm.org/10.1145/2817721.2817750. Cited on pages 25, 27, 28, and 31.

[172] Z. Shakeri Hossein Abad, C. Anslow, and F. Maurer. Multi Surface Interactions with Geospatial Data: A Systematic Review. In *Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces*, ITS '14, pages 69–78, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2587-5. doi: 10.1145/2669485.2669505. URL http://doi.acm.org/10.1145/2669485.2669505. Cited on page 22.

[173] B. Shneiderman and R. E. Mayer. Syntactic/semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Parallel Programming*, 8(3):219–238, 1979. doi: 10.1007/BF00977789. URL http://dx.doi.org/10.1007/BF00977789. Cited on pages 7, 32, 81, and 82.

[174] M. B. Skov, J. Kjeldskov, J. Paay, H. P. Jensen, and M. P. Olsen. Investigating Cross-Device Interaction Techniques: A Case of Card Playing on Handhelds and Tablets. In *Proceedings of the Annual Meeting of the Australian Special Interest Group for Computer Human Interaction*, OzCHI '15, pages 446–454, New York,

NY, USA, 2015. ACM. ISBN 978-1-4503-3673-4. doi: 10.1145/2838739.2838763. URL http://doi.acm.org/10.1145/2838739.2838763. Cited on page 23.

[175] T. Sohn, F. C. Y. Li, A. Battestini, V. Setlur, K. Mori, and H. Horii. Myngle: Unifying and Filtering Web Content for Unplanned Access Between Multiple Personal Devices. In *Proceedings of the 13th International Conference on Ubiquitous Computing*, UbiComp '11, pages 257–266, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0630-0. doi: 10.1145/2030112.2030147. URL http://doi.acm.org/10.1145/2030112.2030147. Cited on page 25.

[176] H. Sørensen, D. Raptis, J. Kjeldskov, and M. B. Skov. The 4C Framework: Principles of Interaction in Digital Ecosystems. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '14, pages 87–97, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2968-2. doi: 10.1145/2632048.2636089. URL http://doi.acm.org/10.1145/2632048.2636089. Cited on pages 16 and 26.

[177] M. Spiegel. XD-Testing: Automated Testing of Cross-DeviceWeb Applications. Master's thesis, ETH Zurich, 2016. Cited on pages 105 and 141.

[178] J. Srivastava, R. Cooley, M. Deshpande, and P.-N. Tan. Web Usage Mining: Discovery and Applications of Usage Patterns from Web Data. *SIGKDD Explor. Newsl.*, 1(2):12–23, Jan. 2000. ISSN 1931-0145. doi: 10.1145/846183.846188. URL http://doi.acm.org/10.1145/846183.846188. Cited on page 37.

[179] Stefano. Jigsaw: An infrastructure for cross-device mashups. Master's thesis, ETH Zurich, 2013. Cited on pages 41, 55, and 141.

[180] N. A. Streitz, J. Geißler, J. M. Haake, and J. Hol. DOLPHIN: Integrated Meeting Support Across Local and Remote Desktop Environments and LiveBoards. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work*, CSCW '94, pages 345–358, New York, NY, USA, 1994. ACM. ISBN 0-89791-689-1. doi: 10.1145/192844.193044. URL http://doi.acm.org/10.1145/192844.193044. Cited on pages 2 and 11.

[181] N. A. Streitz, J. Geißler, T. Holmer, S. Konomi, C. Müller-Tomfelde, W. Reischl, P. Rexroth, P. Seitz, and R. Steinmetz. i-LAND: An Interactive Landscape for Creativity and Innovation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '99, pages 120–127, New York, NY, USA, 1999. ACM. ISBN 0-201-48559-1. doi: 10.1145/302979.303010. URL http://doi.acm.org/10.1145/302979.303010. Cited on pages 2, 11, and 21.

[182] F. Stutz. XDModel: A Framework for Peer-to-Peer Cross-Device Model Management for the Web. Bachelor's thesis, ETH Zurich, 2015. Cited on pages 60 and 141.

[183] D. S. Tan, B. Meyers, and M. Czerwinski. WinCuts: Manipulating Arbitrary Window Regions for More Effective Use of Screen Space. In *CHI '04 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '04, pages 1525–1528, New York, NY, USA, 2004. ACM. ISBN 1-58113-703-6. doi: 10.1145/

985921.986106. URL http://doi.acm.org/10.1145/985921.986106. Cited on pages 25 and 62.

[184] P. Tandler. The BEACH Application Model and Software Framework for Synchronous Collaboration in Ubiquitous Computing Environments. *J. Syst. Softw.*, 69(3):267–296, Jan. 2004. ISSN 0164-1212. doi: 10.1016/S0164-1212(03)00055-4. URL http://dx.doi.org/10.1016/S0164-1212(03)00055-4. Cited on page 2.

[185] L. Terrenghi, A. Quigley, and A. Dix. A Taxonomy for and Analysis of Multi-person-display Ecosystems. *Personal Ubiquitous Comput.*, 13(8):583–598, Nov. 2009. ISSN 1617-4909. doi: 10.1007/s00779-009-0244-5. URL http://dx.doi.org/10.1007/s00779-009-0244-5. Cited on page 16.

[186] V. Triglianos and C. Pautasso. Interactive Scalable Lectures with ASQ. In *Web Engineering, 14th International Conference, ICWE 2014, Toulouse, France, July 1-4, 2014. Proceedings*, pages 515–518, 2014. doi: 10.1007/978-3-319-08245-5_40. URL http://dx.doi.org/10.1007/978-3-319-08245-5_40. Cited on page 23.

[187] J. Turner, J. Alexander, A. Bulling, D. Schmidt, and H. Gellersen. Eye Pull, Eye Push: Moving Objects between Large Screens and Personal Devices with Gaze and Touch. In *Human-Computer Interaction - INTERACT 2013 - 14th IFIP TC 13 International Conference, Cape Town, South Africa, September 2-6, 2013, Proceedings, Part II*, pages 170–186, 2013. doi: 10.1007/978-3-642-40480-1_11. URL http://dx.doi.org/10.1007/978-3-642-40480-1_11. Cited on page 19.

[188] E. Uzun, N. Saxena, and A. Kumar. Pairing Devices for Social Interactions: A Comparative Usability Evaluation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2315–2324, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0228-9. doi: 10.1145/1978942.1979282. URL http://doi.acm.org/10.1145/1978942.1979282. Cited on page 18.

[189] D. T. Wagner, A. Rice, and A. R. Beresford. Device Analyzer: Large-scale Mobile Data Collection. *SIGMETRICS Perform. Eval. Rev.*, 41(4):53–56, Apr. 2014. ISSN 0163-5999. doi: 10.1145/2627534.2627553. URL http://doi.acm.org/10.1145/2627534.2627553. Cited on page 37.

[190] M. Wäljas, K. Segerståhl, K. Väänänen-Vainio-Mattila, and H. Oinas-Kukkonen. Cross-platform Service User Experience: A Field Study and an Initial Framework. In *Proceedings of the 12th International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '10, pages 219–228, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-835-3. doi: 10.1145/1851600.1851637. URL http://doi.acm.org/10.1145/1851600.1851637. Cited on pages 25 and 26.

[191] J. R. Wallace, S. D. Scott, and C. G. MacGregor. Collaborative Sensemaking on a Digital Tabletop and Personal Tablets: Prioritization, Comparisons, and Tableaux. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 3345–3354, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1899-0. doi: 10.1145/2470654.2466458. URL http://doi.acm.org/10.1145/2470654.2466458. Cited on page 22.

[192] M. Weiser. The Computer for the 21st Century. *Scientific American*, 265(3): 66–75, Sept. 1991. Cited on page 1.

[193] D. Wigdor, H. Jiang, C. Forlines, M. Borkin, and C. Shen. WeSpace: The Design Development and Deployment of a Walk-up and Share Multi-surface Visual Collaboration System. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 1237–1246, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-246-7. doi: 10.1145/1518701.1518886. URL `http://doi.acm.org/10.1145/1518701.1518886`. Cited on pages 21 and 22.

[194] H. Wiltse and J. Nichols. PlayByPlay: Collaborative Web Browsing for Desktop and Mobile Devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 1781–1790, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-246-7. doi: 10.1145/1518701.1518975. URL `http://doi.acm.org/10.1145/1518701.1518975`. Cited on page 22.

[195] P. Wozniak, N. Goyal, P. Kucharski, L. Lischke, S. Mayer, and M. Fjeld. RAMPARTS: Supporting Sensemaking with Spatially-Aware Mobile Interactions. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 2447–2460, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3362-7. doi: 10.1145/2858036.2858491. URL `http://doi.acm.org/10.1145/2858036.2858491`. Cited on page 22.

[196] R. Xiao, S. Hudson, and C. Harrison. CapCam: Enabling Rapid, Ad-Hoc, Position-Tracked Interactions Between Devices. In *Proceedings of the 2016 ACM on Interactive Surfaces and Spaces*, ISS '16, pages 169–178, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4248-3. doi: 10.1145/2992154.2992182. URL `http://doi.acm.org/10.1145/2992154.2992182`. Cited on page 18.

[197] J. Yang and D. Wigdor. Panelrama: Enabling Easy Specification of Cross-Device Web Applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 2783–2792, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2473-1. doi: 10.1145/2556288.2557199. URL `http://doi.acm.org/10.1145/2556288.2557199`. Cited on pages 3, 7, 19, 20, 28, 29, 30, 31, 32, and 82.

[198] J. Zagermann, U. Pfeil, R. Rädle, H.-C. Jetter, C. Klokmose, and H. Reiterer. When Tablets Meet Tabletops: The Effect of Tabletop Size on Around-the-Table Collaboration with Personal Tablets. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, pages 5470–5481, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3362-7. doi: 10.1145/2858036.2858224. URL `http://doi.acm.org/10.1145/2858036.2858224`. Cited on page 22.